A Comprehensive Introduction to Software Metric

Oscar Morin

First Edition, 2012

ISBN 978-81-323-4058-4

Published by:
White Word Publications
4735/22 Prakashdeep Bldg,
Ansari Road, Darya Ganj,
Delhi - 110002
Email: info@wtbooks.com

Table of Contents

Introduction

A **software metric** is a measure of some property of a piece of software or its specifications. Since quantitative measurements are essential in all sciences, there is a continuous effort by computer science practitioners and theoreticians to bring similar approaches to software development. The goal is obtaining objective, reproducible and quantifiable measurements, which may have numerous valuable applications in schedule and budget planning, cost estimation, quality assurance testing, software debugging, software performance optimization, and optimal personnel task assignments.

Common software measurements

Common software measurements include:

- Balanced scorecard
- Bugs per line of code
- COCOMO
- Code coverage
- Cohesion
- Comment density
- Connascent software components
- Coupling
- Cyclomatic complexity (McCabe's complexity)
- Function point analysis
- Halstead Complexity
- Instruction path length
- Number of classes and interfaces
- Number of lines of code
- Number of lines of customer requirements
- Program execution time
- Program load time
- Program size (binary)
- Robert Cecil Martin’s software package metrics
- Weighted Micro Function Points

Limitations

As software development is a complex process, with high variance on both methodologies and objectives, it is difficult to define or measure software qualities and

quantities and to determine a valid and concurrent measurement metric, especially when making such a prediction prior to the detail design. Another source of difficulty and debate is in determining which metrics matter, and what they mean. The practical utility of *software* measurements has thus been limited to narrow domains where they include:

- Schedule
- Size/Complexity
- Cost
- Quality

Common goal of measurement may target one or more of the above aspects, or the balance between them as indicator of team's motivation or project performance.

Acceptance and Public Opinion

Some software development practitioners point out that simplistic measurements can cause more harm than good. Others have noted that metrics have become an integral part of the software development process. Impact of measurement on programmers psychology have raised concerns for harmful effects to performance due to stress, performance anxiety, and attempts to cheat the metrics, while others find it to have positive impact on developers value towards their own work, and prevent them being undervalued. Some argue that the definition of many measurement methodologies are imprecise, and consequently it is often unclear how tools for computing them arrive at a particular result, while others argue that imperfect quantification is better than none ("You can't control what you can't measure."). Evidence shows that software metrics are being widely used by government agencies, the US military, NASA, IT consultants, academic institutions, and commercial and academic development estimation software.

Chapter 1

Software Bug

A **software bug** is the common term used to describe an error, flaw, mistake, failure, or fault in a computer program or system that produces an incorrect or unexpected result, or causes it to behave in unintended ways. Most bugs arise from mistakes and errors made by people in either a program's source code or its design, and a few are caused by compilers producing incorrect code. A program that contains a large number of bugs, and/or bugs that seriously interfere with its functionality, is said to be *buggy*. Reports detailing bugs in a program are commonly known as bug reports, fault reports, problem reports, trouble reports, change requests, and so forth.

Effects

Bugs trigger Type I and type II errors that can in turn have a wide variety of ripple effects, with varying levels of inconvenience to the user of the program. Some bugs have only a subtle effect on the program's functionality, and may thus lie undetected for a long time. More serious bugs may cause the program to crash or freeze leading to a denial of service. Others qualify as security bugs and might for example enable a malicious user to bypass access controls in order to obtain unauthorized privileges.

The results of bugs may be extremely serious. Bugs in the code controlling the Therac-25 radiation therapy machine were directly responsible for some patient deaths in the 1980s. In 1996, the European Space Agency's US$1 billion prototype Ariane 5 rocket was destroyed less than a minute after launch, due to a bug in the on-board guidance computer program. In June 1994, a Royal Air Force Chinook crashed into the Mull of Kintyre, killing 29. This was initially dismissed as pilot error, but an investigation by *Computer Weekly* uncovered sufficient evidence to convince a House of Lords inquiry that it may have been caused by a software bug in the aircraft's engine control computer.

In 2002, a study commissioned by the US Department of Commerce' National Institute of Standards and Technology concluded that *software bugs, or errors, are so prevalent and so detrimental that they cost the US economy an estimated $59 billion annually, or about 0.6 percent of the gross domestic product*.

Etymology

The concept that software might contain errors dates back to 1843 in Ada Byron's notes on the analytical engine in which she speaks of the difficulty of preparing program 'cards' for Charles Babbage's Analytical engine:

> “ ...an analyzing process must equally have been performed in order to furnish the Analytical Engine with the necessary operative data; and that herein may also lie a possible source of error. Granted that the actual mechanism is unerring in its processes, the cards may give it wrong orders. ”

Use of the term "bug" to describe inexplicable defects has been a part of engineering jargon for many decades and predates computers and computer software; it may have originally been used in hardware engineering to describe mechanical malfunctions. For instance, Thomas Edison wrote the following words in a letter to an associate in 1878:

> “ It has been just so in all of my inventions. The first step is an intuition, and comes with a burst, then difficulties arise—this thing gives out and *[it is]* then that 'Bugs' — as such little faults and difficulties are called—show themselves and months of intense watching, study and labor are requisite before commercial success or failure is certainly reached. ”

Problems with radar electronics during World War II were referred to as *bug*s (or glitches) and there is additional evidence that the usage dates back much earlier. Baffle Ball, the first mechanical pinball game, was advertised as being "free of bugs" in 1931.

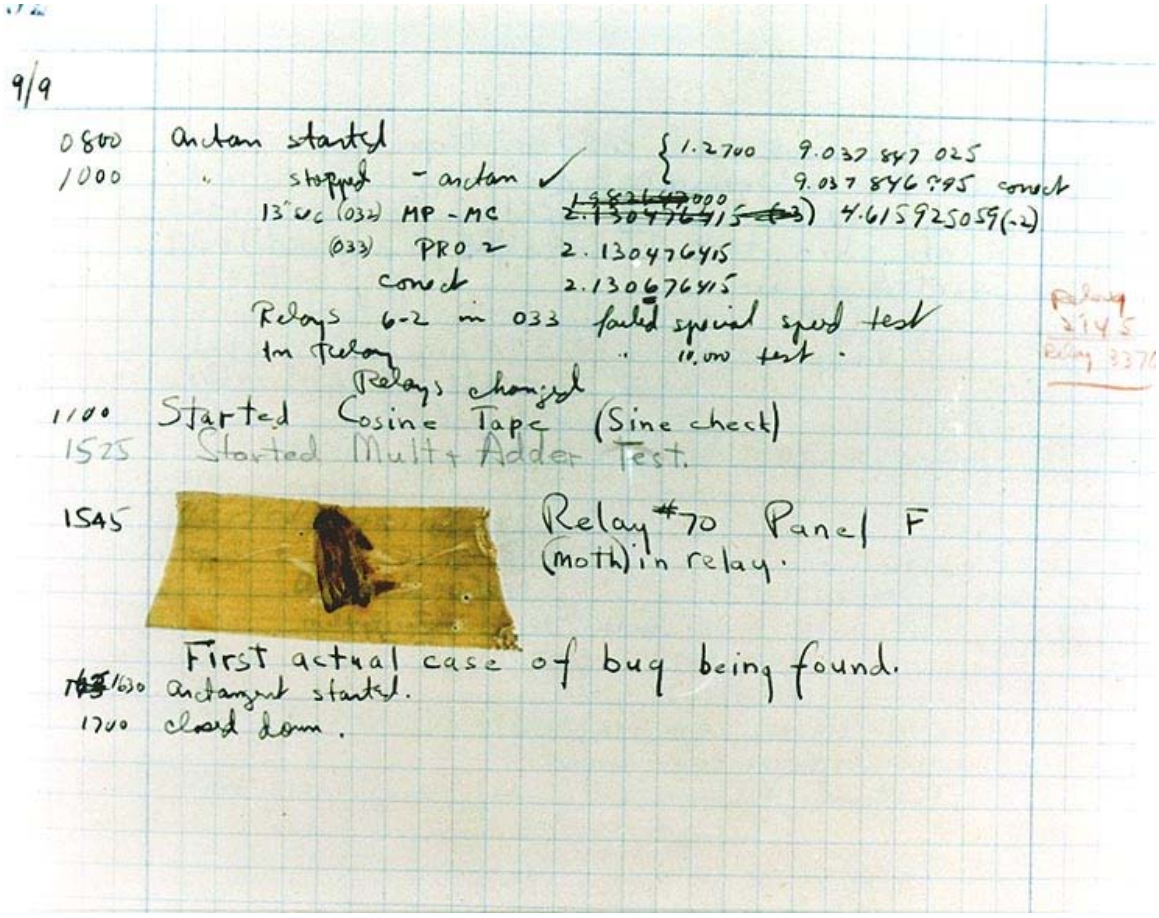


Photo of what is possibly the first real bug found in a computer.

The invention of the term "bug" is often erroneously attributed to Grace Hopper, who publicized the cause of a malfunction in an early electromechanical computer. A typical version of the story is given by this quote:

> “ In 1946, when Hopper was released from active duty, she joined the Harvard Faculty at the Computation Laboratory where she continued her work on the Mark II and Mark III. Operators traced an error in the Mark II to a moth trapped in a relay, coining the term *bug*. This bug was carefully removed and taped to the log book. Stemming from the first bug, today we call errors or glitch's [*sic*] in a program a *bug*. ”

Hopper was not actually the one who found the insect, as she readily acknowledged. The date in the log book was 9 September 1947, although sometimes erroneously reported as 1945. The operators who did find it, including William "Bill" Burke, later of the Naval Weapons Laboratory, Dahlgren, Virginia, were familiar with the engineering term and, amused, kept the insect with the notation "First actual case of bug being found." Hopper loved to recount the story. This log book is on display in the Smithsonian National Museum of American History, complete with moth attached.

While it is certain that the Harvard Mark II operators did not coin the term "bug", it has been suggested that they did coin the related term, "debug". Even this is unlikely, since the Oxford English Dictionary entry for "debug" contains a use of "debugging" in the context of air-plane engines in 1945. *See:* debugging.

Prevention

Bugs are a consequence of the nature of human factors in the programming task. They arise from oversights or mutual misunderstandings made by a software team during specification, design, coding, data entry and documentation. For example: In creating a relatively simple program to sort a list of words into alphabetical order, one's design might fail to consider what should happen when a word contains a hyphen. Perhaps, when converting the abstract design into the chosen programming language, one might inadvertently create an off-by-one error and fail to sort the last word in the list. Finally, when typing the resulting program into the computer, one might accidentally type a '<' where a '>' was intended, perhaps resulting in the words being sorted into reverse alphabetical order. More complex bugs can arise from unintended interactions between different parts of a computer program. This frequently occurs because computer programs can be complex—millions of lines long in some cases—often having been programmed by many people over a great length of time, so that programmers are unable to mentally track every possible way in which parts can interact. Another category of bug called a *race condition* comes about either when a process is running in more than one thread or two or more processes run simultaneously, and the exact order of execution of the critical sequences of code have not been properly synchronized.

The software industry has put much effort into finding methods for preventing programmers from inadvertently introducing bugs while writing software. These include:

Programming style
: While typos in the program code are often caught by the compiler, a bug usually appears when the programmer makes a logic error. Various innovations in programming style and defensive programming are designed to make these bugs less likely, or easier to spot. In some programming languages, so-called typos, especially of symbols or logical/mathematical operators, actually represent logic errors, since the mistyped constructs are accepted by the compiler with a meaning other than that which the programmer intended.

Programming techniques
: Bugs often create inconsistencies in the internal data of a running program. Programs can be written to check the consistency of their own internal data while running. If an inconsistency is encountered, the program can immediately halt, so that the bug can be located and fixed. Alternatively, the program can simply inform the user, attempt to correct the inconsistency, and continue running.

Development methodologies
: There are several schemes for managing programmer activity, so that fewer bugs are produced. Many of these fall under the discipline of software engineering (which addresses software design issues as well). For example, formal program

specifications are used to state the exact behavior of programs, so that design bugs can be eliminated. Unfortunately, formal specifications are impractical or impossible for anything but the shortest programs, because of problems of combinatorial explosion and indeterminacy.

Programming language support

Programming languages often include features which help programmers prevent bugs, such as static type systems, restricted name spaces and modular programming, among others. For example, when a programmer writes (pseudocode) `LET REAL_VALUE PI = "THREE AND A BIT"`, although this may be syntactically correct, the code fails a type check. Depending on the language and implementation, this may be caught by the compiler or at runtime. In addition, many recently-invented languages have deliberately excluded features which can easily lead to bugs, at the expense of making code slower than it need be: the general principle being that, because of Moore's law, computers get faster and software engineers get slower; it is *almost always* better to write simpler, slower code than "clever", inscrutable code, especially considering that maintenance cost is considerable. For example, the Java programming language does not support pointer arithmetic; implementations of some languages such as Pascal and scripting languages often have runtime bounds checking of arrays, at least in a debugging build.

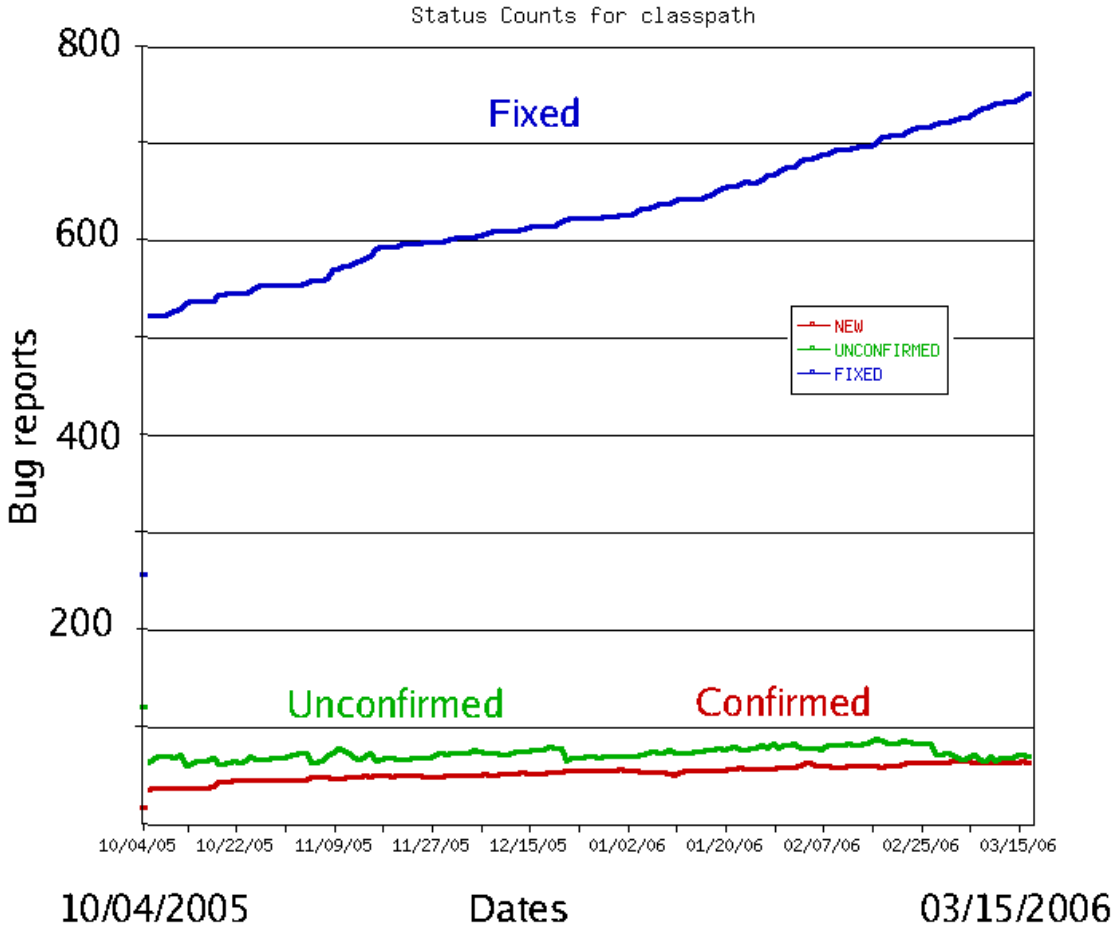
Code analysis

Tools for code analysis help developers by inspecting the program text beyond the compiler's capabilities to spot potential problems. Although in general the problem of finding all programming errors given a specification is not solvable, these tools exploit the fact that human programmers tend to make the same kinds of mistakes when writing software.

Instrumentation

Tools to monitor the performance of the software as it is running, either specifically to find problems such as bottlenecks or to give assurance as to correct working, may be embedded in the code explicitly (perhaps as simple as a statement saying `PRINT "I AM HERE"`), or provided as tools. It is often a surprise to find where most of the time is taken by a piece of code, and this removal of assumptions might cause the code to be rewritten.

Debugging



The typical bug history (GNU Classpath project data). A new bug submitted by the user is *unconfirmed.* Once it has been reproduced by a developer, it is a *confirmed* bug. The confirmed bugs are later *fixed*. Bugs belonging to other categories (unreproducible, will not be fixed, etc.) are usually in the minority

Finding and fixing bugs, or "debugging", has always been a major part of computer programming. Maurice Wilkes, an early computing pioneer, described his realization in the late 1940s that much of the rest of his life would be spent finding mistakes in his own programs. As computer programs grow more complex, bugs become more common and difficult to fix. Often programmers spend more time and effort finding and fixing bugs than writing new code. Software testers are professionals whose primary task is to find bugs, or write code to support testing. On some projects, more resources can be spent on testing than in developing the program.

Usually, the most difficult part of debugging is finding the bug in the source code. Once it is found, correcting it is usually relatively easy. Programs known as debuggers exist to help programmers locate bugs by executing code line by line, watching variable values, and other features to observe program behavior. Without a debugger, code can be added

so that messages or values can be written to a console (for example with *printf* in the C programming language) or to a window or log file to trace program execution or show values.

However, even with the aid of a debugger, locating bugs is something of an art. It is not uncommon for a bug in one section of a program to cause failures in a completely different section, thus making it especially difficult to track (for example, an error in a graphics rendering routine causing a file I/O routine to fail), in an apparently unrelated part of the system.

Sometimes, a bug is not an isolated flaw, but represents an error of thinking or planning on the part of the programmer. Such *logic errors* require a section of the program to be overhauled or rewritten. As a part of Code review, stepping through the code modelling the execution process in one's head or on paper can often find these errors without ever needing to reproduce the bug as such, if it can be shown there is some faulty logic in its implementation.

But more typically, the first step in locating a bug is to reproduce it reliably. Once the bug is reproduced, the programmer can use a debugger or some other tool to monitor the execution of the program in the faulty region, and find the point at which the program went astray.

It is not always easy to reproduce bugs. Some are triggered by inputs to the program which may be difficult for the programmer to re-create. One cause of the Therac-25 radiation machine deaths was a bug (specifically, a race condition) that occurred only when the machine operator very rapidly entered a treatment plan; it took days of practice to become able to do this, so the bug did not manifest in testing or when the manufacturer attempted to duplicate it. Other bugs may disappear when the program is run with a debugger; these are heisenbugs (humorously named after the Heisenberg uncertainty principle.)

Debugging is still a tedious task requiring considerable effort. Since the 1990s, particularly following the Ariane 5 Flight 501 disaster, there has been a renewed interest in the development of effective automated aids to debugging. For instance, methods of static code analysis by abstract interpretation have already made significant achievements, while still remaining much of a work in progress.

As with any creative act, sometimes a flash of inspiration will show a solution, but this is rare and, by definition, cannot be relied on.

There are also classes of bugs that have nothing to do with the code itself. If, for example, one relies on faulty documentation or hardware, the code may be written perfectly properly to what the documentation says, but the bug truly lies in the documentation or hardware, not the code. However, it is common to change the code instead of the other parts of the system, as the cost and time to change it is generally less. Embedded systems frequently have workarounds for hardware bugs, since to make a new version of a ROM

is much cheaper than remanufacturing the hardware, especially if they are commodity items.

Bug management

It is common practice for software to be released with known bugs that are considered non-critical, that is, that do not affect most users' main experience with the product. While software products may, by definition, contain any number of unknown bugs, measurements during testing can provide an estimate of the number of likely bugs remaining; this becomes more reliable the longer a product is tested and developed ("if we had 200 bugs last week, we should have 100 this week"). Most big software projects maintain two lists of "known bugs"— those known to the software team, and those to be told to users. This is not dissimulation, but users are not concerned with the internal workings of the product. The second list informs users about bugs that are not fixed in the current release, or not fixed at all, and a workaround may be offered.

There are various reasons for not fixing bugs:

- The developers often don't have time or it is not economical to fix all non-severe bugs.
- The bug could be fixed in a new version or patch that is not yet released.
- The changes to the code required to fix the bug could be large, expensive, or delay finishing the project.
- Even seemingly simple fixes bring the chance of introducing new unknown bugs into the system. At the end of a test/fix cycle some managers may only allow the most critical bugs to be fixed.
- Users may be relying on the undocumented, buggy behavior, especially if scripts or macros rely on a behavior; it may introduce a breaking change.
- It's "not a bug". A misunderstanding has arisen between expected and provided behavior

Given the above, it is often considered impossible to write completely bug-free software of any real complexity. So bugs are categorized by severity, and low-severity non-critical bugs are tolerated, as they do not affect the proper operation of the system for most users. NASA's SATC managed to reduce the number of errors to fewer than 0.1 per 1000 lines of code (SLOC) but this was not felt to be feasible for any real world projects.

The severity of a bug is not the same as its importance for fixing, and the two should be measured and managed separately. On a Microsoft Windows system a blue screen of death is rather severe, but if it only occurs in extreme circumstances, especially if they are well diagnosed and avoidable, it may be less important to fix than an icon not representing its function well, which though purely aesthetic may confuse thousands of users every single day. This balance, of course, depends on many factors; expert users have different expectations from novices, a niche market is different from a general consumer market, and so on. To better achieve this balance, some software developers use a formalized *bug triage* process (borrowing the medical term), in which each new

bug is assigned a priority based on its severity, frequency, risk, and other predetermined factors.

A school of thought popularized by Eric S. Raymond as Linus's Law says that popular open-source software has more chance of having few or no bugs than other software, because "given enough eyeballs, all bugs are shallow". This assertion has been disputed, however: computer security specialist Elias Levy wrote that "it is easy to hide vulnerabilities in complex, little understood and undocumented source code," because, "even if people are reviewing the code, that doesn't mean they're qualified to do so."

Like any other part of engineering management, bug management must be conducted carefully and intelligently because "what gets measured gets done" and managing purely by bug counts can have unintended consequences. If, for example, developers are rewarded by the number of bugs they fix, they will naturally fix the easiest bugs first—leaving the hardest, and probably most risky or critical, to the last possible moment ("I only have one bug on my list but it says "Make sun rise in West"). If the management ethos is to reward the number of bugs fixed, then some developers may quickly write sloppy code knowing they can fix the bugs later and be rewarded for it, whereas careful, perhaps "slower" developers do not get rewarded for the bugs that were never there.

Security vulnerabilities

Malicious software may attempt to exploit known vulnerabilities in a system — which may or may not be bugs. Viruses are not bugs in themselves — they are typically programs that are doing precisely what they were designed to do. However, viruses are occasionally referred to as such in the popular press.

Common types of computer bugs

- Conceptual error (code is syntactically correct, but the programmer or designer intended it to do something else)

Arithmetic bugs

- Division by zero
- Arithmetic overflow or underflow
- Loss of arithmetic precision due to rounding or numerically unstable algorithms

Logic bugs

- Infinite loops and infinite recursion
- Off by one error, counting one too many or too few when looping

Syntax bugs

- Use of the wrong operator, such as performing assignment instead of equality test. In simple cases often warned by the compiler; in many languages, deliberately guarded against by language syntax

Resource bugs

- Null pointer dereference
- Using an uninitialized variable
- Using an otherwise valid instruction on the wrong data type
- Access violations
- Resource leaks, where a finite system resource such as memory or file handles are exhausted by repeated allocation without release.
- Buffer overflow, in which a program tries to store data past the end of allocated storage. This may or may not lead to an access violation or storage violation. These bugs can form a security vulnerability.
- Excessive recursion which though logically valid causes stack overflow

Multi-threading programming bugs

- Deadlock
- Race condition
- Concurrency errors in critical sections, mutual exclusions and other features of concurrent processing. Time-of-check-to-time-of-use (TOCTOU) is a form of unprotected critical section.

Interfacing bugs

- Incorrect API usage
- Incorrect protocol implementation
- Incorrect hardware handling

Teamworking bugs

- Unpropagated updates; e.g. programmer changes "myAdd" but forgets to change "mySubtract", which uses the same algorithm. These errors are mitigated by the Don't Repeat Yourself philosophy.
- Comments out of date or incorrect: many programmers assume the comments accurately describe the code
- Differences between documentation and the actual product

Chapter 2

COCOMO

The **Constructive Cost Model** (**COCOMO**) is an algorithmic software cost estimation model developed by Barry Boehm. The model uses a basic regression formula, with parameters that are derived from historical project data and current project characteristics.

COCOMO was first published in 1981 Barry W. Boehm's Book *Software engineering economics* as a model for estimating effort, cost, and schedule for software projects. It drew on a study of 63 projects at TRW Aerospace where Barry Boehm was Director of Software Research and Technology in 1981. The study examined projects ranging in size from 2,000 to 100,000 lines of code, and programming languages ranging from assembly to PL/I. These projects were based on the waterfall model of software development which was the prevalent software development process in 1981.

References to this model typically call it *COCOMO 81*. In 1997 *COCOMO II* was developed and finally published in 2000 in the book *Software Cost Estimation with COCOMO II*. COCOMO II is the successor of COCOMO 81 and is better suited for estimating modern software development projects. It provides more support for modern software development processes and an updated project database. The need for the new model came as software development technology moved from mainframe and overnight batch processing to desktop development, code reusability and the use of off-the-shelf software components. Here, we refers to *COCOMO 81*.

COCOMO consists of a hierarchy of three increasingly detailed and accurate forms. The first level, *Basic COCOMO* is good for quick, early, rough order of magnitude estimates of software costs, but its accuracy is limited due to its lack of factors to account for difference in project attributes (*Cost Drivers*). *Intermediate COCOMO* takes these Cost Drivers into account and *Detailed COCOMO* additionally accounts for the influence of individual project phases.

Basic COCOMO

Basic COCOMO computes software development effort (and cost) as a function of program size. Program size is expressed in estimated thousands of lines of code (KLOC)

COCOMO applies to three classes of software projects:

- Organic projects - "small" teams with "good" experience working with "less than rigid" requirements
- Semi-detached projects - "medium" teams with mixed experience working with a mix of rigid and less than rigid requirements
- Embedded projects - developed within a set of "tight" constraints (hardware, software, operational, ...)

The basic COCOMO equations take the form

Effort Applied (E) = $a_b(KLOC)^{b}_{b}$ **[man-months]**
Development Time (D) = $c_b(\text{Effort Applied})^{d}_{b}$ **[months]**
People required (P) = Effort Applied / Development Time **[count]**

where, **KLOC** is the estimated number of delivered lines (expressed in thousands) of code for project, The coefficients a_b, b_b, c_b and d_b are given in the following table.

Software project	a_b	b_b	c_b	d_b
Organic	2.4	1.05	2.5	0.38
Semi-detached	3.0	1.12	2.5	0.35
Embedded	3.6	1.20	2.5	0.32

Basic COCOMO is good for quick estimate of software costs. However it does not account for differences in hardware constraints, personnel quality and experience, use of modern tools and techniques, and so on.

Intermediate COCOMOs

Intermediate COCOMO computes software development effort as function of program size and a set of "cost drivers" that include subjective assessment of product, hardware, personnel and project attributes. This extension considers a set of four "cost drivers",each with a number of subsidiary attributes:-

- Product attributes
  - Required software reliability
  - Size of application database
  - Complexity of the product
- Hardware attributes
  - Run-time performance constraints
  - Memory constraints
  - Volatility of the virtual machine environment
  - Required turnabout time
- Personnel attributes
  - Analyst capability

    - Software engineering capability
    - Applications experience
    - Virtual machine experience
    - Programming language experience
- Project attributes
    - Use of software tools
    - Application of software engineering methods
    - Required development schedule

Each of the 15 attributes receives a rating on a six-point scale that ranges from "very low" to "extra high" (in importance or value). An effort multiplier from the table below applies to the rating. The product of all effort multipliers results in an *effort adjustment factor (EAF)*. Typical values for EAF range from 0.9 to 1.4.

	Ratings					
Cost Drivers	Very Low	Low	Nominal	High	Very High	Extra High
Product attributes						
Required software reliability	0.75	0.88	1.00	1.15	1.40	
Size of application database		0.94	1.00	1.08	1.16	
Complexity of the product	0.70	0.85	1.00	1.15	1.30	1.65
Hardware attributes						
Run-time performance constraints			1.00	1.11	1.30	1.66
Memory constraints			1.00	1.06	1.21	1.56
Volatility of the virtual machine environment		0.87	1.00	1.15	1.30	
Required turnabout time		0.87	1.00	1.07	1.15	
Personnel attributes						
Analyst capability	1.46	1.19	1.00	0.86	0.71	
Applications experience	1.29	1.13	1.00	0.91	0.82	
Software engineer capability	1.42	1.17	1.00	0.86	0.70	
Virtual machine experience	1.21	1.10	1.00	0.90		
Programming language experience	1.14	1.07	1.00	0.95		
Project attributes						
Application of software engineering methods	1.24	1.10	1.00	0.91	0.82	
Use of software tools	1.24	1.10	1.00	0.91	0.83	
Required development schedule	1.23	1.08	1.00	1.04	1.10	

The Intermediate Cocomo formula now takes the form:

$$E=a_i(KLoC)^{(b_i)}.EAF$$

where E is the effort applied in person-months, **KLoC** is the estimated number of thousands of delivered lines of code for the project, and **EAF** is the factor calculated above. The coefficient $\mathbf{a_i}$ and the exponent $\mathbf{b_i}$ are given in the next table.

Software project	$\mathbf{a_i}$	$\mathbf{b_i}$
Organic	3.2	1.05
Semi-detached	3.0	1.12
Embedded	2.8	1.20

The Development time **D** calculation uses **E** in the same way as in the Basic COCOMO.

Detailed COCOMO

Detailed COCOMO - incorporates all characteristics of the intermediate version with an assessment of the cost driver's impact on each step (analysis, design, etc.) of the software engineering process 1. the detailed model uses different efforts multipliers for each cost drivers attribute these **Phase Sensitive** effort multipliers are each to determine the amount of effort required to complete each phase.

In detailed COCOMO, the effort is calculated as function of program size and a set of cost drivers given according to each phase of software life cycle. The phases used in detailed **COCOMO** are requirement planning and software design, detailed design, code and unit test, and integration testing.

Projects using COCOMO

five phases of detailed cocomo are:- -plan and requirement. -system design. -detailed design. -module code and test. -integration and test.

Chapter 3

Code Coverage

Code coverage is a measure used in software testing. It describes the degree to which the source code of a program has been tested. It is a form of testing that inspects the code directly and is therefore a form of white box testing. In time, the use of code coverage has been extended to the field of digital hardware, the contemporary design methodology of which relies on hardware description languages (HDLs).

Code coverage was among the first methods invented for systematic software testing. The first published reference was by Miller and Maloney in *Communications of the ACM* in 1963.

Code coverage is one consideration in the safety certification of avionics equipment. The standard by which avionics gear is certified by the Federal Aviation Administration (FAA) is documented in DO-178B.

Coverage criteria

To measure how well the program is exercised by a test suite, one or more *coverage criteria* are used.

Basic coverage criteria

There are a number of coverage criteria, the main ones being:

- **Function coverage** - Has each function (or subroutine) in the program been called?
- **Statement coverage** - Has each node in the program been executed?
- **Decision coverage** (not the same as **branch coverage**.) - Has every edge in the program been executed? For instance, have the requirements of each branch of each control structure (such as in IF and CASE statements) been met as well as not met?

- **Condition coverage** (or predicate coverage) - Has each boolean sub-expression evaluated both to true and false? This does not necessarily imply decision coverage.
- **Condition/decision coverage** - Both decision and condition coverage should be satisfied.

For example, consider the following C++ function:

```
int foo(int x, int y)
{
    int z = 0;
    if ((x>0) && (y>0)) {
        z = x;
    }
    return z;
}
```

Assume this function is a part of some bigger program and this program was run with some test suite.

- If during this execution function 'foo' was called at least once, then *function coverage* for this function is satisfied.
- *Statement coverage* for this function will be satisfied if it was called e.g. as `foo(1,1)`, as in this case, every line in the function is executed including `z = x;`.
- Tests calling `foo(1,1)` and `foo(0,1)` will satisfy *decision coverage*, as in the first case the `if` condition and the short circuit condition are satisfied and `z = x;` is executed, and in the second neither conditional is satisfied and x is not assigned to z.
- *Condition coverage* can be satisfied with tests that call `foo(1,1)`, `foo(1,0)` and `foo(0,0)`. These are necessary as in the first two cases `(x>0)` evaluates to `true` while in the third it evaluates `false`. At the same time, the first case makes `(y>0)` `true` while the second and third make it `false`.

In languages, like Pascal, where standard boolean operations are not short circuited, condition coverage does not necessarily imply decision coverage. For example, consider the following fragment of code:

```
if a and b then
```

Condition coverage can be satisfied by two tests:

- `a=true`, `b=false`
- `a=false`, `b=true`

However, this set of tests does not satisfy decision coverage as in neither case will the `if` condition be met.

Fault injection may be necessary to ensure that all conditions and branches of exception handling code have adequate coverage during testing.

Modified condition/decision coverage

For safety-critical applications (e.g. for avionics software) it is often required that **modified condition/decision coverage (MC/DC)** is satisifed. This criteria extends condition/decision criteria with requirements that each condition should affect the decision outcome independently. For example, consider the following code:

```
if (a or b) and c then
```

The condition/decision criteria will be satisfied by the following set of tests:

- a=true, b=true, c=true
- a=false, b=false, c=false

However, the above tests set will not satisfy modified condition/decision coverage, since in the first test, the value of 'b' and in the second test the value of 'c' would not influence the output. So, the following test set is needed to satisfy MC/DC:

- a=**false**, b=**false**, c=true
- a=**true**, b=false, c=**true**
- a=false, b=**true**, c=**true**
- a=true, b=true, c=**false**

The bold values influence the output, each variable must be present as an influencing value at least once with false and once with true.

Multiple condition coverage

This criteria requires that all combinations of conditions inside each decision are tested. For example, the code fragment from the previous section will require eight tests:

- a=false, b=false, c=false
- a=false, b=false, c=true
- a=false, b=true, c=false
- a=false, b=true, c=true
- a=true, b=false, c=false
- a=true, b=false, c=true
- a=true, b=true, c=false
- a=true, b=true, c=true

Other coverage criteria

There are further coverage criteria, which are used less often:

- **Linear Code Sequence and Jump (LCSAJ) coverage** - has every LCSAJ been executed?
- **JJ-Path coverage** - have all jump to jump paths (aka LCSAJs) been executed?
- **Path coverage** - Has every possible route through a given part of the code been executed?
- **Entry/exit coverage** - Has every possible call and return of the function been executed?
- **Loop coverage** - Has every possible loop been executed zero times, once, and more than once?

Safety-critical applications are often required to demonstrate that testing achieves 100% of some form of code coverage.

Some of the coverage criteria above are connected. For instance, path coverage implies decision, statement and entry/exit coverage. Decision coverage implies statement coverage, because every statement is part of a branch.

Full path coverage, of the type described above, is usually impractical or impossible. Any module with a succession of *n* decisions in it can have up to 2^n paths within it; loop constructs can result in an infinite number of paths. Many paths may also be infeasible, in that there is no input to the program under test that can cause that particular path to be executed. However, a general-purpose algorithm for identifying infeasible paths has been proven to be impossible (such an algorithm could be used to solve the halting problem). Methods for practical path coverage testing instead attempt to identify classes of code paths that differ only in the number of loop executions, and to achieve "basis path" coverage the tester must cover all the path classes.

In practice

The target software is built with special options or libraries and/or run under a special environment such that every function that is exercised (executed) in the program(s) is mapped back to the function points in the source code. This process allows developers and quality assurance personnel to look for parts of a system that are rarely or never accessed under normal conditions (error handling and the like) and helps reassure test engineers that the most important conditions (function points) have been tested. The resulting output is then analyzed to see what areas of code have not been exercised and the tests are updated to include these areas as necessary. Combined with other code coverage methods, the aim is to develop a rigorous, yet manageable, set of regression tests.

In implementing code coverage policies within a software development environment one must consider the following:

- What are coverage requirements for the end product certification and if so what level of code coverage is required? The typical level of rigor progression is as

follows: Statement, Branch/Decision, Modified Condition/Decision Coverage(MC/DC), LCSAJ (Linear Code Sequence and Jump)
- Will code coverage be measured against tests that verify requirements levied on the system under test (DO-178B)?
- Is the object code generated directly traceable to source code statements? Certain certifications, (ie. DO-178B Level A) require coverage at the assembly level if this is not the case: "Then, additional verification should be performed on the object code to establish the correctness of such generated code sequences" (DO-178B) para-6.4.4.2.

Test engineers can look at code coverage test results to help them devise test cases and input or configuration sets that will increase the code coverage over vital functions. Two common forms of code coverage used by testers are statement (or line) coverage and path (or edge) coverage. Line coverage reports on the execution footprint of testing in terms of which lines of code were executed to complete the test. Edge coverage reports which branches or code decision points were executed to complete the test. They both report a coverage metric, measured as a percentage. The meaning of this depends on what form(s) of code coverage have been used, as 67% path coverage is more comprehensive than 67% statement coverage.

Generally, code coverage tools and libraries exact a performance and/or memory or other resource cost which is unacceptable to normal operations of the software. Thus, they are only used in the lab. As one might expect, there are classes of software that cannot be feasibly subjected to these coverage tests, though a degree of coverage mapping can be approximated through analysis rather than direct testing.

There are also some sorts of defects which are affected by such tools. In particular, some race conditions or similar real time sensitive operations can be masked when run under code coverage environments; and conversely, some of these defects may become easier to find as a result of the additional overhead of the testing code.

Software tools

Tools for C / C++

- BullseyeCoverage
- Cantata++
- Insure++
- IBM Rational Pure Coverage
- Tessy
- Testwell CTC++
- Trucov
- CodeScroll
- TestCocoon
- Semantic Designs Test Coverage for C and C++

Tools for C# .NET

- NCover
- Testwell CTC++ (with Java and C# add on)
- Semantic Designs Test Coverage for C#

Tools for COBOL

- Semantic Designs Test Coverage for COBOL

Tools for Java

- Clover
- Cobertura
- Structure 101
- EMMA
- Jtest
- Serenity
- Testwell CTC++ (with Java and C# add on)
- Semantic Designs Test Coverage for Java

Tools for PHP

- PHPUnit, also need Xdebug to make coverage reports
- Semantic Designs Test Coverage for PHP

Tools for PLSQL

- Semantic Designs Test Coverage for PLSQL

Hardware tools

- Aldec
- Atrenta
- Cadence Design Systems
- JEDA Technologies
- Mentor Graphics
- Nusym Technology
- Simucad Design Automation
- Synopsys

Chapter 4

Cohesion (Computer Science) and Connascent Software Components

Cohesion (computer science)

In computer programming, **cohesion** is a measure of how strongly-related the functionality expressed by the source code of a software module is. Methods of measuring **cohesion** vary from qualitative measures classifying the source text being analyzed using a rubric with a hermeneutics approach to quantitative measures which examine textual characteristics of the source code to arrive at a numerical cohesion score. Cohesion is an ordinal type of measurement and is usually expressed as "high cohesion" or "low cohesion" when being discussed. Modules with high cohesion tend to be preferable because high cohesion is associated with several desirable traits of software including robustness, reliability, reusability, and understandability whereas low cohesion is associated with undesirable traits such as being difficult to maintain, difficult to test, difficult to reuse, and even difficult to understand.

Cohesion is often contrasted with coupling, a different concept. Nonetheless high cohesion often correlates with loose coupling, and vice versa. The software quality metrics of coupling and cohesion were invented by Larry Constantine based on characteristics of "good" programming practices that reduced maintenance and modification costs.

High cohesion

In computer programming, cohesion is a measure of how strongly-related or focused the responsibilities of a single module are. As applied to object-oriented programming, if the methods that serve the given class tend to be similar in many aspects, then the class is said to have high cohesion. In a highly-cohesive system, code readability and the likelihood of reuse is increased, while complexity is kept manageable.

Cohesion is decreased if:

- The functionalities embedded in a class, accessed through its methods, have little in common.

- Methods carry out many varied activities, often using coarsely-grained or unrelated sets of data.

Disadvantages of low cohesion (or "weak cohesion") are:

- Increased difficulty in understanding modules.
- Increased difficulty in maintaining a system, because logical changes in the domain affect multiple modules, and because changes in one module require changes in related modules.
- Increased difficulty in reusing a module because most applications won't need the random set of operations provided by a module.

Types of cohesion

Cohesion is a qualitative measure meaning that the source code text to be measured is examined using a rubric to determine a cohesion classification. The types of cohesion, in order of the worst to the best type, are as follows:

Coincidental cohesion (worst)
: Coincidental cohesion is when parts of a module are grouped arbitrarily; the only relationship between the parts is that they have been grouped together (e.g. a "Utilities" class).

Logical cohesion
: Logical cohesion is when parts of a module are grouped because they logically are categorized to do the same thing, even if they are different by nature (e.g. grouping all mouse and keyboard input handling routines).

Temporal cohesion
: Temporal cohesion is when parts of a module are grouped by when they are processed - the parts are processed at a particular time in program execution (e.g. a function which is called after catching an exception which closes open files, creates an error log, and notifies the user).

Procedural cohesion
: Procedural cohesion is when parts of a module are grouped because they always follow a certain sequence of execution (e.g. a function which checks file permissions and then opens the file).

Communicational cohesion
: Communicational cohesion is when parts of a module are grouped because they operate on the same data (e.g. a module which operates on the same record of information).

Sequential cohesion
: Sequential cohesion is when parts of a module are grouped because the output from one part is the input to another part like an assembly line (e.g. a function which reads data from a file and processes the data).

Functional cohesion (best)

Functional cohesion is when parts of a module are grouped because they all contribute to a single well-defined task of the module (e.g. tokenizing a string of XML).

Although cohesion is a ranking type of scale, the ranks do not indicate a steady progression of improved cohesion. Studies by various people including Larry Constantine, Edward Yourdon, and Steve McConnell indicate that the first two types of cohesion are inferior; communicational and sequential cohesion are very good; and functional cohesion is superior.

While functional cohesion is considered the most desirable type of cohesion for a software module, it may not be achievable. There are cases where communicational cohesion is the highest level of cohesion that can be attained under the circumstances.

Connascent software components

Two software components are connascent if a change in one would require the other to be modified in order to maintain the overall correctness of the system. Connascence is a way to characterize and reason about certain types of complexity in software systems.

Strength

A form of connascence is considered to be stronger if it is more likely to require compensating changes in connascent elements. The stronger the form of connascence, the more difficult, and costly, it is to change the elements in the relationship.

Degree

The acceptability of connascence is related to the degree of its occurrence. Connascence might be acceptable in limited degree but unacceptable in large degree. For example, a function or method that takes two arguments is generally considered acceptable. However it is usually unacceptable for functions or methods to take ten arguments. Elements with a high degree of connascence incur greater difficulty, and cost, of change than elements that have a lower degree.

Locality

Locality matters when analyzing connascence. Stronger forms of connascence are acceptable if the elements involved are closely related. For example, many languages use positional arguments when calling functions or methods. This connascence of position is acceptable due the closeness of caller and callee. Passing arguments to a [web service] positionally is unacceptable due to the relative unrelatedness of the parties. The same strength and degree of connascence will have a higher difficulty, and cost, of change the less closely related the involved elements are.

Types of connascence

This is a list of some types of connascence ordered approximately from weak to strong forms

Connascence of Name
: Connascence of name is when multiple components must agree on the name of an entity.

Connascence of Type
: Connascence of type is when multiple components must agree on the type of an entity.

Connascence of Meaning
: Connascence of meaning is when multiple components must agree on the meaning particular values.

Connascence of Position
: Connascence of positions is when multiple components must agree on the order of values.

Connascence of Algorithm
: Connascence of algorithm is when multiple components must agree on a particular algorithm.

Connascence of Execution (order)
: Connascence of execution is when the order of execution of multiple components is important.

Connascence of Timing
: Connascence of timing is when the timing of the execution of multiple components is important.

Connascence of Identity
: Connascence of identity is when multiple components must reference the entity.

Reducing connascence

Reducing connascence will reduce the cost of change for a software system. One way of reducing connascence is by transforming strong forms of connascence into weaker forms. For example, a method that takes several arguments could be changed to use named parameters. This would change the connascence from position to name. Reducing the degree and increasing locality of involved elements are other ways to reduce connascence.

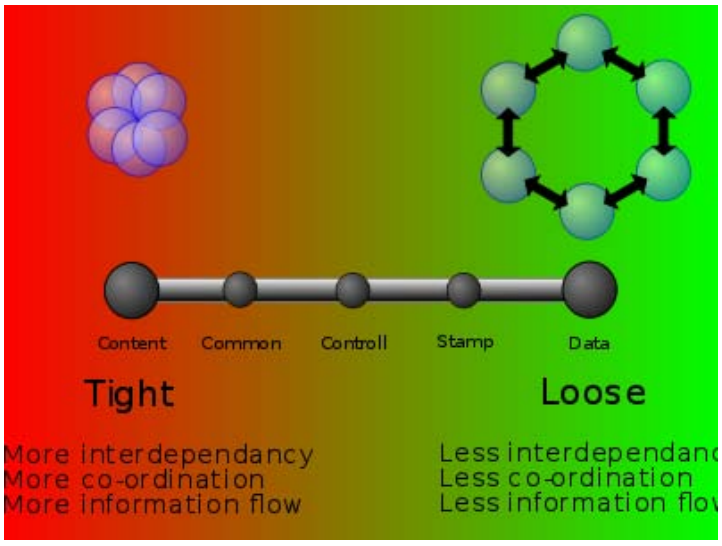
Chapter 5

Coupling (Computer Programming)

In computer science, **coupling** or **dependency** is the degree to which each program module relies on each one of the other modules.

Coupling is usually contrasted with cohesion. Low coupling often correlates with high cohesion, and vice versa. The software quality metrics of coupling and cohesion were invented by Larry Constantine, an original developer of Structured Design who was also an early proponent of these concepts. Low coupling is often a sign of a well-structured computer system and a good design, and when combined with high cohesion, supports the general goals of high readability and maintainability.

Types of coupling



Conceptual model of coupling

Coupling can be "low" (also "loose" and "weak") or "high" (also "tight" and "strong"). Some types of coupling, in order of highest to lowest coupling, are as follows:

Content coupling (high)

Content coupling is when one module modifies or relies on the internal workings of another module (e.g., accessing local data of another module).
Therefore changing the way the second module produces data (location, type, timing) will lead to changing the dependent module.

Common coupling
Common coupling is when two modules share the same global data (e.g., a global variable).
Changing the shared resource implies changing all the modules using it.

External coupling
External coupling occurs when two modules share an externally imposed data format, communication protocol, or device interface.

Control coupling
Control coupling is one module controlling the flow of another, by passing it information on what to do (e.g., passing a what-to-do flag).

Stamp coupling (Data-structured coupling)
Stamp coupling is when modules share a composite data structure and use only a part of it, possibly a different part (e.g., passing a whole record to a function that only needs one field of it).
This may lead to changing the way a module reads a record because a field that the module doesn't need has been modified.

Data coupling
Data coupling is when modules share data through, for example, parameters. Each datum is an elementary piece, and these are the only data shared (e.g., passing an integer to a function that computes a square root).

Message coupling (low)
This is the loosest type of coupling. It can be achieved by state decentralization (as in objects) and component communication is done via parameters or message passing.

No coupling
Modules do not communicate at all with one another.

Object-oriented programming

Subclass Coupling
Describes the relationship between a child and its parent. The child is connected to its parent, but the parent isn't connected to the child.

Temporal coupling
When two actions are bundled together into one module just because they happen to occur at the same time.

Disadvantages

Tightly coupled systems tend to exhibit the following developmental characteristics, which are often seen as disadvantages:

1. A change in one module usually forces a ripple effect of changes in other modules.
2. Assembly of modules might require more effort and/or time due to the increased inter-module dependency.
3. A particular module might be harder to reuse and/or test because dependent modules must be included.

Performance issues

Whether loosely or tightly coupled, a system's performance is often reduced by message and parameter creation, transmission, translation and interpretation overhead.

Message Creation Overhead and Performance
: Since all messages and parameters must possess particular meanings to be consumed (i.e., result in intended logical flow within the receiver), they must be created with a particular meaning. Creating any sort of message requires overhead in either CPU or memory usage. Creating a single integer value message (which might be a reference to a string, array or data structure) requires less overhead than creating a complicated message such as a SOAP message. Longer messages require more CPU and memory to produce. To optimize runtime performance, message length must be minimized and message meaning must be maximized.

Message Transmission Overhead and Performance
: Since a message must be transmitted in full to retain its complete meaning, message transmission must be optimized. Longer messages require more CPU and memory to transmit and receive. Also, when necessary, receivers must reassemble a message into its original state to completely receive it. Hence, to optimize runtime performance, message length must be minimized and message meaning must be maximized.

Message Translation Overhead and Performance
: Message protocols and messages themselves often contain extra information (i.e., packet, structure, definition and language information). Hence, the receiver often needs to translate a message into a more refined form by removing extra characters and structure information and/or by converting values from one type to another. Any sort of translation increases CPU and/or memory overhead. To optimize runtime performance, message form and content must be reduced and refined to maximize its meaning and reduce translation.

Message Interpretation Overhead and Performance
: All messages must be interpreted by the receiver. Simple messages such as integers might not require additional processing to be interpreted. However, complex messages such as SOAP messages require a parser and a string transformer for them to exhibit intended meanings. To optimize runtime performance, messages must be refined and reduced to minimize interpretation overhead.

Solutions

One approach to decreasing coupling is functional design, which seeks to limit the responsibilities of modules along functionality, coupling increases between two classes ***A*** and ***B*** if:

- ***A*** has an attribute that refers to (is of type) ***B***.
- ***A*** calls on services of an object ***B***.
- ***A*** has a method that references ***B*** (via return type or parameter).
- ***A*** is a subclass of (or implements) class ***B***.

Low coupling refers to a relationship in which one module interacts with another module through a simple and stable interface and does not need to be concerned with the other module's internal implementation.

Systems such as CORBA or COM allow objects to communicate with each other without having to know anything about the other object's implementation. Both of these systems even allow for objects to communicate with objects written in other languages.

Coupling versus Cohesion

Coupling and Cohesion are the two terms which very frequently occur together. Together they talk about the quality a module should have. Coupling talks about the inter dependencies between the various modules while cohesion describes how related functions within a module are. Low cohesion implies that module performs tasks which are not very related to each other and hence can create problems as the module becomes large.

Module coupling

Coupling in Software Engineering describes a version of metrics associated with this concept.

For data and control flow coupling:

- $\mathbf{d_i}$: number of input data parameters
- $\mathbf{c_i}$: number of input control parameters
- $\mathbf{d_o}$: number of output data parameters
- $\mathbf{c_o}$: number of output control parameters

For global coupling:

- $\mathbf{g_d}$: number of global variables used as data
- $\mathbf{g_c}$: number of global variables used as control

For environmental coupling:

- **w**: number of modules called (fan-out)
- **r**: number of modules calling the module under consideration (fan-in)

$$\text{Coupling}(C) = 1 - \frac{1}{d_i + 2 \times c_i + d_o + 2 \times c_o + g_d + 2 \times g_c + w + r}$$

`Coupling(C)` makes the value larger the more coupled the module is. This number ranges from approximately 0.67 (low coupling) to 1.0 (highly coupled)

For example, if a module has only a single input and output data parameter

$$C = 1 - \frac{1}{1+0+1+0+0+0+1+0} = 1 - \frac{1}{3} = 0.67$$

If a module has 5 input and output data parameters, an equal number of control parameters, and accesses 10 items of global data, with a fan-in of 3 and a fan-out of 4,

$$C = 1 - \frac{1}{5+2\times5+5+2\times5+10+0+3+4} = 0.98$$

Chapter 6

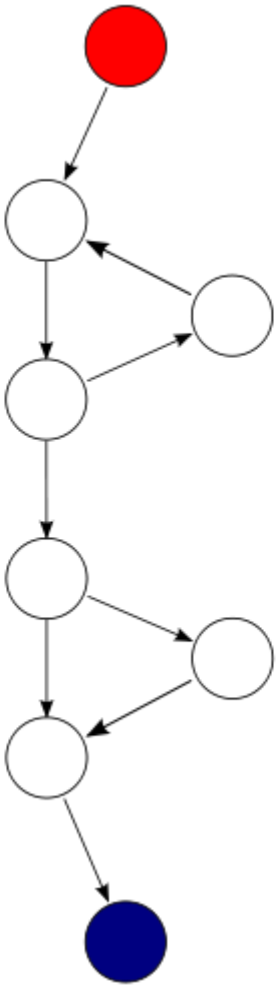
Cyclomatic Complexity

Cyclomatic complexity (or **conditional complexity**) is a software metric (measurement). It was developed by Thomas J. McCabe, Sr. in 1976 and is used to indicate the complexity of a program. It directly measures the number of linearly independent paths through a program's source code. The concept, although not the method, is somewhat similar to that of general text complexity measured by the Flesch-Kincaid Readability Test.

Cyclomatic complexity is computed using the control flow graph of the program: the nodes of the graph correspond to indivisible groups of commands of a program, and a directed edge connects two nodes if the second command might be executed immediately after the first command. Cyclomatic complexity may also be applied to individual functions, modules, methods or classes within a program.

One testing strategy, called **Basis Path Testing** by McCabe who first proposed it, is to test each linearly independent path through the program; in this case, the number of test cases will equal the cyclomatic complexity of the program.

Description



A control flow graph of a simple program. The program begins executing at the red node, then enters a loop (group of three nodes immediately below the red node). On exiting the loop, there is a conditional statement (group below the loop), and finally the program exits at the blue node. For this graph, E = 9, N = 8 and P = 1, so the cyclomatic complexity of the program is 3.

The cyclomatic complexity of a section of source code is the count of the number of linearly independent paths through the source code. For instance, if the source code contained no decision points such as IF statements or FOR loops, the complexity would be 1, since there is only a single path through the code. If the code had a single IF statement containing a single condition there would be two paths through the code, one path where the IF statement is evaluated as TRUE and one path where the IF statement is evaluated as FALSE.

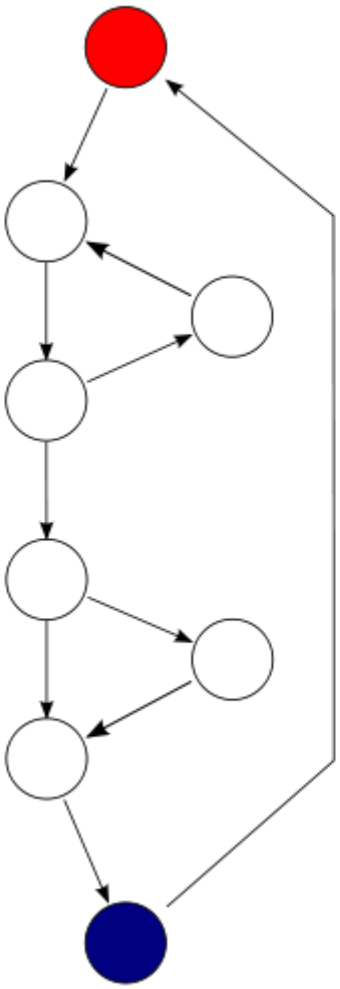
Mathematically, the cyclomatic *complexity* of a structured program is defined with reference to a directed graph containing the basic blocks of the program, with an edge

between two basic blocks if control may pass from the first to the second (the *control flow graph* of the program). The complexity is then defined as:

$$M = E - N + 2P$$

where

M = cyclomatic complexity
E = the number of edges of the graph
N = the number of nodes of the graph
P = the number of connected components



The same function as above, shown as a *strongly connected* control flow graph, for calculation via the alternative method. For this graph, E = 10, N = 8 and P = 1, so the cyclomatic complexity of the program is still 3.

An alternative formulation is to use a graph in which each exit point is connected back to the entry point. In this case, the graph is said to be *strongly connected*, and the cyclomatic

complexity of the program is equal to the cyclomatic *number* of its graph (also known as the first Betti number), which is defined as:

$$M = E - N + P$$

This may be seen as calculating the number of linearly independent cycles that exist in the graph, i.e. those cycles that do not contain other cycles within themselves. Note that because each exit point loops back to the entry point, there is at least one such cycle for each exit point.

For a single program (or subroutine or method), P is always equal to 1. Cyclomatic complexity may, however, be applied to several such programs or subprograms at the same time (e.g., to all of the methods in a class), and in these cases P will be equal to the number of programs in question, as each subprogram will appear as a disconnected subset of the graph.

It can be shown that the cyclomatic complexity of any structured program with only one entrance point and one exit point is equal to the number of decision points (i.e., 'if' statements or conditional loops) contained in that program plus one.

Cyclomatic complexity may be extended to a program with multiple exit points; in this case it is equal to:

$$\pi - s + 2$$

where π is the number of decision points in the program, and s is the number of exit points.

Formal definition

An *even subgraph* of a graph (also known as an Eulerian subgraph) is one where every vertex is incident with an even number of edges; such subgraphs are unions of cycles and isolated vertices. In the following, even subgraphs will be identified with their edge sets, which is equivalent to only considering those even subgraphs which contain all vertices of the full graph.

The set of all even subgraphs of a graph is closed under symmetric difference, and may thus be viewed as a vector space over GF(2); this vector space is called the *cycle space* of the graph. The *cyclomatic number* of the graph is defined as the dimension of this space. Since GF(2) has two elements, the cyclomatic number is also equal to the 2-logarithm of the number of elements in the cycle space.

A basis for the cycle space is easily constructed by first fixing a maximal spanning forest of the graph, and then considering the cycles formed by one edge not in the forest and the path in the forest connecting the endpoints of that edge; these cycles constitute a basis for the cycle space. Hence, the cyclomatic number also equals the number of edges not in a

maximal spanning forest of a graph. Since the number of edges in a maximal spanning forest of a graph is equal to the number of vertices minus the number of components, the formula $E - N + P$ above for the cyclomatic number follows.

For the more topologically inclined, cyclomatic complexity can alternatively be defined as a relative Betti number, the size of a relative homology group:

$$M := b_1(G,t) := \operatorname{rank} H_1(G,t)$$

which is read as "the first homology of the graph *G,* relative to the terminal nodes *t*". This is a technical way of saying "the number of linearly independent paths through the flow graph from an entry to an exit", where:

- "linearly independent" corresponds to homology, and means one does not double-count backtracking;
- "paths" corresponds to *first* homology: a path is a 1-dimensional object;
- "relative" means the path must begin and end at an entry or exit point.

This corresponds to the intuitive notion of cyclomatic complexity, and can be calculated as above.

Alternatively, one can compute this via absolute Betti number (absolute homology – not relative) by identifying (gluing together) all terminal nodes on a given component (or equivalently, draw paths connecting the exits to the entrance), in which case (calling the new, augmented graph $\tilde{G}$, which is), one obtains:

$$M = b_1(\tilde{G}) = \operatorname{rank} H_1(\tilde{G})$$

This corresponds to the characterization of cyclomatic complexity as "number of loops plus number of components".

Etymology / Naming

The name **Cyclomatic Complexity** presents some confusion, as this metric does not only count cycles (loops) in the program. Instead, the name refers to the number of different cycles *in the program control flow graph*, after having added an imagined branch back from the exit node to the entry node.

A better name for popular usage would be **Conditional Complexity**, as "it has been found to be more convenient to count conditions instead of predicates when calculating complexity".

Applications

Limiting complexity during development

One of McCabe's original applications was to limit the complexity of routines during program development; he recommended that programmers should count the complexity of the modules they are developing, and split them into smaller modules whenever the cyclomatic complexity of the module exceeded 10. This practice was adopted by the NIST Structured Testing methodology, with an observation that since McCabe's original publication, the figure of 10 had received substantial corroborating evidence, but that in some circumstances it may be appropriate to relax the restriction and permit modules with a complexity as high as 15. As the methodology acknowledged that there were occasional reasons for going beyond the agreed-upon limit, it phrased its recommendation as: "For each module, either limit cyclomatic complexity to [the agreed-upon limit] or provide a written explanation of why the limit was exceeded."

Implications for Software Testing

Another application of cyclomatic complexity is in determining the number of test cases that are necessary to achieve thorough test coverage of a particular module.

It is useful because of two properties of the cyclomatic complexity, *M*, for a specific module:

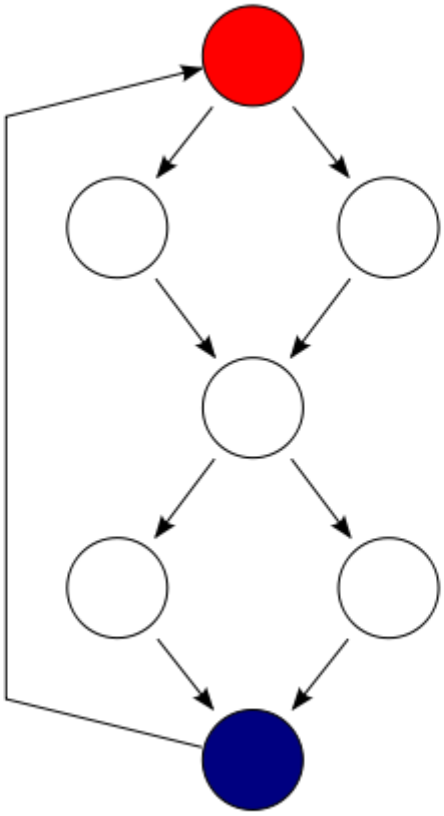
- *M* is an upper bound for the number of test cases that are necessary to achieve a complete branch coverage.
- *M* is a lower bound for the number of paths through the control flow graph (CFG). Assuming each test case takes one path, the number of cases needed to achieve path coverage is equal to the number of paths that can actually be taken. But some paths may be impossible, so although the number of paths through the CFG is clearly an upper bound on the number of test cases needed for path coverage, this latter number (of *possible* paths) is sometimes less than *M*.

All three of the above numbers may be equal: branch coverage $\leq$cyclomatic complexity $\leq$number of paths.

For example, consider a program that consists of two sequential if-then-else statements.

```
if( c1() )
   f1();
else
   f2();

if( c2() )
   f3();
else
   f4();
```



The control flow graph of the source code above; the red circle is the entry point of the function, and the blue circle is the exit point. The exit has been connected to the entry to make the graph strongly connected.

In this example, two test cases are sufficient to achieve a complete branch coverage, while four are necessary for complete path coverage. The cyclomatic complexity of the program is 3 (as the strongly connected graph for the program contains 9 edges, 7 nodes and 1 connected component) (9-7+1).

In general, in order to fully test a module all execution paths through the module should be exercised. This implies a module with a high complexity number requires more testing effort than a module with a lower value since the higher complexity number indicates more pathways through the code. This also implies that a module with higher complexity is more difficult for a programmer to understand since the programmer must understand the different pathways and the results of those pathways.

Unfortunately, it is not always practical to test all possible paths through a program. Considering the example above, each time an additional if-then-else statement is added,

the number of possible paths doubles. As the program grew in this fashion, it would quickly reach the point where testing all of the paths was impractical.

One common testing strategy, espoused for example by the NIST Structured Testing methodology, is to use the cyclomatic complexity of a module to determine the number of white-box tests that are required to obtain sufficient coverage of the module. In almost all cases, according to such a methodology, a module should have at least as many tests as its cyclomatic complexity; in most cases, this number of tests is adequate to exercise all the relevant paths of the function.

As an example of a function that requires more than simply branch coverage to test accurately, consider again the above function, but assume that to avoid a bug occurring, any code that calls either f1() or f3() must also call the other. Assuming that the results of c1() and c2() are independent, that means that the function as presented above contains a bug. Branch coverage would allow us to test the method with just two tests, and one possible set of tests would be to test the following cases:

- c1() returns true and c2() returns true
- c1() returns false and c2() returns false

Neither of these cases exposes the bug. If, however, we use cyclomatic complexity to indicate the number of tests we require, the number increases to 3. We must therefore test one of the following paths:

- c1() returns true and c2() returns false
- c1() returns false and c2() returns true

Either of these tests will expose the bug.

Cohesion

One would also expect that a module with higher complexity would tend to have lower cohesion (less than functional cohesion) than a module with lower complexity. The possible correlation between higher complexity measure with a lower level of cohesion is predicated on a module with more decision points generally implementing more than a single well defined function. A 2005 study showed stronger correlations between complexity metrics and an expert assessment of cohesion in the classes studied than the correlation between the expert's assessment and metrics designed to calculate cohesion.

Correlation to number of defects

A number of studies have investigated cyclomatic complexity's correlation to the number of defects contained in a module. Most such studies find a strong positive correlation between cyclomatic complexity and defects: modules that have the highest complexity tend to also contain the most defects. For example, a 2008 study by metric-monitoring software supplier Enerjy analyzed classes of open-source Java applications and divided

them into two sets based on how commonly faults were found in them. They found strong correlation between cyclomatic complexity and their faultiness, with classes with a combined complexity of 11 having a probability of being fault-prone of just 0.28, rising to 0.98 for classes with a complexity of 74.

However, studies that control for program size (i.e., comparing modules that have different complexities but similar size, typically measured in lines of code) are generally less conclusive, with many finding no significant correlation, while others do find correlation. Some researchers who have studied the area question the validity of the methods used by the studies finding no correlation.

Les Hatton claimed recently (Keynote at TAIC-PART 2008, Windsor, UK, Sept 2008) that McCabe Cyclomatic Complexity has the same prediction ability as lines of code.

Chapter 7

Function Point and Instruction Path Length

Function point

A **function point** is a unit of measurement to express the amount of business functionality an information system provides to a user. The cost (in dollars or hours) of a single unit is calculated from past projects. Function points are the units of measure used by the IFPUG Functional Size Measurement Method. The IFPUG FSM Method is an ISO recognized software metric to size an information system based on the functionality that is perceived by the user of the information system, independent of the technology used to implement the information system. The IFPUG FSM Method (ISO/IEC 20926 Software Engineering - Function Point Counting Practices Manual) is one of five currently recognized ISO standards for functionally sizing software.

Introduction

Function points were defined in 1979 in *A New Way of Looking at Tools* by Allan Albrecht at IBM. The functional user requirements of the software are identified and each one is categorized into one of five types: outputs, inquiries, inputs, internal files, and external interfaces. Once the function is identified and categorized into a type, it is then assessed for complexity and assigned a number of function points. Each of these functional user requirements maps to an end-user business function, such as a data entry for an Input or a user query for an Inquiry. This distinction is important because it tends to make the functions measured in function points map easily into user-oriented requirements, but it also tends to hide internal functions (e.g. algorithms), which also require resources to implement, however, there is no ISO recognized FSM Method that includes algorithmic complexity in the sizing result. Recently there have been different approaches proposed to deal with this perceived weakness, implemented in several commercial software products. The variations of the Albrecht based IFPUG method designed to make up for this (and other weaknesses) include:

- Early and easy function points - Adjusts for problem and data complexity with two questions that yield a somewhat subjective complexity measurement; simplifies measurement by eliminating the need to count data elements.
- Engineering function points - Elements (variable names) and operators (e.g., arithmetic, equality/inequality, Boolean) are counted. This variation highlights computational function. The intent is similar to that of the operator/operand-based Halstead Complexity Measures.
- Bang measure - Defines a function metric based on twelve primitive (simple) counts that affect or show Bang, defined as "the measure of true function to be delivered as perceived by the user." Bang measure may be helpful in evaluating a software unit's value in terms of how much useful function it provides, although there is little evidence in the literature of such application. The use of Bang measure could apply when re-engineering (either complete or piecewise) is being considered, as discussed in Maintenance of Operational Systems—An Overview.
- Feature points - Adds changes to improve applicability to systems with significant internal processing (e.g., operating systems, communications systems). This allows accounting for functions not readily perceivable by the user, but essential for proper operation.
- Weighted Micro Function Points - One of the newer models (2009) which adjusts function points using weights derived from program flow complexity, operand and operator vocabulary, object usage, and algorithmic intricacy.

Instruction path length

In computer performance, the **Instruction path length** is the number of machine code instructions required to execute a section of a computer program. The total path length for the entire program could be deemed a measure of the algorithm's performance on a particular computer hardware. The path length of a simple conditional instruction would normally be considered as equal to 2, one instruction to perform the comparison and another to take a branch if the particular condition is satisfied. The length of time to execute each instruction is not normally considered in determining path length and so path length is merely an indication of relative performance rather than in any sense absolute.

When executing a benchmark program, most of the instruction path length is typically inside the program's inner loop.

Assembly programs

Since there is, typically, a one-to-one relationship between assembly instructions and machine instructions, the instruction path length is frequently taken as the number of assembly instructions required to perform a function or particular section of code.

Performing a simple table lookup on a unsorted list of 1,000 entries might require perhaps 2,000 machine instructions (on average, assuming uniform distribution of input values) , performing the same lookup on a sorted list using a binary search algorithm might require only about 40 machine instructions, a very considerable saving. Expressed in terms of instruction path length, this metric would be reduced in this instance by a massive *factor* of 50 - a reason why actual instruction timings might be a secondary consideration compared to a good choice of algorithm requiring a shorter path length.

The instruction path length of an assembly language program is generally vastly different than the number of source lines of code for that program, because the instruction path length includes only code in the executed control flow for the given input and clearly does not include code that is not relevant for the particular input or unreachable code that can never be executed.

High-level language (HLL) programs

Since one statement written in a high-level language can produce multiple machine instructions of variable number, it is not always possible to determine instruction path length without, for example, an instruction set simulator - that can count the number of 'executed' instructions during simulation. If the high-level language supports and optionally produces an 'assembly list', it is sometimes possible to estimate the instruction path length by examining this list. Unfortunately most high-level language programmers do not have the knowledge to understand the assembly instructions produced and have no way to appreciate the instruction path lengths of their code - except perhaps through anecdote or bad experience. Therefore choice of particular high-level language statements can have dramatic effects on instruction path lengths without the programmer having any means of knowing this in many cases.

Factors determining instruction path length

- in-line code versus the overheads of calling and returning from a function, procedure, or method containing the same statements
- order of items in unsorted lookup list - most frequently occurring items should be placed first to avoid long searches
- choice of algorithm - indexed, binary or linear (item-by-item) search
- calculate afresh versus retain earlier calculated (memoization) - may reduce multiple complex iterations
- read some tables into memory once versus external read afresh each time - avoiding high path length through multiple I/O function calls

Use of Instruction path lengths

From the above, it can be realized that knowledge of instruction path lengths can be used:-

- to choose an appropriate algorithm to minimize overall path lengths for programs in any language
- to monitor how well a program has been optimized in any language
- to determine how efficient particular HLL statements are for any HLL language
- as an approximate measure of overall performance

Chapter 8

Source Lines of Code

Source lines of code (**SLOC**) is a software metric used to measure the size of a software program by counting the number of lines in the text of the program's source code. SLOC is typically used to predict the amount of effort that will be required to develop a program, as well as to estimate programming productivity or effort once the software is produced.

Measurement methods

Many useful comparisons involve only the order of magnitude of lines of code in a project. Software projects can vary between 1 to 100,000,000 or more lines of code. Using lines of code to compare a 10,000 line project to a 100,000 line project is far more useful than when comparing a 20,000 line project with a 21,000 line project. While it is debatable exactly how to measure lines of code, discrepancies of an order of magnitude can be clear indicators of software complexity or man hours.

There are two major types of SLOC measures: physical SLOC (LOC) and logical SLOC (LLOC). Specific definitions of these two measures vary, but the most common definition of physical SLOC is a count of lines in the text of the program's source code including comment lines. Blank lines are also included unless the lines of code in a section consists of more than 25% blank lines. In this case blank lines in excess of 25% are not counted toward lines of code.

Logical SLOC attempts to measure the number of "statements", but their specific definitions are tied to specific computer languages (one simple logical SLOC measure for C-like programming languages is the number of statement-terminating semicolons). It is much easier to create tools that measure physical SLOC, and physical SLOC definitions are easier to explain. However, physical SLOC measures are sensitive to logically irrelevant formatting and style conventions, while logical SLOC is less sensitive to formatting and style conventions. However, SLOC measures are often stated without giving their definition, and logical SLOC can often be significantly different from physical SLOC.

Consider this snippet of C code as an example of the ambiguity encountered when determining SLOC:

```
for (i = 0; i < 100; i += 1) printf("hello"); /* How many lines of code
is this? */
```

In this example we have:

- 1 Physical Lines of Code (LOC)
- 2 Logical Line of Code (LLOC) (for statement and printf statement)
- 1 comment line

Depending on the programmer and/or coding standards, the above "line of code" could be written on many separate lines:

```
/* Now how many lines of code is this? */
for (i = 0; i < 100; i += 1)
{
    printf("hello");
}
```

In this example we have:

- 5 Physical Lines of Code (LOC): is placing braces work to be estimated?
- 2 Logical Line of Code (LLOC): what about all the work writing non-statement lines?
- 1 comment line: tools must account for all code and comments regardless of comment placement.

Even the "logical" and "physical" SLOC values can have a large number of varying definitions. Robert E. Park (while at the Software Engineering Institute) et al. developed a framework for defining SLOC values, to enable people to carefully explain and define the SLOC measure used in a project. For example, most software systems reuse code, and determining which (if any) reused code to include is important when reporting a measure.

Origins

At the time that people began using SLOC as a metric, the most commonly used languages, such as FORTRAN and assembler, were line-oriented languages. These languages were developed at the time when punched cards were the main form of data entry for programming. One punched card usually represented one line of code. It was one discrete object that was easily counted. It was the visible output of the programmer so it made sense to managers to count lines of code as a measurement of a programmer's productivity, even referring to such as "card images". Today, the most commonly used computer languages allow a lot more leeway for formatting. Text lines are no longer limited to 80 or 96 columns, and one line of text no longer necessarily corresponds to one line of code.

Usage of SLOC measures

SLOC measures are somewhat controversial, particularly in the way that they are sometimes misused. Experiments have repeatedly confirmed that effort is highly correlated with SLOC, that is, programs with larger SLOC values take more time to develop. Thus, SLOC can be very effective in estimating effort. However, functionality is less well correlated with SLOC: skilled developers may be able to develop the same functionality with far less code, so one program with less SLOC may exhibit more functionality than another similar program. In particular, SLOC is a poor productivity measure of individuals, since a developer can develop only a few lines and yet be far more productive in terms of functionality than a developer who ends up creating more lines (and generally spending more effort). Good developers may merge multiple code modules into a single module, improving the system yet appearing to have negative productivity because they remove code. Also, especially skilled developers tend to be assigned the most difficult tasks, and thus may sometimes appear less "productive" than other developers on a task by this measure. Furthermore, inexperienced developers often resort to code duplication, which is highly discouraged as it is more bug-prone and costly to maintain, but it results in higher SLOC.

SLOC is particularly ineffective at comparing programs written in different languages unless adjustment factors are applied to normalize languages. Various computer languages balance brevity and clarity in different ways; as an extreme example, most assembly languages would require hundreds of lines of code to perform the same task as a few characters in APL. The following example shows a comparison of a "hello world" program written in C, and the same program written in COBOL - a language known for being particularly verbose.

C

```
#include <stdio.h>

int main() {

printf("\nHello\n");
}
```

COBOL

```
000100 IDENTIFICATION DIVISION.
000200 PROGRAM-ID. HELLOWORLD.
000300
000400*
000500 ENVIRONMENT DIVISION.
000600 CONFIGURATION SECTION.
000700 SOURCE-COMPUTER. RM-COBOL.
000800 OBJECT-COMPUTER. RM-COBOL.
000900
001000 DATA DIVISION.
001100 FILE SECTION.
001200
100000 PROCEDURE DIVISION.
100100
100200 MAIN-LOGIC SECTION.
100300 BEGIN.
100400     DISPLAY " " LINE 1
POSITION 1 ERASE EOS.
100500     DISPLAY "Hello world!"
LINE 15 POSITION 10.
```

```
100600     STOP RUN.
100700 MAIN-LOGIC-EXIT.
100800     EXIT.
```

Lines of code: 4 (excluding whitespace)

Lines of code: 17 (excluding whitespace)

Another increasingly common problem in comparing SLOC metrics is the difference between auto-generated and hand-written code. Modern software tools often have the capability to auto-generate enormous amounts of code with a few clicks of a mouse. For instance, GUI builders automatically generate all the source code for a GUI object simply by dragging an icon onto a workspace. The work involved in creating this code cannot reasonably be compared to the work necessary to write a device driver, for instance. By the same token, a hand-coded custom GUI class could easily be more demanding than a simple device driver; hence the shortcoming of this metric.

There are several cost, schedule, and effort estimation models which use SLOC as an input parameter, including the widely-used Constructive Cost Model (COCOMO) series of models by Barry Boehm et al., PRICE Systems True S and Galorath's SEER-SEM. While these models have shown good predictive power, they are only as good as the estimates (particularly the SLOC estimates) fed to them. Many have advocated the use of function points instead of SLOC as a measure of functionality, but since function points are highly correlated to SLOC (and cannot be automatically measured) this is not a universally held view.

Example

According to Vincent Maraia, the SLOC values for various operating systems in Microsoft's Windows NT product line are as follows:

Year	Operating System	SLOC (Million)
1993	Windows NT 3.1	4-5
1994	Windows NT 3.5	7-8
1996	Windows NT 4.0	11-12
2000	Windows 2000	more than 29

2001	Windows XP	45
2003	Windows Server 2003	50

David A. Wheeler studied the Red Hat distribution of the Linux operating system, and reported that Red Hat Linux version 7.1 (released April 2001) contained over 30 million physical SLOC. He also extrapolated that, had it been developed by conventional proprietary means, it would have required about 8,000 person-years of development effort and would have cost over $1 billion (in year 2000 U.S. dollars).

A similar study was later made of Debian Linux version 2.2 (also known as "Potato"); this version of Linux was originally released in August 2000. This study found that Debian Linux 2.2 included over 55 million SLOC, and if developed in a conventional proprietary way would have required 14,005 person-years and cost $1.9 billion USD to develop. Later runs of the tools used report that the following release of Debian had 104 million SLOC, and as of year 2005, the newest release is going to include over 213 million SLOC.

One can find figures of major operating systems (the various Windows versions have been presented in a table above)

Operating System	**SLOC (Million)**
Debian 2.2	55-59
Debian 3.0	104
Debian 3.1	215
Debian 4.0	283
Debian 5.0	324
OpenSolaris	9.7
FreeBSD	8.8
Mac OS X 10.4	86
Linux kernel 2.6.0	5.2
Linux kernel 2.6.29	11.0
Linux kernel 2.6.32	12.6
Linux kernel 2.6.35	13.5

Relation with security faults

The central enemy of reliability is complexity.
—Geer et al.

A number of experts have claimed a relationship between the number of lines of code in a program and the number of bugs that it contains. This relationship is not simple, since the number of errors per line of code varies greatly according to the language used, the type of quality assurance processes, and level of testing, but it does appear to exist. More

importantly, the number of bugs in a program has been directly related to the number of security faults that are likely to be found in the program.

This has had a number of important implications for system security and these can be seen reflected in operating system design. Firstly, more complex systems are likely to be more insecure simply due to the greater number of lines of code needed to develop them. For this reason, security focused systems such as OpenBSD grow much more slowly than other systems such as Windows and Linux. A second idea, taken up in OpenBSD, Windows and many Linux variants, is that separating code into different sections which run with different security environments (with or without special privileges, for example) ensures that the most security critical segments are small and carefully audited.

Utility

Advantages

1. Scope for Automation of Counting: Since Line of Code is a physical entity; manual counting effort can be easily eliminated by automating the counting process. Small utilities may be developed for counting the LOC in a program. However, a code counting utility developed for a specific language cannot be used for other languages due to the syntactical and structural differences among languages.
2. An Intuitive Metric: Line of Code serves as an intuitive metric for measuring the size of software because it can be seen and the effect of it can be visualized. Function points are said to be more of an objective metric which cannot be imagined as being a physical entity, it exists only in the logical space. This way, LOC comes in handy to express the size of software among programmers with low levels of experience.

Disadvantages

1. Lack of Accountability: Lines of code measure suffers from some fundamental problems. Some think it isn't useful to measure the productivity of a project using only results from the coding phase, which usually accounts for only 30% to 35% of the overall effort.
2. Lack of Cohesion with Functionality: Though experiments have repeatedly confirmed that effort is highly correlated with LOC, functionality is less well correlated with LOC. That is, skilled developers may be able to develop the same functionality with far less code, so one program with less LOC may exhibit more functionality than another similar program. In particular, LOC is a poor productivity measure of individuals, because a developer who develops only a few lines may still be more productive than a developer creating more lines of code - even more: some good refactoring like "extract method" to get rid of redundant code and keep it clean will mostly reduce the lines of code.
3. Adverse Impact on Estimation: Because of the fact presented under point #1, estimates based on lines of code can adversely go wrong, in all possibility.

4. Developer's Experience: Implementation of a specific logic differs based on the level of experience of the developer. Hence, number of lines of code differs from person to person. An experienced developer may implement certain functionality in fewer lines of code than another developer of relatively less experience does, though they use the same language.
5. Difference in Languages: Consider two applications that provide the same functionality (screens, reports, databases). One of the applications is written in C++ and the other application written in a language like COBOL. The number of function points would be exactly the same, but aspects of the application would be different. The lines of code needed to develop the application would certainly not be the same. As a consequence, the amount of effort required to develop the application would be different (hours per function point). Unlike Lines of Code, the number of Function Points will remain constant.
6. Advent of GUI Tools: With the advent of GUI-based programming languages and tools such as Visual Basic, programmers can write relatively little code and achieve high levels of functionality. For example, instead of writing a program to create a window and draw a button, a user with a GUI tool can use drag-and-drop and other mouse operations to place components on a workspace. Code that is automatically generated by a GUI tool is not usually taken into consideration when using LOC methods of measurement. This results in variation between languages; the same task that can be done in a single line of code (or no code at all) in one language may require several lines of code in another.
7. Problems with Multiple Languages: In today's software scenario, software is often developed in more than one language. Very often, a number of languages are employed depending on the complexity and requirements. Tracking and reporting of productivity and defect rates poses a serious problem in this case since defects cannot be attributed to a particular language subsequent to integration of the system. Function Point stands out to be the best measure of size in this case.
8. Lack of Counting Standards: There is no standard definition of what a line of code is. Do comments count? Are data declarations included? What happens if a statement extends over several lines? – These are the questions that often arise. Though organizations like SEI and IEEE have published some guidelines in an attempt to standardize counting, it is difficult to put these into practice especially in the face of newer and newer languages being introduced every year.
9. Psychology: A programmer whose productivity is being measured in lines of code will have an incentive to write unnecessarily verbose code. The more management is focusing on lines of code, the more incentive the programmer has to expand his code with unneeded complexity. This is undesirable since increased complexity can lead to increased cost of maintenance and increased effort required for bug fixing.

In the PBS documentary *Triumph of the Nerds*, Microsoft executive Steve Ballmer criticized the use of counting lines of code:

In IBM there's a religion in software that says you have to count K-LOCs, and a K-LOC is a thousand lines of code. How big a project is it? Oh, it's sort of a 10K-LOC project. This is a 20K-LOCer. And this is 50K-LOCs. And IBM wanted to sort of make it the religion about how we got paid. How much money we made off OS/2, how much they did. How many K-LOCs did you do? And we kept trying to convince them - hey, if we have - a developer's got a good idea and he can get something done in 4K-LOCs instead of 20K-LOCs, should we make less money? Because he's made something smaller and faster, less K-LOC. K-LOCs, K-LOCs, that's the methodology. Ugh! Anyway, that always makes my back just crinkle up at the thought of the whole thing.

Related terms

- KLOC: 1,000 lines of code
  - KDLOC: 1,000 delivered lines of code
  - KSLOC: 1,000 source lines of code
- MLOC: 1,000,000 lines of code
- GLOC: 1,000,000,000 lines of code

Chapter 9

Run Time and Loader (Computing)

Run time (computing)

In computer science, the qualifier **run time**, **run-time**, **runtime**, or **execution time** may have different meanings depending on the context. From a technical perspective it may refer to the time during which a program is running (executing). It contrasts to other phases of program such as compile time, link time, load time, etc. Closely linked to this most basic sense is the state of the entire system on which the program is running, which affects the program's execution and which varies between instances. Therefore, in a software licensing context, "runtime" refers to the broader idea of an installation of a given software or computer program on a computer or server whether running or not.

A "run-time error" is detected after or during the installation or copying of the program, whereas a "compile-time error" is detected by the compiler before the program is so installed and started. Type checking, storage allocation, and even code generation and code optimization may be done at compile-time or upon a run-time, depending on the language and compiler.

Implementation details

In most cases, the execution of a program begins after a loader performed the necessary memory setup and linked the program with any dynamically linked libraries it needs. In some cases a language or implementation will have these tasks done by the language runtime instead, though this is unusual in mainstream languages on common consumer operating systems.

Some program debugging can only be performed (or are more efficient or accurate) when performed at runtime. Logical errors and array bounds checking are examples. For this reason, some programming bugs are not discovered until the program is tested in a "live" environment with real data, despite sophisticated compile-time checking and pre-release testing. In this case, the end user may encounter a *runtime error* message.

Application errors — exceptions

Exception handling is one language feature designed to handle runtime errors, providing a structured way to catch completely unexpected situations as well as predictable errors or unusual results without the amount of inline error checking required of languages without it. More recent advancements in runtime engines enable automated exception handling which provides 'root-cause' debug information for every exception of interest and is implemented independent of the source code, by attaching a special software product to the runtime engine.

Orthographic styling

Some users prefer an orthographic styling in which "run-time" as an adjective is hyphenated, whereas "runtime" as a noun is styled solid (closed up). Examples:

- “Polymorphism requires run-time binding rather than compile-time binding.”
- “The runtime of the algorithm is order n^2.”

As elsewhere with linguistic prescription for the styling of compounds (open, hyphenated, or solid), natural language often pays no heed.

Loader (computing)

In computing, a **loader** is the part of an operating system that is responsible for loading programs, one of the essential stages in the process of starting a program, it means loader is a program that places programs into memory and prepares them for execution. Loading a program involves reading the contents of executable file, the file containing the program text, into memory, and then carrying out other required preparatory tasks to prepare the executable for running. Once loading is complete, the operating system starts the program by passing control to the loaded program code.

All operating systems that support program loading have loaders, apart from systems where code executes directly from ROM or in the case of highly specialized computer systems that only have a fixed set of specialised programs.

In many operating systems the loader is permanently resident in memories, although some operating systems that support virtual memory may allow the loader to be located in a region of memory that is pageable.

In the case of operating systems that support virtual memory, the loader may not actually copy the contents of executable files into memory, but rather may simply declare to the virtual memory subsystem that there is a mapping between a region of memory allocated

to contain the running program's code and the contents of the associated executable file. The virtual memory subsystem is then made aware that pages with that region of memory need to be filled on demand if and when program execution actually hits those areas of unfilled memory. This may mean parts of a program's code are not actually copied into memory until they are actually used, and unused code may never be loaded into memory at all.

Responsibilities

In Unix, the loader is the handler for the system call `execve()`. The Unix loader's tasks include:

1. validation (permissions, memory requirements etc.);
2. copying the program image from the disk into main memory;
3. copying the command-line arguments on the stack;
4. initializing registers (e.g., the stack pointer);
5. jumping to the program entry point (`_start`).

Relocating loaders

Some computers need relocating loaders, which adjust addresses (pointers) in the executable to compensate for variations in the address at which loading starts. The computers which need relocating loaders are those in which pointers are absolute addresses rather than offsets from the program's base address. One well-known example is IBM's System/360 mainframes and their descendants, including the System z9 series.

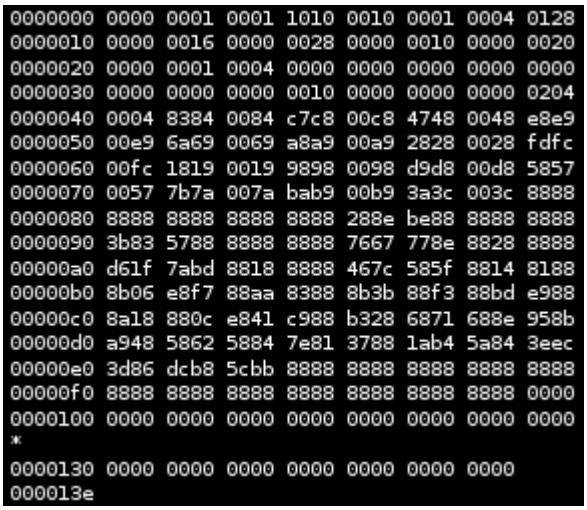
Dynamic linkers

Dynamic linking loaders are another type of loader that load and link shared libraries (like .dll files) to already loaded running programs.

Chapter 10

Binary File and Software Package Metrics

Binary file



The first column numerates the line's starting address, while the * indicates repetition.

A **binary file** (commonly, but not necessarily, with the extension **.bin**) is a computer file which may contain any type of data, encoded in binary form for computer storage and processing purposes; for example, computer document files containing formatted text. Many binary file formats contain parts that can be interpreted as text; binary files that contain *only* textual data—without, for example, any formatting information—are called plain text files. In many cases, plain text files are considered to be different from binary files because binary files are made up of more than just plain text. When downloading, a completely functional program without any installer is also often called program binary, or binaries (as opposed to the source code).

Structure

Binary files are usually thought of as being a sequence of bytes, which means the binary digits (bits) are grouped in eights. Binary files typically contain bytes that are intended to be interpreted as something other than text characters. Compiled computer programs are typical examples; indeed, compiled applications (object files) are sometimes referred to, particularly by programmers, as **binaries**. But binary files can also contain images, sounds, compressed versions of other files, etc. — in short, any type of file content whatsoever.

Some binary files contain headers, blocks of metadata used by a computer program to interpret the data in the file. For example, a GIF file can contain multiple images, and headers are used to identify and describe each block of image data. If a binary file does not contain any headers, it may be called a **flat binary file**.

Manipulation

To send binary files through certain systems (such as e-mail) that do not allow all data values, they are often translated into a plain text representation (using, for example, Base64). This encoding has the disadvantage of increasing the file's size by approximately 30% during the transfer, as well as requiring translation back into binary after receipt.

Microsoft Windows allows the programmer to specify a system call parameter indicating if a file is text or binary; Unix does not, and treats all files as binary. This reflects the fact that the distinction between the two types of files is to a certain extent arbitrary.

Viewing

A hex editor or viewer may be used to view file data as a sequence of hexadecimal (or decimal, binary or ASCII character) values for corresponding bytes of a binary file.

If a binary file is opened in a text editor, each group of eight bits will typically be translated as a single character, and you will see a (probably unintelligible) display of textual characters. If the file is opened in some other application, that application will have its own use for each byte: maybe the application will treat each byte as a number and output a stream of numbers between 0 and 255 — or maybe interpret the numbers in the bytes as colors and display the corresponding picture. If the file is itself treated as an executable and run, then the operating system will attempt to interpret the file as a series of instructions in its machine language.

Interpretation

Standards are very important to binary files. For example, a binary file interpreted by the ASCII character set will result in text being displayed. A custom application can interpret the file differently, a byte may be a sound, or a pixel, or even an entire word. Binary itself

is meaningless, until such time as an executed algorithm defines what should be done with each bit, byte, word or block. Thus, just examining the binary and attempting to match it against known formats can lead to the wrong conclusion as to what it actually represents. This fact can be used in steganography, where an algorithm interprets a binary data file differently to reveal hidden content. Without the algorithm, it is impossible to tell that hidden content exists.

Binary compatibility

Two files that are binary compatible will have the same pattern of zeros and ones in the data portion of the file. The file header, however, may be different.

The term is used most commonly to state that data files produced by one application are exactly the same as data files produced by another application. For example, some software companies produce applications for Windows and the Macintosh that are binary compatible, which means that a file produced in a Windows environment is interchangeable with a file produced on a Macintosh. This avoids many of the conversion problems caused by importing and exporting data.

Software package metrics

The term *software package*, as it is used here, refers to a group of related classes (in the field of object-oriented programming).

- **Number of Classes and Interfaces:** The number of concrete and abstract classes (and interfaces) in the package is an indicator of the extensibility of the package.
- **Afferent Couplings (Ca):** The number of other packages that depend upon classes within the package is an indicator of the package's responsibility.
- **Efferent Couplings (Ce):** The number of other packages that the classes in the package depend upon is an indicator of the package's independence.
- **Abstractness (A):** The ratio of the number of abstract classes (and interfaces) in the analyzed package to the total number of classes in the analyzed package. The range for this metric is 0 to 1, with A=0 indicating a completely concrete package and A=1 indicating a completely abstract package.
- **Instability (I):** The ratio of efferent coupling (Ce) to total coupling (Ce + Ca) such that I = Ce / (Ce + Ca). This metric is an indicator of the package's resilience to change. The range for this metric is 0 to 1, with I=0 indicating a completely stable package and I=1 indicating a completely instable package.
- **Distance from the Main Sequence (D):** The perpendicular distance of a package from the idealized line A + I = 1. This metric is an indicator of the package's balance between abstractness and stability. A package squarely on the main sequence is optimally balanced with respect to its abstractness and stability. Ideal packages are either completely abstract and stable (x=0, y=1) or completely concrete and instable (x=1, y=0). The range for this metric is 0 to 1, with D=0

indicating a package that is coincident with the main sequence and D=1 indicating a package that is as far from the main sequence as possible.

- **Package Dependency Cycles:** Package dependency cycles are reported along with the hierarchical paths of packages participating in package dependency cycles.

Chapter 11

Duplicate Code

Duplicate code is a computer programming term for a sequence of source code that occurs more than once, either within a program or across different programs owned or maintained by the same entity. Duplicate code is generally considered undesirable for a number of reasons. A minimum requirement is usually applied to the quantity of code that must appear in a sequence for it to be considered duplicate rather than coincidentally similar. Sequences of duplicate code are sometimes known as **clones**.

The following are some of the ways in which two code sequences can be duplicates of each other:

- character-for-character identical
- character-for-character identical with white space characters and comments being ignored
- token-for-token identical
- token-for-token identical with occasional variation (i.e., insertion/deletion/modification of tokens)
- functionally identical

How duplicates are created

There are a number of reasons why duplicate code may be created, including:

- Copy and paste programming, in which a section of code is copied "because it works". In most cases this operation involves slight modifications in the cloned code such as renaming variables or inserting/deleting code.
- Functionality that is very similar to that in another part of a program is required and a developer independently writes code that is very similar to what exists elsewhere.
- Plagiarism, where code is simply copied without permission or attribution.

Problems associated with duplicate code

Code duplication is generally considered a mark of poor or lazy programming style. Good coding style is generally associated with code reuse. It may be slightly faster to develop by duplicating code, because the developer need not concern himself with how the code is already used or how it may be used in the future. The difficulty is that original development is only a small fraction of a product's life cycle, and with code duplication the maintenance costs are much higher. Some of the specific problems include:

- **Code bulk affects comprehension**: Code duplication frequently creates long, repeated sections of code that differ in only a few lines or characters. The length of such routines can make it difficult to quickly understand them. This is in contrast to the "best practice" of code decomposition.
- **Purpose masking**: The repetition of largely identical code sections can conceal how they differ from one another, and therefore, what the specific purpose of each code section is. Often, the only difference is in a parameter value. The best practice in such cases is a reusable subroutine.

- **Update anomalies**: Duplicate code contradicts a fundamental principle of database theory that applies here: Avoid redundancy. Non-observance incurs update anomalies, which increase maintenance costs, in that any modification to a redundant piece of code must be made for each duplicate separately. At best, coding and testing time are multiplied by the number of duplications. At worst, some locations may be missed, and for example bugs thought to be fixed may persist in duplicated locations for months or years. The best practice here is a code library.

Detecting duplicate code

A number of different algorithms have been proposed to detect duplicate code. For example:

- Baker's algorithm.
- Rabin–Karp string search algorithm.
- Using Abstract Syntax Trees.
- Visual clone detection.

Example of functionally duplicate code

Consider the following code snippet for calculating the average of an array of integers

```
extern int array1[];
extern int array2[];

int sum1 = 0;
int sum2 = 0;
int average1 = 0;
```

```
int average2 = 0;

for (int i = 0; i < 4; i++)
{
   sum1 += array1[i];
}
average1 = sum1/4;

for (int i = 0; i < 4; i++)
{
   sum2 += array2[i];
}
average2 = sum2/4;
```

The two loops can be rewritten as the single function:

```
int calcAverage (int* Array_of_4)
{
   int sum = 0;
   for (int i = 0; i < 4; i++)
   {
       sum += Array_of_4[i];
   }
   return sum/4;
}
```

Using the above function will give source code that has no loop duplication:

```
extern int array1[];
extern int array2[];

int average1 = calcAverage(array1);
int average2 = calcAverage(array2);
```

Tools

Code duplication analysis tools include:

- Atomiq - commercial
- Black Duck Suite - commercial (software analyzing suite)
- CCFinder (C/C++, Java, COBOL, Fortran, etc. / uncomfortable to compile for non-windows OS)
- Checkstyle (Java)
- CloneAnalyzer (C/C++ and Java / Eclipse plugin only)
- Clone Digger (Python and Java)
- CloneDR - commercial (Many languages supported)
- Copy/Paste Detector (CPD) from PMD (Java, JSP, C, C++, Fortran, PHP)
- ConQAT (Open Source, supports: ABAP, ADA, Cobol, C/C++, C#, Java, PL/I, PL/SQL, Python, Text, Transact SQL, Visual Basic, XML)
- JPlag (Java, C#, C, C++, Scheme and natural language text)
- Pattern Miner (CP Miner) - commercial

- Simian (software) - commercial
- CodePro Analytix - commercial

Chapter 12

Linear Code Sequence and Jump

Linear code sequence and jump (LCSAJ) is a software analysis method used to identify structural units in code under test. Its primary use is with dynamic software analysis to help answer the question "How much testing is enough?". Dynamic software analysis is used to measure the quality and efficacy of software test data, where the quantification is performed in terms of structural units of the code under test. When used to quantify the structural units exercised by a given set of test data, dynamic analysis is also referred to as coverage analysis.

History

The LCSAJ analysis method was devised by Professor Michael Hennell in order to perform quality assessments on the mathematical libraries on which his Nuclear physics research at the University of Liverpool depended. Professor Hennell later founded the Liverpool Data Research Associates (LDRA) company to commercialize the software test-bed produced for this work, resulting in the LDRA Testbed product.

Introduced in 1976, the Linear Code Sequence and Jump (LCSAJ) is now also referred to as the jump-to-jump path (JJ-path).

Definition

An LCSAJ is a software code path fragment consisting of a sequence of code (a Linear Code Sequence, or basic block) followed by a control flow Jump, and consists of the following three items :

- the start of the linear sequence (basic block) of executable statements
- the end of the linear sequence
- the target line to which control flow is transferred at the end of the linear sequence.

When used for coverage analysis, LCSAJs are considered to have a Test Effectiveness Ratio of 3.

Test Effectiveness Ratio

Coverage analysis metrics are used to gauge how much testing has been achieved. The most basic metric is the proportion of statements executed, considered to have a Test Effectiveness Ratio (TER) of one :

$$TER_1 = \frac{number\ of\ statements\ executed\ by\ the\ test\ data}{total\ number\ of\ executable statements}$$

Higher level coverage metrics can also be generated, in particular :

$$TER_2 = \frac{number\ of\ control-flow\ branches\ executed\ by\ the\ test\ data}{total\ number\ of\ control-flow\ branches}$$

$$TER_3 = \frac{number\ of\ LCSAJs\ executed\ by\ the\ test\ data}{total\ number\ of\ LCSAJs}$$

These metrics satisfy a pure hierarchy, whereby when TER3 = 100% has been achieved it follows that TER2 = 100% and TER1 = 100% have also been achieved.

Both the TER1 & TER2 metrics were in use in the 1940s and the third dates from the 1970s. The requirements for achieving TER1 = 100% became an industrial norm during the late 1960s, and was the level originally selected for the DO-178 avionics standard until it was supplemented by the MCDC (Modified Condition/Decision Coverage) additional requirement in 1992. Higher levels TER3 = 100% have been mandated for many other projects, including aerospace, telephony and banking.

Example

Consider the following C code:

```
1.  #include    <stdlib.h>
2.
3.  #include    <string.h>
4.
5.  #include    <math.h>
6.
7.
8.
9.  #define MAXCOLUMNS  26
10.
11. #define MAXROW      20
12.
13. #define MAXCOUNT    90
14.
```

```
15.  #define ITERATIONS  750
16.
17.
18.
19.  int main (void)
20.
21.  {
22.
23.      int count = 0, totals[MAXCOLUMNS], val = 0;
24.
25.
26.
27.      memset (totals, 0, MAXCOLUMNS * sizeof(int));
28.
29.
30.
31.      count = 0;
32.
33.      while ( count < ITERATIONS )
34.
35.      {
36.
37.          val = abs(rand()) % MAXCOLUMNS;
38.
39.          totals[val] += 1;
40.
41.          if ( totals[val] > MAXCOUNT )
42.
43.          {
44.
45.              totals[val] = MAXCOUNT;
46.
47.          }
48.
49.          count++;
50.
51.      }
52.
53.
54.
55.      return (0);
56.
57.
58.
59.  }
60.
```

From this code, the following is a complete list of the LCSAJ triples for this code

LCSAJ Number	Start Line	Finish Line	Jump To Line
1	10	17	28
2	10	21	25
3	10	26	17
4	17	17	28
5	17	21	25
6	17	26	17
7	25	26	17
8	28	28	-1

A coverage level of TER3 = 100% would be achieved when test data is used that causes the execution of each of these LCSAJs at least once.

From this example it can be seen that the basic block identified by an LCSAJ triple may span a decision point, reflecting the conditions that must be in place in order for the LCSAJ to be executed. For instance, LCSAJ 2 for the above example includes the '**while**' statement where the condition '**(count < ITERATIONS)**' evaluates to TRUE.

In addition, it can also be seen that each line of code has an LCSAJ 'density' associated with it; line 17, for instance, appears within 6 unique LCSAJs - i.e. it has an LCSAJ density of 6.

This is helpful when evaluating the maintainability of the code; If a line of code is to be changed then the density is indicative of how many LCSAJs will be affected by that change.

Chapter 13

Software Quality

In the context of software engineering, **software quality** measures how well software is designed (*quality of design*), and how well the software conforms to that design (*quality of conformance*), although there are several different definitions, including conformance to customer expectectations. It is often described as the 'fitness for purpose' of a piece of software.

Whereas *quality of conformance* is concerned with implementation, *quality of design* measures how valid the design and requirements are in creating a worthwhile product.

Definition

One of the challenges of software quality is that "everyone feels they understand it".

In addition to more software specific definitions given below, there are several applicable definitions of quality which are used in business.Quality_(business)#Definitions

Software quality may be defined as conformance to explicitly stated functional and performance requirements, explicitly documented development standards and implicit characteristics that are expected of all professionally developed software.

The three key points in this definition:

1. Software requirements are the foundations from which quality is measured.

   Lack of conformance to requirement is lack of quality.

2. Specified standards define a set of development criteria that guide the management in software engineering.

   If criteria are not followed lack of quality will usually result.

3. A set of implicit requirements often goes unmentioned, for example ease of use, maintainability etc.

If software conforms to its explicit requirements but fails to meet implicit requirements, software quality is suspected.

A definition in Steve McConnell's *Code Complete* divides software into two pieces: **internal** and **external quality characteristics**. External quality characteristics are those parts of a product that face its users, where internal quality characteristics are those that do not.

Another definition by Dr. Tom DeMarco says "a product's quality is a function of how much it changes the world for the better." This can be interpreted as meaning that user satisfaction is more important than anything in determining software quality.

Another definition, coined by Gerald Weinberg in *Quality Software Management: Systems Thinking*, is "Quality is value to some person." This definition stresses that quality is inherently subjective - different people will experience the quality of the same software very differently. One strength of this definition is the questions it invites software teams to consider, such as "Who are the people we want to value our software?" and "What will be valuable to *them*?"

History

Software product quality

- Product quality
  - conformance to requirements or program specification; related to Reliability
- Scalability
- Correctness
- Completeness
- Absence of bugs
- Fault-tolerance
  - Extensibility
  - Maintainability
- Documentation

The Consortium for IT Software Quality (CISQ) was launched in 2009 to standardize the measurement of software product quality. The Consortium's goal is to bring together industry executives from Global 2000 IT organizations, system integrators, outsourcers, and package vendors to jointly address the challenge of standardizing the measurement of IT software quality and to promote a market-based ecosystem to support its deployment.

It is essential to supplement traditional testing – functional, non-functional, and run-time – with measures of application structural quality. Structural quality is the quality of the application's architecture and the degree to which its implementation accords with software engineering best practices. Industry data demonstrate that poor application structural quality results in cost and schedule overruns and creates waste in the form of

rework (up to 45% of development time in some organizations). Moreover, poor structural quality is strongly correlated with high-impact business disruptions due to corrupted data, application outages, security breaches, and performance problems. As in any other field of engineering, an application with good structural software quality costs less to maintain and is easier to understand and change in response to pressing business needs.

Source code quality

A computer has no concept of "well-written" source code. However, from a human point of view source code can be written in a way that has an effect on the effort needed to comprehend its behavior. Many source code programming style guides, which often stress readability and usually language-specific conventions are aimed at reducing the cost of source code maintenance. Some of the issues that affect code quality include:

- Readability
- Ease of maintenance, testing, debugging, fixing, modification and portability
- Low complexity
- Low resource consumption: memory, CPU
- Number of compilation or lint warnings
- Robust input validation and error handling, established by software fault injection

Methods to improve the quality:

- Refactoring
- Code Inspection or software review
- Documenting the code

Software reliability

Software **reliability** is an important facet of software quality. It is defined as "the probability of failure-free operation of a computer program in a specified environment for a specified time".

One of reliability's distinguishing characteristics is that it is objective, measurable, and can be estimated, whereas much of software quality is subjective criteria. This distinction is especially important in the discipline of Software Quality Assurance. These measured criteria are typically called software metrics.

History

With software embedded into many devices today, software failure has caused more than inconvenience. Software errors have even caused human fatalities. The causes have ranged from poorly designed user interfaces to direct programming errors. An example of a programming error that lead to multiple deaths is discussed in Dr. Leveson's paper (PDF). This has resulted in requirements for development of some types software. In the

United States, both the Food and Drug Administration (FDA) and Federal Aviation Administration (FAA) have requirements for software development.

Goal of reliability

The need for a means to objectively determine software reliability comes from the desire to apply the techniques of contemporary engineering fields to the development of software. That desire is a result of the common observation, by both lay-persons and specialists, that computer software does not work the way it ought to. In other words, software is seen to exhibit undesirable behaviour, up to and including outright failure, with consequences for the data which is processed, the machinery on which the software runs, and by extension the people and materials which those machines might negatively affect. The more critical the application of the software to economic and production processes, or to life-sustaining systems, the more important is the need to assess the software's reliability.

Regardless of the criticality of any single software application, it is also more and more frequently observed that software has penetrated deeply into most every aspect of modern life through the technology we use. It is only expected that this infiltration will continue, along with an accompanying dependency on the software by the systems which maintain our society. As software becomes more and more crucial to the operation of the systems on which we depend, the argument goes, it only follows that the software should offer a concomitant level of dependability. In other words, the software should behave in the way it is intended, or even better, in the way it should.

Challenge of reliability

The circular logic of the preceding sentence is not accidental—it is meant to illustrate a fundamental problem in the issue of measuring software reliability, which is the difficulty of determining, in advance, exactly how the software is intended to operate. The problem seems to stem from a common conceptual error in the consideration of software, which is that software in some sense takes on a role which would otherwise be filled by a human being. This is a problem on two levels. Firstly, most modern software performs work which a human could never perform, especially at the high level of reliability that is often expected from software in comparison to humans. Secondly, software is fundamentally incapable of most of the mental capabilities of humans which separate them from mere mechanisms: qualities such as adaptability, general-purpose knowledge, a sense of conceptual and functional context, and common sense.

Nevertheless, most software programs could safely be considered to have a particular, even singular purpose. If the possibility can be allowed that said purpose can be well or even completely defined, it should present a means for at least considering objectively whether the software is, in fact, reliable, by comparing the expected outcome to the actual outcome of running the software in a given environment, with given data. Unfortunately, it is still not known whether it is possible to exhaustively determine either the expected outcome or the actual outcome of the entire set of possible environment and input data to

a given program, without which it is probably impossible to determine the program's reliability with any certainty.

However, various attempts are in the works to attempt to rein in the vastness of the space of software's environmental and input variables, both for actual programs and theoretical descriptions of programs. Such attempts to improve software reliability can be applied at different stages of a program's development, in the case of real software. These stages principally include: requirements, design, programming, testing, and runtime evaluation. The study of theoretical software reliability is predominantly concerned with the concept of correctness, a mathematical field of computer science which is an outgrowth of language and automata theory.

Reliability in program development

Requirements

A program cannot be expected to work as desired if the developers of the program do not, in fact, know the program's desired behaviour in advance, or if they cannot at least determine its desired behaviour in parallel with development, in sufficient detail. What level of detail is considered sufficient is hotly debated. The idea of perfect detail is attractive, but may be impractical, if not actually impossible. This is because the desired behaviour tends to change as the possible range of the behaviour is determined through actual attempts, or more accurately, failed attempts, to achieve it.

Whether a program's desired behaviour can be successfully specified in advance is a moot point if the behaviour cannot be specified at all, and this is the focus of attempts to formalize the process of creating requirements for new software projects. In situ with the formalization effort is an attempt to help inform non-specialists, particularly non-programmers, who commission software projects without sufficient knowledge of what computer software is in fact capable. Communicating this knowledge is made more difficult by the fact that, as hinted above, even programmers cannot always know in advance what is actually possible for software in advance of trying.

Design

While requirements are meant to specify what a program should do, design is meant, at least at a high level, to specify how the program should do it. The usefulness of design is also questioned by some, but those who look to formalize the process of ensuring reliability often offer good software design processes as the most significant means to accomplish it. Software design usually involves the use of more abstract and general means of specifying the parts of the software and what they do. As such, it can be seen as a way to break a large program down into many smaller programs, such that those smaller pieces together do the work of the whole program.

The purposes of high-level design are as follows. It separates what are considered to be problems of architecture, or overall program concept and structure, from problems of

actual coding, which solve problems of actual data processing. It applies additional constraints to the development process by narrowing the scope of the smaller software components, and thereby—it is hoped—removing variables which could increase the likelihood of programming errors. It provides a program template, including the specification of interfaces, which can be shared by different teams of developers working on disparate parts, such that they can know in advance how each of their contributions will interface with those of the other teams. Finally, and perhaps most controversially, it specifies the program independently of the implementation language or languages, thereby removing language-specific biases and limitations which would otherwise creep into the design, perhaps unwittingly on the part of programmer-designers.

Programming

The history of computer programming language development can often be best understood in the light of attempts to master the complexity of computer programs, which otherwise becomes more difficult to understand in proportion (perhaps exponentially) to the size of the programs. (Another way of looking at the evolution of programming languages is simply as a way of getting the computer to do more and more of the work, but this may be a different way of saying the same thing). Lack of understanding of a program's overall structure and functionality is a sure way to fail to detect errors in the program, and thus the use of better languages should, conversely, reduce the number of errors by enabling a better understanding.

Improvements in languages tend to provide incrementally what software design has attempted to do in one fell swoop: consider the software at ever greater levels of abstraction. Such inventions as statement, sub-routine, file, class, template, library, component and more have allowed the arrangement of a program's parts to be specified using abstractions such as layers, hierarchies and modules, which provide structure at different granularities, so that from any point of view the program's code can be imagined to be orderly and comprehensible.

In addition, improvements in languages have enabled more exact control over the shape and use of data elements, culminating in the abstract data type. These data types can be specified to a very fine degree, including how and when they are accessed, and even the state of the data before and after it is accessed.

Software Build and Deployment

Many programming languages such as C and Java require the program "source code" to be translated in to a form that can be executed by a computer. This translation is done by a program called a compiler. Additional operations may be involved to associate, bind, link or package files together in order to create a usable runtime configuration of the software application. The totality of the compiling and assembly process is generically called "building" the software.

The software build is critical to software quality because if any of the generated files are incorrect the software build is likely to fail. And, if the incorrect version of a program is inadvertently used, then testing can lead to false results.

Software builds are typically done in work area unrelated to the runtime area, such as the application server. For this reason, a deployment step is needed to physically transfer the software build products to the runtime area. The deployment procedure may also involve technical parameters, which, if set incorrectly, can also prevent software testing from beginning. For example, a Java application server may have options for parent-first or parent-last class loading. Using the incorrect parameter can cause the application to fail to execute on the application server.

The technical activities supporting software quality including build, deployment, change control and reporting are collectively known as Software configuration management. A number of software tools have arisen to help meet the challenges of configuration management including file control tools and build control tools.

Testing

Software testing, when done correctly, can increase overall software *quality of conformance* by testing that the product conforms to its requirements. Testing includes, but is not limited to:

1. Unit Testing
2. Functional Testing
3. Regression Testing
4. Performance Testing
5. Failover Testing
6. Usability Testing

A number of agile methodologies use testing early in the development cycle to ensure quality in their products. For example, the test-driven development practice, where tests are written before the code they will test, is used in Extreme Programming to ensure quality.

Runtime

Runtime reliability determinations are similar to tests, but go beyond simple confirmation of behaviour to the evaluation of qualities such as performance and interoperability with other code or particular hardware configurations.

Software quality factors

A software quality factor is a non-functional requirement for a software program which is not called up by the customer's contract, but nevertheless is a desirable requirement which enhances the quality of the software program. Note that none of these factors are

binary; that is, they are not "either you have it or you don't" traits. Rather, they are characteristics that one seeks to maximize in one's software to optimize its quality. So rather than asking whether a software product "has" factor *x*, ask instead the *degree* to which it does (or does not).

Some software quality factors are listed here:

Understandability
: Clarity of purpose. This goes further than just a statement of purpose; all of the design and user documentation must be clearly written so that it is easily understandable. This is obviously subjective in that the user context must be taken into account: for instance, if the software product is to be used by software engineers it is not required to be understandable to the layman.

Completeness
: Presence of all constituent parts, with each part fully developed. This means that if the code calls a subroutine from an external library, the software package must provide reference to that library and all required parameters must be passed. All required input data must also be available.

Conciseness
: Minimization of excessive or redundant information or processing. This is important where memory capacity is limited, and it is generally considered good practice to keep lines of code to a minimum. It can be improved by replacing repeated functionality by one subroutine or function which achieves that functionality. It also applies to documents.

Portability
: Ability to be run well and easily on multiple computer configurations. Portability can mean both between different hardware—such as running on a PC as well as a smartphone—and between different operating systems—such as running on both Mac OS X and GNU/Linux.

Consistency
: Uniformity in notation, symbology, appearance, and terminology within itself.

Maintainability
: Propensity to facilitate updates to satisfy new requirements. Thus the software product that is maintainable should be well-documented, should not be complex, and should have spare capacity for memory, storage and processor utilization and other resources.

Testability
: Disposition to support acceptance criteria and evaluation of performance. Such a characteristic must be built-in during the design phase if the product is to be easily testable; a complex design leads to poor testability.

Usability
: Convenience and practicality of use. This is affected by such things as the human-computer interface. The component of the software that has most impact on this is the user interface (UI), which for best usability is usually graphical (i.e. a GUI).

Reliability

Ability to be expected to perform its intended functions satisfactorily. This implies a time factor in that a reliable product is expected to perform correctly over a period of time. It also encompasses environmental considerations in that the product is required to perform correctly in whatever conditions it finds itself (sometimes termed robustness).

Efficiency

Fulfillment of purpose without waste of resources, such as memory, space and processor utilization, network bandwidth, time, etc.

Security

Ability to protect data against unauthorized access and to withstand malicious or inadvertent interference with its operations. Besides the presence of appropriate security mechanisms such as authentication, access control and encryption, security also implies resilience in the face of malicious, intelligent and adaptive attackers.

Measurement of software quality factors

There are varied perspectives within the field on measurement. There are a great many measures that are valued by some professionals—or in some contexts, that are decried as harmful by others. Some believe that quantitative measures of software quality are essential. Others believe that contexts where quantitative measures are useful are quite rare, and so prefer qualitative measures. Several leaders in the field of software testing have written about the difficulty of measuring what we truly want to measure well.

One example of a popular metric is the number of faults encountered in the software. Software that contains few faults is considered by some to have higher quality than software that contains many faults. Questions that can help determine the usefulness of this metric in a particular context include:

1. What constitutes “many faults?” Does this differ depending upon the purpose of the software (e.g., blogging software vs. navigational software)? Does this take into account the size and complexity of the software?
2. Does this account for the importance of the bugs (and the importance to the stakeholders of the people those bugs bug)? Does one try to weight this metric by the severity of the fault, or the incidence of users it affects? If so, how? And if not, how does one know that 100 faults discovered is better than 1000?
3. If the count of faults being discovered is shrinking, how do I know what that means? For example, does that mean that the product is now higher quality than it was before? Or that this is a smaller/less ambitious change than before? Or that fewer tester-hours have gone into the project than before? Or that this project was tested by less skilled testers than before? Or that the team has discovered that fewer faults reported is in their interest?

This last question points to an especially difficult one to manage. All software quality metrics are in some sense measures of human behavior, since humans create software. If a team discovers that they will benefit from a drop in the number of reported bugs, there

is a strong tendency for the team to start reporting fewer defects. That may mean that email begins to circumvent the bug tracking system, or that four or five bugs get lumped into one bug report, or that testers learn not to report minor annoyances. The difficulty is measuring what we mean to measure, without creating incentives for software programmers and testers to consciously or unconsciously "game" the measurements.

Software quality factors cannot be measured because of their vague definitions. It is necessary to find measurements, or metrics, which can be used to quantify them as non-functional requirements. For example, reliability is a software quality factor, but cannot be evaluated in its own right. However, there are related attributes to reliability, which can indeed be measured. Some such attributes are mean time to failure, rate of failure occurrence, and availability of the system. Similarly, an attribute of portability is the number of target-dependent statements in a program.

A scheme that could be used for evaluating software quality factors is given below. For every characteristic, there are a set of questions which are relevant to that characteristic. Some type of scoring formula could be developed based on the answers to these questions, from which a measurement of the characteristic can be obtained.

Understandability

Are variable names descriptive of the physical or functional property represented? Do uniquely recognisable functions contain adequate comments so that their purpose is clear? Are deviations from forward logical flow adequately commented? Are all elements of an array functionally related?...

Completeness

Are all necessary components available? Does any process fail for lack of resources or programming? Are all potential pathways through the code accounted for, including proper error handling?

Conciseness

Is all code reachable? Is any code redundant? How many statements within loops could be placed outside the loop, thus reducing computation time? Are branch decisions too complex?

Portability

Does the program depend upon system or library routines unique to a particular installation? Have machine-dependent statements been flagged and commented? Has dependency on internal bit representation of alphanumeric or special characters been avoided? How much effort would be required to transfer the program from one hardware/software system or environment to another?

Consistency

Is one variable name used to represent different logical or physical entities in the program? Does the program contain only one representation for any given physical or mathematical constant? Are functionally similar arithmetic expressions similarly constructed? Is a consistent scheme used for indentation, nomenclature, the color palette, fonts and other visual elements?

Maintainability

Has some memory capacity been reserved for future expansion? Is the design cohesive—i.e., does each module have distinct, recognizable functionality? Does the software allow for a change in data structures (object-oriented designs are more likely to allow for this)? If the code is procedure-based (rather than object-oriented), is a change likely to require restructuring the main program, or just a module?

Testability

Are complex structures employed in the code? Does the detailed design contain clear pseudo-code? Is the pseudo-code at a higher level of abstraction than the code? If tasking is used in concurrent designs, are schemes available for providing adequate test cases?

Usability

Is a GUI used? Is there adequate on-line help? Is a user manual provided? Are meaningful error messages provided?

Reliability

Are loop indexes range-tested? Is input data checked for range errors? Is divide-by-zero avoided? Is exception handling provided? It is the probability that the software performs its intended functions correctly in a specified period of time under stated operation conditions, but there could also be a problem with the requirement document...

Efficiency

Have functions been optimized for speed? Have repeatedly used blocks of code been formed into subroutines? Has the program been checked for memory leaks or overflow errors?

Security

Does the software protect itself and its data against unauthorized access and use? Does it allow its operator to enforce security policies? Are security mechanisms appropriate,

adequate and correctly implemented? Can the software withstand attacks that can be anticipated in its intended environment?

User's perspective

In addition to the technical qualities of software, the end user's experience also determines the quality of software. This aspect of software quality is called usability. It is hard to quantify the usability of a given software product. Some important questions to be asked are:

- Is the user interface intuitive (self-explanatory/self-documenting)?
- Is it easy to perform simple operations?
- Is it feasible to perform complex operations?
- Does the software give sensible error messages?
- Do widgets behave as expected?
- Is the software well documented?
- Is the user interface responsive or too slow?

Also, the availability of (free or paid) support may factor into the usability of the software.

Chapter 14

Software Development Effort Estimation

Software development efforts estimation is the process of predicting the most realistic use of effort required to develop or maintain software based on incomplete, uncertain and/or noisy input. Effort estimates may be used as input to project plans, iteration plans, budgets, investment analyses, pricing processes and bidding rounds.

State-of-practice

Published surveys on estimation practice suggest that expert estimation is the dominant strategy when estimating software development effort.

Typically, effort estimates are over-optimistic and there is a strong over-confidence in their accuracy. The mean effort overrun seems to be about 30% and not decreasing over time. The strong over-confidence in the accuracy of the effort estimates is illustrated by the finding that, on average, if a software professional is 90% confident or "almost sure" to include the actual effort in a minimum-maximum interval, the observed frequency of including the actual effort is only 60-70% .

Currently the term "effort estimate" is used to denote as different concepts as most likely use of effort (modal value), the effort that corresponds to a probability of 50% of not exceeding (median), the planned effort, the budgeted effort or the effort used to propose a bid or price to the client. This is believed to be unfortunate, because communication problems may occur and because the concepts serve different goals .

History

Software researchers and practitioners have been addressing the problems of effort estimation for software development projects since at least the 1960s; see, e.g., work by Farr and Nelson.

Most of the research has focused on the construction of formal software effort estimation models. The early models were typically based on regression analysis or mathematically derived from theories from other domains. Since then a high number of model building

approaches have been evaluated, such as approaches founded on case-based reasoning, classification and regression trees, simulation, neural networks, Bayesian statistics, lexical analysis of requirement specifications, genetic programming, linear programming, economic production models, soft computing, fuzzy logic modeling, statistical bootstrapping, and combinations of two or more of these models. The perhaps most common estimation products today, e.g., the formal estimation models COCOMO and SLIM have their basis in estimation research conducted in the 1970s and 1980s. The estimation approaches based on functionality-based size measures, e.g., function points, is also based on research conducted in the 1970s and 1980s, but are re-appearing with modified size measures under different labels, such as "use case points" in the 1990s and COSMIC in the 2000s.

Estimation approaches

There are many ways of categorizing estimation approaches, see for example. The top level categories are the following:

- Expert estimation: The quantification step, i.e., the step where the estimate is produced based on judgmental processes.
- Formal estimation model: The quantification step is based on mechanical processes, e.g., the use of a formula derived from historical data.
- Combination-based estimation: The quantification step is based on a judgmental or mechanical combination of estimates from different sources.

Below are examples of estimation approaches within each category.

Estimation approach	Category	Examples of support of implementation of estimation approach
Analogy-based estimation	Formal estimation model	ANGEL, Weighted Micro Function Points
WBS-based (bottom up) estimation	Expert estimation	Project management software, company specific activity templates
Parametric models	Formal estimation model	COCOMO, SLIM, SEER-SEM
Size-based estimation models	Formal estimation model	Function Point Analysis, Use Case Analysis, Story points-based estimation in Agile software development
Group estimation	Expert estimation	Planning poker, Wideband Delphi
Mechanical combination	Combination-based estimation	Average of an analogy-based and a Work breakdown structure-based effort estimate
Judgmental	Combination-	Expert judgment based on estimates from

combination based estimation a parametric model and group estimation

Selection of estimation approach

The evidence on differences in estimation accuracy of different estimation approaches and models suggest that there is no "best approach" and that the relative accuracy of one approach or model in comparison to another depends strongly on the context . This implies that different organizations benefit from different estimation approaches. Findings, summarized in , that may support the selection of estimation approach based on the expected accuracy of an approach include:

- Expert estimation is on average at least as accurate as model-based effort estimation. In particular, situations with unstable relationships and information of high importance not included in the model may suggest use of expert estimation. This assumes, of course, that experts with relevant experience are available.

- Formal estimation models not tailored to a particular organization's own context, may be very inaccurate. Use of own historical data is consequently crucial if one cannot be sure that the estimation model's core relationships (e.g., formula parameters) are based on similar project contexts.

- Formal estimation models may be particularly useful in situations where the model is tailored to the organization's context (either through use of own historical data or that the model is derived from similar projects and contexts), and/or it is likely that the experts' estimates will be subject to a strong degree of wishful thinking.

The most robust finding, in many forecasting domains, is that combination of estimates from independent sources, preferable applying different approaches, will on average improve the estimation accuracy .

In addition, other factors such as ease of understanding and communicating the results of an approach, ease of use of an approach, cost of introduction of an approach should be considered in a selection process.

Uncertainty assessment approaches

The uncertainty of an effort estimate can be described through a prediction interval (PI). An effort PI is based on a stated certainty level and contains a minimum and a maximum effort value. For example, a project leader may estimate that the most likely effort of a project is 1000 work-hours and that it is 90% certain that the actual effort will be between 500 and 2000 work-hours. Then, the interval [500, 2000] work-hours is the 90% PI of the effort estimate of 1000 work-hours. Frequently, other terms are used instead of PI, e.g., prediction bounds, prediction limits, interval prediction, prediction region and, unfortunately, confidence interval. An important difference between confidence interval

and PI is that PI refers to the uncertainty of an estimate, while confidence interval usually refers to the uncertainty associated with the parameters of an estimation model or distribution, e.g., the uncertainty of the mean value of a distribution of effort values. The confidence level of a PI refers to the expected (or subjective) probability that the real value is within the predicted interval.

There are several possible approaches to calculate effort PIs, e.g., formal approaches based on regression or bootstrapping , formal or judgmental approaches based on the distribution of previous estimation error , and pure expert judgment of minimum-maximum effort for a given level of confidence. Expert judgments based on the distribution of previous estimation error has been found to systematically lead to more realistic uncertainty assessment than the traditional minimum-maximum effort intervals in several studies, see for example .

Assessing and interpreting the accuracy of effort estimates

The most common measures of the average estimation accuracy is the MMRE (Mean Magnitude of Relative Error), where MRE is defined as:

MRE = |actual effort − estimated effort| / |actual effort|

This measure has been criticized and there are several alternative measures, such as more symmetric measures , Weighted Mean of Quartiles of relative errors (WMQ) and Mean Variation from Estimate (MVFE) .

A high estimation error cannot automatically be interpreted as an indicator of low estimation ability. Alternative, competing or complementing, reasons include low cost control of project, high complexity of development work, and more delivered functionality than originally estimated. A framework for improved use and interpretation of estimation error measurement is included in .

Psychological issues related to effort estimation

There are many psychological factors potentially explaining the strong tendency towards over-optimistic effort estimates that need to be dealt with to increase accuracy of effort estimates. These factors are essential even when using formal estimation models, because much of the input to these models is judgment-based. Factors that have been demonstrated to be important are: Wishful thinking, anchoring, planning fallacy and cognitive dissonance. A discussion on these and other factors can be found in work by Jørgensen and Grimstad .

- It's easy to estimate what you know.
- It's hard to estimate what you know you don't know.
- It's very hard to estimate things that you don't know you don't know.