Software metric

7.1 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
- Code coverage
- Cohesion
- Comment density[1]
- Connascent software components
- Coupling
- Cyclomatic complexity (McCabe's complexity)
- DSQI (design structure quality index)
- 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
- Function Points and Automated Function Points, an Object Management Group standard[2]
- CISQ automated quality characteristics measures

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.[3][4] 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.[5] Others have noted that metrics have become an integral part of the software development process.[3] 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.[6] 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,[7] while others argue that imperfect quantification is better than none ("You can't control what you can't measure.").[8] Evidence shows that software metrics are being widely used by government agencies, the US military, NASA,[9] IT consultants, academic institutions,[10] and commercial and academic development estimation software.

7.2 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 classes in other packages that depend upon classes within the package is an indicator of the package's responsibility.
- **Efferent Couplings (Ce):** The number of classes in 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 unstable 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 unstable (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

7.3 Programming tool

A **programming tool** or **software development tool** is a computer program that software developers use to create, debug, maintain, or otherwise support other programs and applications. The term usually refers to relatively simple programs, that can be combined together to accomplish a task, much as one might use multiple hand tools to fix a physical object. The ability to use a variety of tools productively is one hallmark of a skilled software engineer.

The most basic tools are a source code editor and a compiler or interpreter, which are used ubiquitously and continuously. Other tools are used more or less depending on the language, development methodology, and individual engineer, and are often used for a discrete task, like a debugger or profiler. Tools may be discrete programs, executed separately – often from the command line – or may be parts of a single large program, called an integrated development environment (IDE). In many cases, particularly for simpler use, simple ad hoc techniques are used instead of a tool, such as print debugging instead of using a debugger, manual timing (of overall program or section of code) instead of a profiler, or tracking bugs in a text file or spreadsheet instead of a bug tracking system.

The distinction between tools and applications is murky. For example, developers use simple databases (such as a file containing a list of important values) all the time as tools. However a full-blown database is usually thought of as an application or software in its own right. For many years, computer-assisted software engineering (CASE) tools were sought after. Successful tools have proven elusive. In one sense, CASE tools emphasized design and architecture support, such as for UML. But the most successful of these tools are IDEs.

List of tools

Software tools come in many forms:

- Binary compatibility analysis: ABI Compliance Checker
- Bug Databases: Comparison of issue tracking systems - Including bug tracking systems
- Build Tools: Build automation, List of build automation software
- Code coverage: Code coverage#Software code coverage tools.
- Code Sharing Sites: Freshmeat, Krugle, Sourceforge, GitHub. See also Code search engines.
- Compilation and linking tools: GNU toolchain, gcc, Microsoft Visual Studio, CodeWarrior, Xcode, ICC
- Debuggers: Debugger#List of debuggers. See also Debugging.
- Disassemblers: Generally reverse-engineering tools.
- Documentation generators: Comparison of documentation generators, help2man, Plain Old Documentation, asciidoc
- Formal methods: Mathematical techniques for specification, development and verification
- GUI interface generators
- Library interface generators: SWIG

- Integration Tools
- Memory debuggers are frequently used in programming languages (such as C and C++) that allow manual memory management and thus the possibility of memory leaks and other problems. They are also useful to optimize efficiency of memory usage. Examples: dmalloc, Electric Fence, Insure++, Valgrind
- Parser generators: Parsing#Parser development software
- Performance analysis or profiling: List of performance analysis tool
- Refactoring Browser
- Revision control: List of revision control software, Comparison of revision control software
- Scripting languages: PHP, Awk, Perl, Python, REXX, Ruby, Shell, Tcl
- Search: grep, find
- Source code Clones/Duplications Finding: Duplicate code#Tools
- Source code formatting: indent
- Source code editor
    - Text editors: List of text editors, Comparison of text editors
- Source code generation tools: Automatic programming#Implementations
- Static code analysis: lint, List of tools for static code analysis
- Unit testing: List of unit testing frameworks

IDEs

Integrated Development Environments combine the features of many tools into one package. They for example make it easier to do specific tasks, such as searching for content only in files in a particular project. IDEs may for example be used for development of enterprise-level applications.

Different aspects of IDEs for specific programming languages can be found in this comparison of integrated development environments.

Programming complexity

Programming complexity (or **software complexity**) is a term that encompasses numerous properties of a piece of software, all of which affect internal interactions. According to several commentators, there is a distinction between the terms complex and complicated. Complicated implies being difficult to understand but with time and effort, ultimately knowable. Complex, on the other hand, describes the interactions between a number of entities. As the number of entities increases,

the number of interactions between them would increase exponentially, and it would get to a point where it would be impossible to know and understand all of them. Similarly, higher levels of complexity in software increase the risk of unintentionally interfering with interactions and so increases the chance of introducing defects when making changes. In more extreme cases, it can make modifying the software virtually impossible. The idea of linking software complexity to the maintainability of the software has been explored extensively by Professor Manny Lehman, who developed his Laws of Software Evolution from his research. He and his co-Author Les Belady explored numerous possible Software Metrics in their oft cited book,[1] that could be used to measure the state of the software, eventually reaching the conclusion that the only practical solution would be to use one that uses deterministic complexity models.

Measures

Many measures of software complexity have been proposed. Many of these, although yielding a good representation of complexity, do not lend themselves to easy measurement. Some of the more commonly used metrics are

- McCabe's cyclomatic complexity metric
- Halsteads software science metrics
- Henry and Kafura introduced Software Structure Metrics Based on Information Flow in 1981[2] which measures complexity as a function of fan in and fan out. They define fan-in of a procedure as the number of local flows into that procedure plus the number of data structures from which that procedure retrieves information. Fan-out is defined as the number of local flows out of that procedure plus the number of data structures that the procedure updates. Local flows relate to data passed to and from procedures that call or are called by, the procedure in question. Henry and Kafura's complexity value is defined as "the procedure length multiplied by the square of fan-in multiplied by fan-out" (Length $\times$(fan-in $\times$ fan-out)^2).

- A Metrics Suite for Object Oriented Design[3] was introduced by Chidamber and Kemerer in 1994 focusing, as the title suggests, on metrics specifically for object oriented code. They introduce six OO complexity metrics; weighted methods per class, coupling between object classes, response for a class, number of children, depth of inheritance tree and lack of cohesion of methods

There are several other metrics that can be used to measure programming complexity:

- Branching complexity (Sneed Metric)
- Data access complexity (Card Metric)
- Data complexity (Chapin Metric)
- Data flow complexity (Elshof Metric)
- Decisional complexity (McClure Metric)

Types

Associated with, and dependent on the complexity of an existing program, is the complexity associated with changing the program. The complexity of a problem can be divided into two parts:[4]

1. Accidental complexity: Relates to difficulties a programmer faces due to the chosen software engineering tools. A better fitting set of tools or a more high-level programming language may reduce it.
2. Essential complexity: Is caused by the characteristics of the problem to be solved and cannot be reduced