Measuring Engineering by Kilian Carolan

"The rationale for using metrics is to improve the software engineering decision making process from a managerial and technical perspective"(Fenton, N.E.)

Over the course of the this report I will discuss the various ways in which the process of software engineering can be measured and analyzed, the toolsets and platforms which can be used to carry out the analyzed. The content of this report will be split into four main sections. In the first I will discuss what data is measured and reported on, in the second section I will discuss where we will compute these metrics, in the third the algorithms which can be used for analyses will be discussed, and in the final part the ethics behind these processes shall be discussed.

Why we want software metrics

Software metrics are important due to the fact we can assign a value to a design attributes thus quantifying their value. By measuring the characteristics of system components such as cyclomatic complexity and then consolidating these measurements, you can quantify attributes of system quality such as maintainability. We can also use software metrics to identify the design components whose quality is below standard. Metrics can identify components with characteristics that deviate from the standard for example you can measure components to discover those with highest complexity, these in turn are most likely to contain defects due to their complexity. (Software Engineering 9 Ian Sommerville 24.4)

WHAT DATA

Software metrics dates back to the late 1960's where productivity was originally measured by the LOC measurement meaning lines of code written by a programmer in a month, and how many mistakes were made in KLOC or thousand lines of code. In 1971 Akiyama produced what was possibly the first model for predicting software quality this also used KLOC where he used this in conjunction with a regression line to create a crude forecasting model for predicting overall software quality, "In other words he was using KLOC as a surrogate measure for program complexity. (Fenton, N.E.)

However, using LOC for a software metric for complexity and functionality has critical drawbacks, firstly it does not take into account different programmer's styles. It also suggests that more is better, when shorter code could in fact be more efficient and solve the problem at hand better. Another key drawback to the LOC approach is that LOC in assembly language is not comparable to LOC in high level languages.

In general software metrics would fall into two main categories control metrics and predictor metrics

1. Control metrics are usually associated with software processes, for examples the average effort and time required to repair reported defects.
2. Predictor metrics are associated with the software itself and are also known as product metrics. Examples include cyclomatic complexity of a module, the average length of identifiers in a program and the number of attributes and operations associated with object classes in a design.

Examples of Software Metric

The following metrics are known as the Chidamber and Kemerer’s suite also known as the CK Suite. These are the six most used metrics for object orientated programing

1. Fan-in/Fan-out: Fan-in is a metric for the number of functions (X) that call another function. Fan-out is the number of functions that are called by function X. High values for fan-in mean that X is closely correlated to the rest of the design. Hence changes in X will have greater importance on the rest of the code. A high value for fan-out suggests that overall complexity of X may be high because of the complexity of the design logic needed to utilize the called methods and function.
2. Length of code: This is a metric for the overall size of a program. In general, the more lines of code written for a program or method, the more complex and prone to bugs the component is likely to be. Length of code has been shown to be one of the most reliable metrics for predicting error proneness in components as length of code is highly correlated to number of bugs.
3. Cyclomatic complexity: This is a measure of the control complexity of a program i.e. the number of linearly independent paths through a program's source code. This control complexity may be related to program understandability
4. Length of identifiers. A measure of the average length of names for identifiers i.e. the names of variables, classes, methods and other such entities in a program. The longer the identifiers the more likely they are to be meaningful hence a more understandable program
5. Depth of conditional nesting: This is a metric which measures the nesting depth of if-statements in a program. Deeply nested if statements are hard to understand and follow and this means they are potentially error-prone
6. Fog-index: This is a measure of the average length of words and sentences in documentation. The higher the value of a documents fog index the more difficult the document is to understand this metric could be particularly useful in an open source or team environment ensuring that all member of the team or an engineer working to improve the code or adapt to their own purposes can easily understand why each method was written

(Software Engineering 9 Ian Sommerville 24.4)

Technical debt, is a concept in [software development](https://en.wikipedia.org/wiki/Software_development) that reflects the implied cost of additional rework caused by choosing an easy solution now instead of using a better approach that would take longer. This is another piece of data measured in the software metrics. This idea essentially comes down to how much extra work would need to be done per unit of time to correct problems created by your code imperfections in other words the maintainability of the code. This creates a tradeoff between getting code released early without being perfect but still useful against the work not being done due to the code not being released whilst it is being perfected. It could be compared to building bridges before you come to the crossing. While this will make the journey faster it delays when you can begin. (Techopedia).

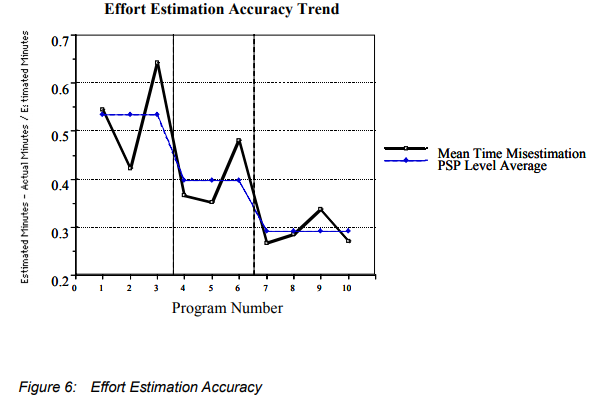
Personal Software Process (Watts S. Humphrey, November 2000)

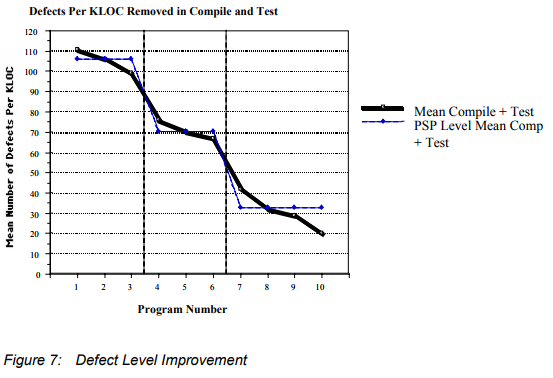
The Personal Software Process (PSP) is a structured software development process created to help software engineers better understand what they were doing and help improve their methodology by keeping track of their expected versus their actual development of code. This process was developed by Watts Humphreys. Specifically, PSP hopes to improve software development by helping an engineer improve their planning skills, helping them make commitments they can keep, improving the quality of their projects and reducing the number of errors in the development.

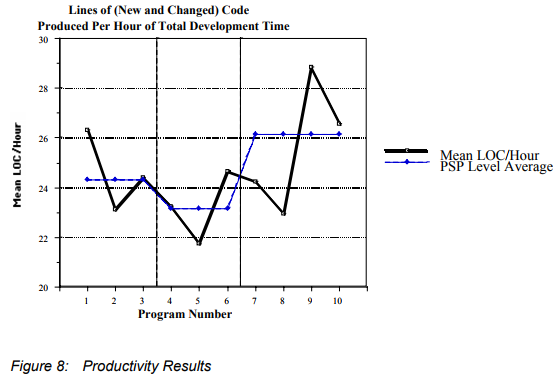
PSP follows a progressive structure in that an engineer will begin integrating PSP into their software development process in stages. The engineer should begin at PSP0 where an engineer would use their existing programming practices but log data like how much time was spent at each phase and log number of defects found. The engineer would then work his way through each of the seven PSP phases. To give a brief breakdown of each stage: PSP0-PSP0.1 introduces process discipline and measurement, PSP1-PSP1.1 introduces estimating and planning, PSP2, PSP2.1 introduces quality management and design and PSP3 introduces cyclical development.

Significant improvements in engineering were recorded in regards to estimating accuracy and early defect removal while not significantly affecting productivity (Hayes), a summary of the results can be found in the following graphs taken from The Personal Software ProcessSM (PSPSM) by Watts S Humphreys

Once at grips with PSP an engineer can then move on to the Team Software Process (TSP), which in conjunction with the PSP gives a concrete functional process structure which is designed to help teams consisting of managers and engineers to organize projects and produce software products of varying sizes. In short the TSP is intended to improve the levels of quality and productivity of a team’s software development. Here a team leader would be introduced who would be in charge of making sure standards are met and to ensuring good communication and that the team understands the direction they are trying to take (<https://www.sei.cmu.edu/library/assets/presentations/intro_tsp_2010_06_27.pdf>.)







PSP has been compared to lighting a candle in a study by Philip M. Johnson in the University of Hawaii. PSP is a candle lit to help look for something in the dark rather than just looking where the streetlight covers. By this they mean that the process is analytically flexible, it encourages situation specific analytics, like how a candle can be moved to navigate through darkness, PSP encourages users to use analytics best suited to their needs. However, PSP is not without inherent problems. The manual nature of PSP can have potential for significant data quality problems, and like how a candle flame can be easily extinguished, PSP is fragile. (Philip M. Johnson)

During this study PSP led the team to LEAP- “lightweight, empirical, anti-measurement dysfunction and portable software process measurement” toolkit. This toolkit automates and normalizes data in that developers still enter most data but the toolkit automates subsequent PSP Analyses and provides analyses that PSP cannot. In this simile the LEAP toolkit replaces the PSP candle with a campfire introducing a higher level tool support thus “increasing light” by improving data quality and decreasing manual analysis required. However, a campfire is not as flexible as a candle, by introducing automation the leap toolkit makes certain analytics easy to collect and others more difficult. The team at the university of Hawaii came to the conclusion that PSP could not be fully automated however they agreed with agile developers that development overheads such as manual data entry doesn’t provide enough return on the man hours needed. However, they went in a different direction to the agile community in that extensive measurement and analysis was still valued

The question then arose “What kinds of useful software analytics could [they] obtain if both collection and analysis were “free”? (Philip M. Johnson). This question led to a decade long study at the university of Hawaii known as Hackystat. Hackystat goes against the conventional wisdom of defining high level goals first and then figuring out what data collection analysis is necessary to achieve them, in fact Hackystat does the opposite. They aimed to develop ways of collecting process and product data whilst minimizing overhead costs for developers, then decide the high level engineering goals that could be supported by analyses on this data.

Hackystat “implements a service oriented architecture”. Sensors attached to the development tools gather process and product data and send it to a server. This is useful as other services can then query and then build higher-level analyses based of data collected by Hackystat. There are four main design features of Hackystat:

1. Client-and-server-side data collection: Instrumentation was developed for client-side tools like editors, build tools, and test tools. Server-side tools were also developed such as configuration management repositories and build servers.
2. Unobtrusive data collection: Using Hackystat client-side tools locally caches data gathered while an engineer works offline. It then sends the data to the Hackystat data repository when the engineer reconnects
3. Fine grained data collection: Hackystat allows for data to be collected on a minute-by-minute or second-by second basis. For example, Hackystat a measures buffer transition i.e. collecting a data instance anytime an engineer changes the active buffer from one file to another. Hackystat allows for an engineer to be tracked as they edit a method, creates a test case for a method or invokes the test.
4. Both personal and group based development: Hackystat allows for engineers to define projects and group work. Hackystat can then keep record when different engineers edit the same file

(Searching under the Streetlight for Useful Software Analytics Philip M. Johnson)

Sonar is another toolset which allows for the computation of software metrics. It is an open source platform used to manage code quality mainly for java developers but through the use of source code other languages can also be supported. Sonar allows for analyses on 7 main facets: Duplicated code, coding standards, unit tests, complex code, potential bugs, comments and design. Sonar works of three main features

1. Sonar contains a set of source code analyzers that are grouped in a Maven plugin and called when needed. The built in analyzers then use the configuration stored in the database. Although Sonar uses maven plugins to run its analysis, it is also capable of analyzing maven plugin projects.
2. Sonar utilizes a database to give results of an analysis, the projects, global configuration and to keep historical analysis .5 database engines are currently supported by Sonar these are Oracle, MySQL, Derby, PostgreSQL and MS SQLServer
3. Sonar also features a web reporting tool which display code quality dashboards on projects, checks for defects and configure analysis.