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.

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.

LOC style metrics would fall under the category of product metrics, as would design metrics and object orientated metrics. We also have the category of requirement metrics such as size requirement metrics but also others like traceability metrics. A third category of software metrics would fall under the heading of process metrics. This would hope to track aspects of the design process such as scheduling and human resource hours. (Software Metrics Alex Boughton )

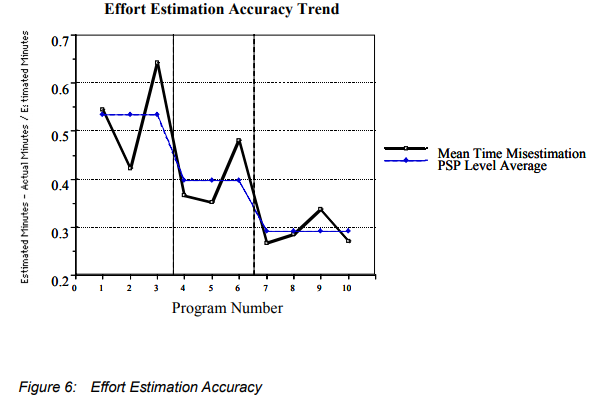
Personal Software Process (Watts S. Humphrey, November 2000)

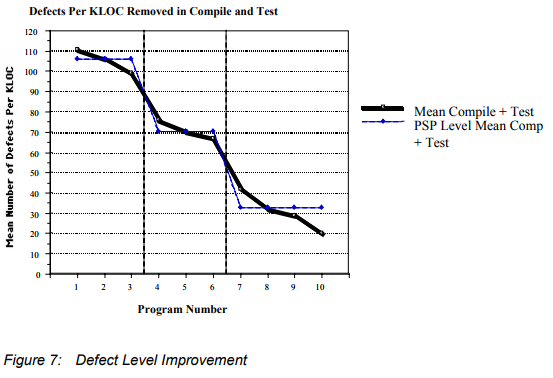
The Personal Software Process (PSP) is a structured software development process created to help software engineers better understand what they were doing and help improve their methodology by keeping track of their expected versus their actual development of code. This process was developed by Watts 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.

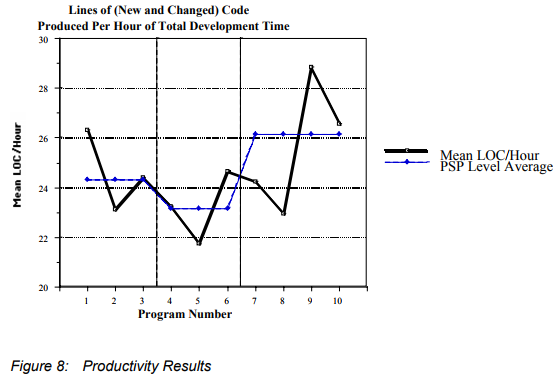
PSP follows a progressive structure in that an engineer will begin integrating PSP into their software development process in stages. The engineer should begin at PSP0 where an engineer would use their existing programming practices but log data like how much time was spent at each phase and log number 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 figure out the data collection analysis necessary to achieve this, in fact Hackystat does the opposite. They aimed to develop ways of collecting software process and product data whilst minimizing overhead costs for developers , then determine what high level engineering goals could be supported by analyses on this data

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 (techopedia), is another piece of data measured in the software metrics.