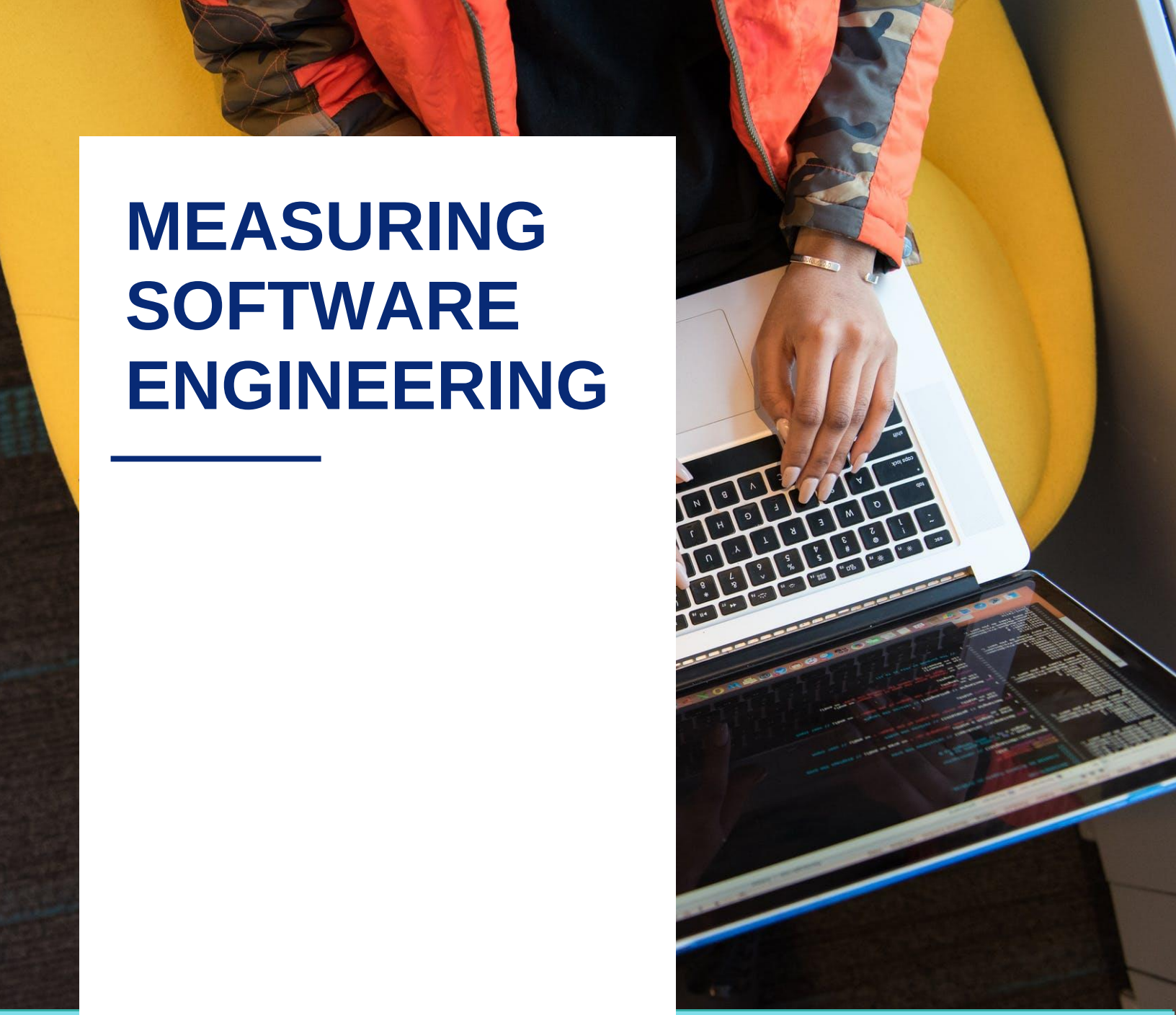

MEASURING SOFTWARE ENGINEERING

SOFTWARE ENGINEERING CSU3301

STEPHEN ROWE - 14319662

Before discussing the methods and tools available for measuring software engineering, it is important to first clarify the goals that we want to achieve, to guide how we approach this topic. These measurements should ideally allow us to:

- Maintain software codebases
- Improved assessment of individual engineers
- Improve the design stages of software development
- Identify new skill developers should be dedicating time to
- Identify bottlenecks in development
- Automate assessment, reduce dependency on manager intuition
- Test and validate novel ideas and study their usefulness/effectiveness

The use of software metrics remained as a very experimental field from the 60s through to the 90s, (Norman E. Fenton, 1999), and was typically seen as a more frivolous and begrudgingly undertaken exercise within this industry at large, with little synchronization between academic advances and practical industry application. Early attempts focused on LOC or KLOC (lines of codes, or kilo-lines of code) and were directly compared with code defects, which overall served as a very crude measurement of effort and complexity and quickly ran into issues with the advent of high-level and object oriented languages (that is, measurements of the use of these languages were incompatible with the measurement of equivalent Assembly programs). The field gradually evolved into a more meta-focused analysis while approaching the turn of the century, prioritizing instead a code-independent measurement of software complexity (through function point analysis and McCabe's cyclomatic number (McCabe, Dec. 1976)), practical implementation of metrics programs, and the measurement of the effectiveness and usefulness of certain software measurement techniques.

An example of an algorithm used for software metrics is McCabe Cyclomatic Complexity, as mentioned previously, which is a method for determining the complexity of a function or section of code within a program. As complexity of

code can be a consistent bottleneck, especially with respect to development in teams or when introducing new developers, a solid formulaic approach to determining the complexity of a piece of code can be an invaluable asset in software development measurement. Cyclomatic Complexity is seen as a "magic number" (Stadlbauer, 2018), produced by constructing a flow-graph consisting of Nodes and Edges, with each node representing an instruction and each edge representing the flow of operations. The equation is $M = E - N + 2$, which represents the number of paths that can be taken depending on the input to the function. It scrutinizes branches and is unaffected by LOC, and as a result, is a powerful determination of the difficulty in maintaining and understanding the code for future use. An example of McCabe's method being implemented in industry is Hewlett Packard's rule that any modules that produce a cyclomatic complexity value greater than 16 should be completely re-written (McCabe, Dec. 1976) (although a lower threshold is more often expected today), and McCabe's complexity value is typically included in analysis of code bases in most modern software measurement systems. Despite this however, the method has its detractors (Hummel, 2014), suggesting that this method on its own is a far more useful method for determining how extensive testing should be on the software as opposed to its difficulty of human interpretation, and that in order to achieve its intended result, should be used in conjunction with analysis of nesting depth, code length, and the context that a developer should have full understanding of before changing any single line of code.

Innovations introduced by Watts Humphrey, in particular the Personal Software Process (Humphrey W. , 1999) and Team Software Process (Humphrey W. S., 2000) serve as a foundation to software measurement, and aimed to introducing routines to a developer's process in order to reduce code defects, create a quantifiable measurement to allow for more realistic future project commitments, and consistent individual improvement. This approach added significant overhead to developers, as it required consistent personal logging of time spent, errors made, and accumulation of other historical data. It was

intended to be progressively introduced to the developers workflow, starting out at PSP0, in which the developer would consistently input data about their process as they worked, followed by a post-mortem in which the developer would suggest approaches that could improve their workflow going forward (known as the personal process improvement plan). Data from this stage would then go into PSP1, in which the developer would be assigned a new project/task, and use their historical data to estimate time spent on the new project, and repeat the routines from PSP0 and compare, accumulating in a consistent framework for estimations going forward. Building on top of this, PSP2 introduces a code review in which the developer assesses the code defects they introduced in the process and provides analysis techniques to more easily identify defects going forward. The Personal Software Process directly ties into the Team Software Process, with each individual developer carrying out the routines outlined by the PSP, with a team manager defining goals and allocating tasks and roles, which in total, allowed for a self-managing team. The advantages of the PSP approach are threefold: (i) individual assessment adds a layer of personal responsibility to the developer over their own management and growth, both in a persistent focus on building upon previous work, (ii) allowing for a certain amount of flexibility so that PSP can be incorporated into other development methodologies, and (iii) an incremental introduction to each aspect of PSP opens the process up to (typically slow-moving) industrial embracement. However, attempts to rectify the overhead demands of PSP introduced their own set of issues, as attempts to automate the process are outlined by Philip M Johnson in the development of Hackystat (Johnson, Searching under the Streetlight for Useful Software Analytics, 2012).

Hackystat serves as an excellent case study of automated software measurement, as it is open source and well documented, and despite the fact that it is not commonly used today, I will look at it in some detail as it offers a view of a similar scope of data-collection that is used today in services such as Pluralsight Flow, as well as a view of the privacy issues and discomforts in

implementing such a system. Developed at the University of Hawaii, Monoa, its primary contribution to the field was the use of telemetry-based analysis, achieved through plug-in integrations to IDEs. Hackystat was an evolution of the *Leap Toolkit* (Johnson, 2012), which more closely followed the PSP model while offering automated personal repositories of user activities. Issues arose with its incompatibility with agile development, and its inflexibility regarding having to develop new software to integrate a specific measurement, as opposed to the flexibility of simply creating a new spreadsheet for relevant measurement (which PSP offered). Hackystat took a different approach, of integrating client-side Sensors into the software already used by developers in their workflow (IDE, build tools, etc.). It outlines four types of telemetry that was useful for analysis (Johnson, 2004):

- Development telemetry (based on tool usage, file changes, command line invocations)
- Build telemetry (based on test software such as JUnit, or build tools)
- Execution telemetry (studying stress and performance of the build)
- Usage telemetry (based on user interaction with the software)

The intention was that this system would produce decision-making value regardless how far into the project the developers were when implementation of Hackystat was introduced. It also allows for an incremental and experimental path to project management. A case study the developers of Hackystat gave was that, through experimentation with this collected data, they were able to identify categories of build errors, consider the advantages of enforcing full build tests before commits, then enforce new rules among the development team and see practically immediate results through telemetric measurement of these changes in practice, all while adding no work overhead for the developers themselves. They were also able to notice when code complexity and number of defects were rising and assess segments of otherwise functional code that were contributing to these factors. The data produced by Hackystat through telemetry was relatively simple, second-by-second logs of events, however this allows for

flexible statistical analysis with little to no additional work for the developers themselves. (Johnson, 2004)

Hackystat works by installing plug-in sensors to the applications that developers were using, which sent the aggregated data to the central Web server that is overviewing the process, in an XML structure. The sensor plugin would have to be developed for each IDE/Build system/etc. individually for compatibility, such as an Eclipse sensor, Ant sensor, Emacs sensor, etc. Each of these plug-ins would produce XML in a structure of one of several categories outlined by Hackystat documentation, such as *Activity* data type for IDE-related data collection, *FileMetric* data type for file size/metadata, *Build* data type for build outcomes and errors produced (including the developer that ran the build, a timestamp, syntactic errors), *Perf* data type for load-testing and memory-usage, and *Commit* for commits to project repositories (which would naturally be gathered in the git repository itself, but this allowed for use of the data in relation to other Hackystat statistics) and configuration management, *CLI* for command-line invocations related to the project. This is not an exhaustive list but presents the extent of information gathered. Further iterations of Hackystat introduced features such as the ability to prioritize which project artifacts to continuously inspect, and the Software ICU, which assess the project's health through succinct summaries of the overall project (coverage, complexity, coupling, churn, etc) to be used for comparison with other projects.

When it came to privacy concerns and push back against such heavy-duty surveillance, the Hackystat developers initially expected the service to be uncontroversial due to the fact that it was so code-oriented as opposed to relying on the inter-personal intuition of a manager, or the continuous, partially subjective self-analysis of PSP (Johnson, 2012). However, in practice, they noticed that while statistics such as Coverage, Complexity, Coupling and LOC were comfortably accepted, the gathering of information based on Development Time, comparison on user commits based on quantity, the number of times the

software was built, and the number of times the test units were run, were met with more reluctance. These metrics were far more dependent on individual workstyle and habits of individual employees, and the outputted results of the Software ICU may reward certain non-consequential habits over others. This introduced a balancing-plate dynamic in the introduction of telemetric software measurement between the following, with these remaining the underpinning issues regarding privacy in this field:

- Trust between the developers and management for the information to be used appropriately
- Scope of the data collected may expand, and the unobtrusive and consistent nature of the data collection removes a massive amount of control from the user over what data is collected
- Data collected may not be representative of the effort and contribution of a team member to the project when considering outside factors, with this information being open to being used as a managerial crutch as opposed to an aid.

With respect to the rewarding of habits as mentioned previously, Snipes argues that (Will Snipes, 2013) an even closer analysis of employee approaches to issues such as a straight-forward bug fix in an controlled academic environment can identify approaches that produce faster and more effective results. In the paper, he studies the approach of multiple developers solving the same problem with the use of Markov chains, and focuses particularly on the “Searcher” (i.e. a developer that uses structured navigation through the source code through referencing) and the “Browser” (i.e. a developer that uses unstructured navigation through the source code), and highlights that the lack of repeated tasks in the Searcher’s approach and the fact that the same result could be produced in less steps shows a clear benefit over the Browser’s approach. I believe that this shows a potential solution path to the privacy scope issue highlighted above, in that a controlled experimental environment with much higher scrutiny may produce more useful evidence of the benefits of habitual changes than a catch-all data vacuum approach in a regular work environment.

However, this approach in a workplace environment has two significant issues, as highlighted earlier by Fenton (Norman E. Fenton, 1999) in respect to the overhead this may produce in software development, with measurement overhead often being the first things to suffer in the face of a deadline, and the wall between academic research and industry application.

Returning to the discussion of algorithms used for software analytics, Code Churn is the rewriting of code that has already been committed, typically within a short time span. It is a natural part of software development, as it is mostly concerned with prototyping solutions, experimenting with different implementations, and it common in the early stages of development. It is not a bad quality, and a healthy level can denote that the project is progressing smoothly. However, code churn can be a major bottleneck when its primary causes are code perfectionism (in cases where it adds little additional value to the project) and in parts of the project that are already suitable for production. (Pluralsight, 2020). It is important to align code churn with real-world situations affecting the project and the point in the project's lifecycle, as it can be hard to interpret a code churn statistic if it is in isolation. For example, in the case that a project specification wasn't met and is pointed out by a client, it is likely that there will be an uptick in the code churn as a result, which is not a concerning issue. It can be used effectively to determine where requirements are not clear and can be used to identify developer burnout. Code churn is a fundamentally straightforward measurement, as it keeps a record over a span of (let's say) three weeks of commits, measures blocks that have been modified and not modified, and lines that have been added, deleted and modified within files of the project. Lines deleted and lines modified are the main interest and can be compared with lines added to see if the project is progressing or stagnating. Visual Studio Team Foundation Server (Microsoft, 2017) measures total churn as [Lines Added] + [Lines Deleted] + [Lines Modified].

Pluralsight Flow is a more modern equivalent to Hackystat. It focuses primarily on project Git repositories as opposed to plug-in software for its analytics (Pluralsight, 2020). It provides three levels of metrics:

- Quarterly and annual trends (at all levels of the organization)
- Remote Meetings and Scrum Meetings – mixing agile development into the analytics
- Granular analysis – team member work patterns across commit, pull requests and ticket activity.

Granular analysis is calculated on a per-engineer, per-week basis for easy comparisons, with the primary metrics being commits per day, "impact" (a proprietary metric that combines complexity and quantity) and efficiency (percentage of code that is going into production, with a value of about 75% being seen as healthy as 100% efficiency would indicate that development is being rushed). It also consists of views depending on the role played by an employee, for example a Reviewer and Reviewee view, where receptiveness to feedback is measured, percentage of unreviewed pull requests and contribution to pull request reviews are all calculated. Pull requests can be categorized into three categories of importance, i.e. High Activity PRs, Long Running PRs, Unreviewed PRs, which allow the manager to prioritize issues.

The software allows for analysis of each sprint, with a focus on Pull Request resolution, which includes time to resolve a pull request, time to first comment, volume of follow-on comments, and the impact of the reviewer on the pull request discussion. This would ideally create a standard within the company for the timeframe expected for pull request resolution and scrutinize outlier PRs. This shows how software analytics can easily be implemented into typical Agile development methods, as it readily improves feedback for Scrum meetings. The dynamic between reviewers within the company can also be produce, which highlights the pull request reviewers that are most active and the employees they are interacting with. Using git-centered analysis software, such as Flow and WayDev, shows the significant insights that this can produce into the dynamics

between employees within a company, allow managers to identify where real-world issues may be affecting development, encourage developers to get involved in multiple aspects of the system as opposed to specializing and restricting themselves, and is able to achieve similar and better results without the need for the installation of plugins and tracking of time that Hackystat focused on.

As a privacy-focused person, I went into the topic of software analytics and measurement with a certain amount of unease overall but, studying these systems more in-depth and seeing how data is actually collected and how it can be effectively used, has relaxed that feeling a lot. While I still believe that extreme examples such as the Humanyze badges (Weller, 2016) are an egregious invasion of privacy and are inconsiderate to social disorders such as autism, I otherwise feel that there are very viable options that are able to strike the balance between effective assessment with constant data collection, and respecting the privacy of employees. I definitely reassessed how I weighed privacy over performance improvements while approaching the topic, and now being able to identify ways in which these software could improve my own day-to-day work habits and productivity and how they can be used to relax instead of inflame tensions within a software engineering team (under the right management). However, I do feel that the dynamic between employer and employee will be an unfortunate driving force behind further privacy-encroaching technologies in the workforce (in other words, if your employee says we're using this new system now, employees won't have much say). I hope the effectiveness of more reasonable system such as Flow will be enough to counteract this.

References

Hummel, D. B. (2014, May 20). Retrieved from https://www.cqse.eu/en/news/blog/mccabe-cyclomatic-complexity/

Humphrey, W. (1999, November 1). Retrieved from https://resources.sei.cmu.edu/library/asset-view.cfm?assetid=5283

Humphrey, W. S. (2000, November 1). Retrieved from https://resources.sei.cmu.edu/library/asset-view.cfm?assetid=5287

Johnson, P. M. (2004). *Improving Software Develment Management through Software Project Telemetry*.

Johnson, P. M. (2012). *Searching under the Streetlight for Useful Software Analytics*.

Johnson, P. M. (2012). Searching under the Streetlight for Useful Software Analytics.

McCabe, T. J. (Dec. 1976). A Complexity Measure. *IEEE Transactions on Software Engineering, vol. SE-2, no. 4*, pp. 308-320.

Microsoft. (2017). Retrieved from https://docs.microsoft.com/en-us/azure/devops/report/sql-reports/perspective-code-analyze-report-code-churn-coverage?view=azure-devops-2020

Norman E. Fenton, M. N. (1999). Software metrics: successes, failures and new directions. *The Journal of Systems and Software*. Retrieved from https://www.academia.edu/9574064/Software_metrics_successes_failures_and_new_directions

Pluralsight. (2020). Retrieved from https://www.pluralsight.com/blog/tutorials/code-churn

Pluralsight. (2020, May 20). Retrieved from https://www.youtube.com/watch?v=KPuIb6G6amM

Stadlbauer, R. (2018, April 10). Retrieved from https://dzone.com/articles/what-exactly-is-mccabe-cyclomatic-complexity

Weller, C. (2016, October 20). *Business Insider* . Retrieved from https://www.businessinsider.com/humanyze-badges-watch-and-listen-employees-2016-10?r=US&IR=T

Will Snipes, V. A. (2013). *Towards Recognizing and Rewarding Efficient Developer Work Patterns*.