Trinity College Dublin

Measuring Engineering: A Report

By:

Amaya Aita

[15317530]

CS3012: Software Engineering

Prof. Stephen Barrett

Wednesday 6th December 2017

Introduction

This report explores the various forms of measurement and assessment used to evaluate software engineering processes. Attention is paid to ethical issues, and the factors which effect the adoption of the measurement techniques. Where possible, software applications and platforms that facilitate the measurements are discussed.

Measures Based on the Software System

The original software engineering (SE) measurements were based around lines of code (LOC), and were discussed in academic circles in the 1970s and 1980s [1, 2]. LOC were used as early measures of productivity, and defects per thousand lines of code was an early software quality estimate. LOC has serious draw backs as a measure of productivity, effort, complexity, and quality. One issue is the wide variety of programming languages that use different LOC to complete the same task. Similarly, the complexity of a piece of software is not solely related to LOC. Issues such as the number of branches or paths through a system contribute to the overall complexity.

One early measure of software size independent of programming language is Function Points [3]. Function Points seek to size each element of a software system. This size should reflect the amount of functionality a user can expect from the system or component. From an organizational point of view, this measure can then be used to estimate resource costs, by using measured components or systems as a predictor. As SE measurement matured, several ISO standards were developed for sizing software systems based on function points. These include COSMIC, FiSMA and NESMA.

A measure known as Cyclomatic Complexity is one of the foundational metrics of direct measurements [4]. This measure represents the number of linearly independent paths through a program. Its calculation follows a graph based approach, using a control flow graph. It is a measure of the maximum number of branches that may need to be investigated to find a bug. This measure can be applied to a software system at various levels to give a more complete picture of complexity. Individual source code files and methods can be scored for complexity. This measure can then inform the software engineering process by implementing processes designed to reduce the type of identified complexity. A software measurement tool called McCabe IQ emerged from this research. This tool now includes many complexity measures that relate to code complexity, data complexity, complexity in relation to object orientated structure, and metrics related to a programs difficulty to comprehend or create. Design Complexity Metric is a code complexity metric which is included in McCabe IQ. This metric measures the amount of interaction between modules in a system. Essential Complexity Metric is another available measurement used to estimate the structuredness and quality of code.

The data specific metrics in McCabe IQ include Data Reference Metric (DR), Data Reference Severity Metric (DR_severity) and Maintenance Severity Metric (maint_severity). DR is a measure that calculates the number of times a data related variable is used in the code, high DR_severity indicates many data related variables are used in the code, and maint_severity measures how difficult a module is to keep maintained.

Object Orientated Programming adds structure and relations between code elements. When structure is added to a system, new measures become available. Simply calculating the same measures as before at finer-granularity increases the depth of analysis that is possible. There are multiple Object Orientated measures available in McCabe IQ. The measure representing depth in the class hierarchy is one of the simplest measures available. Number of Children (NOC), which is a count of the number of classes that derive from a class, is a useful value measure of a classes importance. Weighted Methods Per Class (WMC) provides an object centric estimate of class size

In 1977, Maurice Howard Halstead published a collection of new software metrics [6]. These metrics are Program Vocabulary, Program Length, Calculated Program Length, Volume, Difficulty, Time Required to Program, Effort, and Number of Delivered Bugs. These measures are used in various software measurement solutions such as CMT++ , CMTJava, and MaCabe IQ.

Measures based around testing are another class of metric for software systems. As testing became an integral part of the SE process, so too did metrics around tests. The number of tests which pass is the simplest metric based around testing. Code coverage is one of the most important measures calculated in relation to testing. Code Coverage can be conceptualized in terms of the control flow graph on which Cyclomatic Complexity is calculated. If each path in the control flow graph is tested, the code is fully covered. If code is not fully covered, each untested path could potentially hide unexpected behavior. It is well known that for non-trivial software systems, manually testing of all possible execution paths is practically impossible. However, SE processes and paradigms, such as automated testing and Test-Driven Development, emerged to help increase code coverage [8, 9].

Code quality is sometimes characterized by the existence of Code Smells [5]. A Code Smell is a design anti-pattern, or bad practice. A Code Smell might not create a bug, but “smelly” code is considered more difficult to maintain and extend. Smells are identified through experience, and are not formally identifiable. Taxonomies of smells for various languages have been created through expert collaboration. Several software measurement systems provide a metric which counts smells. These include Checkstyle, Sonar, PMD, and FindBugs.

The metrics are also known as direct measurements, as they relate directly to the outputs of the SE process i.e. source code files, methods and tests. These direct measurements examine engineering output at different points in the development cycle to assess the quality of the SE process. They are easy to collect and require no additional input from users. These measures have remained popular and have enjoyed widespread adoption over time. One of the main issues with using these measures in isolation is that they ignore most of the complexity and intricate factors that affect the SE process. For example, these measures have no information about the real-life work environment in which SE takes place. Interpersonal issues in a real-world environment can have a significant effect on the process and are not a directly measurable quantity [7].

Measurement Systems based on these direct measures have remained popular, mainly because they are easy to integrate, are unobtrusive, and create very few ethical issues. Since the data collected relates only to the outputs of the engineering process, engineers consider it “fair game”. Some examples of systems which use direct measure are DevCreek, Ohloh, Atlassian, CAST, Parasoft, McCabe, Coverity, and Sonar [7]. Many companies now have inhouse data teams where these types of measures are monitored and visualized in real-time to help steer management decisions.

Organisational & Goal Driven Measures

Organisational measures of the software engineering process evolved alongside the progression of direct measurements. These can be thought of as measurements of software project meta information. One of the first measurements of this kind used in software engineering is GQM (Goal-Question Metric), whose origins is in the field of Total Quality Management [10]. GQM is a model consisting of three levels: Goal, Question and Metric. For every object, a set of organisational goals is established.

Next, questions relating to the object must be formulated. The answers to these questions should give information about progress towards the established goals. Once this set of questions has been created a method to answer them is needed. The final step in GQM is to chose metrics which will provide answers to the questions. If the object under investigation is a software engineering process, such as the use of a productivity tool, we can define goals (improve productivity) and questions (has the tool improved productivity) that lead to a choice of goal specific measurement (A/B test results or questioners). The GQM process requires extensive engineer input and has a high overhead. In the example we just gave understanding and defining the goals and questions is an overhead that would require input from software engineers. The measurement step could also eat up development time if it requires measures like structured questioning of system users. [11]

Another organisational approach which use Bayesian Belief Nets (BBNs) to combine goal driven measurement and direct measurements has been suggested [11]. This system embeds organisational goals and questions into the structure of the net. Then using simple direct measures, analysis of the software engineering process is performed. The system forces users to make explicit the usually hidden assumptions in the decision-making process. Uncertainty and causal relationships are explicitly defined, and the probabilistic nature of the system allows forecasting even when faced with missing data.

Personal Software Process (PSP)

In 1995, a software engineer named Watts Humphrey released a book called ‘A Discipline for Software Engineering’. It explained, in detail, how goal driven organisational software processes could be applied to the individual. At the time, his work was ground-breaking, and it lead to a host of new software engineering measurement techniques and solutions [12]. Today, his work is referred to as Personal Software Process, or PSP. PSP’s main high-level goals are to improve project estimation and quality assurance. It is applied to an individual, not an organization, and it follows an iterative, three stage process.

In the first stage, the individual records all relevant data surrounding the planning development, and post mortem of a project. Various improvement techniques are added in this stage, such as coding standards. The individual is also tasked with devising an improvement plan. In the second stage, data collected from the first stage is used to estimate the time a new project will take, and how large it will be. The actual time taken, and size of the new project, is recorded during it’s execution, as with the rest of the data compiled in the first stage. Subsequent projects use this historical data for forecasting completion time. In the third stage, two new steps are added: design review and code review. The focus of these steps is to remove current faults and prevent future ones. Recording the design decisions that lead to faults, and the steps required to remove them, improves future project processes.

This original form of PSP, is purely a manual process, in which an individual enters values related to the work he or she is completing. Typical PSP systems require more than 500 unique values that the user needs to calculate and input. These systems are flexible, and can adapt to the requirements of a certain project, or organisation. Essentially, they allow for measurement adaption to fit any new requirements.

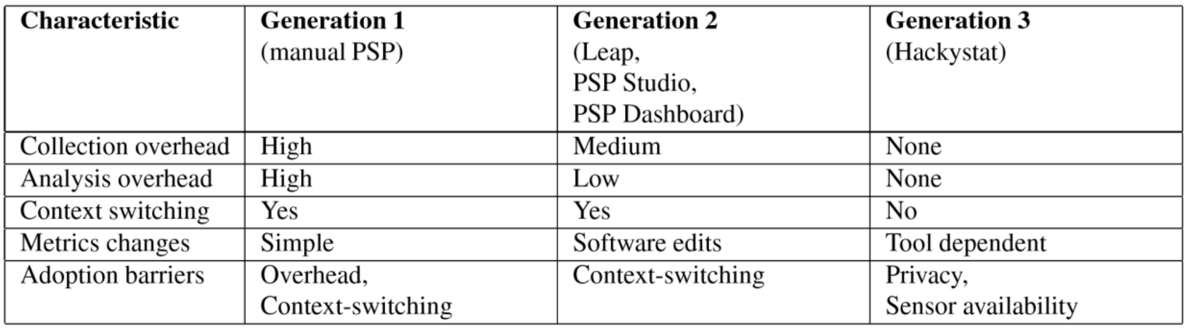
PSP makes two resounding claims. The first is that the use of personal metrics can provide significant benefits to the individual’s engineering process. Although there is very little debate engaged over this claim, as several studies easily confirmed it [13, 14, 15], the overheads associated with the collection of these metrics creates a very high barrier to adoption. The second claim of PSP is that these benefits are large enough that a user trained in PSP will continue to use the system long after the training period has ended. However, this does not appear to be the case, as many software practitioners believe the benefits of these metrics do not outweigh the overheads [7, 11, 16].

These overheads associated with data collection, along with some other reoccurring problems like errors in data entries due to human fault [11], could be solved by the automation of PSP. Even Humphrey himself suggested that automation could be a step forward for PSP when he said, “It would be nice to have a tool to automatically gather the PSP data. Because judgment is involved in most personal process data, however, no such tool exists or is likely in the near future” [12]. In the past 20 years, several attempts have been made to automate PSP but, unfortunately, none have found success yet.

Automated Interaction Tracking

Leap, PSP Studio and PSP Dashboard are all attempts at automating the collection and analysis of PSP data. None of the systems achieve full automation, but, instead, they reduce the number of measures the user must input by hand. Where possible, these second-generation systems calculate the values that are required, such as time spent on tasks. Dialogue boxes are presented to the user at predetermined points that prompt the input of PSP data. The overhead is reduced by efficient data entry, and some automation.

An extensive study conducted in 2003 compared the second-generation systems to the original PSP specification *(Figure 1)*. The metrics used in the new systems are focused on measuring software edits, while the original PSP uses simple and flexible metrics. Therefore, the new systems remove a lot of the flexibility from the original, which is one of the major benefits of the PSP approach. Also, software based measurements and scripted dialogue boxes cannot adequately capture unexpected events. Overall, the new systems were found to only have a small effect on increasing the adoption rate, so they did not achieve the main goal of making the benefits of PSP outweigh the perceived overhead [16].



Characteristic	**Generation 1** (manual PSP)	**Generation 2** (Leap, PSP Studio, PSP Dashboard)	**Generation 3** (Hackystat)
Collection overhead	High	Medium	None
Analysis overhead	High	Low	None
Context switching	Yes	Yes	No
Metrics changes	Simple	Software edits	Tool dependent
Adoption barriers	Overhead, Context-switching	Context-switching	Privacy, Sensor availability

Figure 1: Three generations of approaches to metrics for individuals [16].

10 years on, a 2013 review of second-generation systems described the ethical issues that arose with their use, with concern mainly surrounding the tracking of user data. Without anonymization, many workers reported feeling uncomfortable with fine-grain analysis being done of their work. From an engineer's perspective, measurements, such as time spent on tasks, are perceived as somewhat dangerous in the hands of management. However, since the input of sensitive data is mainly reported manually, the engineer is given significant control over the personal information being tracked by the system [7]. Second-generation systems teeter on the edge of acceptable monitoring.

The third-generation of PSP is vastly different from the original PSP and second-generation. Hackystat [11, 16, 18] and PROM (PRO Metrics) [17] are examples of third-generation systems. These systems offer a totally automated approach as they no longer need user supplied data to calculate metrics. To achieve this level of automation, the measurements are based on the live work of a user, rather than manual inputs or software outputs. For this to happen, an unobtrusive monitoring process is combined with engineering tools. The monitoring system then tracks the use of developer tools, and pushes the collected data to a centralised service. This centralised service, or data warehouse, stores the combined data from all monitored tools and services. This storage solution creates a queryable knowledge base, where trends and outlier processes can be identified . Third-generation systems can then monitor the development process at a fine-grained level. The type of data recorded can include second by second editing information, switches between files or contexts, and developer communication within the system.

The extensive real-time monitoring of developer tools provided by third-generation systems has allowed for creation of next generation applications, such as Zorro. Zorro is an automated system that determines if an individual is using test driven development correctly [19]. It is built on top of Hackystat, and monitors the creation of unit tests. The order in which source code and tests are added is constantly examined, and deviations from best practise are recorded. Before third-generation systems, there was no way to check if the testing methodology was used, or if the engineer added tests after the source code was written. In a similar fashion to that of Zorro, any software engineering process with established best practice can be monitored with its own application.

However, third-generation systems have encountered backlash, as they raise the same ethical concerns as second-generation, but at a much higher level. As they are fully automated in terms of data collection and analysis, they have a very low barrier to adoption. That being said, different barriers to adoption have been introduced in the form of perceived social or political obtrusiveness, despite the unobrusiveness of the interface itself. Many engineers have reacted suspiciously to third-generation systems, like Hackystat and PROM, due to the potential misuse of the data they collect. This data is so detailed, that privacy violations are a major concern. Collectively, developers have said they do not trust the organisations they work for, or third-party organisations, to store this type of information [1].

The main technical differences between Hackystat and PROM are the tools and languages that each support. Hackystat only supports Java development, whereas PROM supports a host of languages. PROM offers Manager Views, which is a major design difference from Hackystat, which only supports views of the data from individual engineers. The introduction of Manager Views to PROM confirmed the suspicions of engineers, who were fearful organisations could use this data to measure their individual performance.

Comparison of Traditional Measurement Approaches

A 2013 study compared the traditional measurement approaches of software engineering, in terms of adoption and supported analysis. *Figure 2* shows the adoption barriers and automation levels for the four types of software engineering measurement systems used [7]. The breadth of analysis possible using each system is represented by circle size in the graph.

Direct measurement and third-generation systems both have high automation levels. This reduces data collection overhead, and lowers the barrier to adoption. However, even though a higher level of analysis is possible using third-generation systems, their adoption is hampered by aforementioned privacy and ethical issues. Direct measurement systems, such as Sonar, offer less analytical insight, but are used extensively. Visibly, PSP type systems offer the richest form of analysis due to how flexible they are. In theory, they can capture all information, however, their lack of automation completely hampers adoption, with some estimates of adoption rates as low as zero [11] . Lastly, the fourth approach shown in *Figure 2* is the Agile development process. The adoption rate of this process is average, however engineers who subscribe to it are often passionate in it’s defence and promotion.

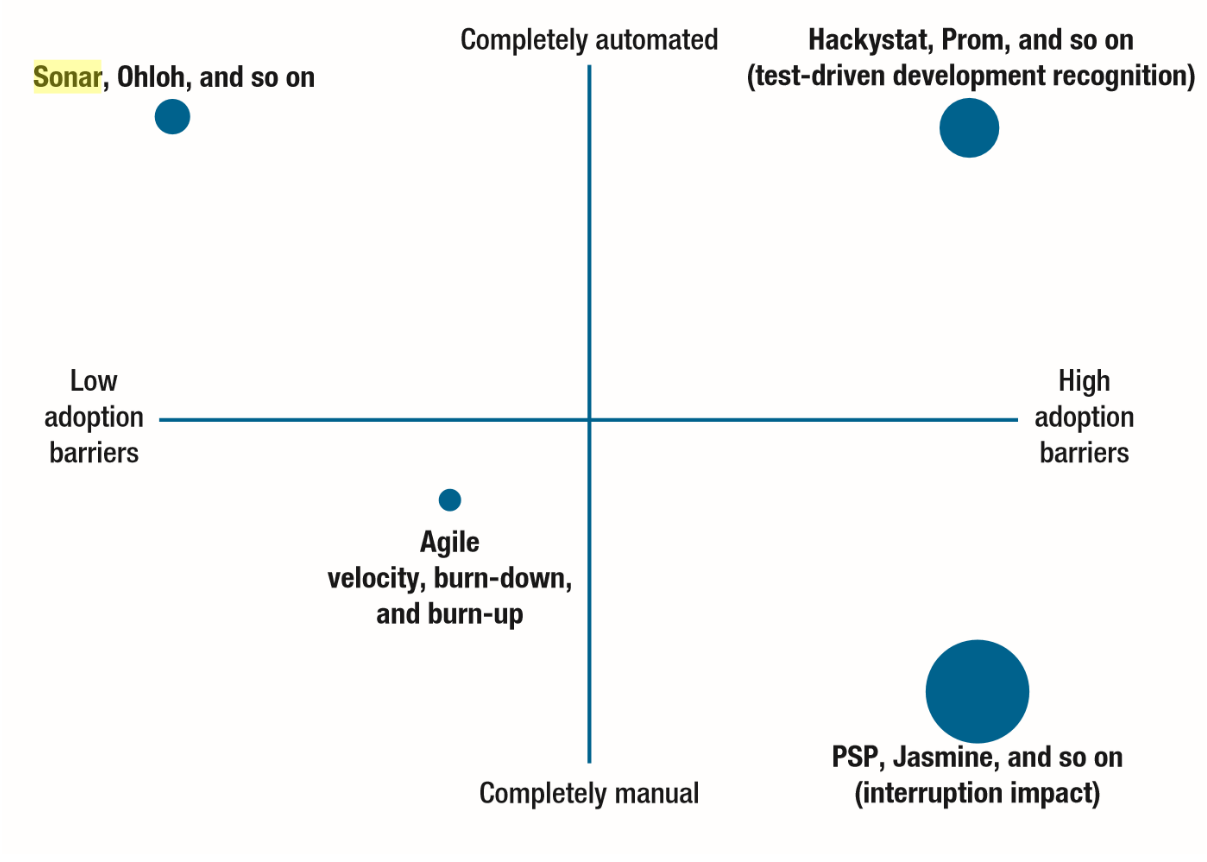


Figure 2: A classification for software analytics approaches, including automation, adoption barriers, and the breadth of possible analytics the approach supports. In the parentheses are analytics that would be difficult [7].

Agile

Agile software development is a simple, yet a powerful process. It is more often applied to teams than to individuals. The agile process flow requires measurements and estimations at various stages. It is characterised by short iterations of the process, and rapid response to issues and defects [20]. Agile development introduces some new measures and terminology to the field of software engineering.

Effort Estimation and Software Size are widely used measures in agile processes. They are most commonly performed using subjective estimation, or an estimate from the team based on previous experience and iterations. Using a group estimate is often referred to as Planning Poker. Another measure used in Agile workflows is Velocity (Productivity). It is a measure of the overall productivity of the team, and needs to be monitored over several iterations for accurate estimation. Burndown Chart is also an important measurement tool used to measure progress. It most often shows the amount of work left in the current iteration. In addition, a second Burndown Chart is often used to show the amount of work remaining across iterations until the release version of the system. Cumulative Flow Diagrams (CFD) are used to track the previous amounts of work carried out over many iterations. They're useful for estimation of future velocity and effort. The Re-Work Metric is used to record all changes to the initial plan during an iteration. This fits with the Agile principles that embrace change. Earned Business Value is a useful measurement tool that tells you how much of the final product is complete, or how much value the current version of the system provides to a user. Finally, Total Effort is an estimate that is required to quantify the total number of iterations and the resource needs of each iteration.

Estimating and calculating these Agile metrics is mostly manual work, but, unlike PSP, rapid iterations and relatively simple measures mean that the automation barrier is not considered a significant a limiting factor to adoption. Furthermore, ethical concerns do not arise as the data collected is not as detailed as second/third-generation systems, and does not refer to individual contributions, giving more privacy to the engineer. The main factor limiting this methods adoption is the breadth of possible analysis, which is much lower when compared to other available techniques.

Data Mining

Increased adoption of version control, collaborative repositories, and open source libraries have, in combination, facilitated new styles of analysis in SE processes. This widespread adoption increased the perceived value of properly documented commit messages, pull requests, and bug reports. Creating this type of documentation has now become a standard part of the SE process. The volume of documentation that is now available, combined with modern analytic techniques, creates opportunities for examining SE processes in a new and exciting way, known as Data Mining. The main Data Mining techniques employed in process analysis are learned classifiers, clusters mined from the data, decision trees, association rules, and mined graph structures [22]. The use of these techniques on the SE process is sometimes referred to as Software Intelligence (SI).

A common method of Data Mining is to view project repositories through the lens of graph theory. Software dependency connections and software project structure can be represented graphically using the large volume of public repositories available today. Using graph based mining methods, insight into SE processes can be derived from these repository graphs. In a study conducted on Github repositories, a large collection of historical data was mined to investigate process metrics and outcomes related to continuous integration (CI) testing. The outcomes of this experiment found that CI increases productivity with little negative effect, "Integration improves the productivity of project teams, who can integrate more outside contributions, without an observable diminishment in code quality" [23].

The Data Mining techniques that fall under the umbrella of Natural Language Processing (NLP) have applications in SE process analysis. Using NLP on unstructured developer data, such as emails, bug reports, and commit messages, has been shown to enhance SI [22]. Text matching, topic modelling, named entity recognition, and a host of other techniques can be used to extract structured information from unstructured text. There are many types of analysis that this new structured information unlocks, for example, mining developer communications to create frequently asked question and answer lists. NLP for SI is an active area of research with great potential for creating insight into SE processes.

But again, there are some ethical concerns surrounding the mining of data. Many developers believe that organisations do not have the right to mine internal developer conversations and emails [22].

Social Analysis

Analysing the social aspect of SE provides a unique insight into SE processes. Social systems are often modelled as networks, which can be traversed to calculate various types of social measures. The possibilities and power of network analysis in a social setting has a large role to play in measurement and assessment of software processes.

The number of academic studies in the social SE field are on the increase, with the majority of these giving rise to useful applications or metrics. A 2013 report discusses a social analysis system that identifies technology experts. This system uses repository information and developer connections to find said experts [24]. This could be used as a measure to aid effective communication and issue resolution within an organisation. Two years later, a 2015 study focused on individual productivity when compared to peers, and found that network effects have a large influence. Being connected to productive individuals increases productivity, even when other factors are controlled [25]. This type of insight, when combined with productivity measurement, could aid in selection of team members to increase productivity across an organisation. Lastly, an analysis of pull request information from Github found that acceptance rates are heavily affected by social factors. The prevailing assumption in this area is that code quality is the strongest factor that affects acceptance, eclipsing all other factors. This study refutes that opinion, claiming that the public availability of user information creates a social status around successful project contributions and ownership. In addition to social status, the previous relationship between the owner and requester was a strong signal of acceptance [26].

Gamification

Gamification is the application of game design techniques, such as a reward system or leader boards, to any applicable system. The Gamification design pattern is growing in popularity, and has already been applied to SE processes by the system Gamiware, “A gamification platform for software process improvement“ [30]. This system was designed to increase motivation in software engineering projects, using gaming initiatives that don’t create significant process overhead, and it found remarkable success.

Adding a social gaming elements to version control and public repositories helped drive users to these systems. Elements such as GitHub’s public visualisation of user commits is an obvious use of gamification to motivate engagement. [27] In turn, other techniques that use these repositories for data are then boosted by the increased volume , due to gamifaction.

One of the main uses of gamification is to encourage users to do tasks which are perceived as undesirable. An example of this is getting engineers to adhere to best practices. A gamification system for test driven development exists where virtual rewards are used to motivate engineers to adhere to the Test Driven process. The study was a success and shows the power of these techniques to encourage the adoption of SE processes . [28]

Gamification in SE is a relatively new field, a study of its application found that basic game elements are showing promising results. The study states that new fields often use the simplest available techniques for initial research direction. However there is scope for major developments in SE process by adopting the complex and mature gamification measures [29]

In terms of ethical issues gamification, like social analysis, is often looked at negatively. Gamification is sometimes seen as manipulation and its use in the work place is controversial. The addictive elements that many techniques use can have negative effects when not used with care. Traditional methods of increasing adoption and reducing churn usually involve adding additional value for the user, while gamification employs psychological tricks to encourage engagement. This lack of added value causes gamification of developer processes to be sometimes viewed by engineers as a waste of resources.

References

[1] Putnam, L.H., 1978. A general empirical solution to the macro software sizing and estimating problem. IEEE Transactions on Software Engineering 4 (4), 345±361.

[2] Boehm, B.W., 1981. Software Engineering Economics. Prentice-Hall, New York.

[3] Albrecht, A.J., 1979. Measuring Application Development. Proceedings of IBM Applications Development joint SHARE/GUIDE symposium. Monterey CA, pp. 83±92.

[4] McCabe, T., 1976. A software complexity measure. IEEE Transactions on Software Engineering 2 (4), 308±320.

[5] Tufano, M.; Palomba, F.; Bavota, G.; Oliveto, R.; Di Penta, M.; De Lucia, A.; Poshyvanyk, D. (2015-05-01). "When and Why Your Code Starts to Smell Bad" (PDF). 2015 IEEE/ACM 37th

[6] Halstead, Maurice H. (1977). Elements of Software Science. Amsterdam: Elsevier North-Holland, Inc. ISBN 0-444-00205-7.

[7] M. Johnson, Philip. (2013). Searching under the Streetlight for Useful Software Analytics. Software, IEEE. 30. 57-63. 10.1109/MS.2013.69.

[8] Glenford J. Myers (2004). The Art of Software Testing, 2nd edition. Wiley. ISBN 0-471-46912-2.

[9] Hagenberg Research – 2009 DOI: 10.1007/978-3-642-02127-5

[10] Basili, V.R., Rombach, H.D., 1988. The TAME project: Towards improvement-oriented software environments. IEEE Transactions on Software Engineering 14 (6), 758±773.

[11] NE Fenton, M Neil "Software metrics: successes, failures and new directions" Journal of Systems and Software 47 (2), 149-157

[12] Watts S. Humphrey A Discipline for Software Engineering, SEI series in software engineering, 19, reprint Addison-Wesley, 1995 ISBN 0201546108, 9780201546101

[13] S. Khajenoori and I. Hirmanpour. An experiential report on the implications of the Personal Software Process for softwarequalityimprovement. InProceedings oftheFifthInternational Conference on Software Quality, pages 303–312, October 1995.

[14] M. Ramsey. Experiences teaching the Personal Software Process in academia and industry. In Proceedings of the 1996 SEPG Conference, 1996.

[15] B. Shostak. Adapting the Personal Software Process to industry. Software Process Newsletter #5, Winter 1996.

[16] P.M. Johnson et al., "Beyond the Personal Software Process: Metrics Collection and Analysis for the Differently Disciplined," Proc. 25th Int'l Conf. Software Eng. (ICSE 03), IEEE CS, 2003, pp. 641–646.

[17] A. Sillitti, A. Janes, G. Succi, and T. Vernazza, “Collecting, integrating and analyzing software metrics and personal software process data,” in Proceedings of the 29th Euromicro Conference. IEEE, 2003, pp. 336– 342.

[18] Johnson, Philip M., et al. "Improving software development management through software project telemetry." IEEE software 22.4 (2005): 76-85.

[19] Johnson, Philip M., and Hongbing Kou. "Automated recognition of test-driven development with Zorro." Agile Conference (AGILE), 2007. IEEE, 2007.

[20] Taghi Javdani , Hazura Zulzalil, Abd. Azim Abd. Ghani, Abubakar Md. Sultan, On the current measurement practices in agile software development, International Journal of Computer Science Issues, 2012, Vol. 9, Issue 4, No. 3, pp. 127-133.

[21] Hassan, A.E. and T. Xie, Software intelligence: the future of mining software engineering data, in Proceedings of the FSE/SDP workshop on Future of software engineering research2010, ACM: Santa Fe, New Mexico, USA. p. 161-166.

[22] Di Penta, M. (2012) Mining developers' communication to assess software quality: Promises, challenges, perils. In Emerging Trends in Software Metrics (WETSoM), 2012 3rd International Workshop on. IEEE.

[23] Quality and productivity outcomes relating to continuous integration in GitHub

[24] Dittrich, Andrew, Mehmet Hadi Gunes, and Sergiu Dascalu. (2013) "Network Analysis of Software Repositories: Identifying Subject Matter Experts." Complex Networks. Springer Berlin Heidelberg. pp. 187-198.

[25] “Network Effects on Worker Productivity” 2015

[26] “Influence of social and technical factors for evaluating contribution in GitHub”

[27] L. Singer and K. Schneider, "It was a bit of a race: Gamification of version control," Games and Software Engineering (GAS), 2012 2nd International Workshop on, Zurich, 2012, pp. 5-8.

[28] W. Snipes, V. Augustine, A. R. Nair and E. Murphy-Hill, "Towards recognizing and rewarding efficient developer work patterns," 2013 35th International Conference on Software Engineering (ICSE), San Francisco, CA, 2013, pp. 1277-1280.

[29] “Gamification in software engineering – A systematic mapping”

[30] “Gamiware: A gamification platform for software process improvement