Trinity College Dublin
Coláiste na Tríonóide, Baile Átha Cliath
The University of Dublin

CS3012 Software Engineering
Assignment #4:Measuring Software Engineering

Jevgenijus Cistiakovas,

November 7, 2019

Contents

1 Introduction

Software metrics are considered a vital part of software engineering.[6] Given that software industry is still full of productivity and quality issues, software measurements are seen as necessary means

that can aid in analysing and improving software processes. In particular, software measurements provide support for planning, monitoring, controlling and evaluating the software process.[2]

A software metric is defined as method of quantitatively determining the extent to which software process, product, or project possesses a certain attribute. This does not only include the formula used for determining a metric value, but also the methods for presenting and interpreting it.[3] In particular, commercial tools put a lot of emphasis on presenting rather simple metrics in a sophisticated way that allow the customers to better interpret the results.

Software measurement is an active area of research with books and articles being published. However, the methods developed by researchers and methods actually used within software industry can widely differ. It is important to consider the both the academic methods as well as commercial solutions available.

Agile development methods are becoming the new norm in software industry.[17][14] It is therefore important to consider the metrics and methods that measure Agile Software Development process.

This report talks about the different ways software process can be measured including the algorithmic approaches available in Measurable Data(2). It examines some popular solutions available for gathering and analysing data in software development in Tools Available(3).Finally, the ethical side of collecting and using metrics to help with decision making is discussed in Ethics(4). As a whole this report presents the independent research done as a part of CS3012 Software Engineering module at Trinity College Dublin.

2 Measurable Data

Software measurement can be said to consist of measurements of products, processes and resources, with developers being part of the resources of an organisation.[4] Examples of external(higher level) attributes would be quality, complexity and maintainability for product; quality, cost-effectiveness and stability for process; productivity, reliability for resources. All those higher level metrics are built upon lower level metrics such as maintainability would built on modularity of code base.

This section considers different attributes that can be measured, methods that can be used to measure them and the usage of this metric to define useful and perhaps more higher level metrics.

2.1 Size

One of the most obvious and simplest metrics as well as one of the most popular property to measure is software size. Measuring the size of software is straightforward if a relatively simple measure is used. It should be noted however that size can only correctly indicate how much of an entity is present, it cannot directly tell how much effort and cost was or need to be put in, or how productive really the developers were. Size is nevertheless still a useful metric as it can be used to predict attributes such as development time and resources, it is also commonly used as a component for computing indirect metrics such as for measuring bug density. Overall, size is a useful metric that can be used as a base for deriving other attributes. Size is commonly used for normalization as it is commonly viewed that many other metrics scale in a predictable manner as the size of the program grows. It is not unreasonable to suggest that bigger program might take longer time to write and might have more bugs in it than program of a smaller size. At the same time developers and managers need to be careful when basing their assumptions as relations between attributes are not always as simple as it might seem to be. The subsections below consider mainly the methods of measuring size of a product rather than a process as every products inherently has a size than can be measured.

2.1.1 Code Size - LOC

Back in the early days of computing, the size of a computer program used to be determined by the number of punched cards used to record the program.[16] Now that the programs are stored on hard disks, an equivalent measure could be the number of bytes needed to store the program, but perhaps a better choice would be lines of code(LOC) metric. Being probably one of the oldest and most debatable metrics, it is still used due to its dangerous simplicity and ease of calculation. It has been known for a long time now that measuring LOC comes with a lot of questions and issues.[16] Firstly, it is obvious that not all of the lines of code are the same. It is common for among programmers to use blank lines to make code more readable in their opinion. Blank lines are of little interest to those interested in real size of the program. Similarly, it is considered a good practice to generously document the code. Unlike blank lines, comment lines require at least some effort to write, but perhaps not as much effort as the actual code. A common solution is either discard comments at all or to give code and comment lines different weight when calculating LOC. Even if agreement is achieved on those basic issues, there come another issues related to different programming languages having different verbosity as well as different requirements for separating code into lines. For example, what in "pretty written" Java would take 20 lines of code with multiple for loops, can be often expressed as a single list comprehension in Python. What LOC tells in this case is that code in Java probably took more keystrokes to type. This cannot tell which code was easier to write for developer. The metric is also highly dependent on a style used by the developers. For example, Google's style guide for C++ requires all lines to be shorter than 80 character for their internal reasons.[8] It would be perhaps unreasonable to set such a requirement for a new style guide. As a result, this leads to inability to compare the same metric between different companies and projects.

The choice of how to count LOC should always be based on how the metric is to be used. It is sometimes desirable to count comment lines and code lines separately to then find a comment density and try to correlate it with the quality of code written. Other times, it is preferred to measure only the number of executable statements in the final code. While it misses all the helper code written during development such as prototypes and internal tools, it would be a good size metric for a final product. It is also common to differentiate between the new code developed, code that was reused such as for example standard libraries, code that was partially reused.

An alternative to LOC as a code size metric is token count that was defined by Maurice Halstead. He defines a length of a program to be the sum of total number of occurrences of operators plus total occurrences of operands, where for example operands are variables and operators are arithmetic symbols or command names such as WHILE. Using the count of operators and operands, Halstead also other derived metrics such as program level, programming effort, program difficulty, etc.

Code size and LOC is a perfect example of how carefully metrics and methods to be used need to be studied as even such a seemingly simple metric as code size can have many aspects to it.

2.1.2 Design size

Before any software is written, it is necessary being designed. Designing software is a long and complex process that takes a lot of effort. In the standard waterfall model of development process, a design is fully developed before any software is written. Thus design size together with design complexity can serve as an indicator for approximating the amount of effort required for development of the product. Measuring design or requirements size usually comes from the methodology used for design. If object oriented approach and UML diagrams are used then size can derived from the number of classes, objects, interfaces and associations present as well as the number of different design patterns used. During use case design and analysis, the number of use cases can be used

as a size measure. For Agile, the number of user user stories can be used. All these methods of measuring size can be of high value within a single project, but due to different methodologies and styles used, it becomes problematic to compare design size of different projects between different organizations.

2.1.3 Functionality Size

Many software engineers argue that the amount of business functionality in a product best expresses the size of the product. Back in 1979 Allan Albrecht at IBM defined function points (FPs) as a unit of measure of amount of functionality. This approach measures the functionality of specification documents, but can also be applied at a later stage of product's size to refine the size estimate and thus the cost or productivity estimate. The advantage of FPs is that they are generally independent of the specification model or technique used. The number of of FPs is defined as a product of unadjusted function point count(UFC) and Technical complexity factor (TCF):

$$FP = UFC \times TCF$$

Where UFC is computed as a weighted sum of items of different variety. The items can of the following types[11]:

1. External input types: data or control input items provided by the user.
2. External output types: output data types to the user.
3. External inquiry types: interactive inputs requiring a response.
4. External file types: files that are passed or shared between the system and other systems.
5. Internal file types: files that are used and shared inside the system.

Each of there items is assigned a subjective "complexity": simple, average, or complex. Then, a weight is assigned based on both type and complexity level varying from 3 (for simple external input) to 15 (for complex external file). UFC is then:

$$UFC = \sum_{i=1}^{15} (\text{Number of items of variety i}) \times (weight_i)$$

TFC is a sum of 14 differently weighted factors such as whether the product needs to accommodate for performance or reusability, etc.

The advantage of FP measurement is that it can be obtained based on the system requirements in the early stage of software development. FPs is a well understood and studied method. The notion of FPs for object-oriented software is extended by introducing object points that can be computed directly from class diagram.

FPs and UFC can also be used to estimate the code size.

It has been claimed that function point analysis is hard to do properly and is unnecessarily complex is used for resource estimation.[5] Despite having many limitations, it has been used in practice for many years, in particular it has been used to report progress and define payment in contracts as well as for estimating the cost of development and complexity of the product.

2.2 Measuring Structure

While size is an important internal attribute of software product, there are other useful internal attributes. One of such attribute is structure. Structure in part can be separated into control flow structure such as the order in which instructions are executed, and data flow structure such as how data is handled by the program.[4] Briand et al.[1] defines properties of such structural attributes as length, complexity, cohesion and coupling. Attributes such as complexity are of high interest to engineers and managers as potentially cost and effort estimates can be derived from complexity. Properties such as cohesion which assesses the tightness with which related program features are "grouped together" in systems or modules[1], and coupling which captures the amount of relationship between the elements belonging to different modules[1] can be potentially used to define such higher level attributes as maintainability and testability of product.

It is common to view the structure of a software product as a directed graph resembling the DFA nature of software and computers. Looking at a program in terms of its control flow graph allows to more formally analyse and derive more formal ways to measure it. In particular, McCabe proposed the cyclomatic number of a program's flowgraph as a measure of program complexity. The McCabe's cyclomatic number measure the number of linearly independent paths through the program. The cyclomatic number is defined as follows:

$$\text{Cyclomatic complexity of flowgraph F is } v(F) = e - n + 2$$

with e being the number edges, n being number of nodes. The claim is that the higher the number the more complex is the code. Cyclomatic number definition is also defined for workgraph sequencing and workgraph nesting to allow for bottom calculation for complex control graphs. Unlike with LOC measure, with cyclomatic number it is not obvious why the claim holds true. It does however follow all the properties of complexity measure as described by Briand et al.[1] One important property is that the complexity of a system is not less than the sum of the complexities of any of its two independent sub parts which matches to some extent the intuitive model of complexity.

Applications of McCabe's cyclomatic complexity include limiting complexity during development by setting a limit to the allowed cyclomatic number. It can also be used to estimate the required effort for writing tests. The limitation of cyclomatic complexity is that the original paper is vague on some details of the metric and hence different tools might report different cyclomatic number for the same code.

2.3 Measuring External Attributes

The principal goal of software engineering is to develop high quality software effectively. The most important metric for a software product is thus quality.

Boehm et al. and McCall et al. described quality as being composed of many different components. Examples of some important quality factors would be reliability, maintainability, usability, testability and efficiency. Those quality factors are in turn composed of different criteria that can be measured. Some of these factors can be calculated using the static metrics described in preceding sections. For example, testability can be derived from a cyclomatic number. Maintainability can be derived from a modularity static metric. Maintainability on the other hand is related to the process of maintaining the product, hence it is best calculated from the measures of the process such as effectiveness of the process of maintaining a product.

Decomposing quality metric into many components and then trying to correctly calculate metric for each component requires a lot of time and effort. Often rough and approximate measures are sufficient. One measurement that is universally used is number of defects, where defect means a

known error, fault, or failure. Quality of the product can then be obtained from defect density which is defined as follows

$$\text{Defect density} = \frac{\text{Number of known defects}}{\text{Product size}}$$

Defect density is defined in terms of size. Therefore all the issues and limitations related to measuring product size as describes previously in Measurable Data section(2) are also transferred to defect density. Similarly to how there are different size in code, there exist different types of defects. It is necessary to differentiate between pre-release faults, detecting of which indicates good testability and good testing process as well as perhaps lack of quality from the developers, ad between post-release faults. With post-release fault density it is also important to differentiate between defect found by customers and defects found internally. Any commercial company would want to minimize the fault density related to customer. Lastly, defects can have different effect - not all faults lead to software failures. Some minor defect might never be noticed by the users.

2.4 Measuring Process - Agile Approach

The measures described above mainly concerned with measuring attributes of a product. It is often also desirable to measure the state of development process that can directly affect the cost and time needed for development and maintenance of a product. This section will look at how and what process can be measured. In particular, it will focus on agile development as it is currently dominating development process. However, it was reported that the use of metrics in Agile software development is similar to Traditional software development.[10]

The most popular factor agile teams are interested in measuring are velocity, effort estimation accuracy, testing performance and code quality.

Velocity measures the amount of work done with respect to time. Issues' velocity measures capability of a team to complete issues planned for a sprint. One metric that can be used is number of issues completed during the sprint. Knowledge of velocity helps to assess and improve planning, to identify bottlenecks.[13] The change of velocity over time is also important. New teams can expect to see increase in their velocity as team members develop relationships and become accustomed to work process.

In order to get more insight into the reasons behind velocity, it it worthwhile to look into development speed. A significant factor that affect development speed is time it takes for commit to be reviewed. An agile team should strive to minimize time it takes for commit to be integrated as often a developer can be blocked waiting for that commit to be reviewed. If it takes long to review a commit, it is a significant sign for a manager as it might mean anything from team being too busy to knowledge not being properly shared within a team.

Testing is an integral part of agile development process. As mentioned previously, testing efforts can be estimated statically. Agile teams are also interested in measuring testing process. For example, testing performance can be measured by means of such metrics as unit test duration, average time to fix an error, average number of iterations in code review phase.

It is important to note that measures concerned with process usually concern with time as unlike a product, a process is a continuous process in which time determines cost.

2.5 Measuring developers

The engine behind software development is developers. All the attributes defining each individual developer such as quality and productivity in the optimistic scenario contribute to the overall quality and efficiency of process and product.

The choice of metric to measure depends on the goal the measurement is trying to achieve. One of the reasons to measure software engineers is economical - that is the goal is to identify the strong and weak performers. In a commercial world developer needs to necessary bring value to the company. If a developer is determined to not perform to the set standards, then the company needs to look into it. Often the outcome is that a developer gets sacked from the job. Knowing the strong performers within the team is also of great use to the company, as it might decide to promote the talents to higher position. A large company with a big number of teams could benefit from ensuring that all the teams are of similar performance level. High performers can be asked to join weaker teams to bring them to the average standard. The work of the manager is of course not just finding low performers and asking them to leave the company, the key to the work of manager is to ensure that the team is performing to the full potential. In this case, a trend is of more use that an atomic metric - for example the change in productivity and quality of a developer. Once a manager detects a worrying trend, he/she can then look talk to the developer and look into lower level metrics in an attempt to identify the cause of the change. For example, a drop in quality might indicate that a developer is under the pressure of deadline or is stressed.

While metrics are of the greatest use for managers of the teams, there are also helpful for regular developers. One use case is for example that developer can use metrics as records of his/her work .

The reasons described are not unique to software development. There are mostly universal across the market. The details on how the software engineers can be measured, of what constitutes to a happy, healthy and well performing developers is what is unique to this field.

There are many low level metrics that can be measured about an individual developer. Essentially, every action performed by a developer on a computer can be recorded and used to define a metric. For example, this includes such metrics as lines of code written/modified by the engineer as well as number of emails sent to colleagues or the complexity change caused by the engineer's change in the codebase. These lower level metrics are used to measure higher level attributes such as overall quality of work done or productivity.

Productivity is normally defined, from an economic view, as the effectiveness of productive effort.[12] Productivity can be measured as a single ratio of metrics. For example, as a ratio of *Work Size/Effort* where work size can be measured in lines of code or Function points, and effort can be measured in terms of time and cost. This productivity metric essentially measures the productivity of a developer in developing the product. The productivity metric suffers from the same issues as each component within the ratio. It relies upon the ability of work size metric to correctly capture the amount of work done by the engineer taking into account factors such as complexity of the work done, importance of the work done, impact of the work done, quality of the work done.

The metrics considered previously were measuring the actions done by a developer. However, a developer is a human and its biological parameters can be measured. As argued by Marieke van Vugt et al. 2019 in [18], biometric sensors can potentially be used to measure productivity. As it has been noted, measuring productivity correctly is hard. In particular, productivity in the field of software development requires sometimes singular focus, and sometimes distraction.[18]. Vugt provides examples of some of the biometric measures that can be used. These include measuring pupil size, hearth-rate variability, and EEG. All these metrics provide some information on the person's attention state. However, with software engineering requiring more than mechanical concentration on a single thing, these metrics cannot be used a single source of productivity measure. Biometric sensors can nevertheless be useful for the developer as they aid in identifying distractions and low productivity periods. Knowing when a developer is the least concentrated, can help the colleagues to decide whether to interrupt his/her or not. Developers should strive to reduce the amount or interruption and distraction during a high concentration period as it has direct consequences on the productivity.

While measuring each individual software engineer might be of a good help to manager, for example in identifying when a particular person needs help, or as a proof of high/low job performance by an employee, the software is normally developed in team, and productivity of a team is of greater importance that productivity of a single team member. For example, the productivity of a senior developer might drop as he/she is dealing with teaching skills or knowledge to junior developers to help junior developer to be independently productive. The overall productivity of the team will probably increase in the long-term.[9]

Overall, there are many ways and methods of how individual software engineers can be measured ranging from LOC to pupil size. The challenge is in interpreting the results correctly and appreciating the fact that no metric is fully correct.

3 Tools Available

Developing solution to gather and analyse metrics takes developers time and thus introduces overhead. Therefore, there appeared companies that specialise in developing and providing generic solutions to measuring software process. Such measurement programs and services are an important source of control over cost and quality of development process within software organisations.[6] Therefore, it is important to consider the popular public products available as they in part determine the general direction of the practical measurement in software industry.

3.1 Git Analytics Tools

Many tools available on the market can be described as Git analytics tools. They also call themselves as Engineering Intelligence tools. These tools such as GitPrime or Velocity by Code Climate work by monitoring and analysing git repositories. All the platforms include such natural git metrics as activity counts and averages. For example, the number of pushes per day for each engineer/team/repository. The main way the platforms try to attract customers is by providing more advanced metrics such as measuring individual's impact, productivity, rework as well as providing ways of visualizing this data. Such advanced metrics are usually computed by proprietary algorithms.

3.1.1 GitPrime

As they describe themselves, GitPrime is an organizational tool, pioneering a different way of measuring and communicating about productivity in software engineering. GitPrime has "Git" in its name for a reason. The platform operates by watching customer's git repository and retrieving useful metrics that are then presented to the customer in a more visual manner. The platform initially worked of only data from git, but now it works with many services providing git repositories such as GitHub or BitBucket.

Compared to Velocity, GirPrime puts more focus on individual engineers. For example, it allows to measure efficiency of engineer submitting code using such aspects as responsiveness, number of comments addresses, receptiveness and number of preview pull requests. Similarly, it measures engineers reviewing code using such metrics as time it takes reviewer to first comment on a new pull request, or influence - whether comments get addressed by the submitter.

Overall, the key analytics provided by GirPrime are:

- Impact - Impact of a change is based on the relative difficulty of the change as determined by their algorithm.

- Churn - Churn measures the amount of rework. A lot of rework can be a sign that a developer is stuck. It also represents a potentially wasted effort.
- tt100 Productive - It is time in hours required to write a hundred lines of code after churn. GitPrime claims that over time this metric is a good indicator of productivity.

3.1.2 Velocity by Code Climate

The tool is focused mainly of pull requests (PR). Therefore their tool works of both source-code level data as well as collaborative work data. Velocity provides different metrics about pull requests. For example, its review cycle metric calculates the number of times a PR goes back and forth between the reviewer and the contributor. They claim this can help in identifying bottleneck PRs. Velocity provides detailed metrics about both: individual engineers and team as well as about each individual pull request. Velocity puts more emphasis on collaboration and provides a lot of metrics related to code review process and pull requests' related details.

3.2 Estimancy

Estimancy claims to be "The first AI-based software solution for Enterprise Application Outsourcing Management and Software Spend Estimation". Estimancy provide a service of estimating the software size to develop based on requirement specifications in natural language. Estimancy provides a Microsoft Word extension that can process a specification written in natural language and automatically detect units of work, and after user's approval estimates can be calculated.

3.3 Polyspace by Mathworks

Polyspace is a family of product by famous company MathWorks that is a developer of Matlab. Polyspace is a static code analysis tool that uses formal methods to prove the absence of critical run-time error in source code for C, C++, and Ada programming languages. Polyspace family consist of two products: Code Prover and Bug Finder. Polyspace products allow the user to generate different software metrics such as

- Comment density of a source file
- Cyclomatic complexity
- Number of lines, parameters, call levels, etc. in a function
- Identified run-time errors in the software

3.4 Codacy

Codacy is an automated code analysis/quality tool. In other words, Codacy is a hosted automated code review service. Codacy automatically applies static analysis code patterns to a project and grades it so a customer can take a first glance of its health. Codacy reports back the impact in code quality for every commit or pull request: new issues introduced, code coverage, code duplication or code complexity. It also provides suggestions on how code can be improved. Codacy supports Git as the only source mode management technology. It however support many programming languages including Scala, Java, PHP, Python, etc. An interesting feature of Cadacity is that it also looks for security issues in the code. Security is an important attribute defining quality of a product.

4 Ethics

Based on a large number of publications in the area of software measurement and also based on the informations presented in this report, it can be said that measuring software is indeed a challenging problem. The complexity of the issue adds to an opinion that some aspects of measurements in software engineering are unethical.

Metrics by themselves do not usually have any ethical issues, unless they reveal very personal information. The main ethical questions are raised when considering the methods used to collect data, the nature of the data collected and the way those metrics are later used. Collection of data and nature of data raises ethical questions such as what data and to what extent can an employer collect data about its employees, and how transparent does the company needs to be about the data it is collecting. While collecting the LOC of commits submitted poses no ethical issues, collection of biometric data such as person's posture or hearth rate is more controversial.

Software engineers do not like to be measured, so the solution might be to not disclose the fact they are being measured to them. This is not a viable solution and will only lead to loss of trust. Trust is very important when talking about measurement of developers. Often software engineers do appreciate the value metrics such history as periods of high concentration or methodologies such as personal software process can bring. The problem comes with this information being used by the others that the person do not trust. While the goal of measurement is to quantify the interested attribute to allow for objective decision making, there is still an issue of whether the metric captures all the details to allow for objective presentation. It is argued that metrics should be used only for triggering and initiating a more qualitative research into the issue as certain details cannot be correctly quantified.[15]

When considering third party services used for measurement, the issue of trust becomes more acute. It is more than trust between two people, it is now trust between two organisations. The company has to rely on the service provider being honest and implementing good security solution to the data processed and using good algorithms with good correlation as algorithms are often proprietary.

The view that some aspects cannot be correctly quantified into metrics raises a question of whether decision making based on metrics is ethical. On one hand, we should strive to make the process of decision making to be objective and correct. On the other hand, all known methods for measuring such attributes as productivity have their limitations. This means that metrics do not always correctly reflect the reality and thus decision might be incorrect. As well as that all the methods are developed by humans and thus are full biases in addition bugs and errors. This especially applies to automated measurement tools such as those described in Tools Available section (3). As the tools and methods that use computational intelligence are being developed, it is important to remember that those techniques are build on the idea of computer learning. Therefore, a computer can also learn human biases. This leads to an issue of measuring the quality of the tool indented to measure the other tools/processes/engineers.

Knowledge that decisions are being made based on some specific metrics can also provoke some people to try and take advantage of weaknesses of the algorithm being used. This can lead to an "arms race" between measuring body and the people being measured.

I believe that a solution to many issues is transparency. The employer needs to be transparent about what is measured, how it is measured, how it is further processed and how a final result is being used. Every part of the method used to for example calculate productivity metrics needs to have a clear scientific justification. Every part of the measurement pipeline should be under control of both employees and managers. Every developer should have ability to protest against an aspect of the measurement pipeline. In summary, the measurement pipeline should be a collective asset that should bring a clear value to every participant - both engineers and managers. Otherwise, it

will lead to unhappiness, feeling of being restricted and being under full control, questions about ethics. It is important, as software engineering is to some extent a creative job where happiness directly affect productivity.[7]

5 Conclusion

In conclusion, measuring software engineering process is a complex problem. It can however bring an enormous benefit to both the organisation and its employees. Software development process and in particular software developers participating in it generate a lot of measurable data. The key problem is combining the metrics to allow for measurement of such concepts as productivity, quality, complexity. There exist companies that specialise in providing commercial tools for collection and analysis of organisation's data. As the current dominating development process is Agile and the current dominating version-control system is Git, the most popular metrics are extracted from Git and Agile workflows. Despite metrics providing visible value, some aspect of measuring are sometimes viewed as unethical.

References

[1] L. C. Briand, S. Morasca, and V. R. Basili. Property-based software engineering measurement. *IEEE Transactions on Software Engineering*, 22(1):68–86, Jan 1996.

[2] Lionel C. Briand, Sandro Morasca, and Victor R. Basili. An operational process for goal-driven definition of measures. *IEEE Trans. Softw. Eng.*, 28(12):1106–1125, December 2002.

[3] Michael K. Daskalantonakis. A practical view of software measurement and implementation experiences within motorola. *IEEE Trans. Softw. Eng.*, 18(11):998–1010, November 1992.

[4] Norman Fenton and James Bieman. *Software Metrics: A Rigorous and Practical Approach, Third Edition*. CRC Press, Inc., Boca Raton, FL, USA, 3rd edition, 2014.

[5] Norman E. Fenton and Martin Neil. Software metrics: Success, failures and new directions. *J. Syst. Softw.*, 47(2-3):149–157, July 1999.

[6] A. Gopal, M. S. Krishnan, T. Mukhopadhyay, and D. R. Goldenson. Measurement programs in software development: determinants of success. *IEEE Transactions on Software Engineering*, 28(9):863–875, Sep. 2002.

[7] Daniel Graziotin and Fabian Fagerholm. *Happiness and the Productivity of Software Engineers*, pages 109–124. Apress, Berkeley, CA, 2019.

[8] Google Inc. Google c++ style guide.

[9] Andrew J. Ko. *Individual, Team, Organization, and Market: Four Lenses of Productivity*, pages 49–55. Apress, Berkeley, CA, 2019.

[10] Eetu Kupiainen, Mika V. Mäntylä, and Juha Itkonen. Using metrics in agile and lean software development - a systematic literature review of industrial studies. *Inf. Softw. Technol.*, 62(C):143–163, June 2015.

[11] Hareton Leung and Zhang Fan. Software cost estimation. 04 2001.

[12] Edson Oliveira, Davi Viana, Marco Cristo, and Tayana Conte. How have software engineering researchers been measuring software productivity? - a systematic mapping study. pages 76–87, 01 2017.

[13] Prabhat Ram, Pilar Rodríguez, and Markku Oivo. Software process measurement and related challenges in agile software development: A multiple case study. 09 2018.

[14] HP report (2015). Agile is the new normal: Adopting agile project management, May 2015.

[15] Caitlin Sadowski and Thomas Zimmermann. *Rethinking Productivity in Software Engineering*. 01 2019.

[16] V.Y.Shen W.M.Zage S.D.Conte, H.E.Dunsmore. A software metrics survey. *MCC Technical Report STP-284-86*.

[17] Ayca Tarhan and Seda Gunes Yilmaz. Systematic analyses and comparison of development performance and product quality of incremental process and agile process. *Information and Software Technology*, 56(5):477 – 494, 2014. Performance in Software Development.

[18] Marieke van Vugt. *Using Biometric Sensors to Measure Productivity*, pages 159–167. Apress, Berkeley, CA, 2019.