Student Name: Nissimol Aji **Student Number:17321973**

Measuring Software Engineering

Student Name: Nissimol Aji
Student Number:17321973
Module:CS3021

Table of Contents

Student Name: Nissimol Aji **Student Number:17321973**

Introduction

Software Engineers are high in demand. The rate of software Engineers has increased drastically over the decade. With this considerable rise we start investigating different ways in which we can identify an engineer that is productive as supposed to someone that is unproductive. Throughout this report I will discuss different metrics used for measurability along with tools that are available. I will also examine different algorithmic approaches available and give a view of the ethical implications that are involved.

Measurable Data

Measuring a developer's productivity is a tough puzzle to solve. So How Should we measure a developers Productivity?
The fact that there is no objective measure of developer productivity doesn't mean you can't measure it. It just means that you have to measure it subjectively[1].That's why the first and most critical part of understanding a developer's productivity is having a priority order of what tasks are the most important or have the most dependent items.[2]Here I will be analysing the key metrics that can be measurable and the metrics that should not be used to measure developers productivity.

Key Metrics worth mentioning

- **Code coverage:** This indicates the code that is covered during a testcase. The higher the percentage of code tested; it is less likely to have a software bug presented in the code. Some studies have suggested that increasing code coverage above 70-

[1] **https://dev.to/nickhodges/can-developer-productivity-be-measured-1npo**
2 https://www.7pace.com/blog/how-to-measure-developer-productivity

80% is time consuming and therefore leads to a relatively slow bug detection rate.[3]

- **Agile process metrics:** The basic metrics used for agile process are Leadtime, cycle time and velocity.

1) **Leadtime**: This is how long it takes to go from an idea till the software is deployed. Lowering the lead time improves how responsive software developers are to customers. The graph below shows the different aspects involved in developing a software and the time it takes till the deployment stage.

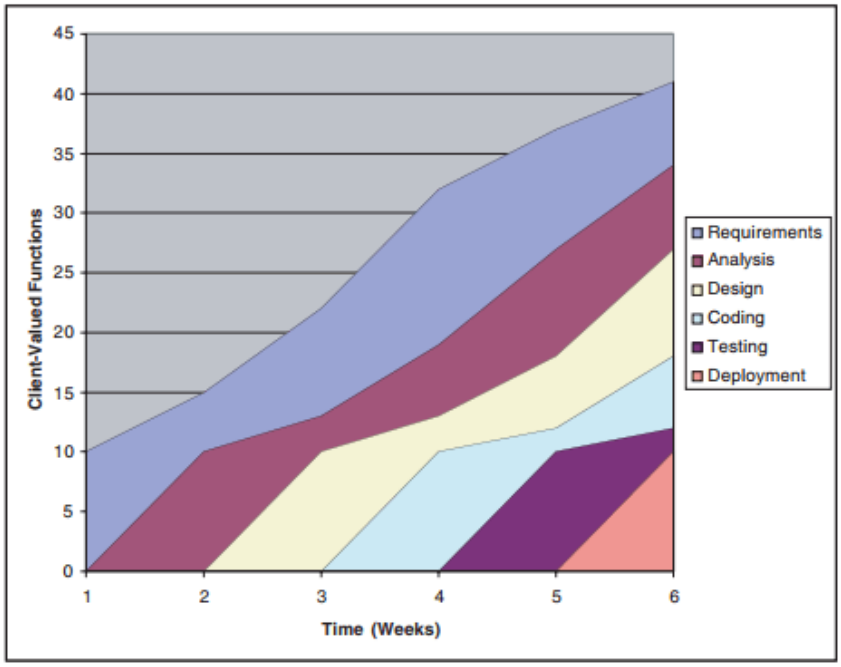


Figure 5–2
Cumulative flow of system inventory.

4

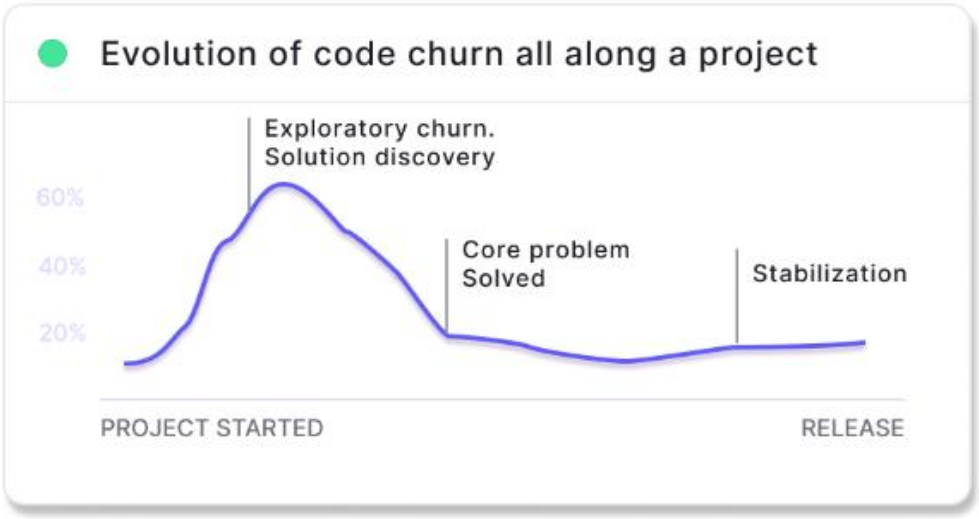
2) **Cycle time:** How long it takes to change a software system and to implement that change.

3) **Velocity:** Measured by setting a sprint and for each sprint certain tasks are assigned. This set of tasks are to

[3] **https://www.linkedin.com/pulse/metrics-help-support-effective-measurement-your-vendors-ben-saunders/**
[4] https://stackify.com/track-software-metrics/

be completed before the next sprint meeting. This meeting usually takes place every two or three weeks. The number of tasks completed by a team is the velocity. Having a high velocity means that the team is performing well.

- **Code churn:** Its measured by taking the code that was modified, added and deleted over short period of time for example for a few weeks. This was then divided by the total number of lines of code added. Code churns will vary between teams, individual, different types of projects and where those projects are in the development cycle. [5]A sudden increase in churn rate may indicate that a developer is experiencing difficulty in solving a particular problem. The graph below shows the progression of the code churn from the beginning of the project.



How NOT to measure developer's productivity

- **Lines of code:** [3] More lines of code doesn't mean good quality. Why solve a problem by writing a code that contains 500 lines when it can be solved by 100 lines. What we should be looking

[5]**https://anaxi.com/blog/2019/07/31/how-to-use-software-productivity-metrics-the-right-way/**

for is code that is readable and maintainable. If no one in the team can understand the code that is written then it's useless.

- **Commits:** By looking at code commits in isolation we can inaccurately measure a developer's productivity. Again, it's necessary to evaluate whether the code committed is of high quality than executing unnecessary commits as a means to illustrate productivity.

- **Hours worked:** The most productive developers accomplish more work and solve more difficult problems in less time. This is a poor way to measure productivity. In a study from Stanford University, researchers found that employees made to work 60 hours per week often accomplish less than employees who only work 40. The findings implied that people who are overworked may even begin to clock negative productivity. This would likely be characterized by an increase in errors or oversights that workers then need to later correct.[6]

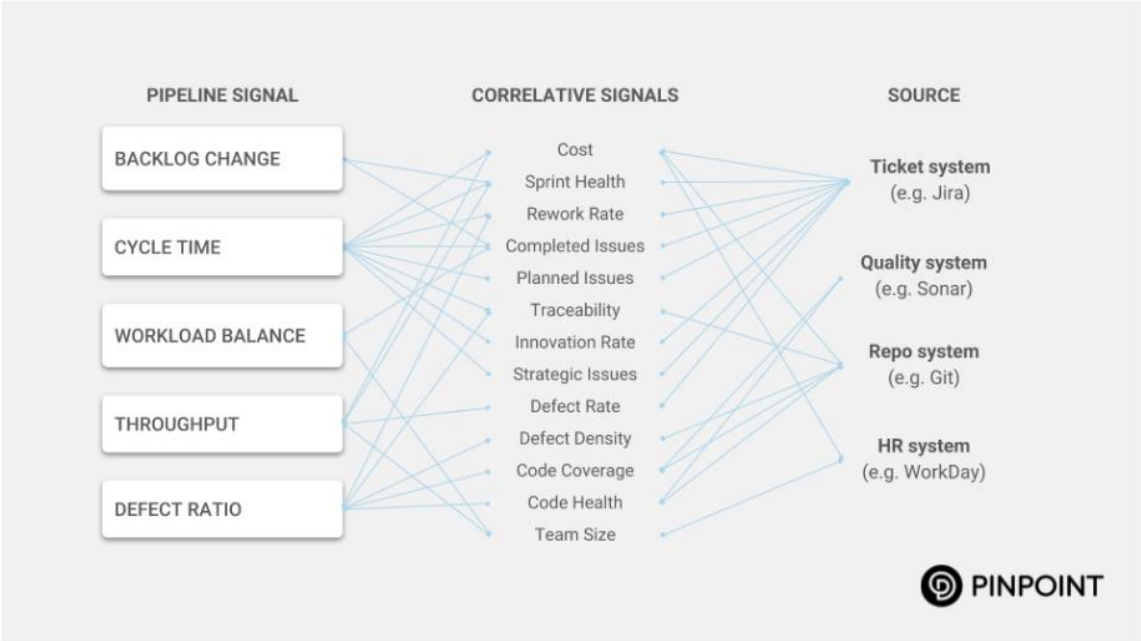
- **Bugs fixed:** A developer could fix one bug that nobody managed to solve and it could take a full week. While another developer could fix twenty detailed bugs in the meantime. This is not an effective way of measurability. Below is an illustration of what happens if bug fixed method was introduced.

[6] **https://www.7pace.com/blog/how-to-measure-developer-productivity**

Student Name: Nissimol Aji **Student Number:17321973**

Tools Available

- **Pinpoint:** This tool looks at which signals would be most reflective of a pipeline performance in a team. Signals also known as metrics. Some of the signals include Backlog change, Cycle time, Workload balance, throughput, defect ratio. Pinpoint uses data science to make correlations to other signals. Here is an example of how pinpoint makes correlation to other signals.



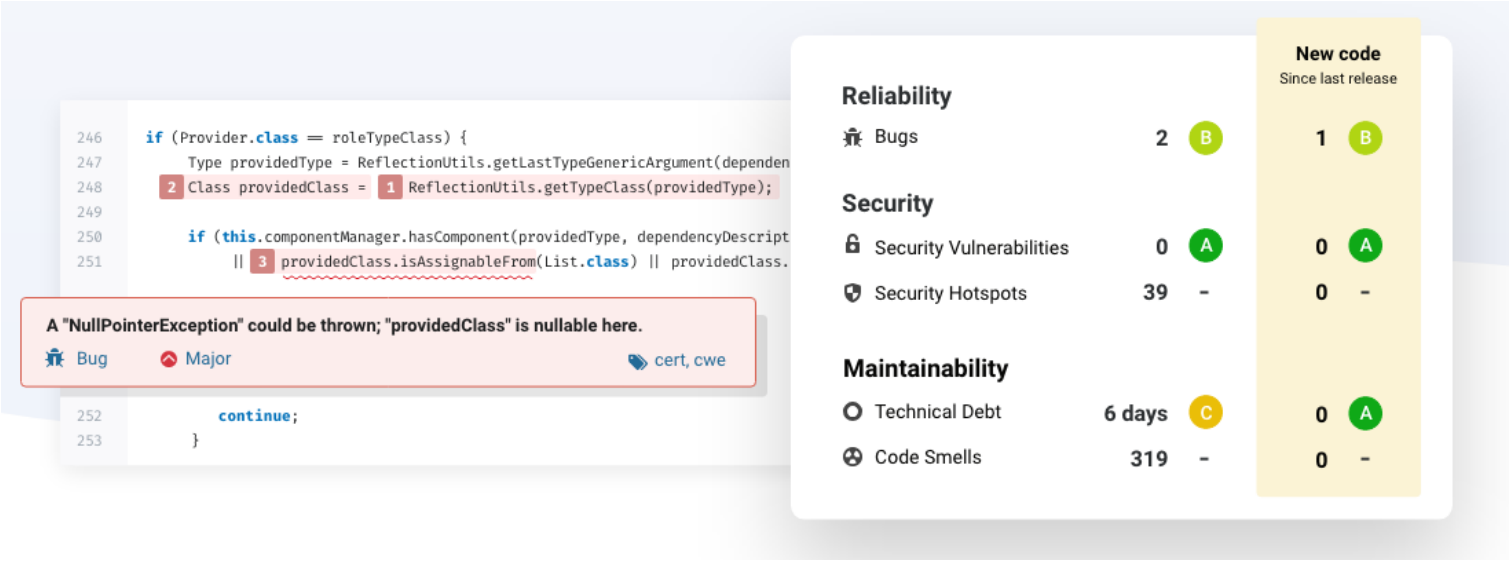
The more signals, the more powerful the insights they deliver. For example, if the cycle time is trending in the wrong direction the team will receive a diagnostic on where and why.[7]

- **The personal Software Process (PSP):** This software concentrates on the work of individual developers. PSP is a structured software development process that assists engineers in managing software quality from the start of a project to its completion. The basic measures of PSP are: Development time, Defects and size. The following are the four stages in PSP.
    1. **PSP0(Personal Measurement):** Adds a coding standard, size measurement and process improvement proposal.
    2. **PSP1(Personal Planning):** Adds schedule and task planning. There is a method for estimating the size and the development time of a new program using personal data.
    3. **PSP2(Personal Quality)**: Adds design specification along with defect prevention. This step introduces defect management.
    4. **PSP3(Scaling up):** This is the final step which shows how to scale up multiple process to developing systems with many Lines of Code (LOC).

- **SonarQube:** It is a web-based code quality analysis tool implemented in JAVA and is able to analyse the code of about 25 different programming languages. It covers a wide range of quality checks which includes Architecture and Design, Complexity, Duplications, Coding Rules, Potential Bugs, unit test etc. SonarQube provides details of different errors and coding quality level analysis it helps developers to improve the

[7] **https://insights.pinpoint.com/signals-vs-noise**

code quality and also helps to improve the coding skills. The developer can improve knowledge about the coding standards, best practices and etc. Regularly use of the SonarQube leads developers to identify the coding standard violations and they tend to adhere to those standards even at the time of coding.[8]The diagram shows how SonarQube detects the quality of code.



Different Algorithmic Approaches

- **Halstead's Software Metrics**
  A computer program consists of tokens that are classified as operators and operands. Halstead's metrics counts the tokens and determines which are operators and operands. The following base measures are used:

n1 = Number of distinct operators.

n2 = Number of distinct operands.

N1 = Total number of occurrences of operators.

N2 = Total number of occurrences of operands.

[8] **https://www.tatvasoft.com/blog/introduction-to-sonarqube-sonarlint/**

The table below shows the major equations formed when using the measures mentioned above. [9]

- Vocabulary(n): - the sum of the number of operands and operators.
- Length(N): - the sum of the total number of operators and operands in the program.
- Volume(V): - the information contents of the program measured in bits. The computation of v is based on the number of operations and operands in an algorithm.
- Level(L): -The inverse of the error proneness of the program so low-level program is more likely to have more errors than high level program.
- Difficulty Level(D): - this is also known as error proneness and the program is proportional to the number of unique operators in the program.
- Effort(E): - its proportional to the volume and the difficulty level.

[9] **https://flylib.com/books/en/1.428.1/halsteads_software_science.html**

- Number of delivered Bugs(B): - it correlates to the overall complexity of the software.

  Halstead's work has had a great impact in software measurement. He concluded that the code complexity of a program increases when the level decreases and volume increases. However, his studies have received a lot of criticism including methodology, derivations of equations, human memory models, and others

- **McCabe Cyclomatic Complexity:** This is a software quality metric that quantifies the complexity of a software program. Complexity is found by measuring the number of linearly independent paths throughout the program. The higher the

Vocabulary (n)	$n = n_1 + n_2$
Length (N)	$N = N_1 + N_2$ $= n_1 \log_2(n_1) + n_2 \log_2(n_2)$
Volume (V)	$V = N \log_2(n)$ $= N \log_2(n_1 + n_2)$
Level (L)	$L = V^* / V$ $= (2/n_1) \times (n_2/N_2)$
Difficulty (D) (inverse of level)	$D = V / V^*$ $= (n_1/2) \times (N_2/n_2)$
Effort (E)	$E = V / L$
Faults (B)	$B = V / S^*$

where V * is the minimum volume represented by a built-in function performing the task of the entire program, and S * is the mean number of mental discriminations (decisions) between errors (S * is 3,000 according to Halstead).

number, more complex the code would be.
The following equation is derived from the control flow graph of a program:

Cyclomatic complexity: *E-N+2P*

E: *number of edges*

N: *number of nodes.*
P: *number of disconnected parts of the flow graph for e.g. Subroutine.*

If the programs contain a high McCabe number i.e. >10 then it's likely to be difficult to understand and it has a higher probability of containing errors.

Ethics

There are a number of ethical implications around measuring software engineering mostly because of the large amount of data that is being gathered. ***The developers have the right to know what exactly they are being measured, how it is used and if it's going to affect their career.*** When monitoring someone's performance the data should be handled carefully and in a secure way. For example, after analysing the results it should be deleted. ***The results should not be given to a third party or sold to other organizations in any way.*** Another scenario would be to use employees unique ID instead of using their actual name.

The productivity of an employee can also depend on factors outside of the work. No one is perfect we all make mistakes. No one can be productive every day. There are days when we need breaks due to personal matters, health problems etc. As well as there are days when we try our best but everything is going in the wrong direction. The thought of knowing this could affect the measurement can lead to further stress. This can enormously affect a developer's performance. Therefore, it greatly matters when proposing a statistical model to take all these factors into account.

Furthermore, ***the use of measurement can cause an effect on how the team operates.*** If certain aspects of the development are being measured then the team will spend more time focusing on those aspects and can deviate from other components. This can

have a negative impact on the team's progression and the quality of the software.

Conclusion

There are many different ways in which a software engineering can be measured. After looking at the different metrics and the ethical implications associated with them, measurability is quite difficult.

One can argue that it is impossible and we shouldn't measure a developer's productivity. But I believe that technology is always evolving and the software industry is growing drastically, we should consider analysing multiple factors that would enable us to achieve this.

Reference

- 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.
- https://pdfs.semanticscholar.org/4c43/5041f078e555737b49b04bd3ea6a92502036.pdf
- https://www.sonarqube.org/
- http://sunnyday.mit.edu/16.355/metrics.pdf
- http://ro.ecu.edu.au/cgi/viewcontent.cgi?article=2101&context=theses