Measuring Engineering

Liam Sherlock
17308853
Software Engineering CS3012

Table of Contents

Abstract

This essay covers a brief history of software engineering, the ways in which the software engineering process can be measured and assessed through measurable data, the computational platforms that can be used to assess such data, algorithmic approaches to measure the data, and finally the ethics behind using such data.

Introduction

Software engineering is defined as "a systematic approach to the analysis, design, assessment, implementation, maintenance and reengineering of software, that is, the application of engineering to software."[1] According to the University of Minnesota Duluth there are 7 main principles of Software Engineering. Separation of concerns, modularity, abstraction, anticipation of change, generality, incremental development, and consistency.[2]

Separation of concerns:

Separation of concerns is the recognition that humans need to work within a limited context. As described in G. A. Miller's paper *The magical number seven, plus or minus two: some limits on our capacity for processing information* he describes how the mind is limited to working with approximately seven units of data where a unit is a single abstraction or concept. While abstractions can be grown it takes time and use for it to serve as a unit.[3] As software engineers need to solve complex issues it is important to be able to separate the different parts and sections of a product. Allowing the engineer to focus on one section at a time.

Modularity:

As a specialization of the separation of concerns, modularity is the practice of separating software into components based off their functionality and responsibility.

[1] Laplante, in reference 1
[2] University of Minnesota Duluth in reference 2
[3] G. A. Miller in reference 3

Abstraction:

Another specialization of the separation of concerns, abstraction involves separating the behavior of software components from their implementation, separating what an object does and how it achieves that goal. A failure to properly abstract may cause issues such unnecessary coupling.

Anticipation of change:

The building of a product is a process of communication between the software engineers and their clients. As the clients learn about what the technology can provide for them the developer learns more about the values and priorities of the clients. As such the developers must anticipate the possibility of change in the requirements, writing code in such a way to allow for the ease of change.

Generality:

Relating to the anticipation of change is important when designing software that it is free from unusual restrictions and limitations. A common example of such a restriction is the “year 2000” issue where a two digit year limitation caused an unexpected issue. The longer code survives beyond its expected lifetime the more likely such issues may occur and as such foresight must be brought into the development process to steer away from such issues.

Incremental Development:

Incremental development is the process of slowly building out software in increments, adding use cases one at a time. It allows the verification process to be simplified as any additional errors will be related to the add portion, as the portion is small and partly isolated it is easier to correct.

Consistency:

Consistency is the assertion of the fact that it is easier to do things in a familiar context. As an example, a style guide lays out a consistent manner to lay out code. This not only makes the code easier to understand but it allows the programmer to not have to worry about how they should style their code allowing them to focus solely on the implementation. From a higher level, consistency where possible helps the programmer focus on what is important. This is closely tied to the implementation of code as it is essential that code is designed to present a consistent interface to the user.

History

In the 1960s the term software engineering began appearing in debates over what the engineering process might mean for software and in 1968 the NATO Science Committee sponsored a conference on software engineering in 1968 marking the start of software engineering as a profession.[4] At the time programming was reserved for a select group with access to a mainframe, programming on punchcards. As it required painstaking planning, programs were not particularly complex, their value coming from the speed in which they could solve calculations. As such it was not until programmers started solving complex problems without existent solutions in commercial settings that software engineering became required. This lead to the rise of the central problems in software engineering, code verification, team coordination, and version management.[5] As the field has grown many of these questions have good solutions such as version control systems, automated testing, and high level languages. However, the most difficult question lies in the more human aspects of software engineering. This leads into the main topic of this report. How can we use the data generated by the software engineering process to assess and enhance the software process.

[4] Reference 4
[5] Reference 5

Measurable Data

In the current state of things, data has become a dominant source of truth. With companies building systems to process and extract knowledge and insights from structured and unstructured data. Applying the process of data science to the data created by the software engineering process could allow for an increase in productivity amongst engineers. Below is a list of concrete data points which could be used to evaluate the software engineer.

Metrics:

Employee Focused

- Online Presence
- Work Hours/Holidays
- Personal Info
- Pay + Expenses claimed
- Interaction with training
- Personal Software Progress
- Programs Used in a Day
- Websites Used in a Day
- Code Commits
    - Lines of Code
- Projects Worked On
    - Hours committed to project
    - Task completion time
- Software Quality
    - Defect Density
    - Defect Removal Rate
    - Test Coverage
    - Failure Rate
    - System Availability
    - System Reliability
    - Total Hours of Downtime
    - Code Readability
- Software Design
    - Complexity
    - Cohesion
    - Coupling
    - Inheritance
- Email/Slack Usage

- Fitbit

Environment:

- Office Space/ Office Layout
- Usage of Office Space
    - CCTV
- Weather
- Temperature and Lighting
- Office Acoustics

An obvious issue here is creating accurate models for software engineers. While there is a lot of metrics that can be used the majority of them cary their own pitfalls. Some common examples are lines of code. While some may claim that the more lines of code the more productive the engineer. The obvious counterexample is that different languages use different amounts of code. Two pieces of code that achieve the same result may have widely different lines of code but one is in python and the other is written in java. It is likely that the python code contains less lines of code. One might suggest that you then view the lines of code through the context of a particular language. But then it becomes an issue of code quality, the two pieces of code might achieve the same thing but one is ridden with issues or terribly inefficient while having more lines of code. Another problematic is total work hours as the quality of work done in an hour might drastically change person to person. One employee might spend more time achieving the same result.

There are numerous issues with most data points and finding an effective way to utilize the data. Additionally how much generalization is acceptable, if instead of lines of code you look at commits pushed, it may give a more general view of progress but then you most “productive” employee may be committing ever couple lines of code inflating his “productivity”. An even more abstract view would be to look at bugs or features provided for. But once again the amount of work required for any given bug or feature widely differs.

As I have covered above finding the right metric that accurately measures and assess an engineers productivity is incredibly difficult. However that does not mean that they are all together useless. Many of these data points may be used to assist in creating a more productive environment for the engineers. For example, if an engineer needs help with a particular language or library, the companies commit history could be used

to help the engineer track down those with knowledge in the relative language or library. If a correlation is found between quieter office spaces and employee satisfaction that information could be used to inform the managers that office renovations could benefit the company. In that way data can be useful but it will always be difficult to link it directly to engineer performance as shown above.

Tools and Computational Platforms

While metrics may be hard to generate a market does exist for platforms to capitalize on the uses of such metrics. While many platforms exist to assist the engineer to be more productive, there is a set of products designed to monitor the data produced by engineers.

On the low level exist tools that help provide transparency and some level of measurement. This are common place in the office like Trello and git. These tools help a team manage and organize themselves while providing limited measurements on the actions of the developers. Additionally tools like Atlassian's Jira are built specifically for specific algorithmic approaches. Helping teams plan and track their progress in a workflow built for an agile style of development.

On the higher level are data tools such as humanyze which leverage custom data sources with their algorithms and models to analyze and measure relevant business insights. Using data from the lower level and their own employee badge tracking tool they generate data visualizations and analysis on the employes. Measuring employee interaction and productivity. Additionally are tools like steelcase which monitor an employee's heal as many believe that a happy employee is a productive employee. They measure things such as an employee's posture, heat rate, breathing to help improve the user's overall health and wellbeing.

It is worth noting while each system may have its own sophisticated systems, they all rely on users being able to accurately apply changes based on the data. The results are useless if the results are not applied correctly.

Algorithmic Approaches

Combined with the tools I described above a number of algorithmic approaches have been designed to improve the engineering process. One of the more traditional methodologies is the Waterfall method. It is seen a sequential development approach to software development following a general sequence of events.

1. Gather and document client requirements
2. Document and design the system
3. Code and unit tests the system
4. System testing
5. User acceptance testing
6. Fix any issues
7. Deliver final product

In between each of the initial design stages contains a substage consisting of interaction between the client and development team where the developers can report their progress and the client can provide feedback and approve the next stage of progress. Once the initial design stages are complete the developers are largely independent to implement the designed system.

This approach helps make sure the developers and customers are on the same page when it comes to what will be delivered and when they will be delivered. Additionally as it is broken into several defined stages it is simple to measure progress of the overall project. Since the whole project is defined completely at the start of the process it is less likely for there to be any structural issues in the design.

Unfortunately this method also comes with several downsides. During the initial design stages clients may have a hard time gathering all of the required documents and requirements. Additionally it may be hard for the client to be able to fully visualize the final product they will be given. Additionally given that the client only gives feedback at the start there is a possibility that the client will not be satisfied with the final product and by the time they see what will be delivered changes may be hard and expensive to implement.

Agile on the other hand is an iterative based approach to development. Emphasizing the “move fast and break things” mentality, prioritizing a rapid delivery of complete components. Rather than have several defined stages time is broken into “sprints” of a defined duration. During the “sprint” the developer is expected to achieve a list of deliverables planned at the start of the “sprint”. If any work is unfinished during that sprint, the work is reprioritized and the information is used to help plan future “sprints”. The work is reviewed and evaluated by the team and customer as the work is completed. Unlike the waterfall method agile requires a large amount of customer interaction throughout the duration of the project.

The main advantages of such a method is that as the customer has frequent interactions they can help make decisions and changes throughout development. Additionally agile can quickly produce minimal viable products if time is prioritized. As the process focuses on the client it allows for a larger focus on the users wants and needs. However like the waterfall method it comes with its own downsides. Firstly it relies on high customer involvement, which if present greatly improves the product however some customers may not want or have time for such interaction. Additionally the iterative nature of agile development commonly leads to frequent refactoring as the full scope of the system’s architecture is usually not considered in the initial design.

One of the main frameworks that utilize the agile mindset is the scrum method. It is based on continuous adjustment to fluctuating factors. Designed to help teams naturally adapt to change and user requirements.

The final methodology that I will cover is the spiral model. When placed on a diagram the process looks like a spiral. Each loop of the spiral is called a phase and the amount of phases required is based off the project being developed. Each phase is divided into 4 stages.

1. Objective determination and alternative solutions.
2. Identifying and resolving risks.
3. Developing next version.
4. Review and plan for the next phase.

Throughout one phase the team will gather requirements from the customer and identify the objectives of the phase. Alternative solutions will be proposed and the team will plan for the phase ahread. The second step is to identify all possible solutions and

select the best solution possible. The risks are also identified such that they may be resolved through planning. Finally a prototype is built to help in the next stage, product development. Where features are developed and tested. Finally the last stage is to review and plan for the next phase, allowing customers to evaluate the current version of the product.

The main advantages of this model include its risk handling, flexibility, and customer involvement. Containing a large amount of risk assessment and handling, the model puts the most emphasis on risk compared to any other model. It is good for large projects with many unknown risks. Additionally as there are multiple phases client requests can be incorporated and accommodated for. Finally as the customer follows the whole development process it is likely that the final project fulfills their requirements.

Some downsides are that it is more complex than the other models, additionally it tends to be more expensive as it is difficult to know the amount of phases needed at the start of the project. Leading to time estimation being quite difficult.

However it is worth noting that one does not need to strictly follow any of the methodologies above. It is common for companies to have properties of both methodologies, fulfilling their needs and requirements.

Ethical Concerns

In search of solid ethical principles for software engineers to abide by The Computer Society and the IEEE have decided on 8 principles that promote software engineers to commit themselves to making software engineering a respected profession in accordance with their commitment to the health, safety, and welfare of the public.[6]

1. Public - Software engineers should act with the public interest.
2. Client and Employer - Software engineers should act in the best interests of their clients and employer consistent with the public interest.
3. Product - Software engineers should ensure their products meet the highest professional standards.

[6] Reference 9

4. Judgement - Software engineers should maintain integrity and independence in their professional judgement.
5. Management - Software engineering managers should promote an ethical approach to the management of software development.
6. Profession - Software engineers should advance the integrity and reputation of the profession.
7. Colleagues - Software engineers should be fair and supportive to their colleagues.
8. Self - Software engineers should participate in lifelong learning regarding their profession and promote an ethical approach to its practice.

Once defined the question becomes does the monitoring and assessment of software engineers promote the ethics as stated above. I believe that it is a mixed bag that depends on the definition of public interest. While a more productive workplace is conducive to the public interest the question remains if the close monitoring of the employees is consistent with the public interest. Primarily if the monitoring and evaluation of an employee's every action promotes their welfare. Personally I would vote no in the collection of data involving the engineer as a person but support data collection towards the work an engineer produces. Once the data stretches into their personal well being instead of the work they are producing is the line I see.

Conclusion

Overall as a profession software engineering has an abundance of tools to improve its overall productivity and efficiency. Especially as the use of social tools and technologies enter the mainstream in the enterprise world. It becomes more and more likely that any individual will have to deal with a program that processes their every action. While I have ethical concerns over the abundance of personal monitoring and potential data abuse these tools can definitely be beneficial in the progress toward a more productive workplace. While none of the tools are likely to be a silver bullet that multiplies an engineers efficiency they are likely a step in the right direction.

References

1. https://books.google.ie/books?id=pFHYk0KWAEgC&lpg=PP1&dq=What+Every+Engineer+Should+Know+about+Software+Engineering.&pg=PA1&redir_esc=y&hl=en#v=onepage&q&f=false
2. https://www.d.umn.edu/~gshute/softeng/principles.html
3. http://www2.psych.utoronto.ca/users/peterson/psy430s2001/Miller%20GA%20Magical%20Seven%20Psych%20Review%201955.pdf
4. https://learn.saylor.org/mod/page/view.php?id=12353
5. https://faculty.washington.edu/ajko/books/cooperative-software-development/history.html
6. https://www.sebokwiki.org/wiki/Software_Engineering_Features_-_Models,_Methods,_Tools,_Standards,_and_Metrics
7. https://www.seguetech.com/waterfall-vs-agile-methodology/
8. https://www.geeksforgeeks.org/software-engineering-spiral-model/
9. https://www.computer.org/education/code-of-ethics
10. https://www.humanyze.com/elements/
11. https://www.atlassian.com/software/jira