Darragh Masterson 16325463

Measuring Software Engineering: A Report

Contents

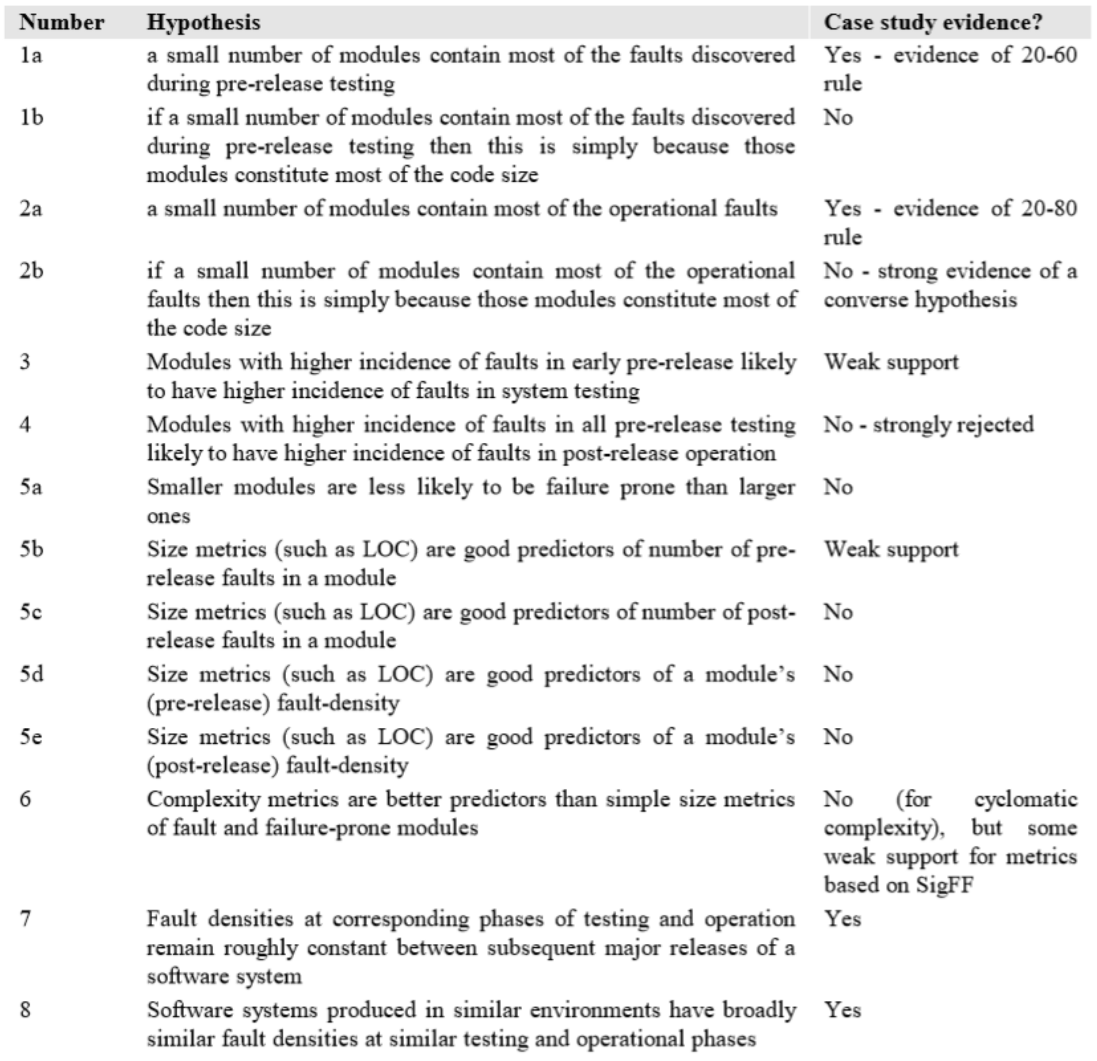
What is software engineering?

Software engineering is defined as engineering discipline whose focus is the cost-effective development of high-quality software systems. These software systems however are intangible and only exist inside the world of computers. They don’t follow any laws of the real world which makes them different from other types of engineering and makes them difficult to understand. Over the years, software engineering has had to develop its own principles and rules to follow for it to work. These rules were also for different pieces of software to be consistent with one another. Now software development is even more important in this modern world where society relies on technology and the software that runs it. For example, we now live in a world where people trust software enough that they trust it with important things in their life such as money. It is the responsibility of software engineers to build systems to keep such important things secure and reliable.

How to measure software engineering

The problem with software not existing in the real world is that there are no rules and constraints. The way that software works has been decided on and developed by humans and as a result it makes it unclear just how the best way to write software is. This also makes it unclear of the best way to measure it. the obvious way for a company to measure the work done by their software engineers would be simply to see how much code they are able to write. This would be clear evidence of work being done. Another simple method would be measuring code by the number of defects, or the lack of defects. The lines of code was a measure of quantity and the number of defects a measure of quality, however these metrics do not cover the complexity of software engineering. More importantly these metrics are hard to use to predict how good the software is and how well it will work. It is hard to see a correlation between the length of a programme and the quality of it. The number of defects may also be smaller in development and could easily grow as the project grows and even after it is released. Counting the number of defects may also discourage software engineers to taking risks and allow them to play it safe and therefore not making programmes of quality that they are able to.

A hypothesis test was done to see just how much these basic metrics would be able to tell about code written. The results being displayed in the table below.



Number	Hypothesis	Case study evidence?
1a	a small number of modules contain most of the faults discovered during pre-release testing	Yes - evidence of 20-60 rule
1b	if a small number of modules contain most of the faults discovered during pre-release testing then this is simply because those modules constitute most of the code size	No
2a	a small number of modules contain most of the operational faults	Yes - evidence of 20-80 rule
2b	if a small number of modules contain most of the operational faults then this is simply because those modules constitute most of the code size	No - strong evidence of a converse hypothesis
3	Modules with higher incidence of faults in early pre-release likely to have higher incidence of faults in system testing	Weak support
4	Modules with higher incidence of faults in all pre-release testing likely to have higher incidence of faults in post-release operation	No - strongly rejected
5a	Smaller modules are less likely to be failure prone than larger ones	No
5b	Size metrics (such as LOC) are good predictors of number of pre-release faults in a module	Weak support
5c	Size metrics (such as LOC) are good predictors of number of post-release faults in a module	No
5d	Size metrics (such as LOC) are good predictors of a module's (pre-release) fault-density	No
5e	Size metrics (such as LOC) are good predictors of a module's (post-release) fault-density	No
6	Complexity metrics are better predictors than simple size metrics of fault and failure-prone modules	No (for cyclomatic complexity), but some weak support for metrics based on SigFF
7	Fault densities at corresponding phases of testing and operation remain roughly constant between subsequent major releases of a software system	Yes
8	Software systems produced in similar environments have broadly similar fault densities at similar testing and operational phases	Yes

Even though you cannot make any assumptions from this test, it strongly suggests that these metrics are not enough to measure the complexity and how well a piece of software will work.

The line of code measurement became even more redundant as it discriminated against certain programming languages where it was harder to write more lines of code. An example is that it is harder and takes longer to write a line of code in assembly language in assembly language than it would be to in a higher-level language, such as java. In some languages more can be done in a line of code, but also more can go wrong in a single line. This led to new metrics needing to be developed in order to be able to properly measure software engineering.

From the hypothesis test it was understood better how to use the basic metrics but there was also now a desire to be able to predict the quality of a final product.

Computational platforms

A solution still using the same metrics was developed. This involves the use of Bayesian belief nets which involves simulation and probability. The execution of these Bayesian belief networks has only become feasible through algorithms and the software that is now used to develop and run them.

A Bayesian belief network is a visual web of nodes, such as the one pictured (fig 1.2) which work together with a related set of probability figures. The graph shows which nodes are related and the probability table shows the probability for each state of a node.

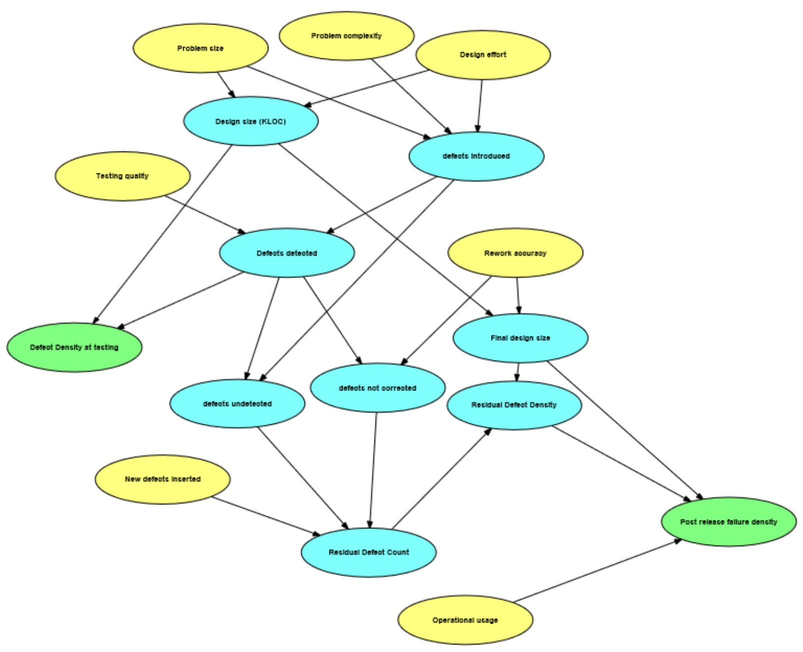


Fig 1.2

When data is entered into the model it can tell the complexity of each node. This is because it considers the nodes that rely on each other and therefore can tell which will be more complex to implement. An example of data being entered into the same Bayesian belief network as above can be seen below.

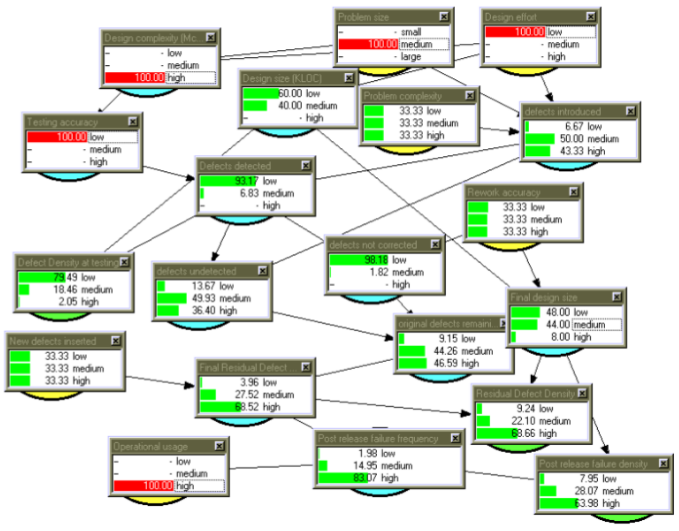


Fig 1.3.

This shows how the same metrics can be used to investigate just how many factors can affect how a software will run, especially on a deeper level.

Measurements of software engineering without metrics were developed an example of this is test driven development. This is a method rather than a metric, however, software can be used to measure if software engineers are using this method. This is when developers set tests for their code and then attempt to get all these tests to pass. There have been claims about how test-driven development will naturally result in 100% coverage code, and will produce higher quality code with less defects in it. For example, when test driven development was introduced at Microsoft it significantly reduced the defect rates where it was used. When test driven development was introduced in an IBM team it improved the quality by 50%. However, it was also found to produce less reliable code in other cases.

Zorro architecture was used to gather, sort and make sense of this data. HackyStat is used to collect this data, this is then passed down to Software Development System Analysis which organises the data. The Zorro Architecture then makes sense of this data. For it to be an appropriate measure of test-driven development it must have the necessary behaviours and recognise when test driven development is happening. A study showed that Zorro architecture was able to recognise the behaviours of test-driven development to a level of 89% accuracy. This shows how other metrics can accurately be measured which may also be useful when measuring software engineering.

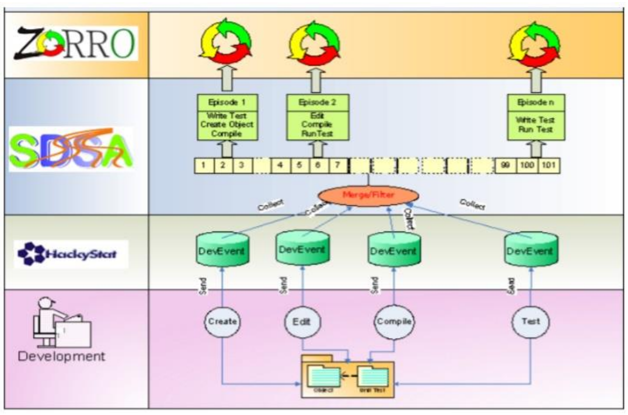


fig 1.4

Project telemetry is another way of measuring software metrics. However, this is more than just seeing the number of lines of code and the bugs, it involves constantly monitoring and collecting data from the developers' system. This is a "highly automated communications process by which measurements are made and other data collected at remote or inaccessible points and transmitted to receiving equipment for monitoring, display, and recording.". Nasa for example use this type of measurement for their operations. This can be applied to the world of software as well. The data must be collected automatically and time stamped, accessible to both developers and managers and analysis must be done regularly, making short term predictions. This data is collected using HackyStat which involves installing sensors into many of the developers' tools in their system. After installation data is then able to monitor and store data on the developer.

This data allowed the 95 build errors that were encountered to be partitioned into different categories. It also showed the differences between new and experienced developers. It could measure how regularly build failures occurred for each developer. It was also discovered that there is no relationship between build failure and the number of lines of code committed and active time before the commit. Meaning that these metrics are not causal of build failures. Although this is not a perfect solution to measuring software engineering, it is undoubtably a better measurement than just using the basic metrics alone.

Another measure of outcome is by modelling dynamical influence in human interaction. The idea behind this comes from the idea that one thing happening can affect the outcome of another entity. The analogy of a domino knocking over another domino is used to explain the concept. For this to work it needs to be understood just how the two of these entities will interact with each other. From the initial state if it is understood how components will interact with each other one will be able to predict how the whole system will work. This can be brought into software engineering as it is something that has the data and can have a lot of entities interacting with each other inside of a piece of software.

Another problem with software development is that the same traditional methods have been continuously been used. These methods such as the waterfall model were used for more than 40 years, meaning that there was no moving forward in this area. This was not changed until it was challenged by a group of software engineers that wanted to try implement agility into the software development process. This new method posed a solution to the old problems associated with software engineering. However, this was a big change and traditionalists were cautious to change over to trying agile methods. Although this is not a measurement of software development it is a platform of improvement and is now being implemented by companies such as IBM, Microsoft and Google.

Another solution that involves gathering data from the software engineers is by recognising what they are doing right and rewarding them for doing so. This is done through monitoring patterns of the developer working on a certain task and then compare these work patterns to those of a developer using priori defined practices. The system will give feedback to the developer in the form of a game on how to adopt better techniques and habits in the form of a game.

However, for this to work the patterns to monitor must be understood. Systems such as Mylyn Monitor, HackyStat or PROM can be used to gather this type of data. The required data is the commands issued in the development environment, the commands to the code and finally the commands outside the working environment. An example of the data tracked was how the developers looked online for solutions to a certain bug. A study showed that a developer that used unstructured navigation took more than twice as many steps to find their solution. This is an inefficient way of finding a solution and shows that this data can be used to find such inefficiencies. An example of a habit used by the inefficient developer is that on average the find text command was used four times every time they tried to fix a bug. The other developer, however, used this feature less than once. This shows that it is more efficient to use other methods such as the find references command.

It is then important to encourage software developers to follow such commands. This could be done through a proposed game which would encourage developers to follow these patterns by rewarding them in the game when done so.

Ethics concerns

When gathering data, the ethics of doing this has to be taken into account. The act of trying to become more productive will result in employees having to work harder. The need for this workload and the fact that they will be constantly monitored may result in stress and anxiety to the employees being monitored. This is especially the case when falling behind on work. Not only is this ethically wrong but it may also create a negative impact on work if employees are not able to deal with working under this type of pressure.

There is also the ethics of holding the data on employees. There can be GDPR concerns from these employees that the company may use this data in the wrong way or continue to use it after moving on from a company. So, when collecting this data, a company must make it accessible and let the data subjects know exactly how it is being used and how it will be used.

Lastly there is a problem with how the data may be perceived. Data can be misinterpreted, and this could result in employees being treated unfairly as a result. An example in project telemetry is that HackyStat measures a metric called Active Time. Which measures time working on the files in tools such as IDEs or Microsoft office. However other things would not be considered, such as reading and replying to emails or having meetings with other employees. So, a measurement of 4 hours for an 8 hour work day would look like the employee has not done enough work. An employee may have done a lot more work than this however and management may try to punish a developer as a result.

Conclusion

So, to conclude Software Engineering is so far a flawed process and there is no perfect way or making or measuring it. The methods mentioned above can be used in different scenarios but there is no method that will always work in every scenario.

As the report *No Silver Bullet – Essence and Accident in Software Engineering* says there is no development in software engineering that can revolutionise the whole industry to the magnitude that is happening in hardware. This is true as there is no system that can completely get rid of errors in software engineering as it is too error prone.

Bibliography

Brooks, F. P. (1986). *No Silver Bullet – Essence and Accident in Software Engineering.* Chapel Hill: University of North Carolina.

Fenton, N. E., & Neil, M. (1998). Software Metrics: Successes, Failures and New Directions. *Journal of Systems and Software*.

Gandomani, T., Zulzalil, H., Ghani, A. A., Sultan, A. B., & Nafchi, M. Z. (2013). Obstacles in moving to agile software development methods; At a Glance. *Journal of Computer Science*, 620-625.

Johnson, P. M. (2007). *Automated Recognition of Test-Driven Development with Zorro.* Honolulu: University of Hawai'i.

Johnson, P. M., Kou, H., Agustin, J., Chan, C., Moore, C., Miglani, J., . . . Doane, W. E. (2003). *Beyond the Personal Software Process: Metrics collection and analysis for the differently disciplined.* Honululu: University of Hawaii.

Johnson, P. M., Kou, H., Paulding, M., Zhang, Q., Kagawa, A., & Yamashita, T. (2005). Improving Software Development Management through Software Project Telemetry. *IEEE software 22.4*, pp. 76-85.

Pan, W., Dong, W., Cebrian, M., Kim, T., Fowler, J. H., & Pentland, A. (. (2012, March). Modeling Dynamical Influence in Human Interaction. *IEEE Signal Processing*, pp. 77-86.

Snipes, W., Augustine, V., Nail, A. R., & Murphy-Hill, E. (2013). Towards Recognizing and Rewarding Efficient Developer Work Patterns. *IEEE*, pp. 1276-1280.

Sommerville, I. (2006). *Software Engineering 8th Edition.* Addison Wesley.