Ida Fantenberg Niklasson

CS3012 Software Engineering

15 November 2018

A report on the measurements of the Software Engineering process

Introduction

A software engineering's work is so much more complex than just writing code. It is about working with a client and find out what the client needs. Moreover, finding a solution, implement a good design for the solution and testing the solution. The design should not only fulfil these needs but also work for future problems and unexpected requirements. The code needs to work for many years and be able to keep up with the development of technology. In other words, the code needs to be reliable.

Since the software engineering process is complex, the way to measure the process is complex as well. This report will focus on how to measure the software engineering process in terms of measurable data as well as platforms and approaches where these measurable data is accessible.

In the end, the ethics about this subject will be discussed. If we find that it is possible to asses the software engineering process, is it right to do so?

Measurable data

The software engineering process is quite complex and hard to measure. To be able to measure the process you need to have some specific data which you can analyse and of course, that is measurable. Before discussing what kind of data there are, we will look at the waterfall model. There are a lot more software process models and they aren't necessary to describe the different data but, they visualize what data you can get from the software engineering process.

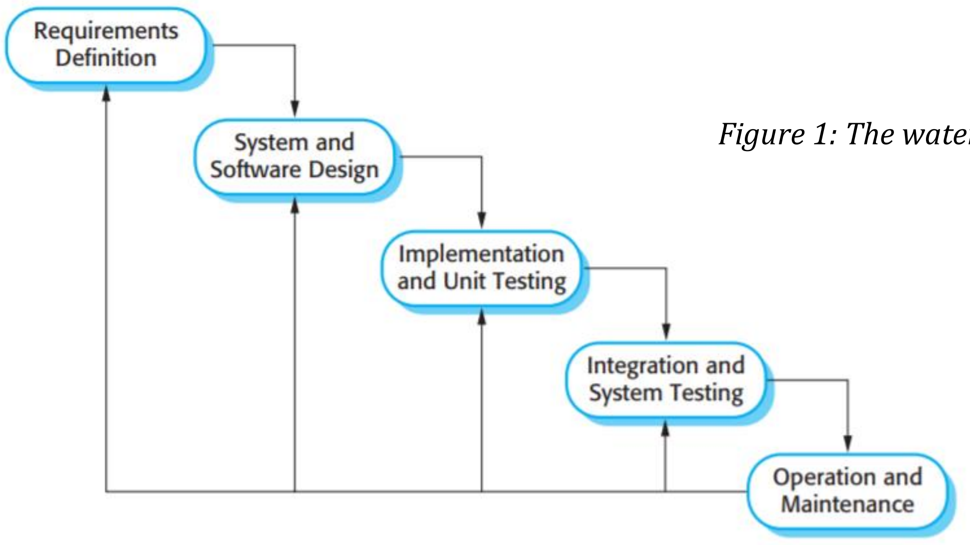

Requirements Definition
System and Software Design
Implementation and Unit Testing
Integration and System Testing
Operation and Maintenance

Figure 1: The waterfall model

The five bubbles in figure 1 represent the different stages in the process. The waterfall model is based on a plan-driven process where you have to plan and schedule each part of the process before starting to work on them.

Requirements Definition is the first part where you together with the customer analyse the need of the system user. Thereafter you obtain specific and detailed requirements and goals for the system.

System and Software Design is about the architecture of the system, in other words, the design. How the different part of the system should be implemented and the relationship between them.

Implementation and Unit Testing is about creating the design in different program unit. Unit testing is also an important part to make sure everything works as it should.

Integration and System testing is about integrating the different program units into one complete system. Then a final testing needs to be done before handing over the system to the customer.

Operation and Maintenance, the last part of the process. This is about having the system in use and correcting error that weren't found earlier in the process. Improving the implementation and develop the system according to new requirements which may occur is also a part of this final part.

From the description of the software engineering process is it easier to now find and categorize different data. A big part of the process is of course writing lines of codes therefor, you could easily measure the work by number of lines. The lines itself does not say much about the work though, since a program with a lot of code can be very ineffective. If the work would be assessed by the lines of code the programmer could easily make more comments and whitespaces to make it look like they are doing a better job. Measuring the code in that way would lead to a code with less quality. Moreover, solutions with lots of code could make the program less efficient. At the same time a solution with less code do not always have to better. Since the process is way more complex the measured data need to be as well. The lines of codes are data that is measurable, but the process cannot be assessed only according to that data.

Personal Software Process (PSP) and the Team Software Process (TSP) are two ways to find valuable measurable data. Both the PSP and TSP are designed to help developers to understand the development process. They are also supposed to help the developer to better understand the process. Johnson, P are describing these two analytics in the article "Searching under the Streetlight for Useful Software Analytics. In the article the connection to searching under the streetlight is that you can search for and collect data that is easy to get, in other words are in the streetlight. Then again, as with the number of lines with code, the data that is the easiest to get most often is not the data that is most useful. Notably is also that if the data is easy to use that could also implicate that the data itself is not as useable.

PSP and TSP were created by Watts Humphrey. He created these because it was here he found that engineers, both individual and in small and big organisations, encountered the biggest difficulties. To summarize the principle of PSP and TSP, you could to start with look at how productive the engineer is, the quality of the code and how much time the engineer spent on the project. The productivity could be measured by lines of code and how many commits the engineer made. Furthermore, it could also be number of messages sent to the project's member since communication is also a big part of the work.

From the unit testing and system testing part of the process you could get data on the quality of the code. Testing is a way to know where the code requires improvements. Moreover, the testing does not only improve the code but also the process itself. The better test you have the more can be modified in a better and faster way. A possible way to measure how the testing is affecting the code is by code churn. Code churn is when an engineer's code needs to be modified in a short period of time. To compare with counting the lines of code again, let's say there are two people writing code and of them are writing significantly more code than the other. If it then turns out the one who has written the most code, needs to change most of it, then it wasn't that productive kind of work. To put it another way code churn is a measure on whether the code that is added is useful or not.

Another measurable data that is related to testing is code coverage. When the code is tested a certain percentage of the code is covered, that is called code coverage. That percentage gives a number on how good your tests are and could give an answer on if you would need more tests. At the same time, you maybe need less tests. There could a case where you write some code and then test that part of the code. Later when you write more code you maybe add several more tests to test the next part. Instead you maybe could have modified the first tests instead of writing new tests and testing the same thing repeatedly. The best-case scenario would be that every line of the code would be executed at least once during a test. Then the code coverage would be 100. But the code coverage is not only a measure on how good the testing is, it is also a helpful measure on the quality of the code. If a part of the code is not being tested, you cannot know the quality of the code. Although some code can be very hard to test.

All the mentioned data above is about the code itself. From the waterfall process model, you can see that the software engineering process is more than just writing code. The code is the product of the process but in the process, there is also the individual of the engineer. Studies have shown that a worker will be more productive if the worker is happy. A productive worker does not necessarily have to be happy but, a happy worker is more likely to be productive. The happiness of a worker can, according to the article "Measuring Happiness Using Wearable Technology", be measured by sensors who is measuring the physical activity. They mean that physical activity would imply increasing happiness and therefor, also increase the productivity. The easiest example of a physical activity would be to have a break from the job.

Whit this in mind we can go back to measurable parts of the process. The parts including codes can be measured by the examples above. Those parts could also, together with all the other parts be easily measured in how much time needed to complete the task. An engineer that completes the work as fast as possible is more likely to not have implemented a well thought out solution. A solution that is well thought out but takes a little longer to write will probably be better. The time of a task is something that needs to be considered, especially because nearly all projects have a deadline. Although the time should not be considered by itself.

Cycle time is a key measurement for time. The concept cycle time comes from the lean thinking, meaning how much time the process for a product takes from the beginning to the end. You can measure the cycle time for the whole engineering process or for only a part of it, for example cycle time for system testing. This is an easy thing to measure if you only have the tools to do it. Cycle time is mostly an average value for doing the specific task. The cycle time itself may not contribute to much but it is a good guideline to know about how much time a task should take. If the task takes much more time than the cycle time you can suspect that there might be something wrong.

A lot of these different measurable data that have been mentioned above shouldn't be considered in isolation. A combination of different data will make a better analyse on the situation. It might still not give a fair assessment of the situation. What kind of assessment that suits the situation will be different from time to time. The data mentioned above are only a few of all data that could be measured.

Computational platforms/ Tools

It would take way too much time to get all this data manually. Therefor a computational platform or tool is necessary to gather all this data for you and your project team. There are a lot of different platforms and tools available on the market that help you get the data needed. These tools do not only help you gather the data they also present it in a way to give feedback on the engineer's and project team's work. Most of these testing tools are automated, in other words they are collecting data while you are working with the code. Listed below are some testing tools that are available.

Toggl

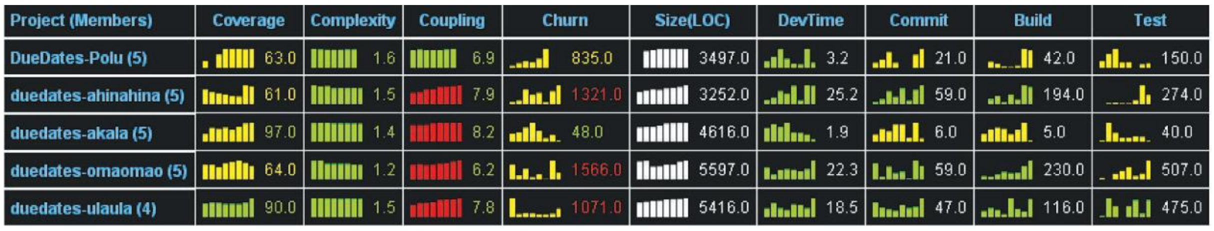
The time, as mentioned as a measurable data, is very valuable in most of the projects. It is easy to se how much time in total you have spent on a project, you only have to calculate the time between the start of the project and the end of the project. That time, however, doesn't say much about the engineer's performance.

Toggl is a tool that tells you more specific how you spent your time in a project. By the reports you get from Toggl you will be able to see where in the project time were spent. Then the time can be compared to the worked that was done during that time, and in that way calculate the productivity. This is a great tool for both the individual engineer and the project manager.

For the individual engineer this tool can help you understand what takes time in your working process. Certain tasks need more time but there could be a task that took longer time that you expected. By knowing these things, you can change the way you work. The result from a certain part of the process was not maybe as good as desired, a little time spent on that task implicates that there is where you should spend more time for a better result. The same principle applies for managers and their teams. It is important for a project team to be effective and be careful with their time spent. Some things may not be as important as other things. From the reports Toggl can generate the manager will have a better understanding how the team is working. That is also a helpful tool for planning future projects. If the manager better knows how the team is spending their time the timeline will be more accurate for the project.

Hackystat

A little more complex testing framework is Hackystat. Hackystat gather process and product data from the team's work. Sensors are attached to the development tools and any other tools they are using to get the data which are then sent to a web server. Hackystat can from the data collected generate graphs and diagram to represent different metrics. Some examples on these metrics are code coverage, complexity, code churn, size, commits and tests. By comparing these graphs and diagrams to each other you can find valuable patterns about the engineers and the team's behaviour. Hackystat was developed to facilitate the process to identify the cause of a problem in a project.



Project (Members)	Coverage	Complexity	Coupling	Churn	Size(LOC)	DevTime	Commit	Build	Test
DueDates-Polu (5)	63.0	1.6	6.9	835.0	3497.0	3.2	21.0	42.0	150.0
duedates-ahinahina (5)	61.0	1.5	7.9	1321.0	3252.0	25.2	59.0	194.0	274.0
duedates-akala (5)	97.0	1.4	8.2	48.0	4616.0	1.9	6.0	5.0	40.0
duedates-omaomao (5)	64.0	1.2	6.2	1566.0	5597.0	22.3	59.0	230.0	507.0
duedates-ulaula (4)	90.0	1.5	7.8	1071.0	5416.0	18.5	47.0	116.0	475.0

Figure 2: Metrics presented by Hackystat

Below is a list on the four key design features of Hackystat:

1. **Client- and server-side data collection**
   Hackystat has developed editors, build- and test tools for the client-side but also configuration and management repositories for the server-side.
2. **Unobtrusive data collection**
   The developers do not need to paus their work to manually record what they have been working on. In other words, unobtrusive data collection means that the developers do not notice when the data is collected.
3. **Fine-grained data collection**
   By using the client-side tools data can be collected very often which implicate a detailed data collection.
4. **Both personal and group-based development**
   Hackystat tracks data for the individual developer but the developer can also create project in Hackystat which keeps a track on the interaction between the team members.

GitHub

The popular repository service GitHub can also gather data from the software engineering process. There are several different toolsets for GitHub that obtain data from the repositories in GitHub to represent the data to the user. Since a project are divided into different repositories data can be represented for each repository and commit. Which make a very good visualization of the data. It is also possible to get raw data from GitHub to make your own graphs and diagrams or use the information in other ways. By using repositories in general and doing commits is a great way to track your work. There is always documentation for each commit which are very valuable for the analysis of the project.

Algorithmic approaches

Algorithmic approaches can be seen in two different ways in this case, either as an algorithm to calculate data or as an algorithm for an engineering process. Above measurable data is data you get from the current engineering process. Instead of trying to find ways to calculate new data the engineering process could be developed to make other kind of data available. So instead of making complex calculations it can, with a little effort, in the long run be easier to change the engineering process. GitHub and other repositories are a prime example of that kind of change. By doing the commits to the repositories you can collect certain kind of data that would be hard to do otherwise. Below the two different kind of algorithmic approaches will be discussed.

One outcome of doing commits to a repository is code churn. Each time a commit is made to GitHub the lines of code are either green, red or just white. The colours describe if the line of code has been deleted, added or not changed. Although there isn't a colour for when the line has been changed, when calculating code churn, one added one deleted line will equal zero. To calculate the code churn, the lines of code will be counted when the first commit is made. When the next commit is made all lines that have been added or changed will be counted. From this number the lines that have been changed will be subtracted and then you have the result.

Let's say that you have a piece of code that you must change many times because you get new directives from your client. If you then do commits very often there will be a lot of code churn since you are changing the code each time. If you instead only did one commit after you done all the changes, you will only see the code churn once. That is to say, the code churn will look different depending on how often the commits are made but, the algorithm for calculating it will be the same every time.

Code coverage is even easier to calculate. An algorithm for code coverage is not really needed since the code coverage is calculated when the code is executed. When the code is executed each line that has been executed will be counted and then divided to the total number of lines in the code. The tricky thing about this algorithm though is that you may have elements in the code that you don't want to be calculated. Then you will have to exclude those elements from the code coverage result. It is possible to exclude code but then the you need to be aware of that before calculating the code coverage.

There is a lot of different algorithms to calculate different kind of time in the process. Before explaining some of these the Parkinson's Law is worth mentioning. The definition of Parkinson's Law is "work expands so as to fill the time available for its completion". The relevance for this is that how much time the engineer has until deadline might affect the engineers work. It is quite impossible to set a deadline perfectly according to the work that needs to be done. Imagine that there are two engineers with the same task but one of them have significant more time to complete the task. Then the engineers will probably not complete the task at the same time.

Having said that, time is often used together with other metrics. Two examples are Open/close rates and MTTR. Open/close rates are according to an article from

TechBeacon calculated by "how many production issues are reported and closed within a specific time period". MTTR stands for mean time to repair and is the total time of maintenance divided by number of repairs. Lines of codes divided by a certain time is also a measure that is being used.

Now to a way that the engineering process can be changed instead. One approach that has been tested is gamification. Gamification is about making the software engineering environment to be like a game. The idea is that this will make the software engineering process more fun and at the same time learn and adapt the software engineering behaviour. In a study made, a certain number of commits made you level up in the game, that was the software engineering process. As in a game they also had a leader board so that the students in the study could get regular updates. Even though this approach was not very popular by all students in the study they developed a behaviour to commit more often.

In a company a similar approach could be used in the real work. Even though the environment wouldn't be a game, they could do weekly or monthly challenges to make the engineers a little bit more excited. People in general get stimulated by achieving goals and getting credits for their work. In other words, a project team's productivity could benefit from having smaller goals on the way to the final goal. If the aim is to really embrace the approach of a game, it would be possible to also have a leader board at a company. The engineers are probably working on different tasks, therefor, an idea would be to compete about having the lowest weekly code churn or code coverage. That would also be a great tool for the manager to collect data about the work, in a funnier way.

Ethics

Collecting data

The first and a very extensive question about the ethics of this topic is if the data should be gathered at all. If a lot of information is collected there is also a risk that the information will be misused. In big companies with many employees and much information stored data can get leaked and personal data can fall into the wrong hand. When you sign up for different software's they often collect sensitive data. Most of the time they have user terms which you are obligated to read through, but most people don't do it. One reason for that is that you must accept the term to be able to use the software. You are probably never forced to use a particular software but at some point, you have to use at least one. That is to say that at some point you also must accept some user terms.

There are also a lot of different location systems used in companies to track the employees. In healthcare for example, the current location of the staff and patients are very important. In an office the location of the staff is not as important but still very helpful. There are different badges the employees could have with them so that their location would be able to track. The pager system is also a popular solution where ´you page someone and then they get a notification or a smaller message where they should go.

There will always be an ethic issue when tracking and collecting location data of the employees. Firstly, the employees might not feel comfortable being tracked and for different reasons don't want the company to know exactly where you are. Since it is the device that is being tracked it is possible for the employee to put the device away to avoid being tracked. That would damage the whole tracking system since the location of that person would be misled, then the location system is not reliable.

Another issue that employees have with a tracking system is how the collected data is handled. There are no problems with being tracked by the manager if they know that only the manager could access their location. But as always, there are an uncertainty who could access the location data. This is a difficult matter and a question whether being able to track the location of the staff is possible without the employees feeling concerned.

Accuracy of the data

Then say that you at last gather the information, how do you know that the data you get are accurate in the assessment of the engineering process?

A study was made to investigate the behaviour of a developer. They recorded a video of the developer's screen while she was working. The aim with the study was to investigate the behaviour of a developer compared to how successful their tasks were. Record a video of the screen in a study is possible because the one in the study are in on the condition and it's probably not for a longer period. Have this approach in the real workplace would though be a big ethic issue. The employee would probably feel observed all the time, feeling like the manager is standing behind you and watching every step you take. This might have a negative impact on the mood which could implicate that the employee would be less productive.

In addition, recording a video of every employee would result in very big amount of data. There are no reasons to collect that amount of data because the manager will not have the time to go through everything. In such big amount of data, there will also be a lot of unusable data. Collecting that much data would of course also be a matter of privacy.

The Hackystat framework is similar to the case above, even though they don't collect as much information. Hackystat collect data from the developer's tool which will result in a big amount of data too. The difference is that the gathered data is only regarding the developer's work. It is still an ethical question, especially whether the employees know that data is collected or not. An issue Hackystat have had is that the employees have complained about that they know that data is gathered but, not exactly when it is gathered.

Coupled with everything above, let's go back to the ethical about the key question how the engineering process can be measured and assessed. As mentioned, there are a lot of different measurable data that can be used to assess the engineer's work. You can get a number on how good someone is doing their job and compare with others. The ethical question in this is if you can trust those numbers. By discussing the case where the engineers have the same task but different amount of time, it will be clear that a number might not tell the whole story.

The engineer who had more time for her task can be more thorough and her code will probably be more thought out. The other engineer with less time will not have that time to carefully think about a good solution but stills gets the job done. They both complete the task even though the engineer with more time had a better solution. Then is she a better engineer since she had the better solution? Or is the other engineer a better engineer because she could complete the task in less time? The same case could in theory be done with the same person. Then the only thing that will separate the different cases are the time, not the skills.

If you then would look at their software engineering process you could for example look at the code churn. The engineer who did consider the solution before starting to write code will probably have less code churn. Since that engineer had a better solution, she didn't have to change the code as much. Developers also works with different approaches. Some might have a testing approach where they don't think as much about the solution, they keep on writing and testing the code along the way instead. Someone who has that approach will have a lot more code churn than someone who once again, has a more thorough solution from the beginning.

Conclusions

When assessing the work of a software engineer it is important to be using the right metrics for that unique case. The manager needs to carefully consider which metrics should be selected. There is not a specific metric that is better than the others, which metric or metrics should be used for assessing the work depends on what kind of information the manager needs.

In my opinion, the best way to make a fair assessment is to compare different metrics to each other. To collect data from the development tools, like in Hackystat, only regarding the software engineering process and not any private information. There should then be a limited access to the stored data and it should of course be stored in a secure way.

References

- Sommerville, Ian. (2011). Software Engineering, 9th edition. Pearson Education, Inc.
- Johnson, Philip. (2013). Searching under the Streetlight for Useful Software Analytics. http://www.citeulike.org/group/3370/article/12458067
- https://en.wikipedia.org/wiki/Personal_software_process
- https://blog.gitprime.com/why-code-churn-matters/
- https://confluence.atlassian.com/clover/about-code-coverage-71599496.html
- https://hbr.org/2012/01/positive-intelligence
- http://csdl.ics.hawaii.edu/research/hackystat/
- https://stackify.com/track-software-metrics/
- E. B. Passos, D. B. Medeiros, P. A. S. Neto and E. W. G. Clua, (2011) "Turning Real-World Software Development into a Game," Games and Digital Entertainment (SBGAMES), 2011 Brazilian Symposium on, Salvador, 2011, pp. 260-269.,
- http://lore.ua.ac.be/Teaching/SE3BAC/slides2007/10MetricsOld.pdf
- https://docs.microsoft.com/en-us/azure/devops/report/sql-reports/perspective-code-analyze-report-code-churn-coverage?view=tfs-2018
- https://blog.gitprime.com/6-causes-of-code-churn-and-what-you-should-do-about-them/
- https://docs.microsoft.com/en-us/visualstudio/test/using-code-coverage-to-determine-how-much-code-is-being-tested?view=vs-2017
- https://en.wikipedia.org/wiki/Parkinson%27s_law
- https://techbeacon.com/9-metrics-can-make-difference-todays-software-development-teams
- https://www.klipfolio.com/blog/cycle-time-software-development
- https://www.fiixsoftware.com/mean-time-to-repair-maintenance/
- 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.
- Martin P. Robillard, Wesley Coelho, and Gail C. Murphy. 2004. How Effective Developers Investigate Source Code: An Exploratory Study. IEEE Trans. Softw. Eng. 30, 12 (December 2004), 889-903.
- R. Want, A. Hopper, a. Veronica Falc and J. Gibbons. The active badge location system. ACM Trans. Inf. Syst., 10(1):91–102, 1992.