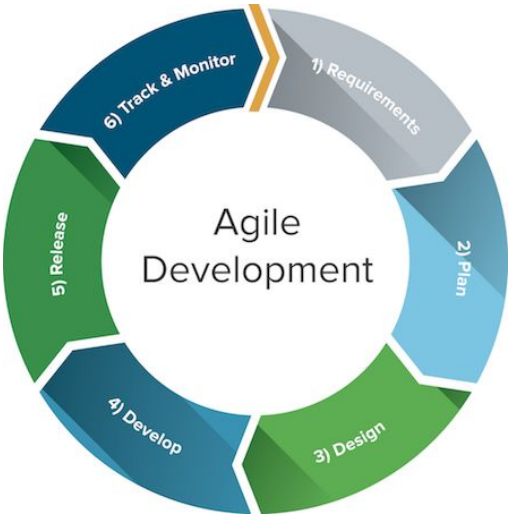
Measuring Software Engineering - An Essay

Software engineering is an incredibly diverse field of work. Ranging from developing the sturdy low level Operating Systems functionality to putting together the beautiful front end to a website. As with any field of work, software engineers have a responsibility to both produce code at a reasonable rate and also create high quality code. Measuring these is key to grading how well a software engineer is performing and is therefore a key point of interest for managers. How we actually go about this is a difficult and complex task which I hope to shed some light on in this essay.

The Software Engineering Process

In order to discuss what we can actually measure we must first understand what the software engineering process involves. There are many models that software engineers follow but probably the most widely used method is the agile method. The agile method, for those who don't know it, is a six stage cycle for producing a piece of software or functionality. It starts with an idea for a piece of functionality or requirements for software that someone wants made. At this stage the team has to decide what will be required by the project or program. Then a plan is generated for how the solution can be tackled or solved. This decides who will be working on what and when everything needs to be done by. Next the software will be designed around the requirements and deadline. This part would decide what functionality will be implemented and who will do what. The next stage is when the actual development on the agreed upon features begins. During this stage features of the program will be created in teams or individually to fulfill the specifications to the best of the developers ability. Once this period is complete the developers release the completed features. This can be pushing to an app store, deploying an update to the cloud or just simply pushing to a repository. From here the changes are tracked and monitored to see if they are functioning as intended or if there are any unexpected issues that have arisen in a production environment. This is the general flow of the agile method.



The agile method doesn't necessarily specify the inner workings of the development period, it can be further broken down. This will however differ on a team to team

basis. Some commonly used techniques in here are version control systems, task management systems and daily standup meetings. Version control systems are used in almost every software engineering team around the world in some way or another. It allows the developers to work on different pieces of work without breaking each-other's builds every time they change a line of code. These version control systems allow developers to create “commits” which are individual collections of changes to a file. Generally each commit will encompass one or more program features/ fixes. Furthermore if several people want to use each-other's commits or are working on a large feature away from the main team they will use a “branch”. This means their commits will be kept separate from the rest of the team but they will be able to pass their commits between each-other. Then when the team are finished working on the feature they can merge the branch and all of its commits back into the master branch, thus the feature is implemented. Task management systems are also increasingly being used in teams to track who is working on what. The way it works is generally with either an actual board or a virtual visualisation of a board with several columns. Generally the columns will follow something along the lines of “To-do”, “In progress”, “Complete”. Initially all tasks will be under the to-do heading. A developer will then take, or be instructed to take one or more tasks and place them under the in progress heading. They will also add some sort of tag to the tasks to flag them as their tasks. The developer will then work on these tasks until completion at which point they will be moved to the complete column. Some groups will use a “blocked” column which means that a task cannot be completed yet as it is dependent on something else, be it a knowledge block or a programming dependency. Finally stand-up meetings are sometimes used by teams during development. These are generally short (~ten - fifteen minutes) meetings held standing up. This is to encourage it to be completed quickly. In this meeting each team member gives a quick summary of: what they did the day before, what they are working on that day and what issues they’ve encountered. If a team member has been blocked by some sort of programming dependency they would explain it here, or if someone had concerns whether the team would reach their deadline at this point. So in summary the software engineering process generally follows the agile development cycle but can also have many variations during the development phase that will decide how the team works together.

How it can be measured

Measuring the software engineering process is difficult and there are many approaches. Some good and some misleading at best. In this section I will discuss how these measurements can be taken. These measurements can be categorised into either autonomous or manual. Meaning that some of these measurements will have to be gathered by hand or automatically as the work is being done. First of all I would like to discuss how we can measure software engineering manually and then

later I will discuss how some of this can be automated. One way to measure is to note by hand at the stand-up sessions, who is completing how much work and who is getting blocked or working slowly. One advantage of measuring like this is that the person who is measuring is likely to understand what sort of work is being completed and how hard that work is. Meaning despite the fact that someone completed thirty tasks and someone else completed three. They can calibrate based on how hard the tasks were. This allows them to judge people in a more general sense. However this person might be biased and therefore this method can only provide general idea of how much work is being done. Another way that the work can be measured by hand is by monitoring the task board and watching to see who completes what tasks and who gets blocked a lot. This is similar to the method described above for stand-up meetings, however this can be better as there can be a level of separation between the observer and the team, this can make for overall more unbiased opinions. This however still isn't guaranteed and so it's still only best used for creating a general overview of the work distribution. So as can be seen by these examples the actual measurements made by these two methods are inherently biased by the fact a human made the observations.

Now we will look at some programmatic approaches to the problem and see if we can do any better. There are several ways in which we can measure the engineers in an automated manner. We can capture from the version control system how often commits are made. We can also read from the version control system how many lines of code are written by each developer. Alternatively if you are using a virtual task board, there are methods for scraping how many tasks members complete or how long it takes someone to complete a particular task. These can all be done with simple hooks in the version control system or generally online task management boards will have statistic visualizations. These can be scraped and dumped to a database to be analyzed at a later stage. This data will be impartial as it has been grabbed by an automatic hook that doesn't differ between one's best friend and worst enemy. What's also great about this method is the data can be made anonymous at this point, this would hide who is actually being judged from the assessment stage of the measurement process. Another way of gathering data is running software on the developers devices to log where they go during the day or what websites they visit. This can allow the observer to figure out if they are becoming too distracted or if they are deliberately not working as hard as they need to. There are however issues with this method as well that I will discuss later in this essay. At this point we now have our methods of gathering data from an individual or group of software engineers that should be able to provide us with some abstract knowledge on whether work is being accomplished.

How it can be assessed

Once data has been collected, the question of how to analyze this data comes around. Again there are almost as many ways to do this analysis as there are types of data. Like measurement techniques we can broadly categorize the analysis into two disciplines. Either manual or automatic. If we are going to look through the data by hand then we should have some way of either visualizing the data or not very much data to go through by hand. As humans are innately prone to human error it's best left to machines to go through large quantities of data. However if we only have a small amount of data or a good visualization for the data it is possible for a person to do this task. Now regardless of whether the data was gathered by hand we need to understand how to analyse each type of data we have gathered. The number of tasks that the engineer completes can be useful for analysis. Some would say it's possible to simply linearly compare the number of tasks that one engineer completes against the number that one of their colleagues complete. However there are of course drawbacks to this. Firstly tasks might be of different difficulty/ scale. This would mean that someone could complete one task that would be of a similar scale to another five tasks together. Another difference is that some tasks might have different expected durations or someone might be working on several different tasks simultaneously meaning they put none in the complete section despite the fact they did a lot of work. This is why task completion can be too vague of a metric. It requires the tasks to be finely tuned and the developers to work on exactly one task at a time. Therefore analyzing by tasks can be difficult and inaccurate. Another way of analyzing the data is looking at the number of program lines each developer wrote in some time period and directly comparing that with the number that another wrote. The problem with this is again that these lines mightn't have similar value. For example throwing together a simple user interface for an android app can be done with a gui that will generate the code for you. You could generate a couple hundred lines in a matter of minutes by this method. But someone else could have created an implementation of quicksort for the program which is no more than twenty lines long. Since the thing we actually want to measure is the amount of effort each software engineer is contributing, this method wouldn't work as it would claim the user interface in this example required much more effort than the quicksort implementation. A happy medium between the extremes of measuring by lines and measuring by tasks is to measure by commits. Commits generally represent a useable noteworthy block of code towards a feature. Therefore in our previous example of the user interface and quicksort implementation, both developers would have had similar commits, probably one or two at most. There are of course drawbacks to comparing developers on their number of commits. Firstly you would have to programmatically gather the data on how many commits they produced. This is because it's just not feasible to employ someone fulltime to manually count how many commits each person made. You would also have to standardise between the engineers how much effort each commit should represent. Also if or when the

developers realise they are graded based on the number of commits they make they will start committing in smaller and smaller increments until eventually it's reduced to effectively counting lines of code again. This is because the grading effectively incentives more commits, which is the same as incentivising smaller commits. This can make commits another bad scale for measuring engineers.
For these reasons I believe that the best way to assess the engineers is by the team leader or project manager via standups. This is overall the most accurate way as they can understand the team much better than a program and furthermore this is the exact reason why the standup meetings exist.

Computational Platforms

Measuring the engineers isn't necessarily a process that needs to take place within the company or group. There are services who offer to measure engineers for you. For example you can install programs that will sit on a developers laptop and observe which websites they visit and which programs they open and how long they have them open. This data will be offloaded to their servers. From there they will analyse the engineers and see which are spending time on what applications. From there they can notify the team manager if for example some developer is spending a lot of time on social media websites instead of working on their integrated development environment or vice versa. Another example of the work being done elsewhere is if the data was analysed onto a cloud provider such as Google, Amazon or many other options. They offer distributed computing services, this means you can just send off the data to them. From there the data can be analysed, meaning that the data can be processed at a reduced cost for the company or group (as they don't have to supply the hardware). This makes offloading the data an invaluable resource to companies or groups who wish to measure their software engineers. Another computational platform for analysing engineers is to use the built in functionality of Github or Trello. These companies will provide ways of gathering statistics on how many contributions to the project each engineer makes. Github also allows repository owners to execute custom code whenever there is a commit to the repository using commit "hooks". This offloads a lot of the work and furthermore by automating it on Github's servers the group doesn't need to worry about hosting the code or servers. This can make the whole process more straightforward and easy for the group. Another system that groups use to for monitoring software engineers is Trello, Asana or Jira. These systems allow the project leaders or managers to see visualisations of for example how many tasks an engineer completed in a given time period.

Algorithmic Approaches

Once we know how we want to assess we can look at the sort of approach we want to take from a programmatic perspective. The goal of the program is to calculate if there are any outliers from the data set. A good way of judging this is by calculating the standard deviation of all employees. First we have to sum together for each employee how many contribution events they performed, be it tasks, commits or lines of code. This can be performed by just iterating over the list and building a new list or dictionary of employees to contributions. From here we can calculate the average number of contributions per engineer by getting the total number of contributions and dividing by the number of developers. From there we can calculate the standard deviation of each developer from the mean. This is under the assumption that the contributions of each developer are pseudo-random and therefore approximately form a normal distribution. If we then sort developers by their standard deviation it should become obvious if there are any large outliers.

Another approach to finding outliers is to use a correlation coefficient. In the same way as above we need to calculate how many contributions each developer produced. We assign each developer a number between zero and the number of developers. Then we sort the developers by the number of contributions they made (low to high). From here if we take the number of contributions to be the y values and the x values to be the number of contributions we should be able to use the formula for calculating a correlation coefficient to figure out how lined up all the developers are. If there are large deviations we should get a value closer to zero and if they are lined up we should get a value close to one. There are pros and cons to this. It can let you know when you do have to actually take a look at team performance and allow you to spot when there might be a problem instead of telling you there definitely is a problem with xyz individual. However if there can be a very steep difference between the worst and everyone is in a line between them this algorithm won't produce any warnings. One way this could be counteracted by employing a minimum number of contributions required. This means the system will also throw a flag for low contributing workers. This however might be down to a group where the understanding is that they produce fewer more meaningful contributions. So a better solution would actually be to measure the difference between the highest contributing worker and the lowest contributing worker. If this exceeds some number you could also manually trigger a flag. Again this could be a team where it is only expected to produce one or two contributions a day. Therefore even a disparity of 1 might cause a flag every single day. This means that realistically the best way to solve the problem is to find for the manager or observer, the lowest contributing developer or engineer each day and they can then make the decision on whether there is a substantial difference between them and the mean and whether it is accounted for.

Ethical Concerns

In recent years there has been huge improvements globally on the awareness of data protection and data privacy. People have become more and more aware of the type of information software can collect from them and the implications this will have on them. So understandably in software engineering measurement similar ethical concerns have arisen. The question arises is it reasonable that someone can measure to the second how much work you are doing? Is it reasonable that your employer can track your every movement and action? Well the easy answer is no, they shouldn't be because it's your life nobody should have that much information on you. But in reality employers are making an investment in a software engineer to produce some amount of work for an agreed upon cost. Since your work is now their money they should also be permitted to know what's becoming of their investment. This is why the whole agile process exists, why we have stand up meetings and task boards. The employer doesn't want to be swindled out of their money. But to what extent should they know. This is where the ethics comes in. They are certainly allowed some knowledge but to what extent. I think they have a right to know when you are working vs when you are browsing social media as this is clearly a violation of what you promise to do for them. Knowing your exact location at any time is too much information in my opinion. This is unnecessary as you could work wherever you please in software engineering and therefore knowing someone's location doesn't let them know if you are working or not. There is a caveat to allowing them any information. Why should they need direct knowledge of any level about how much you work on a minute to minute basis. Shouldn't it be plenty that the work is completed to the specifications on time? If it's not then the team has failed anyways and should get repercussions. If the work won't be finished on time the developers should be able to tell the client during the agile planning or specification phase. Thereafter they have taken on the responsibility to complete it. As an analogy you can imagine taking your car to a garage to have it serviced. You wouldn't stand there and ensure it's being worked on at every second of the day. You would assume that because you have payed for them to do it you trust them to complete the work. If they don't then you don't go back to that garage. Similarly I believe that if you are paying for software engineers to produce some code there should be a level of trust that they will complete the task. There are of course even further caveats to that. In particular it might be important for the team leader to know how their team is performing. So for that reason I do believe some level of measurement is required. So I believe that there is a requirement for the measurement but it must be done within legal bounds and the engineers must explicitly give their permission for it.

Conclusion

So in conclusion, the software engineering process can follow many different flows but the most common is the agile method, this method's development section is commonly the measured period. The development period can be broken down into several different processes; the version control system, task management system and standup meetings. These can then be either observed programmatically or observed manually and the general. Furthermore this data can either be assessed by hand or automatically with a program. If one does this automatically there are platforms available upon which this assessment can be run, eg Amazon Web Services, or places where the data can be pulled from, for example Github. In terms of the actual algorithmic approaches there are two main methods that can be used, either calculating the standard deviation or using the correlation coefficient to detect when manual analysis is required. This is all only useful though if the engineers actually consent to being measured or if the team actually wants some sort of measurement to be employed.

References

https://www.smartsheet.com/agile-vs-scrum-vs-waterfall-vs-kanban