Measuring Software Engineering

Daniel Nugent

18326304

Introduction

What is Software Engineering?

The Bureau of Labor Statistics (IEEE) defines software engineering as the systematic application of scientific and technological knowledge, methods and experience to the design, implementation, testing and documentation of software. Software engineering is a computer discipline. The term first appeared in 1965, in a list of services offered by companies in a June edition of Computers and Automation, but did not become prevalent until the 1970's.

Why and what should we measure?

I think the answer is both straightforward, and not at the same time. The straightforward answer is really why do we measure anything. Do we want to measure how good a software engineer is at doing his/her job? Is it more at an individual level or a team level? Do we only care about the quality of the code they write or also how they influence others at doing their job? Are we results driven or is it also important to know what behavioural patterns caused the success/failure of some software? These are some questions I think about what measuring Software Engineering should encompass. Circling back to the why part more, and I think when you observe what is happening software, we can start to get some idea into how important it is to measure it. If you compare how large applications are from maybe the early 90's to today, the size of the programs is probably the biggest disparity between the two. In the early 90's hard drive space was usually no larger than a few hundred megabytes and ram no larger than 16MB typically and that was considered A LOT at the time. Even the most basic applications nowadays would occupy more than the whole hard drive in the 90's. The scale at which codebases operate at nowadays is putting a lot of pressure on companies to track, document, format to just make it manageable for their software engineers to modify and add to the code base without it collapsing. If we don't have a way of understanding who is critical to the maintenance and development of such a code base and who is not, or even detrimental (by straining resources and being counterproductive) we are setting ourselves up to sub optimally develop software not only in terms of development time but also more important costs. It is essential that we can determine any leaks or issues within the development process and fix them fast. The need to automate such a system has become increasingly important as we don't want to have employ humans to monitor and evaluate the work of other humans who *should* be doing their job. And even if they are doing their job as requested, this may not be enough if the required infrastructure and version control training and oversight is not in place in this company.

<u>Measuring techniques and concepts</u>

Source lines of code

A commonly used method of analysing software engineering is Sources lines of code. It simply counts how many lines of code a software engineer has used in a repo or codebase to determine the productivity of that software engineer. It's very easy to see the flaw in this approach, but because of how easy it is to implement and understand the method, it is used frequently on repository hosting websites and version control systems such as SVN. I used SVN in first year and I remember the lecturer using this exact method to determine who on team contributed more than others. It was really frustrating because you could simply copy paste code or write long winded inefficient code to give the illusion of being a better software engineer. Bill Gates said that "Measuring programming progress by lines of code is like measuring aircraft building progress by weight". This approach is so flawed that I think it's laughable that it can be even though of as effective is any capacity. Source lines of code is very similar to saying "a book has to be x length long". It encourages filler text and discourages careful thought processes in the creation of the book.

I think when we think of which systems are used in the world today, too much emphasis is put on costs and overhead and often disregarding the quality of system which is being used, and Source line of code is a brilliant example of that. It is just so simple to count lines and display the information in a meeting of a nice chart to show who is contributing more than others to software illiterate people who are often in charge in determining which employees get promoted and which get let go, so in that regard I think it's irresponsible and unfair to use it professionally at all today.

I think what I'm trying to say is we need to properly evaluate what system should be used in every single regard and not overlooking any particular one.

Hours spent programming

I don't want to spend too much time discussing this metric because I feel like many of the arguments for and against Source lines of code as a measure of productivity in a software engineer also apply to this metric. The main thing that it tells us is how long a specific software engineer or many software engineers took to complete a specific task. We can use it in some capacity to determine which software engineers may be better at certain specific tasks. But I think issues start to arise when you realise what tasks are typically distributed in the software engineering field. Not everyone is given the same program to write. We tend to spread the work around to where everyone is doing something different as either an individual or a team.

Team A may be required to build a frontend and team B may be required to build a backend. There are so many factors in building each of these systems that it is very difficult estimate with good accuracy as to how long each of these tasks should take to complete. We have to look at previous experience of individuals on each team, cohesion between the team members and also the complexity of the system. So, it is therefore hard to say if an arbitrary number of hours to complete the task is "good". Putting pressure on people to complete a task can cause people to write poor code which mightn't work in every aspect (not enough testing) and not well documented or just being hard to understand. Rushing software tends to backfire and I think project managers should overestimate the time required rather than underestimating and then inevitably having to delay the project. Although hard deadline can motivate people to work harder also so it is a bit of a double-edged sword in that regard. It prevents procrastinating but may cause impulsive decisions to be made

Code Coverage

Provided that we can write meaningful test suites to determine programming accuracy at accomplishing a specified task, we can determine/measure a software engineers' accuracy or completeness of a problem via running tests on their code. The main disadvantage of using tests to determine accuracy is the fact that someone has to write these tests and their tests might have issues such as not testing as edge cases. Nearly all large software firms use test suites on a regular basis as they save time in the development process in the long term. Code coverage can give a reasonable estimation on how long it may take for an engineer to complete a task by using the current coverage and time worked on the problem to estimate how long it will take to fix all of the issues. Not only does it aid in potentially estimating completion time it also serves as a good metric of how competent a software engineer is at solving tasks and checking for completeness of problems they encounter. I.e. checking that the code they were told to write does what it was intended to do. It avoids many of the issues pertaining to source lines of code and time spent programming as metrics of software engineering. It is able to determine redundant code which doesn't run at all, and also point the software engineer in the right direction by telling them exactly what is wrong with their code / where exactly the issues are.

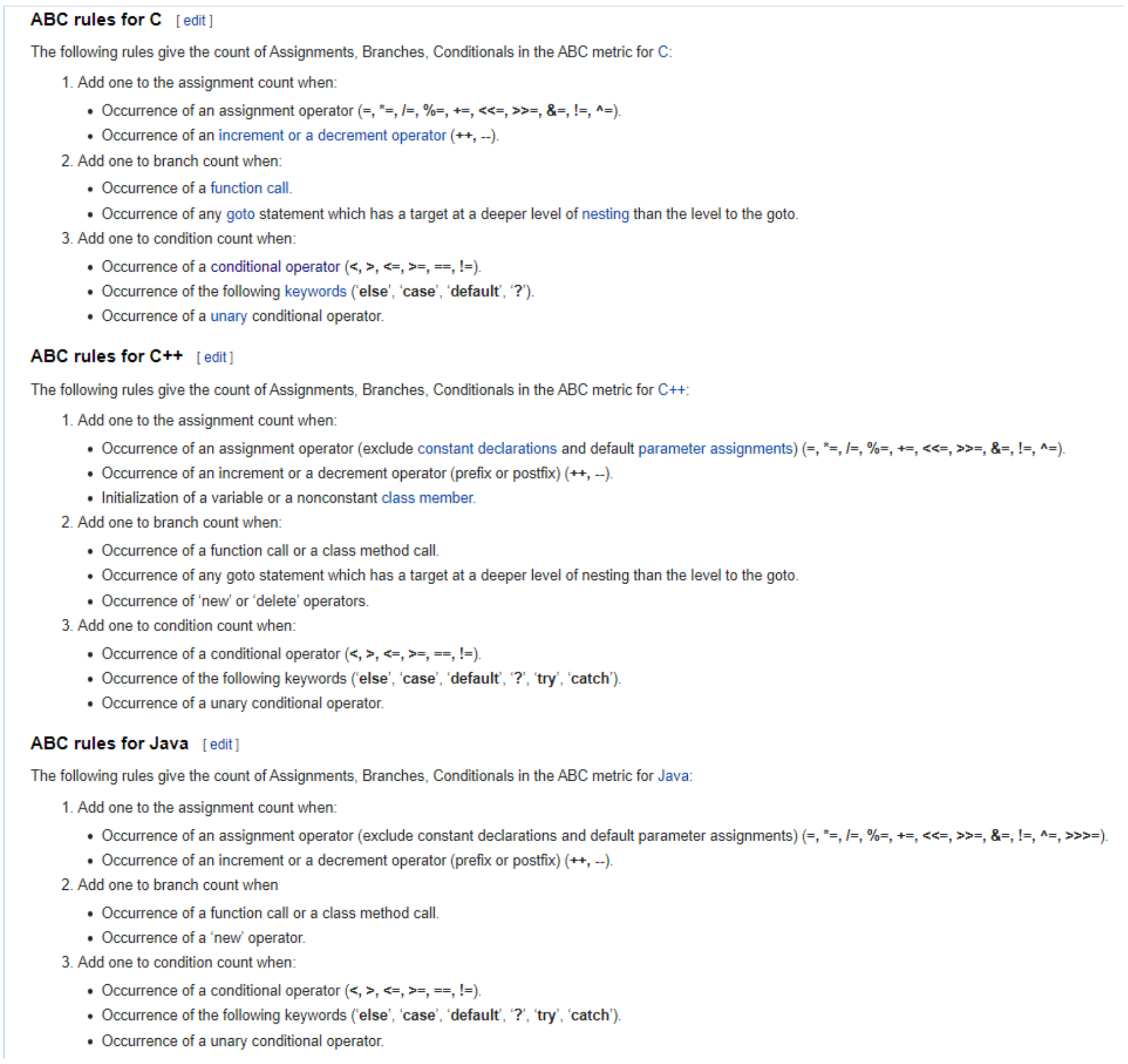
There are five main coverage criteria and they are Function coverage, Statement coverage, Edge coverage, Branch coverage and condition coverage. Function coverage checks that every function has been called, statement coverage checks that each statement has been executed, edge coverage checks that each edge in the control flow graph has been executed, branch coverage checks that each branch (conditional if/else statements typically) has been executed and conditional coverage checks that Boolean expression has been evaluated to true and false. One possible issue I see with code coverage is that not all tasks are created equally. So, it may be hard to compare different completeness results across projects. Another issue I see is that it doesn't consider the efficiency of a program. Maybe it is important that the code runs in no slower than $O(n^2)$. Overall I think code coverage is a step in the right direction for software metrics.

ABC Software Metric

ABC software metric is a much more recent addition to possible ways to measure software engineering. It was introduced in 1997 by Jerry Fitzpatrick to be an improvement over Source lines of code. The ABC software metric defines an ABC score as triple of values which represent assignments (A), number of branches (B) and number of conditions (C). It is a 3-D vector <assignments, branches, conditionals>. It is rounded to the nearest tenth by conventions.

$$| < ABCvector > | = \sqrt{(A^2 + B^2 + C^2}$$

Its purpose was to improvement on Source lines of code by improving upon its shortcomings. One such shortcoming was the fact that different programming style could result in more or less lines of code for the same result. So, by accounting for this, the ABC metric can very often give two programs which do the same thing, the same score by eliminating redundant code. It has a few notable advantages. One which we already discussed is being able to give different coding styles a more accurate score. It is very accurate at calculating how long a project will take to complete. This is done via estimating the ABC score for the project and getting a running average of the score on a particular day. It also improves upon the Source lines of code bug rate calculating which was previously calculated as *NBugs/LOC.* Now we calculate it by *NBugs/ABC Score.* It is a linear metric so files, classes, functions etc can be scored by summing the scores in a file of the sub modules such as classes, functions etc. This only apples to ABC scores, scalar ABC scores. The one last notable advantage over Source lines of code is being able to compare programs written in different languages as ABC scores use assignments, branches and conditionals to determine score which are part of all programming languages, however many languages use syntactic sugar to shorten code which could give the false illusion that one programming is a bad programmer, when in fact much of the work he does is in C or assembly whilst his colleague use Python or JavaScript. ABC has rules for different languages to constitute what is an assignment branch or conditionals (see picture next page).



ABC rules for C [edit]

The following rules give the count of Assignments, Branches, Conditionals in the ABC metric for C:

1. Add one to the assignment count when:
   - Occurrence of an assignment operator (**=, *=, /=, %=, +=, <<=, >>=, &=, !=, ^=**).
   - Occurrence of an increment or a decrement operator (**++, --**).
2. Add one to branch count when:
   - Occurrence of a function call.
   - Occurrence of any goto statement which has a target at a deeper level of nesting than the level to the goto.
3. Add one to condition count when:
   - Occurrence of a conditional operator (**<, >, <=, >=, ==, !=**).
   - Occurrence of the following keywords ('**else**', '**case**', '**default**', '**?**').
   - Occurrence of a unary conditional operator.

ABC rules for C++ [edit]

The following rules give the count of Assignments, Branches, Conditionals in the ABC metric for C++:

1. Add one to the assignment count when:
   - Occurrence of an assignment operator (exclude constant declarations and default parameter assignments) (**=, *=, /=, %=, +=, <<=, >>=, &=, !=, ^=**).
   - Occurrence of an increment or a decrement operator (prefix or postfix) (**++, --**).
   - Initialization of a variable or a nonconstant class member.
2. Add one to branch count when:
   - Occurrence of a function call or a class method call.
   - Occurrence of any goto statement which has a target at a deeper level of nesting than the level to the goto.
   - Occurrence of 'new' or 'delete' operators.
3. Add one to condition count when:
   - Occurrence of a conditional operator (**<, >, <=, >=, ==, !=**).
   - Occurrence of the following keywords ('**else**', '**case**', '**default**', '**?**', '**try**', '**catch**').
   - Occurrence of a unary conditional operator.

ABC rules for Java [edit]

The following rules give the count of Assignments, Branches, Conditionals in the ABC metric for Java:

1. Add one to the assignment count when:
   - Occurrence of an assignment operator (exclude constant declarations and default parameter assignments) (**=, *=, /=, %=, +=, <<=, >>=, &=, !=, ^=, >>>=**).
   - Occurrence of an increment or a decrement operator (prefix or postfix) (**++, --**).
2. Add one to branch count when
   - Occurrence of a function call or a class method call.
   - Occurrence of a 'new' operator.
3. Add one to condition count when:
   - Occurrence of a conditional operator (**<, >, <=, >=, ==, !=**).
   - Occurrence of the following keywords ('**else**', '**case**', '**default**', '**?**', '**try**', '**catch**').
   - Occurrence of a unary conditional operator.

ABC metric score rules for several programming languages

Halstead complexity measures

Halstead complexity measures are software metrics introduced by Maurice Howard Halstead in 1977. The purpose of these metrics was to identify measurable properties of software and relations between them.

There are several measures to be calculated

Base numbers:

- n_1 = number of distinct operators.
- n_2 = number of distinct operands.
- N_1 = total numbers of operators.
- N_2 = total number of operands.

Measures

- Program vocabulary: $n = n_1 + n_2$.
- Program length: $N = N_1 + N_2$.
- Calculated estimated program *Length:* $\hat{N} = n_1 log_2 n_1 + n_2 log_2 n_2$.
- Volume $V = N \; x \, n_1 log_2 n$.
- Difficulty*: D =* $\frac{n_1}{2} \; x \; \frac{N_2}{n_2}$.
- Effort*: E = D x V .*

Difficulty pertains to measuring the difficulty of writing or reading a program.

Effort pertains to measuring coding time using the following equation.

Time: $T = \frac{E}{18} s$

We can estimate the number of bugs in the code using the bugs measurement.

$$Bugs: \; B = \frac{E^{\frac{2}{3}}}{3000} \; or \; \frac{V}{3000}.$$

Halstead's complexity measures have some advantage over the more primitive measures of software engineering. First of all, it only counts distinct operands and operators, thus not counting "copy-pasted" code. This factors into the calculation of time taken and bugs estimated as reused code is far less likely to contain bugs in it or take a long time to write for a software engineer. It is effective of predicting the rate of error in a program and maintenance effort. Also, it can be used for any programming language. The biggest disadvantage of Halstead's complexity measures is the dependence of the complete code. You can't use it on a half-finished algorithm and expect accurate results on when you will finish the program. Overall it's pretty good at it's intended purpose of measuring certain things about a program such as error rate and completion time.

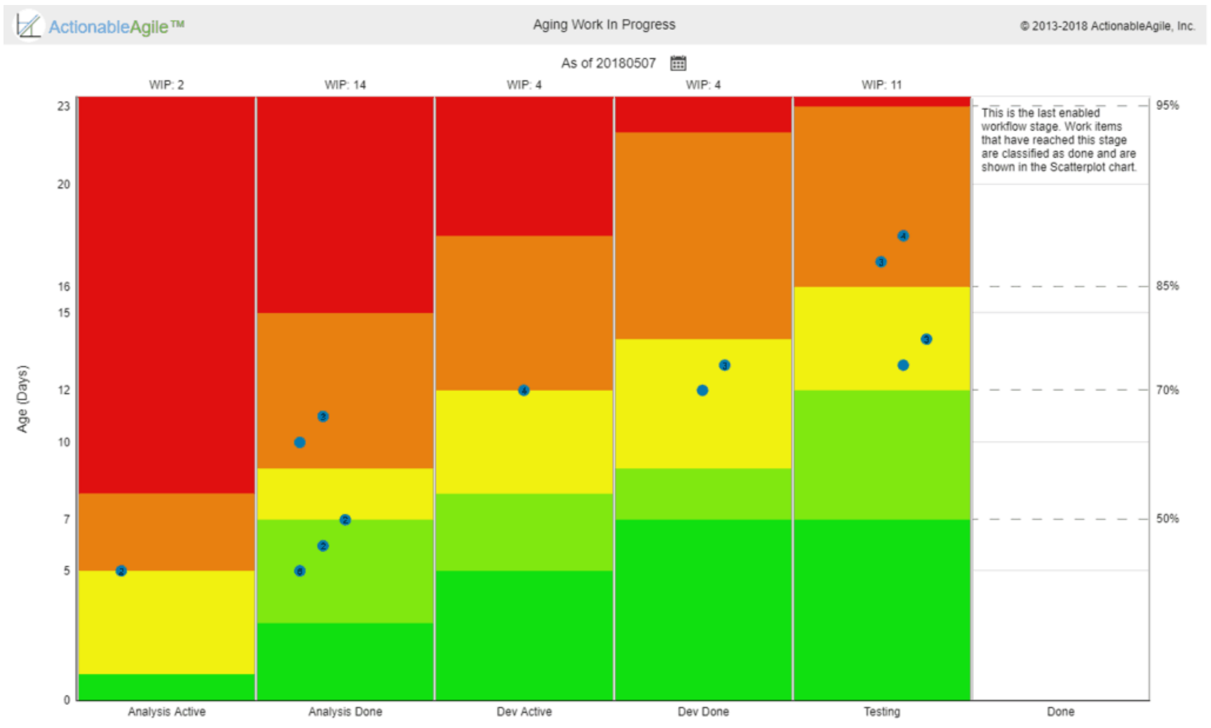
Flow metrics

Flow metrics are a collection of metrics which encompass the act of work being done and waiting periods in between such work being completed. A consistent flow is often required to stock bottlenecks slowing down production, such as for example on a car assembly line, where if there is a bottleneck in making engines, no cars can be completed and sold. Flow metrics are ultimately used to speed up product development and efficiency by eliminating such bottlenecks, and if such bottlenecks are identified, to do something about it (give more resources to those bottlenecks, are removing them from areas which are already performing very efficiently). Flow metrics don't only pertain to software engineering, they can be used in many industries which is an advantage to them as we saw in our earlier example.

The first flow metric we are going to be looking at is Work in Progress (WIP). This involves tracking work that has started but has yet to be complete. I think it's safe to say that uncompleted work is of no value to a software company when it comes to customers or profits. By keeping track of what work is in progress we can analyse if we are being spread too thin by having "too many tabs open at once", as if we don't have many completed tasks, but many are marked as being WIP at once, this could be an issue as they are not being useful for customers or the business.

Next, we will look at how queues can be beneficial in a flow metrics context. Queues tend to form when work waits between different stages. Often queues for the vast majority of work item's total lifecycle so it is prudent that how they affect your engineers and when they tend to occur. We can use efficiency diagrams to compare and contrast between WIP and work in queues. This is useful so we can locate where exactly there are flow bottlenecks in our development and allocate resources to fix such bottlenecks. The speed of a products development is onlyh as fast as the slowest moving part, so instead of rushing certain aspects of the product (software in our case), we should spread our resources as evenly as humanly possible instead to get the product into customer's hands more efficiently and faster. Cycle time is another term used for queues in flow metrics.

Lastly, we will look at Work Item Age. Essentially this measures the elapsed time between when the work item started and the current time. Time contrasts queues in the sense that it provides transparency in which items are flowing well and which aren't. This is the best metric to use if you want to determine when a work item that has already started will finish. If the work item hasn't started yet, it would be better to use the previously aforementioned queues or cycle times.



A visual graphic or Work item age in an agile context.

Online platforms

Github

Github is a popular online platform used to host the repositories of software engineers around the world. Github by default has a few built in metrics that can be used to track commits made by other developers in a repository. By default, it has basic functionality for metric such as commit dates, percentage of code base committed / source lines of code, and overall is useful to get a gist of who are the main contributors to a repo and who are less so.

Thankfully, many 3rd party online platform can be integrated with Github to seamlessly improve the overall experience for end users. One such platform is called GitColony. One such powerful feature provided via GitColony is partial reviews. This can be leveraged to analyse and check code before a pull request is made to ensure that no obvious bugs are present in code and will save time in the future from a software developer to review the whole pull request. These partial reviews are saved so you always know if you have already partially reviewed a certain section of code.

GitColony mentions a very powerful quote on their website which really resonated with me. “Ask programmers to review 10 lines of code, they’ll find 10 issues. Ask them to do 500 lines and they’ll say it looks good”. Obviously, this is a marketing ploy they leverage to market their features, but it definitely has some truth behind it.

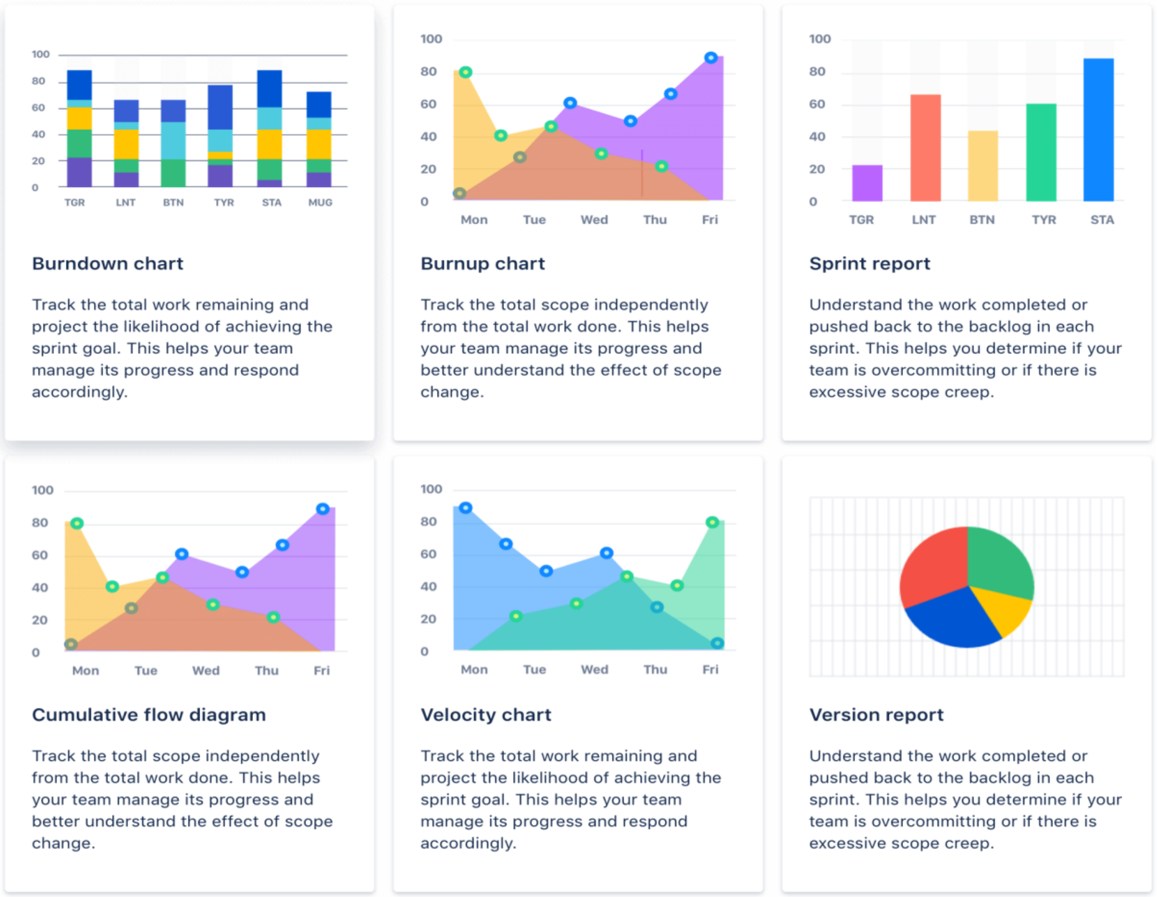
Another cool feature on their platform is that code reviews can now be fully traced. We can see exactly when and who reviewed certain code. This allows us to easy track potential issues that arise in future to be solved by getting to the root source of the problem quickly and discreetly.

Virtual pull requests and Linked pull requests are two more powerful features that help us save time. Virtual pull requests occur on GitColony and are only visible in a vacuum, essentially meaning that they can’t be accidentally merged by anyone and break a build. It allows people to vote on if they think it should be merged in, which can help with disagreements and save time. Linked pull requests allow you to link an external repository on GitHub as a pull request from the same origin branch back into main repository branch. This can allow developers to have their own private repository and do testing on it themselves without potentially interfering with other developer’s work. Overall, I think GitColony is a strict improvement on GitHub with additional features which help track, manage and effectively measure work being done on a repository. It has a free trial to test the waters to see if it suits your needs before paying. It does not have a free tier which is a disadvantage of it.

Jira

Jira is a software development tool aimed at Agile development. It has many features such as scrum boards, Kanban boards, roadmaps and Agile reporting among many others. The scrum board can be used to show and document what work is in progress, what work has yet to be started and what work is done, all at a glance in a nice interface. Users can also see pull requests, commits, comments, branches and deployment status for each work item.

The Kanban boards work very similarly to the scrum boards. The roadmaps feature shows when different work items are estimated to be completed by and when new ones are to be started. The Agile reporting section is where the true power of Jira is in a measuring software engineering context. It has many eloquent graphs which can be used to predict which work items may be late or not on schedule which we have already discussed can have detrimental effects on the success of a project. Overall Jira is an excellent online platform for both hobbyists and more professional oriented workloads. It has paid and free tier options for both the new programmer and the enterprises looking to leverage the capabilities of Jira.



Burndown chart

Track the total work remaining and project the likelihood of achieving the sprint goal. This helps your team manage its progress and respond accordingly.

Burnup chart

Track the total scope independently from the total work done. This helps your team manage its progress and better understand the effect of scope change.

Sprint report

Understand the work completed or pushed back to the backlog in each sprint. This helps you determine if your team is overcommitting or if there is excessive scope creep.

Cumulative flow diagram

Track the total scope independently from the total work done. This helps your team manage its progress and better understand the effect of scope change.

Velocity chart

Track the total work remaining and project the likelihood of achieving the sprint goal. This helps your team manage its progress and respond accordingly.

Version report

Understand the work completed or pushed back to the backlog in each sprint. This helps you determine if your team is overcommitting or if there is excessive scope creep.

Some of the Agile reporting graphs on Jira's online platform

Conclusion

In this assignment I was tasked to write about several of the methodologies and online platforming tools available to measure software engineering. They range in capabilities and applications drastically as we have seen. From primitive measuring techniques such as Source lines of code and hours spent programming to far more intricate and complex ones such as ABC metrics and Halstead's complexity measures. There are many online platforms to enable enterprises and new developers to perform their job in a more synergistic and comprehensive manner. They allow tech leads to analyse and quickly resolve issues in the code base and in new pull requests from developers. I think there is a tool or methodology for any developers regarding skill level or price-wise available to use which is indicative of the advancements in software development in the past 50 years or so.