Mike Kelly
18318983

Software Measurement

Introduction

To make progress, it is important to compare different states of the journey. Comparison is essential, as there may be many different routes to the goal, and one can spend their time most efficiently on the route that yields the most progress. This might be seen in weightlifting, where progress can be measured in weight lifted, and the routes being the regimens used during training. The effect of the regimens might vary depending on the stage of progress, and so comparison and variability of routes (to obtain different values on which to compare) lends itself to optimization. This is a very fundamental principle which can be applied almost everywhere. It is found in nature, in which plants utilize tropism such as phototropism to grow towards sunlight (in which the shape of the stem curves towards sunlight, using energy generated from photosynthesis as its measurement). This can and be used anywhere measurement is possible. In areas where measurement is easy, it is easy to perform large scale comparison to find value, like the most valuable salesperson in a company. One company may try to poach valuable salespeople to gain an advantage in the market. Here we have outlined the possibility and values of mensuration. However, fields such as software engineering, in which it is difficult to measure the value a programmer brings to a project (even if the value of the project is easy to measure), it can be difficult to identify optimal approaches on an individual level. We might be able to identify one project as having more value than another, but it is difficult to know why without careful analysis. If we want to more easily identify how to make good software, we need to become better at measuring the value programmers bring to a project, and then identify the routes they take. Therein lies the problem (and possibilities) of software measurement.

Software Development Lifecycle

The SDLC is a description of the phases that software goes through in its lifecycle, upon which methodologies for creating and maintain software are based. There are typically six phases a piece of software will go through (any amount of times).

- Planning:
  Planning is when senior members of the team collaborate with clients and auxiliary teams such as sales and marketing to investigate what approaches are available to the project, and verifying its feasibility. Consultants may be used to identify risks and advise on optimal approaches.

- Defining:
  After the requirements analysis, the product and its requirements must me strictly defined (Software Requirement Specification), and approved by all collaborators such as the team, the team's organization, and the client. This ensures that all participants are ready to move forward with the project.

- Designing:
  A Design Document Specification (DDS) is made, with a draft of possible approaches to the project, in order to meet the SRS. Stakeholders discuss and agree upon a design for the project, based upon their preference for risk, robustness, modularity, budget, and time. The internal design of the project is planned as closely as possible.

- Building:
  The DDS is materialized into actual code. This phase is intended to be straightforward, as if the DDS is detailed, it is easy to map the design into code. This phase primarily involves the development team, with less input from other stakeholders.

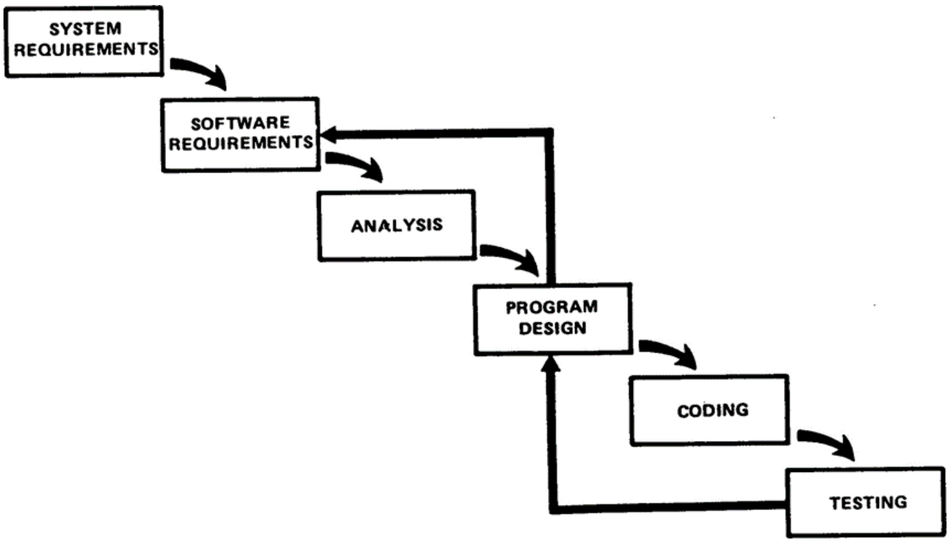
- Testing:
  While testing can be performed at all stages of the software lifecycle, it is important to thoroughly test after building, as it is vital that all defects that have been overlooked or accidentally introduced our found and resolved before moving onto the next stage.

- Deployment:
  Deployment can be realized in a multitude of ways. If the stakeholders are confident in the project's integrity, it can be released outright. However, if they are more cautious, they may opt for a phased deployment, in which issues can be found with the less impact to users. This is also the longest phase, as it includes the maintenance of the product. Generally, the team size is reduced after deployment. Trade offs and code-debt tend to appear in this phase and may increase the maintenance cost.

SDLC Models

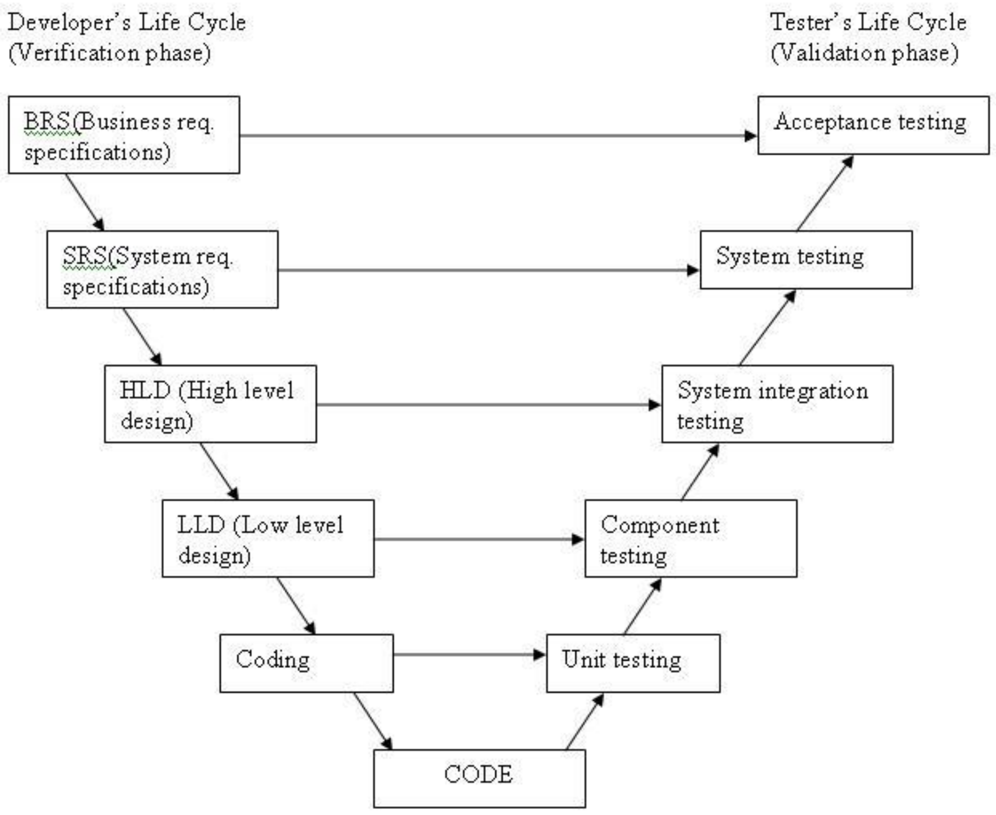
Depending on the requirements of the project, different models for its lifecycle may be better suited to its realization. All stakeholders must understand and approve of the model taken, as there will be different expectations of everyone's involvement at each stage of the SDLC depending on the approach. The attractions and risks of each methodology will be outlined.

- Waterfall



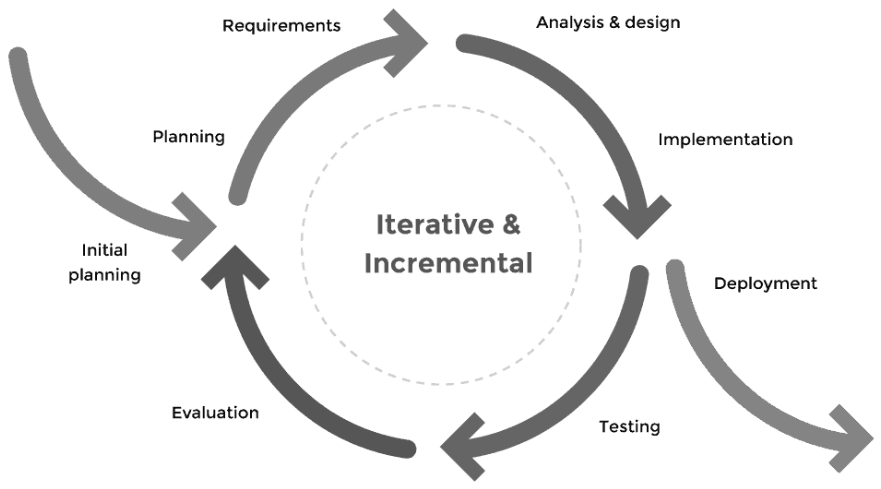
Waterfall is among oldest software development model. It was first explicitly defined in Royce (1970), although not by the name of waterfall. In it, Royce outlines distinct phases that are performed to their completion before moving to the next phase. The requirements and design phase can be revisited should a flaw be found. This methodology has the benefit of clear, and easily identifying progress. Large teams are able to coordinate more easily, as everyone knows what is expected of them at all times. However, this strictness comes at a cost, the largest of which is that while it is easy to ensure a product fulfils its initial requirements, it does not assist in finding accurate requirements, leaving them ambiguously to what the stakeholders define at the beginning, when they might not know exactly what is needed. Changing existing requirements is also expensive, as reverting to a previous state is impeded by the cost of throwing away what was done in the meantime. The result is often software that does not meet what the client truly wanted.

- V-Model



The V-Model, consisting of pairs of phases, each pair having a Verification phase and a Validation phase, is one of the successors to the Waterfall method. Developed simultaneously and independently in the early 1990's by the American and German military, this methodology allows for users to participate in the development and maintenance of the project. It also helps ensure that the project is on an acceptable path, as testing is rigorously performed with each phase. It has been criticized for being a naively simple model of software development, inflexible, and for lacking coherence, as there are many interpretations of the model. Any version of the model that is universally accepted is too trivial to make use of. Therefor it is important that the version used is well understood by stakeholders.

- Iterative



The iterative model has no single source. It was experimented with and practiced in a variety of software projects, and in other fields. The idea of the model is to develop a system in iterations, in which each iteration builds upon the previous one. With this, the user is involved in every phase. Every increment of the project is delivered to the user, allowing the project to coincide with what the user wants (even if it is not known at the beginning of the project). A big attraction is that a working product is delivered with each iteration. It may not have realized all of the requirements, but it will be functional.

Software Metrics

In order to do anything meaningful with software metrics, they must be computable, objective, easy to obtain, and most importantly, relevant. If the metrics are computable, then it is easy to optimize large scale software projects. Metrics must be objective, so that that the measurements possess integrity. They must be easy to obtain, as otherwise measuring the software development process may distract from the development process itself. Finally, relevance is important, as otherwise development will be altered to satisfy arbitrary metrics that won't add any value to the project. Few metrics fulfils all of these requirements, however a combination of metrics, used correctly, may provide useful insight into a project.

The most fundamental metric is lines of code (LOC). There are several definitions for what counts as a LOC, for example whether to include comments or not. More strict definitions tend to be more relevant than others. Properties of LOC are that it is computable, objective, easy to obtain, however of little relevance to the value added to the project. It may be used in conjunction with other metrics to discern value. If one task is written in twice, with each version a different length, it may be desirable to choose the shorter version, as it may be more efficient and easier to obtain. This should only be used as an indicator however, and caution should always be taken when using LOC as a metric.

The number of commits shows how frequently a programmer saves his work with a revision control system. On its own this has little relevance, as it is each programmer's preference to make large or small commits. However, it may be correlated with quality of code, and smaller commits are potentially a good practiced that can be learned by others.

Lead time is the amount of time it takes for a piece of software to be developed and delivered to a customer. This fulfils all of the desired properties, and it is clear that a shorter lead time is preferable. It represents how responsive the team is to the client.

Time spent per tasks is computable, objective, and easy to obtain. It is relevant within a team and has the most impacted when analysed as a trend, instead of individual values.

Team velocity is the amount of software units a team completes in a sprint (Agile methodology). It is a discrete number, so it is computable, objective (to the extent that a unit is agreed upon as completed), and easy to compute. It is most relevant to the team that it is taken. It can't easily be compared to other teams, as the units they are working on may differ in many ways. It is a good approximation for progress in a team.

Code churn and efficiency are closely related. If someone writes a module, that later has lots of additions, modifications, or deletions, it may imply that the original code was inefficient. This is a discrete number, and is relevant, as it tends to find inefficient modules.

Application Crash Rate is simple metric in which the application fails is divided by its uses to determine its likelihood of crashing. This is one of the metrics used to determine if an application is acceptable to deliver.

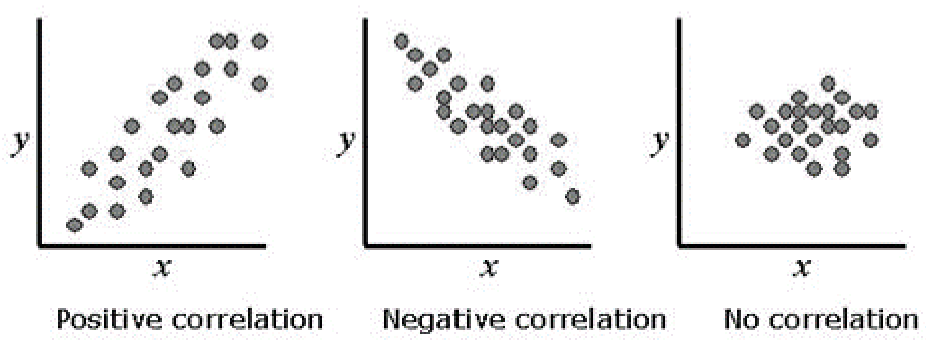
Function points are a metric to identify how useful a product is to its users. It is relevant, as it shows how much the application offers the user and is easily computed. It is not objective nor easy to obtain though, as there it cannot be measured directly. It does have the benefit of being a metric that applies across platforms.

Algorithmic Approaches to Measuring Software Engineering

There are a number of approaches to use the aforementioned metric to produce more relevant metrics which represent software engineering as a whole, as opposed to components of it. There are a number of machine learning and statistical methods to identify individuals in a team that are the most impactful. By identifying these individuals, we can compare what they are doing against what others are. Hopefully, this will find habits that more successful programmers possess, and are able to teach to others in order to improve team efficiency. Algorithms also provide the possibility of identify tasks that are likely to encounter difficulties before the team notices.

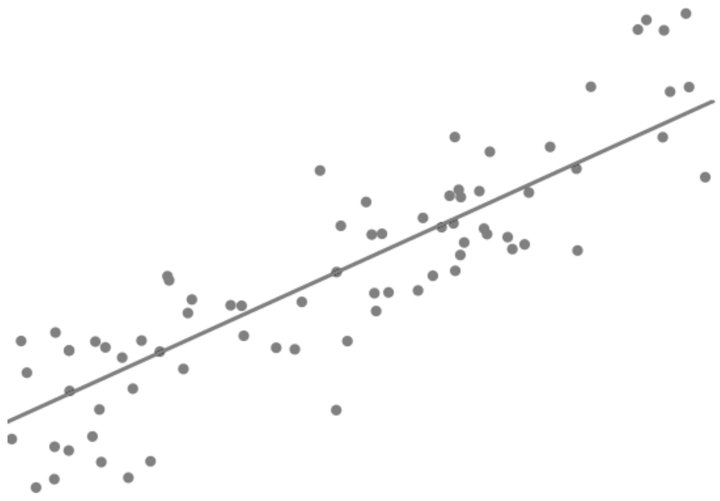
- Identifying Difficult Tasks
  One method of finding individuals with the greatest impact is identifying who completes the hardest problems. While this is a very ambiguous metric, an easily computable proxy can be used. If it is assumed that easier tasks are finished quickly, than tasks with a longer completion time are more difficult. If we were to give each tasks a value of one when it is first planned, and increased by one each subsequent day, and award the current value of the tasks to the programmer who completed it, we can compare the maximum value awarded to each programmer. This is not a precise assumption, so care is needed when taking this approach, however it is very easy to implement, and helps incentivise a team not to leave tasks to the side.

- Correlation



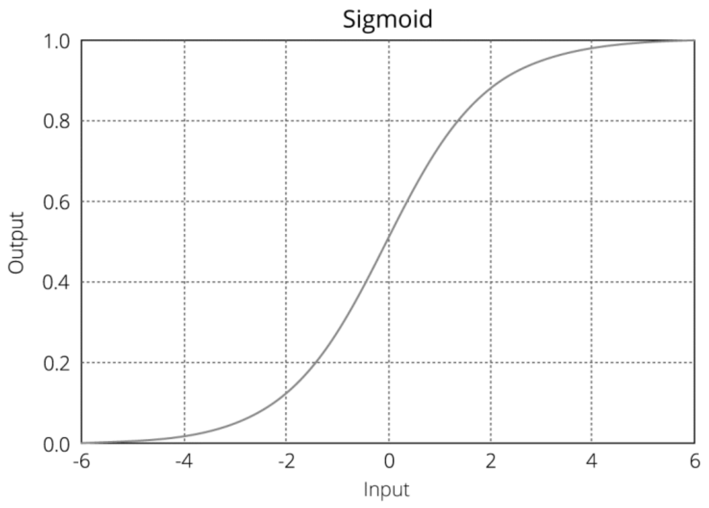
While very basic, by examining the correlation between metrics, we are able to decide what metrics are best used in subsequent algorithms to optimize software development.

- Linear Regression



An approach to find the relationship between two variables is linear regression. If we use metrics we desire to change, such as lead time, and time spent per task, we want to identify other metrics which correlate with smaller outcomes. An easy example is runtime and programming language. For similar tasks, some programming languages will tend to result in less time to complete the same task. By analysing this, we can choose languages which better achieve our goals. Another example is comparing time spent per task with the number of commits.

- Logistic Regression



If we are examining a binary result, for example if a task was completed in its sprint, we can use logistic regression to compare parameters with the likelihood that it was or not. Using this, we can identify tasks that are likely to not meet its deadline, before the sprint is finished, allowing extra resources to be diverted towards it. An example of a possible parameter is chode churn, or perhaps even just code deletions.

- Bayse's Theorem

$$\overset{\text{Posterior}}{P(A|B)} = \frac{\overset{\text{Likelihood}}{P(B|A)}\overset{\text{Prior}}{P(A)}}{\underset{\text{Normalizing constant}}{P(B)}}$$

$$P(B) = \sum_{Y} P(B|A)P(A)$$

We can use Bayse's Theorem to predict the probability of an event occurring, if another known event has occurred. This may allow us to gain a more accurate estimation of project delays as a result of sick days for example, or unexpected time off.

Computational Platforms

There already exists several computational platforms that are available for software engineering teams to utilize and increase their efficiency. Often seen as the successor to more personal assessments of progress, such as code reviews and the Personal Software Process (PSP). Most come with the recommendation to use them as supplementary, as opposed to replacing existing practices. Their goal is to capture what other processes might miss, or to scale their effectiveness.

- Hackystat
  "A framework for collection, analysis, visualization, interpretation, annotation, and dissemination of software development process and product data." – HackyStat.

  Hackystat is a tool that directly obtains metrics from the tools a programmer uses, to obtain data. Emacs, for example, is a text editor which Hackystat can obtain raw data from, such as lines of code, tab switching, and time spent per session, and uploads it to a web server which can be queired. This allows the programmer to see their metrics and allows them to perform further analysis or visualization.

- Stackify
  Stackify is a commercial product with the goal of increasing software developer efficiency. It provides app performance measurement, code profiling, error tracing, and other insights to assist a developer in their work. Rather than analysing the project abstractly, Stackify is a tool for providing concrete insights to assist the developer, helping increase the efficiency of not just the developers time, but also their code.

- GitPrime
  GitPrime provides a dashboard of a project's commits, ticket activity, and PRs. It provides a useful interface for a project manager to get an insight into what a day involves for their team, and to more easily identify bottlenecks and where progress is flowing smoothly.

- SonarQube
  SonarQube is a platform for continuous inspection of code quality. It provides automatic reviews which detect bugs, defects, and duplicated code. It can also be used to assess test coverage, complexity, and security vulnerabilities.

- Code Climate Velocity
  Velocity uses data from revision control systems to provide teams with data on continuous delivery, process effectiveness, bottlenecks, and personal development. It is intended for project managers and engineering executives who are concerned with long term goals of their company.

Ethics

For personal reasons, developers should be concerned with the ethics of the measurement of their work. They hold the right to be concerned for what information is taken from them by their employer. Some aspects they have the legal right to retain, under General Data Protection Regulations. What remains is data that the company has the legal right to, if they seek it, such as metadata of their work. However, the employee reserves the right to cease their employment if they are not comfortable with employers having this information. In the worst-case scenario, data an employer collects can be used to manipulate employees to the company's goals without the knowledge of the employees themselves. Some managers may wish to protect company interests by distancing disruptive or dissatisfied employees to prevent unionization or organized resistance to company policies. Technology can empower this, so employee rights are protected by GDPR to prevent their personal information, such as messages to be used against them. Despite this, there is still a lot of tangible good that can be gained from software engineering measurement, that is to the benefit of all parties. Through careful use of software metrics, in such a way that incentives employees through support (as opposed to causing them resentment by using as a cudgel), managers are able to optimize their team and help their employees along the way.

Conclusion

Software engineering measurement is an extremely promising tool, which can help facilitate companies increasing their efficiency, and programmers in learning and adapting more quickly. There are a variety of options available, some more closely related to employee's personal details than others. With proper information and consent, they present an opportunity improve a team with little cost. They are quickly improving, with much fewer manual data gatherings than in previous versions and provides the ability for programmers to work alongside the tools to ensure code is reliable. Trust is necessary that the company won't misuse any data it collects to ensure a healthy relationship.

Bibliography

Code Climate Velocity. Retrieved from https://codeclimate.com/velocity/pricing

GitPrime. Retrieved from https://stackshare.io/gitprime

HackyStat. Retrieved from https://hackystat.github.io

Royce, W.R. (1970). Managing the Development of Large Software Systems. *Retrieved from* http://www-scf.usc.edu/~csci201/lectures/Lecture11/royce1970.pdf

SonarQube. Retrieved from https://www.sonarqube.org/developer-edition/?utm_source=bing&utm_medium=cpc&utm_campaign=SonarQube-Bing&utm_content=sonarqube

Stackify. Retrieved from https://stackify.com/