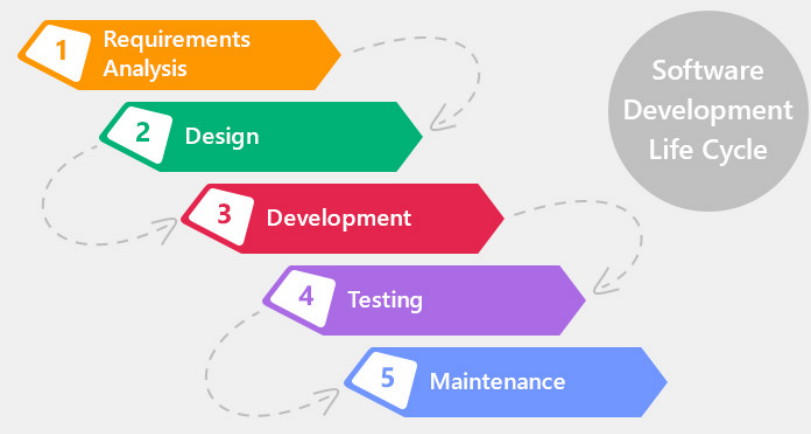
Tiernan O'Connor
16319419

To deliver a report that considers the ways in which the software engineering process can be measured and assessed in terms of measurable data, an overview of the computational platforms available to perform this work, the algorithmic approaches available, and the ethical concerns surrounding this kind of analytics.

This report has been written in order to understand and appreciate the different ways in which software engineering as a process, can be measured and assessed with regards to measurable data. I will also discuss the various platforms which are at the disposal of every software engineering, providing them with a base on which to perform their work and the different algorithmic approaches available. To conclude, I will finish with the ethics behind this process and how it is valued in today's society.

In order to discuss how we can measure the software engineering process, I will begin by explaining what this process consists of. The software engineering process involves the division of work into several subsections or distinct phases, in order to increase efficiency and maximise project output. This process is sometimes called the life cycle and is used on the development of a software project. The software engineering process is a complex and long one, consisting of many different activities and stages.



Requirement Analysis: This first phase of the software development lifecycle involves extracting the requirements of the desired software product. The project manager and stakeholders are focused primarily on this stage. It is essentially market research is undertaken, similar to that prior to the launch of any product in which questions are asked in order to justify the feasibility and demand for such a product. What will it do? Who will use it? What input will it have? What output will it provide? These sort of questions are asked in the analysis stage and the possibility of including all these requirements into the system to be designed is investigated. Finally, specification occurs which consists of putting together a guideline or exact description of the software which needs to be written for such a system. This involves complex mathematics and fine-tuning. The document created then serves the purpose of guideline for the next phase of the model.

Design: In this phase, the system and software design are prepared from the requirement specifications which were studied in the first phase. is concerned with making sure the software system will meet the requirements of the product, as well as ensuring that future requirements can be addressed. System Design refers to an abstract representation of that system. The system design specifications serve as input for the next phase of the model.

In this phase, test strategy is devised, indicating what needs to be tested and how it should be tested.

Development: The development stage consists of the most obvious aspect of the software engineering process – the coding. This phase is, of course, the top priority for the software engineer or developer. A common misconception is that this stage requires the most work, however, this is most commonly not the case. Ensuring the first two phases of the process are carried out correctly ensures that the coding of the system can be a smooth task. Reducing the product design to a block of code is easier when the guideline produced in the analysis phase can be accessed and the test strategy has already been devised.

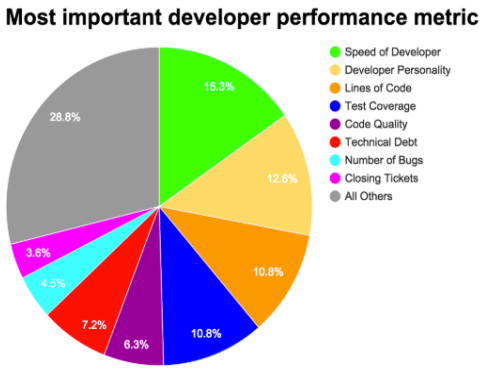
Testing: This phase of the software engineering process revolves around checking for bugs and errors in programs and is particularly useful when code created by a group of software engineers must work together in harmony. The first prototype of the system will be beta tested by customers in order to establish any initial bugs and the error handling of the system. These issues are sent to the engineering team who will solve them before the finished product is sent out to the customers.

Maintenance: This aspect of the software engineering process is undoubtedly the most pivotal. After the system has been deployed and people who are to use it finally have access to it, training and support are vital to the success of the product. People are often hesitant to change and so, software products can often end up being unused by people who don't know how to use it or who haven't received the necessary training. If the relevant training and support are offered to those the product was built for ensures there is no confusion in how to use it. Maintenance ensures software can deal with newly discovered problems and requirements into the future after the initial development of the software.

The software engineering process is a long one, consisting of different stages in order to ensure that each aspect of the process can be carried out in an efficient and hassle-free manner. Essentially, every phase of the process is there to make the next phase easier for the relevant team carrying out the task, be it developers, market researchers or strategists. There are many ways in which this process can be assessed in terms of measurable data

Measuring Data

There is no denying that data is the most crucial component of the software engineering process. Another vital component is measuring this data. What makes a software engineering effective and reliable? How do we calculate this? It's not an easy task. There is not just one way to measure the performance of a software engineer. For comparison, how does one pick the best model of car? There are many factors which we take into our decision. Questions that could be asked maybe – does it look aesthetically pleasing? How fast can it go? How environmentally friendly is it? What is its zero to sixty time. Similarly, there are many factors we must take into consideration when assessing the ability of a software engineer. One such factor we must take into consideration is, of course, the quality of their code. Assessing the quality of their code is similar to assessing the quality of any type of data. Does it make sense? Is it as concise as possible? Did they source it from somewhere else? Measuring data in the software engineering process is vital. There are several different methods used to assess the quality of code, with many debates around the effectiveness of these methods. Some of these methods can be seen in the below diagram which I will proceed to discuss.



Number of Commits: The number of commits refers to the number of times the code has been altered in any form. This can include removal of code, the addition of code or changing existing code. When assessing the quality of an engineer's work, it is always better to see a large number in comparison to a small number of commits. A large number of commits doesn't imply that the software engineer is a good one however it does imply that they are doing a lot of work and focused on the project. A large number of commits shows that the author has been consistently editing their code and removing any bugs, fixing errors and overall increasing the efficiency of the program.

Lines of Code: This is a software engineering metric used to measure code based on the size of it i.e. the number of lines present in the program. I think it is safe to say anyone who knows a thing or two about programming knows that condensing the number of lines of code is better than adding more lines. Again, a small number of lines does not necessarily imply that the code is of a high quality, however, it is a good way to measure the efficiency of a developer. The source lines of code come down to the big debate – Whether it is better to have created x lines of code than to have cut x lines of code from a program. LOC is often used in these kinds of arguments, where developers and other related personnel talk about whether "bigger is better" or whether a software codebase should "go on a diet."

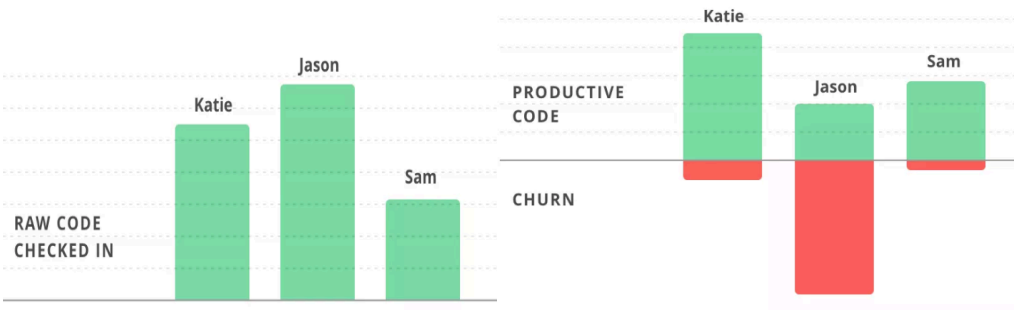
Lead Time: Lead time is defined as the time that elapses between the establishment of a specific requirement and the fulfilment of this requirement. Put simply, the lead time is the time between a developer being made aware a certain program must be written and the time when the writing of said program is complete. Of course, lead time can vary greatly depending on factors such as the size of the requirement, or whether the project is being carried out by an individual or a team. Thus, lead time can be viewed as an ineffective way to measure the quality of code, however, if there is an initial estimated lead time to compare it to, it can be just as effective as any of the other methods listed above.

Code Debt: Think of the following scenario. You are aware that a certain aspect of functionality must be added to your project and you have identified two ways of doing so. The first method will be quick to implement, however, it is messy and may be difficult to alter in the future. The second is clean but will take much longer to implement than the first method. Ward Cunningham provided a metaphor for code debt. It was as follows - to do things in a quick but messy way will provide us with technical debt. This can be compared to financial debt in that the interest payments we must pay for technical debt arise in the form of the additional time taken to fix issues brought on by the quick but messy code that was chosen. In saying that, it was also mentioned that doing the quick and dirty method may somethings be a sensible thing to do. If a program needs to have added functionality in a short period of time, the quick and dirty method is the way to go however the problem arises

when organisations rely on the method too much and are faced with prolonged periods of interest payments.

Bug Fixing: This refers to the improvement of the code in terms of the removal of bugs and the time a software engineer spends fixing bugs. This can be assessed in many ways. The decreased number of bugs found in future uses of the code shows the quality work done by the software developers in terms of their time spent refactoring and architectural improvements. Another way of measuring the level of bug fixing present in a program can be to look at the ratio of bug reopening. The ratio compares bugs which were supposed to be fixed but was reported again against usual bugs. A decreased ratio of bug reopening shows the developer has spent significant time error checking. When the product is being used by the customer, another way to assess the level of bug fixing is to compare the number of bugs found by the development team and the number of bugs found by customers.

Code Churning: This refers to when an engineer rewrites their own code in a short period of time. It can also be explained as code that has lines added, modified or deleted from one version to another. The main purpose of measuring the level of code churning is to allow the management team to assess the quality of the code being produced by developers. Think of it as writing a cover letter and then deleting it and starting again. When you finished you decide to rephrase the entire thing. You technically wrote three cover letters, but in the end, only one was sent to the employer so they are only familiar with one cover letter worth of 'accomplishment' from all that effort. The same is true with code. Raw code can consist of lots of churning, which the below example illustrates. Thus from looking at the raw code, we cannot assume that Jason has done the most work because as we see his raw code consisted of roughly 2/3^{rds} of churn.



When churn starts to increase, this can be an indicator that something is going wrong. High levels of churning can indicate that the developer is having issues identifying the solution for a specific problem. A high churn rate may also mean that a developer is under-engaged or represent the presence of an indecisive product team that has the developer running in circles.

Team Projects vs Individual Projects: Whether a project is carried out by a team or not has an impact on measuring the quality of the code produced. When assessing teamwork, there is the hope that a team has been working so as to cross-check their own code between themselves whereas you don't have this when a software engineer is working solo. Whether or not the project has been conducted by a team affects the relevance and the occurrence of the other metrics listed above.

Clearly, there are several metrics to consider when assessing the quality of code. There are many different ways to monitor the performance of a developer, with methods having their own respective pros and cons. Although it is important to measure the quality of the code, it

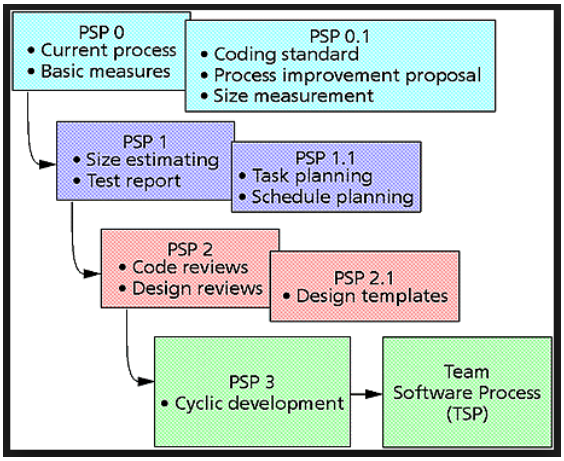
can sometimes appear that these metrics may not be suitable in measuring ability for some specific projects.

Available Platforms

After having talked about assessing code and the metrics used to do so, the question of - where can work be shared in order for it to be so easily accessed and assessed – may be asked. Since the break of developing and the software engineering era, the number of platforms for the code to be written on has skyrocketed. There are now immeasurable amounts of platforms at the disposal of developers which can be used to work from. In this section, I will shed some light on the most popular platforms used today and they aid the software engineer in increasing efficiency and productivity. Developing processes which can improve code quality has been a thing for many years. Researchers and industrial leaders began to realize that software process, plans, and methodologies for producing software, could help to produce accurate project deadlines, help keep software projects on budget and teach programmers to be more productive and knowledgeable of their own programming style.

In 1976, Deming & Juran introduced software inspections, paving the way for the many other processes and platforms. Organisations used these inspections in order to improve software quality. Some years later, the Capability Maturity Model (CMM) surfaced, focusing on the management system and the support and assistance provided to the development engineers. A huge significant step in software quality improvement came with the Personal Software Process (PSP). In recent years, as we know, there has been a huge emphasis on the assessment and improvement of the software engineering process. PSP allows software engineers to assess their progress and improve performance by tracking their predicted and actual development code. It was created by Watts Humphrey who felt the need for individual software engineers to acquire a disciplined and effective approach to writing programs. The aim of PSP was to apply the principles of the CMM to the software development practices of a single developer. It is very similar to the CMM except its focus is on the personal process of the developer.

What is unique to the PSP is that it can be suited to the individual developer. PSP is divided into certain stages as illustrated above. The baseline process PSP0 provides a structure to the programs and is used for gathering data on the software. The PSP0 stage can be split into 3 phases – planning, development, and a post-mortem. Planning and development revolve around constructing projects and recording data on them. During the post-mortem, the developer ensures all the data collected has been recorded and analysed correctly. Once enough data from work has been gathered, this allows the programmers to analyse their own performance and gives insight into how they can improve the personal process. PSP0.1 advances the process by adding a coding standard, a size measurement and the development of a personal process improvement plan (PIP). In the PIP, the engineer records ideas for improving his own process.



The later stages of the process such as PSP1 and PSP2 introduce further planning, quality management, and design. The aim of all this is to create efficiency and allow developers to plan their time carefully, maximising output. However after some research was conducted on the performance of PSP by the University of Hawaii in which they checked over 30,000 data values generated by classroom use of the PSP, they found that its manual nature could result in incorrect process conclusions despite a low overall error rate. As it is not possible to fully automate PSP, which results in the need for significant manual data entry, many believe that it doesn't provide enough return on investment. As a result, the leap toolkit was developed in order to combat the issues faced by PSP.

Leap stands for Lightweight, Empirical, Anti-measurement dysfunction and portable software process measurement. The Leap toolkit differs from PSP in that it automates and normalises data analysis. Although the developer still manually enters most data, the toolkit automates subsequent PSP analyses and in some cases provides analyses (such as various forms of regression) that the PSP doesn't provide. The approach is lightweight because it doesn't prescribe the sequence of development activities (unlike the PSP). Developers themselves control their data files, removing the risk of measurements dysfunction. It maintains data about only the individual developer's activities and doesn't reference developers' names in the data files. Leap is portable as it creates a repository of personal process data that developers can keep with them as they move from project to project and organization to organization.

Developers soon came to agree with Humphrey in that the PSP approach could never be fully automated and would inevitably require significant manual data entry. This is particularly true when each project significantly differs from the previous one, so as to render historical data inappropriate for comparison. As a result, Leap saw significant improvements in the way data was measured however it made some things increasingly difficult. As a result, The University of Hawaii proceeded to develop a data collection tool called Hackystat.

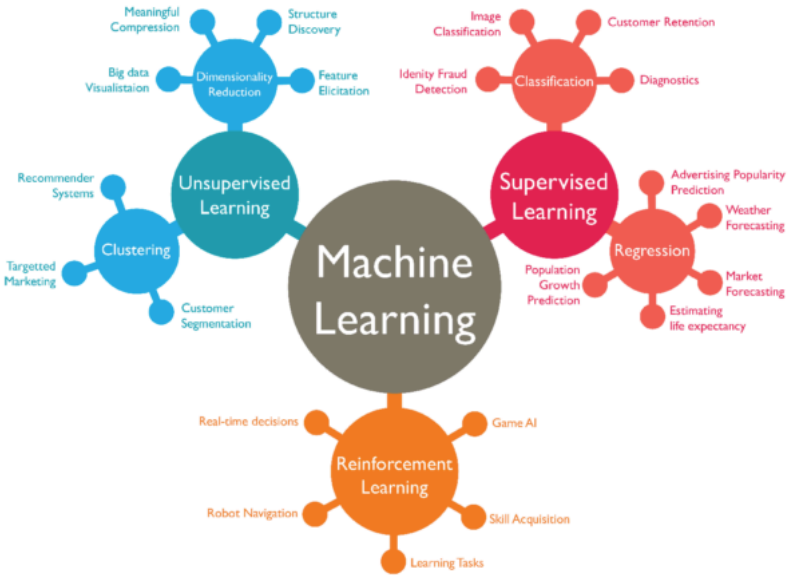
Hackystat describes itself as "A framework for the collection, analysis, visualization, interpretation, annotation, and dissemination of the software development process and product data." It is a platform that is available for researchers, practitioners as well as educators. Focusing on what Hackystat does for the developer, it can be used as infrastructure to support professional development, either proprietary or open source, by facilitating the collection and analysis of information useful for quality assurance, project planning, and resource management. Hackystat "implements a service-oriented architecture in which sensors attached to development tools gather process and product data and send it to a server, which other services can query to build higher-level analyses". Hackystat focuses on unobtrusive data collection. A common issue faced by developers means they are

continuously forced to halt production because of having to record what they have already produced. Hackystat put a system in place so that it could track a programmer as they edit a method and carry out tests.
Tying in the ethics associated with software engineering, a lot of developers were not happy that Hackystat was automatically continuously recording data without their knowledge. In addition to this Hackystat management had easy access to all this information, something which GDPR would cry at the thought of today. As a result, Hackystat quickly transformed from a fantastic invention in terms of saving time for developers into an invasive, unethical platform which developers were inevitably unhappy with.

The list of platforms mentioned above paved the way for many more after them. Popular platforms for software engineers today include Code Climate, Analytics as a service (AaaS) and Gitcop. **The wide variety of different platforms has led to difficulty for software engineers in choosing their desired one.**

Algorithmic Approaches

Having platforms available to measure software engineers performance is a necessary component in the software engineering process. The third aspect of my report is to highlight the different algorithms that can then be used to measure the performance of a software engineer. The data collected can be used across various algorithms and allow us to analyse the data obtained in order to get a better understanding of it. There are various different algorithm approaches to analyse data collected during the software engineering process. These range from computational intelligence to artificial intelligence



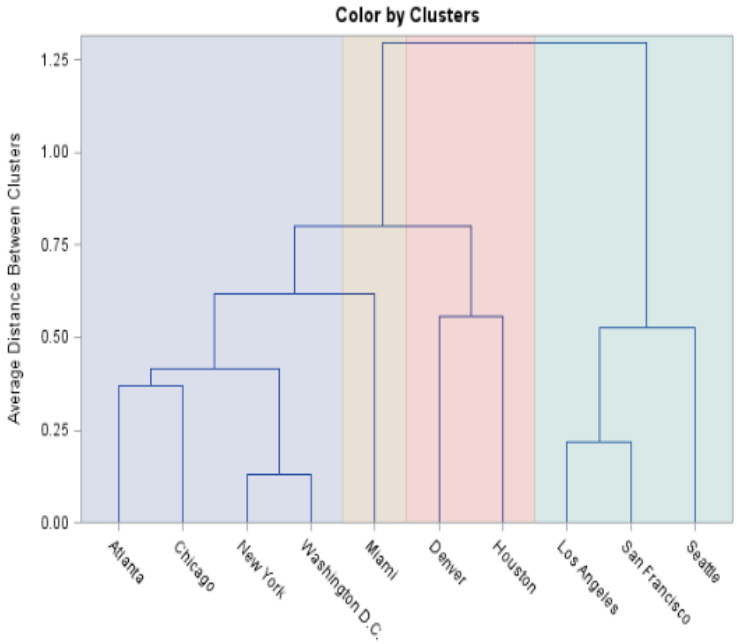
Firstly, I'd like to focus on machine learning, an element of artificial intelligence. Machine learning can be defined as the science of getting computers to learn and act as humans do. This is done by improving their learning over time by feeding them data and information in the form of observations and real-world interactions. Essentially, it is the science of getting computers to act without being programmed. It has become more and more useful in today's society as it uses algorithms to parse data, learn from it and them make predictions on real-world things. Machine learning is divided into three sections – supervised learning,

unsupervised learning, and reinforcement learning. These range in concept despite all being aspects of machine learning.

Unsupervised learning is carried out on data which is unsupervised, meaning the data is purely input data with no corresponding output variables. Similarly, supervised learning is carried out on data which is supervised, meaning the data contains input variables as well as output variables. The algorithms for each thus varies in their approach. Reinforcement learning is where an agent learns how to behave in a specific environment by performing actions and seeing the results. Reinforcement learning algorithms include game theory and centre around decision making. Looking at the above diagram, I could talk all day about machine learning algorithms however for the purpose of this report, I will focus on a few.

A major aspect of unsupervised learning is clustering. Clustering is carried out in order to establish if any groups are present within a data set. When I say groups here, I mean sets of observations which are quite similar, while being quite dissimilar to another group of observations. Clustering can be carried out using hierarchical or k-means clustering. Hierarchical clustering produces a dendrogram of values. A dendrogram is a tree-like structure placing observations into groups based on a dissimilarity matrix constructed. From looking at the below dendrogram below, we can see New York and Washington D.C are quite similar while being very different to Seattle, San Francisco, and Los Angeles. Useful applications of clustering are customer segmentation and target marketing.

Another large aspect of unsupervised learning is principal component analysis. PCA is a dimension reduction technique which extracts important variables (in the form of components) from large sets of variables in datasets. The principal components are found through the linear combination of coefficients that represent the greatest proportion of variance in the data. This is found through the Eigen-decomposition of the covariance matrix of the data.



The result of this decomposition is a matrix of eigenvectors and corresponding eigenvalues. The benefit of PCA is that it can summarise data through far fewer variables. With fewer variables, visualizations and results are more meaningful and impactful. PCA helps to reduce extremely large and complex data set into few variables, making it easier to analyse and work with.

Moving onto supervised learning, we can talk about classification and regression. Classification is the term used to describe finding a rule or method which allows us to determine which category or group a new observation would fit into. Using classification algorithms such as k-Nearest Neighbours and Discriminant Analysis, we can classify the new data points into the dataset with high accuracy.

Linear Discriminant Analysis (LDA) and Quadratic Discriminant Analysis (QDA) are both supervised statistical techniques which are used in order to reveal the internal structure of the data in question. This structure then allows for the classification of future observations. These methods use the knowledge of labelled data (i.e. those whose group membership is known) in order to classify the group membership of the unlabelled data. LDA and QDA both assume the use of a distribution of the data. Once the distributions and the parameters for these distributions are introduced we can start to quantify uncertainty over the structure of the data. This means we can start to talk about the probability of group assignment. Classification algorithms are widely used today particularly in image classification, diagnostics and identify fraud detection.

Reinforcement learning is the third and final aspect of machine learning. As I said previously, reinforcement learning concerns agents learning how to behave in certain environments, based on observation and decision making and the results these cause. The environment refers to the object that the agent is acting on e.g. a video game, while the agent represents the reinforcement learning algorithm. The algorithm starts by a state being sent from the environment to the agent, which then takes an action in response to that state. After that, the environment then sends a pair of next state and reward back to the agent. From this, the agent updates its knowledge with the reward returned by the environment to evaluate its last action. This continues until the system reaches a final state.

To summarise, there are many different algorithmic approaches available for the analysis of data obtained during the software engineering process. I discussed machine learning and the different aspects within it. These included supervised and unsupervised in addition to reinforcement learning. These algorithms vary in their approach to analysing data and are chosen depending on what type of data we are working with.

Ethics

After having discussed the software engineering process, how to measure the data and where to collect it from, one has to wonder – is it all ethical? Because of the huge role computers play in every industry in today's society, software engineers, those responsible for implementing the programs utilised by computers, of course, have significant opportunities to do good but also cause harm. As discussed previously in regards to Hackystat, unethical practices can cause issues in the software engineering process. To ensure software engineers do not cause harm in their work, they must commit themselves to make software engineering a beneficial and respected profession. This is done by following the 8 principles of software engineering which I will proceed to discuss. These include an obligation to the general public, the client and employer, judgement, the product, the profession, colleagues, the engineer themselves and management.

The obligation to the public: This is perhaps the most obvious obligation of the software engineer. Software engineers must act in the interest of the public. After all, the public is who their software will affect and so care must be taken to ensure they accept the full responsibility of their work. Developers are expected to balance the interest of the employer, client and the public. Focusing on the interest of one aspect can be detrimental. Software should only be made accessible to the public when it is believed to be safe, meet specifications and pass relevant tests. A very interest aspect of the software engineers obligation to the public involves considering factors such as economic disadvantage, disability, and allocation of resources and how these can affect access to the benefits of the software.

The obligation to the client and employer: This obligation includes no knowing use of illegal or unethical software, the proper authorised use of client/employers property and ensuring relevant documents that need to be approved are done so by someone qualified to do so. A big aspect of this obligation is keeping any confidential information obtained in their professional work, confidential to themselves where relevant. This is to ensure everything is kept legal and within the best interest of themselves and the public. With GDPR being a huge buzzword nowadays, software engineers must ensure that nothing breaches this strict policy.

The obligation to their judgement: This part of the Software Engineering Code of Ethics seeks to see software engineers maintain integrity and independence in their judgement. This includes focusing on only promoting documents prepared under their supervision and within an area which they feel competent in.

The obligation to the product: The developer's obligation to the product is clear in that they must strive to ensure they meet the highest professional standards possible. Areas of this obligation include ensuring they are qualified to produce the relevant product and that they are themselves capable of understanding the specification for the software it requires. The software engineer should strive to fully understand the specifications for software on which they work.

The obligation to the profession: Software engineers are obliged to help develop an organisational environment which is favourable to acting ethically including promoting public knowledge of software engineering. Software engineers should strive to support each other in striving to follow this code. In the best interest of the profession, software engineers should not act in their own interest at the expense of the profession, client or employer.

The obligation to their colleagues: This obligation involves software engineers being fair to and supporting their colleagues. This obligation is similar to that of any good co-worker in that they should assist colleagues in professional development, review the work of others when relevant and give a fair hearing to the opinions, concerns, or complaints of a colleague.

The obligation to themselves: In regards to this obligation, software engineers shall continually endeavour to further their knowledge of developments in the analysis, specification, design, development, maintenance and testing of software and related documents, together with the management of the development process.

The obligation of management: Software engineering managers and leaders should promote an ethical approach to the management of software development and maintenance. In particular, those managing or leading software engineers should ensure software engineers developing products are informed of standards before being beginning work on the project. The obligation of management ranges from ensuring work is only assigned after taking into account appropriate contributions of education and experience tempered with a desire to further that education and experience.

In conclusion, there are many ways to assess the quality of software engineers work in today's world. Various platforms which are at the disposal of every software engineering, providing them with a base to perform their work. Different algorithmic approaches can be used by software engineers in order to work with data collected. During this process, it is important for developers to remember to follow the code of ethics previously outlined in this report.

Bibliography

- https://www.linkedin.com/pulse/what-software-development-life-cycle-sdlc-phases-private-limited/
- https://www.techopedia.com/definition/8073/lines-of-code-loc
- https://stackify.com/measuring-software-development-productivity/
- https://www.agilealliance.org/glossary/lead-time
- https://martinfowler.com/bliki/TechnicalDebt.html
- https://blog.gitprime.com/why-code-churn-matters/
- https://pdfs.semanticscholar.org/e97d/2611d22c5ee8f8b70c0dc00bff626c54d324.pdf
- http://csdl.ics.hawaii.edu/research/psp-data-quality
- https://ieeexplore.ieee.org/document/6509376
- https://hackystat.github.io/
- https://www.techemergence.com/what-is-machine-learning/
- https://towardsdatascience.com/introduction-to-machine-learning-db7c668822c4
- https://towardsdatascience.com/introduction-to-various-reinforcement-learning-algorithms-i-q-learning-sarsa-dqn-ddpg-72a5e0cb6287
- https://ethics.acm.org/code-of-ethics/software-engineering-code/