MEASURING SOFTWARE ENGINEERING

Paul Byrne: 15317444

CONTENTS

SOFTWARE ENGINEERING PROCESS

The software engineering process is used to manage the conceptualization, development and maintenance of software. It involves the process of dividing software development work into distinct phases to improve design and project management. There are several key phases that we use to describe the software engineering process:

1. **Planning:** This planning phase involves defining and outlining the requirements of the software application.

2. **Requirement Analysis:** This phase involves the process of defining and further analysing each outlined requirement of the planning stage. Each requirement is inputted into a project management system.

3. **Design:** The design phase takes all of the requirements and starts to plan the product. This phase may include defining what programming languages to use, the architecture of the software application, the database relationships and much more.

4. **Implementation and Coding:** This is the phase in which the software application becomes more than just an idea. At this point the developers begin to write the code for the software, the hardware for the servers is set up, the design is further improved and the first test cases are written.

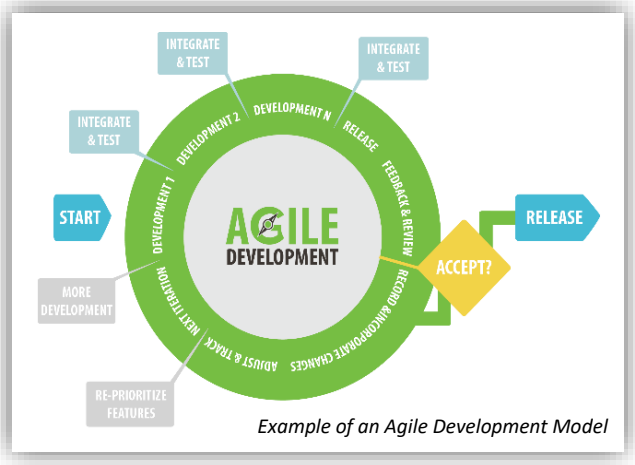
5. **Testing:** Possibly the most important phase of the software engineering process. Testers test the functionality of the software in order to identify and eliminate any bugs found. Testers begin executing test cases from the test plans to ensure that the software has been validated and that all of the requirements have been met.

6. **Deployment:** In the deployment phase the software is delivered/deployed to the customer for their use provided the testing phase has been completed successfully.

7. **Maintenance:** This is the phase in which ongoing care is taken for the developed software after it has been released for customer use. It is used to solve any problems that may present themselves while the customer is using the software. This phase also includes any further improvements made to the software over time.

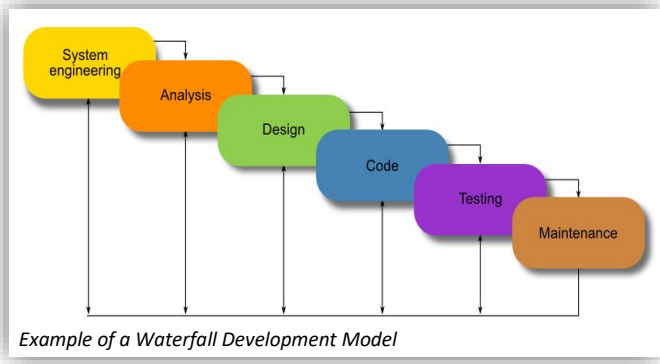
There are many different software engineering process methodologies which give a detailed yet simplified representation of a software process. The different methodologies represent the software engineering process from many different perspectives. Some well-known methodologies include:

- **Agile Development:** Agile Software Development is an umbrella term for several iterative and incremental software development methodologies. While each of these methodologies is unique in its approach, they all share a common vision and core values. They all fundamentally incorporate iteration and the continuous feedback that it provides to successfully refine and deliver a software system. Continuous planning, testing, integration and other forms of evolution are all key actions applied to both the project and the software. All agile methods focus on empowering people to collaborate and make decisions together quickly and effectively.



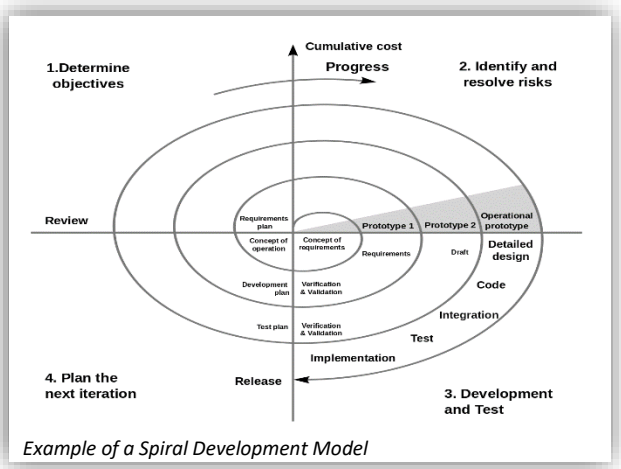
Example of an Agile Development Model

- **Waterfall Development:** Waterfall development is a sequential software development approach, where each fundamental activity of a process is represented as a separate phase arranged in linear order. The result of each phase in waterfall development should ideally be approved and the next phase shouldn't be started until the previous one has been completely finished. As this is not the case and phases overlap feeding information to each other, the software process is therefore not a simple linear but instead involves feedback from one phase to another. As this development method has a relatively rigid structure and is particularly hard to accommodate change when the process is underway, the waterfall development method should only be applied when requirements are well understood and unlikely to change radically during development.



Example of a Waterfall Development Model

- **Spiral Development:** The spiral development design includes the best features from the waterfall development and rapid prototyping methodologies. It introduces a new risk-assessment component which many felt the other methodologies neglected. The approach is represented in spiral form rather than a sequence of activities. Each loop in the spiral represents a different phase in the development process. The spiral model has four phases: Determining Objectives, Identifying and Resolving Risks, Developing and Testing and Planning of the next Iteration. A software development process repeatedly passes through these phases in iterations (each iteration is represented by a spiral in the model).



Example of a Spiral Development Model

MEASUREMENT & ASSESSMENT

As there is such a broad range of tools for measuring and assessing software development productivity, it can be particularly difficult to identify what metrics matter the most and whether or not there is one single metric that accurately reflects the software development process. Choosing the suitable metrics to measure software development productivity requires considerate thought and care to support the specific task or objective that a software developer is trying to successfully complete. Two main areas that we can explore in order to measure the development of software are within the code itself and in the individuals and teams responsible for the software development.

Assessing the Code:

- **Time for Code Completion:** The evaluation of time spent from when a developer begins writing code to when the code is finished and available to customers is a surprisingly impactful means for measuring development. When a developer finishes writing the code for whatever purpose it serves, there will always be a delay in time before the new code is made available to the customer. During this time the code may be reviewed and undergo a series of tests to ensure that it successfully delivers the correct functionality that the developer intended. At any time during this process, a developer may discover an issue that requires a bug fix or code alteration. Depending on the amount of issues discovered by the developer and the time it takes to resolve each one, the software engineering productivity may suffer drastically as the developer is delaying his/her response to something else and has to switch back and forth and rebuild the context every time.
- **Code Churn:** Another particularly effective means of evaluating software productivity can be obtained from when we calculate the percentage of a developer's code representing an edit to their own recent work. Another name for this is code churn. This form of software productivity assessment is usually measured as lines of modified, added and deleted code over a short period of time. This form of measurement allows us to control the software development process and in particular its quality. A spike in code churn can be an indicator that something is wrong with the development process. When a software developer is seen to be spending weeks'

worth of iteration on the same feature with little or no progress, code churn will spot this. Churn rate may also help identify if certain developers are experiencing difficulty solving a problem as well as indicating when a developer is under-engaged.

- **Unit Testing:** Unit testing plays an important part in the productivity of the software engineering process and for this reason it also provides us with an effective measurement tool to assess the level of development of a certain project. Every line of code written without a test to validate it significantly increases the cost of adding tests at a later date than if the tests had been written at the same time as the code. The longer a bug exists in the code, the more expensive it is to resolve it. These reasons stress the importance of writing unit tests throughout the software development process in order to avoid any unnecessary delays in the progression of the project. Assessing the validity of code as it is written is an important means of measuring productivity as without testing the code, software developers may face hours of unnecessary debugging which will only delay the software development process.

Assessing the Individuals & Teams:

- **Team Meetings:** Examination of the team meetings in a software development project can give an accurate indication of just how well a particular team is progressing. One example in which it is particularly effective is when we investigate how often a team finds themselves struggling to schedule group meetings to suit all members of the group. For a team that regularly struggles to identify suitable meeting times, we may conclude that group meetings are happening less frequently and in turn delay the software development process from any major progress. It is for this reason that it is important to identify the meeting times that suit all members of a team early in the software development process as well as establishing the times and locations for when each group meeting will take place. If a group often find themselves running out of time during their scheduled meetings this might be an indication that the group needs to dedicate more time to preparation before the meeting and again, may contribute to the delay in any sufficient progress of the software development process. Another useful metric for evaluating productivity might be through the level of participation of each member of the group. If all members do not fully participate in the team meetings, this could also withhold a software development project from any significant progress.
- **Customer Satisfaction:** Customer satisfaction is a particularly effective means of measuring the progress of a software development process. It is important that the software in question satisfies the customer's needs and that they are happy with what the development team have produced. Regular check-ins with a customer ensures them of the adequate progress being made and also avoids any development of unwanted features which in turn saves time in the long run. Customer satisfaction is vital in the software development process as it may result in future recommendations of the software development services if the software in question delivers value and has no serious deficiencies.
- **Meeting Targets:** A software developer or team's ability to hit targets and release dates reveals a significant amount of information about their productivity. It is important for a realistic target to be set in order to accurately determine whether or not the software developers are making any remarkable progress. Deadlines are set in order to ensure that the implementation of a particular feature is met as to avoid any potential back-log in productivity. The initial release date of a certain piece of software is often regarded as the most important target as it has a

very important impact on related marketing as well as promotional and publicity campaigns. A developer or team of developers capable of meeting their targets exhibits excellent attention to detail and organisation as well as ensuring customers of the adequate progress being made.

COMPUTATIONAL PLATFORMS AVAILABLE

While the Measurement & Assessment section above discusses many different approaches to measuring the software engineering process, none of these examples make use of the computational platforms available to the software engineering community today. The University of Hawaii at Manoa have looked for analytics to help developers understand and improve the development process and its products. Since research first began in 1996, the Collaborative Software Development Laboratory (CSDL) at the University of Hawaii discovered the high-impact analytics that could be yielded from Watts Humphrey's studies of the Personal Software Process (PSP). The original PSP showed three things:

1. The ways in which organisational software process analytics could be adapted to the individual developer.
2. The ways in which these analytics could be used to improve performance.
3. The practices displayed in an incremental manner which could be modified to suit academic and professional adoption.

Humphrey's version of the PSP was very tedious and required manual calculations which demanded substantial effort. While this method had various advantages and disadvantages, the manual aspect of the PSP raised concerns about the reliability of the analytics. In response to this concern, the Leap Toolkit was developed.

Although it still required manual input, the Leap Toolkit addressed some of the problems with the original version of PSP by automating and normalising data analysis. The Leap Toolkit allowed developers to control their own data files in an attempt to avoid flawed metrics as well as maintaining data only relevant to the individual developer's activities. The Leap Toolkit also created a repository of personal process data which enabled data portability.

Aware of the development overhead created by the PSP and Leap Toolkit, the CSDL began a new project, The Hackystat Project, focusing on developing ways to collect software process and product data with little to no overhead for developers. Hackystat had a service-oriented architecture implemented into it in which sensors were attached to development tools to gather process and product data which could be queried to build higher-level analyses. To date, Hackystat has led to a variety of technical innovations in the world of software engineering such as:

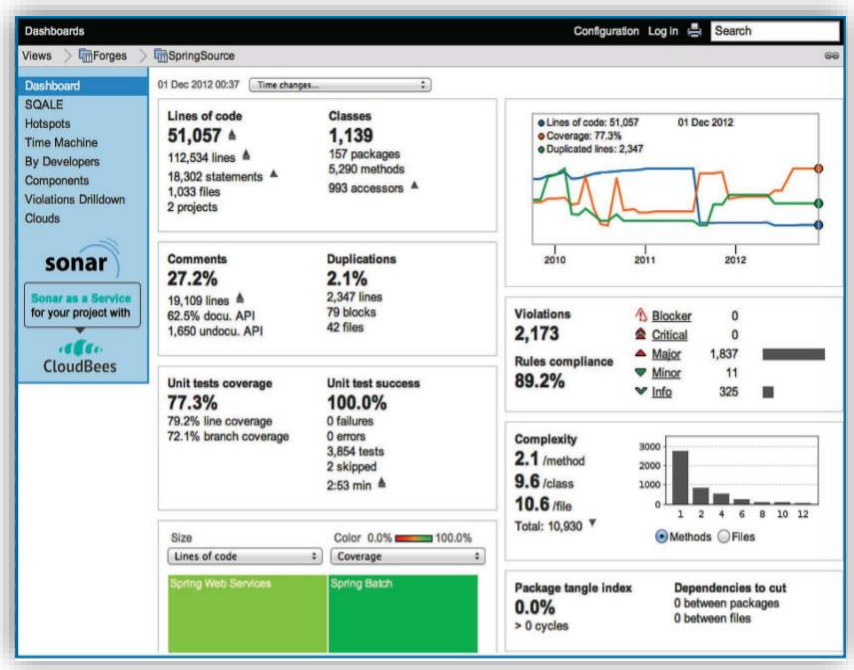
- Support for high-performance-computing software development.
- Methodology for prioritizing software development artefacts.
- A means of assessing a software engineering project's health (The Software ICU).

While Hackystat proved to be an extremely effective computational platform for measuring software engineering metrics, the Hackystat-based Zorro system provided an even more sophisticated application

of development behaviour data with the functionality to determine the extent at which developers use test-first design methods. Likewise, to the response of the Hackystat, the Hackystat-based Zorro system received a tremendous amount of criticism in which software developer's found the computational platform to be privacy-invasive.

As the demand for software product analytics increased, so did the services available for measuring this data. Most of these services were typically built from one or more of three basic sources:

1. A configuration management system.
2. A build system.
3. A defect-tracking system.



Sonar (Software Engineering Analytics) - Sample User Interface

By applying analytic techniques to data provided, these services simply display any analytics found in a user-friendly manner. This may include analytics such as coverage, complexity, security etc.

While there have been many significant improvements made to the computational platforms available for analysing software engineering metrics, services like Sonar (shown above) focus considerably more on the product characteristics rather than the behaviour of the developers themselves. Services like Sonar also fail to detect any correlation between interruptions and software productivity. While the services available today present a lot of positive analytics to encourage software productivity, I believe that significant improvements can be made to these computational platforms. With that being said, considering the level of improvement made over the last decade, there appears to be a very positive future in the area of computational platforms available for analysing software metrics.

ALGORITHMIC APPROACHES

As discussed previously, the modelling of the software engineering process is generally performed using the likes of waterfall and agile models. Further advancement of these models such as integration of algorithmic approaches in phases such as planning, design, implementation and deployment can improve the overall quality of a product. There are many different algorithmic approaches suitable for integration in the software development process. For the purpose of exploring the topic in question, I will discuss two fields which I found particularly interesting: Artificial Intelligence (AI) and Computational Intelligence (CI), both of which appear to hold a very important role for the future of technology.

Artificial Intelligence (AI):

In recent years, the ideal of integrating AI techniques into the software development process in order to improve the overall quality of a product has been posed by many. AI concerns itself with making machines intelligent, while software engineering is the process of defining, designing and deploying some of the most difficult and complex systems. Software engineers can avail of existing software in order to tackle challenges posed by the production of whatever new software that they are working on. This is where algorithms used in the world of AI can be useful.

These algorithms are designed in such a way to deal with one of the most demanding challenges posed to software engineers; replicating intelligent behaviour. For this reason, it is no surprise that the world of software engineering has started integrating many of the algorithms, methods and techniques emerging from the AI community.

The three most common AI techniques used in the software engineering community today are:

1. **Computational Search and Optimisation Techniques:** The redevelopment of the challenges that arise during the software engineering process as optimisation problems that can be tackled using computational search.
2. **Fuzzy and Probabilistic Methods for Reasoning:** The application of AI techniques developed to handle real world problems.
3. **Classification, Learning & Prediction:** During the early stages of the software engineering process, AI techniques have proved to be particularly useful for modelling and predicting software costs.

All of these AI methods are commonly integrated into software engineering and can be seen to optimise either the engineering process itself or the products that it produces. Whether the problem in question is one expressed in probability, developed as a system for predicting something or identified by a need to learn from experience, software engineers will always strive to optimise their approach as efficiently as possible.

While recent studies of AI integration with the software engineering process show promising results, there are still many improvements to be made. The idea of using AI techniques for software engineering requires us to think more about the level of development that will be made in the AI community in the coming years. Not only is AI rapidly changing, but so is the nature of software forcing us to reconsider the ways in which the software development process is implemented. Within this ever changing world

of software development, AI techniques are proving to be a particularly resourceful tool since the inspiration for these techniques stems from human intelligence.

Computational Intelligence (CI):

The software engineering process faces a continuous struggle to meet the ever-increasing demand of customers who require new software systems. With this demand comes new challenges in meeting the requirements of the customer as well as increasing levels of complexity to complete such tasks. As discussed above, there is evident growth in related disciplines such as AI having an impact on the software development process. Computational Intelligence is a new emerging area which has also been seen to provide optimisation techniques to the software development process. CI refers to the ability of a computer to learn a specific task from data or experimental observations. CI is often regarded a subset of AI and the main difference between the two is that CI is based on soft computing methods, which enable adaption to many situations, distinguishable from AI which is based on hard computing techniques. CI is a set of nature-inspired computational methodologies and approaches to address complex real-world problems to which mathematical or traditional modelling might not be suitable. The five main principles of CI and its applications are:

1. **Fuzzy Logic:** A branch of mathematics that allows computers to model real world problems similar to the way that people do. This is done through the use of fuzzy sets. The degree to which an object is a member of a traditional set is strictly 0 or 1 but with fuzzy sets, can be any value between 0 and 1. The flexible membership requirements in fuzzy sets allow for partial membership in a set.
2. **Neural Networks:** Neural network computing is an approach that simulates the human brain's neurons. Neural networks enable the process of learning from experimental data.
3. **Evolutionary Computation:** Optimization algorithms like evolutionary programming and genetic algorithms have proved useful in solving difficult optimization problems.
4. **Learning Theory:** Learning theory in CI attempts to simulate accurate reasoning to that of humans. It brings together cognitive, emotional and environmental effects and experiences to which it can then make predictions from.
5. **Probabilistic Methods:** Probabilistic methods aim to evaluate outcomes of systems mostly defined by the randomness of the real world. Probabilistic methods define the possible solutions based on prior knowledge.

On examination of its potential applications, it is evident that CI technologies could play a significant role in the software development process and may even present innovative approaches to existing methodologies.

One example of how CI applications may be beneficially integrated into software development can be seen in the area of software reliability. The use of neural networks may play an important role in developing more realistic models of reliability than the standard models that exist today. Another way in which evolutionary computation is of interest to the software engineering community can be seen in the area of software development via the evolution of a population of programs as opposed to the development of programs from scratch. This approach may involve competition among programs for certain resources. The fuzzy logic component of CI may also be a useful resource to the software engineering process in cases concerning the incompleteness of data. As discussed above, fuzzy logic uses

degrees of truth rather than the standard Boolean values; true or false. Essentially, fuzzy logic may be able to reasonably estimate data in cases of incompleteness based on previous knowledge.

With reference to the details explored above, CI proves to be an impressively helpful tool in the software engineering community demonstrating its usefulness in requirements engineering, design, quality assurance, reusability, cost estimation, validation and verification as well as maintenance. Evidently, it provides essential methodologies and algorithms to the software engineering process.

ETHICS & CONCERNS

It is not uncommon for many large companies around the world to frequently monitor metrics related to employee productivity. In the world of software engineering, so much of what a developer works with is data-driven feedback. As discussed above, analysation of unit testing, code churn, customer satisfaction etc., provide us with the relevant data upon which we can prioritize future efforts. What these software engineering metrics lack is an accurate measurement of the productivity of the developers themselves. While there are many benefits to the measurement of software engineering, there are also some ethical questions to answer. In this section, I will outline the various ethical complications that exist in the area of software engineering metrics as well as the reasons for such a lack of metrics to accurately represent the productivity of a software engineer themselves.

As discussed in the 'Computational Platforms Available' section above, there is an obvious ethical issue regarding the invasion of privacy when assessing various software engineering metrics. Not only is this an issue in employment roles relating to software engineering, but the collection of employee data appears to be an issue in a variety of working roles worldwide. Personally, I see no issues with the collection of this data, provided it is solely used to assess and better improve the performance of an employee. As long as the metrics collected are kept purely professional and avoid tracking any personal information regarding the employee, I think the collection of employee data is a very beneficial practice within an organisation.

With that being said, there are also a few considerations that need to be taken when assessing the productivity of an employee. There are many reasons as to why an employee may be exhibiting a lack of productivity which may give an inaccurate reflection of the work the employee is doing. Some common causes of poor employee performance include:

- Lack of employee training.
- Failure to understand the appropriate approaches to performing a task.
- Personal problems.
- Job dissatisfaction.
- Boredom/lack of challenging tasks.

Without consideration of the potential issues above, the data collected from an employee may give a distorted representation of an employee's performance. For this reason it is easily comprehensible as to why many have concerns about the measurement and assessment of employee data including employees working in software development.

While the main ethical issue concerned with measuring and assessing software engineering productivity stems from privacy invasion, we can't help but notice a lack of metrics to assess the productivity of the

engineer themselves. Many of the tools used today for measuring the software engineering process (XP, TDD, Agile, Scrum, etc.) were designed to increase developer productivity but fail to assess the efficiency of the developer themselves. Again, the lack of these metrics stems from privacy invasion. But what if we didn't use metrics at all in this case? In my opinion, there is a very obvious solution to measuring employee performance without actually using any metrics: supervisors and employees should meet on a one-to-one basis regularly and frequently, in order to establish a relationship, gather a knowledge of what's going on as well as determining the early warnings of possible developing problems. The important part of this approach is to ensure that the meetings happen on a regular basis. In doing so, an employee never goes too long without giving feedback. As well as this, keeping important notes of each meeting will ensure that the supervisor can spot problems easily and offer positive reinforcement for observed improvement.

Along with the invasion of privacy associated with many forms of measuring productivity, there are also a variety of other issues with some of the commonly used metrics today:

- **Hours Worked:** Assessment of an employee's worked hours is a commonly used metric regardless of the fact that it doesn't give accurate representations of an employee's work. A simple example is if one person works 10 hours instead of 8, they should get 125% of the work done which is simply not necessarily the case and instead, working too many hours may even lead to a decrease in productivity.
- **Source Lines of Code (SLOC):** Problems when assessing a software developer's productivity based on the number of lines written in their code include:
    - Developers can add extra lines of code to increase their performing appearance.
    - A program written in 50% less lines than another program may be more efficient or performant.
    - Sometimes developers gain from the deletion of code.
    - 1000 lines of code containing lots of bugs is worse than 500 lines of code containing no bugs at all.
- **Time Estimation:** Failure to meet estimated deadlines is a very common means of assessing an employees productivity. Problems with this metric includes:
    - Unexpected distractions may result in a task taking twice as long to complete than originally estimated.
    - Doesn't account for cases when a developer is overly and inconsistently optimistic with their estimations.
    - Doesn't account for cases when the customer asked for something unrealistic/impossible, which could only have been discovered during the coding process stage of the software engineering process.

It is hard to determine a single metric that doesn't raise either ethical or problematic concerns. The software engineering process is particularly difficult to measure as there doesn't seem to exist a reliable, objective metric of developer productivity. With that being said, the emerging measurement tools such as Scrum appear to produce relatively useful metrics based on physical data, however, there is no ignoring the fact that the community of software engineering is lacking a tool which can accurately represent the productivity of the software engineer themselves in a way that doesn't present any ethical or problematic concerns.

SOURCES

- http://ieeexplore.ieee.org/stamp/stamp.jsp?arnumber=614837
- http://ieeexplore.ieee.org/document/7847759/
- http://ieeexplore.ieee.org/stamp/stamp.jsp?arnumber=7847759&tag=1
- http://www.citeulike.org/group/3370/article/12458067
- https://dev9.com/blog-posts/2015/1/the-myth-of-developer-productivity