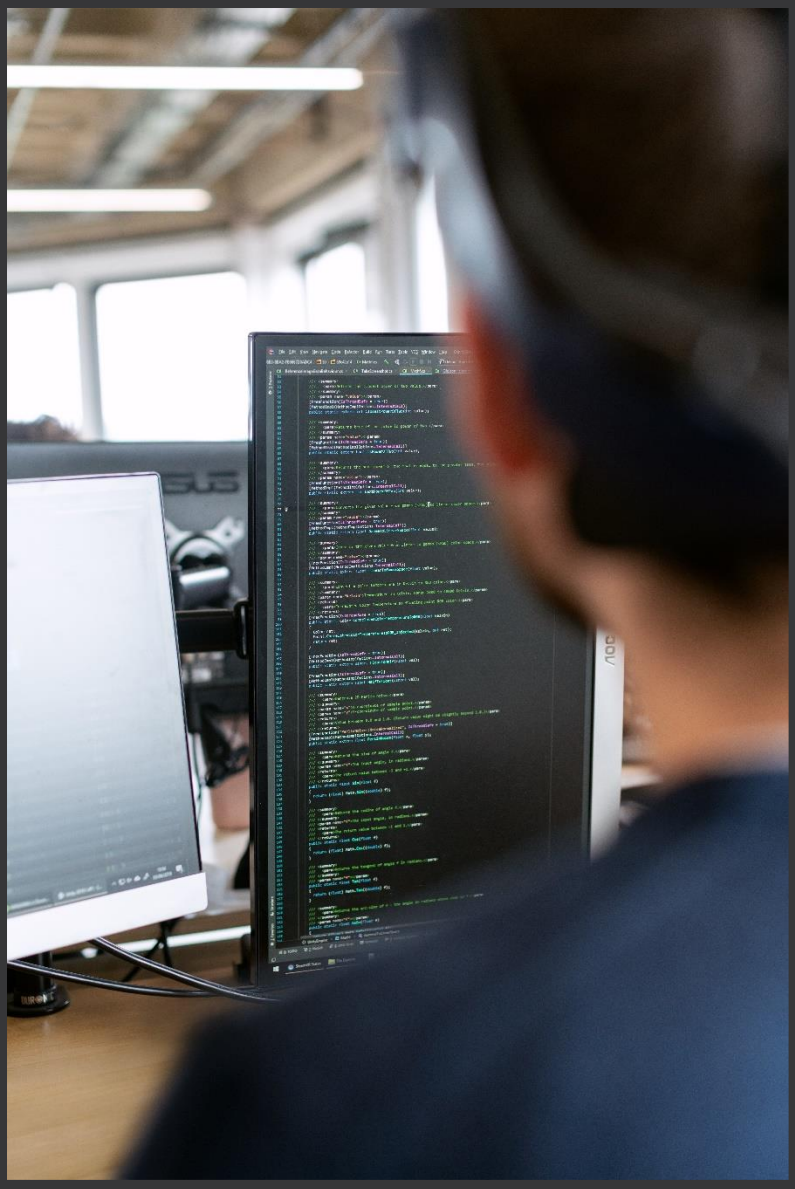


Measuring Software Engineering

Metrics, Frameworks, and Methodologies in Software Engineering Measurement, and their Ethical Implications

Claire Cassidy | 16325301 | 15/11/2020

Introduction:

The twenty-first century has ushered in an era of technological ubiquity, where businesses from all corners of the market pivot away from activities in the industrial sector to deliver service and knowledge-based offerings.[1] Such companies depend increasingly to developments in data analytics such as Big Data to remain competitive and improve their products. However, while companies may use these developments to analyse their target audience, they also claim benefits from turning the techniques inwards to examine their own internal processes. Indeed, no industry has been more closely intertwined with emergent data analytics technologies than the software engineering industry, being adjacent to related theoretical advancements. Beginning in the 1960's and 70's, software engineers and computer scientists began to apply these analysis techniques to their own workflows and determine what benefits could be gleaned from examining the software engineering process itself.

Concerted efforts began to be made to move the industry towards a standardised measurement or set of measurements of efficiency in relation to the software development process. This began with an analysis of what could be considered reliable and accurate metrics by which process efficiency could be measured. Given the abstract nature of computer science and the complexity of the software engineering process in general, such a task was not as straightforward as it may appear. Metrics may be deceptive in the degree to which they reflect the underlying source, and to this day controversy persists over which metrics to use and how to apply them. I will discuss such metrics in the section Metrics in Software Engineering.

As software development has grown exponentially in the previous decades, so too has the userbase for tools that perform these metric gathering operations and analyses automatically. These tools are discussed in the section *Software Measurement Platforms & Frameworks*. In the modern day, complex processing techniques are applied to the huge digital footprint left behind by developers as they work. Such large datasets require powerful algorithms to extract trends and patterns. I discuss the algorithmic approaches to software metric analysis in the section *Algorithmic Approaches to Analysis*. Finally, given the personal aspect of the data being collected and the pressure on management to improve productivity, there is reasonable concern that such data may be used in unethical ways. I discuss these ethical issues in the fourth section, *Ethical Considerations*.

Metrics in Software Engineering

Before beginning an analysis of the software engineering process, one must decide on a set of measurements that reflect the projected success of the engineering process underway. In choosing these metrics, we must select some quantifiable qualities of the software engineering process that we have found to, or assume to, contribute to maximising quality of our end product. We can then measure these qualities in practical scenarios to predict the degree to which we are verging on a quality product.

For the estimation to have any bearing on reality, such metrics must be chosen carefully. Naturally, difficulties have emerged in the field of software measurement relating to the ambiguous degree to which conventional, 'common-sense' metrics actually forecast the quality of the solution, and in general in determining empirically what qualities are truly relevant to the assessment. Further complicating this is the idea of what software *quality* even entails – of course, one may have an intuitive feel for what constitutes a quality product from experience, but tying this experience to empirical data generated in the process of its construction is not straightforward and requires an engineering mindset. Professional standards of software quality have generally settled on a unified measure of compliance to management and customer expectations - *conformance to requirements* and *fitness for use* respectively [3].

Software engineering metrics are divided into two categories, *product metrics* and *process metrics* [4]. Product metrics deal with those measurements that can be retrieved directly from source code along with factors generated in the algorithmic analysis of this code. Process metrics include measurement factors of the engineering process itself that may be gleaned from the tools used by the programming team or by documentation maintained by management.

Table 1: Example Product Metrics and Process Metrics used in Industry

Product Metrics	Process Metrics
Lines of Code (LOC)	Man hours
Development defects (e.g. compiler errors)	Testing Effort
Production defects (e.g. user-reported bugs)	Bug-fixing Rate
Code complexity	Productivity
Performance	Mean Time to Failure (MTTF)

Such factors aim to give an estimation of sub-components of software quality, such as *capability*, *usability*, *performance*, *reliability*, and *maintainability*. Sometimes, new metrics are only quantifiable once the product is released, such as the mean time between user bug reports. Alternatively, some metrics are

only relevant in certain contexts; for example, contractors of safety-critical software systems such as air traffic control systems mandate that the *Mean Time to Failure* (MTTF) of a system (i.e. the time between system crashes) be above a certain threshold, often many months to years.[5] Project variation thus makes it difficult to create a standard measurement process that can reliably capture the true quality of any arbitrary software development project.

The efficacy of many commonly used metrics themselves are also questioned. Concerns have been raised ad nauseum regarding the use of primitive metrics such as Lines of Code (LOC), defect density and in-person months of work, which were among the first metrics devised in the 1960s and 70s.[5] These concerns are well-founded; for instance, LOC can incentivise sloppy, meandering code when used as a means for assessing developer productivity. Incentivising fewer lines of code and elegant solutions risks jeopardising the readability and maintainability of the codebase if not carefully controlled. Furthermore, verbosity varies greatly between languages, and can complicate metrics when applied to codebases using many languages.

Another example of a controversial metric is *defect density*. In prior decades, production defects could be estimated by measuring the quantity of development defects, in an effort to predict end-product quality. The intuition was that modules encountering more errors in development were more likely to contain hard-to-spot bugs. However a study published in 1999 by Norman E. Fenton and Martin Neal failed to find a correlation, and conversely uncovered that complex modules were more likely to be tested thoroughly, whereas simpler modules were neglected.[5] Other industry-standard metrics for predicting end-product quality also failed to be evidenced, leading the researchers to conclude quite pessimistically that “the credibility of metrics as a whole has been lowered because of the crude way in which these kinds of metrics have been used”.

Following their recommendation, more rigour was injected into the analysis techniques. Analyses using only one or two metrics was abandoned and multi-metric, “non-isolationist” approaches became standard. This helped limit the error incurred by outlying poor-quality metrics and laid the foundation for the first automated testing platforms in the late 1990s, which are discussed in section three. Some systems specialised in sets of related, unconventional metrics and made reasoned judgments in sub-domains of the development process, rather than making lofty judgments of the entire system. For example, systems emerged dedicated to measuring testing effort by examining the density of unit test invocations and code coverage.

The twenty-first century has seen a new class of metrics enter mainstream analysis, heralded by advancements in the areas of big data, the internet of things, and behavioural analysis. These metrics are extraneous to data gathered from source code, development tools and conventional management records, but nonetheless have been found to have a considerable impact on the productivity of a team. Such metrics measure the social composition of software teams, the physical environment in which these teams work, and the mood of employees.

For example, researchers have found that conformist behaviours induce a 1.7% in rise in the productivity of a given worker for every 10% rise in collective co-worker productivity.[6] The addition of a well-trained co-worker to a worker's social network also increases that worker's productivity by 0.7%. Thus, easily acquired statistics on team composition can empower management to rebalance teams and plan promotions to maximise global productivity. Such metrics also expert developers in a particular area be identified via team composition visualisations. This allows for faster bug report resolutions, and identification of potential mentoring setups, critical team departures, and lucrative collaboration opportunities. Such high-level metrics are obtained as a product of analysing low-level statistics mined from source code repositories and are typically grouped under *team cohesiveness* metrics.[8]

Happiness and comfort in the work environment have also been shown to be powerful metrics for predicting performance. Research shows a happy employee is 37% more productive and 300% more creative relative to unhappy colleagues. Capitalising on these findings is as simple as making small adjustments to environment such as air conditioning. Wearable tech has been developed for the collection of statistics related to "human and social activities that have previously only been understood in qualitative terms", such as heart rate monitors and devices that measure intonation in conversation between employees. [9] The obvious ethical concerns arising from these metrics will be discussed in section four.

Finally, a discussion is warranted on how metrics are only relevant in the context of the underlying process they seek to measure. New development paradigms such as AGILE have emerged in the 21st century, rendering some conventional metrics obsolete.[10] For example, process metrics are forced to undergo a radical transformation since as conventional metrics presuppose detailed requirements documentation. The ad-hoc nature of AGILE development necessitates the use of quality metrics in assessing and minimising risk. Ever-evolving user stories and the absence of requirements documents often render effort estimation ahead of time inaccurate. The dearth of theoretical research forces most teams to resort to a pooled subjective estimation, often done via the *planning poker* technique. Another approach is to track *velocity*, which is defined as the turnover-per-iteration (SCRUM sprint) of user stories. This can be compared against the remaining user stories to estimate time to completion. The underlying assumption that stories take roughly constant effort throughout the development cycle may also introduce error, however these metrics can still be of use to refine future stories given a project that is under- or over-budget. Upon reading the current literature, it is clear that much theoretical work needs to be done to support modern development paradigms such as AGILE.

Practice	*Main aim*	*Basis of practice*
Software size	Estimation of software size/effort	User Stories
Velocity	Overall productivity of team	User Story points
Burndown chart	Progress monitoring	User Stories
Cumulative flow	observation of lead time and WIP queue depth	Work in Process/Progress
Responding to change	indicator of ability of team to hand over product quality	Defects fixing cost
Earned Business Value	Monitoring business value delivered to customer	Business value
Total Estimation Effort	Planning and budgeting	User Stories and Re-works

Figure 1: Commonly used metrics in AGILE development and their motivations, courtesy of Javdani, et al. [10]

For all their controversies, the metrics mentioned are producing practical results. Many crude metrics of the 1960s and 70s are still relevant today, albeit subject to more effective analysis strategies. Indeed, it must be noted that "the field isn't converging on a single best approach, nor are the latest approaches intrinsically better than earlier ones. Rather, the community has been exploring the space of trade-offs among expressiveness, simplicity, and social acceptability".[11]

In the early days of measurement, practical application of theory was hindered by financiers' views of metric collection as a superfluous overhead to be added to projects that were oftentimes already over-budget. This changed with the advent of the platforms discussed in the next section, which seek to mitigate these concerns by providing automatic and invisible data gathering and processing.

Software Measurement Platforms & Frameworks

When considering software engineering metrics and their analyses, we rely on software measurement platforms and frameworks to facilitate practical application. The modern market is awash with tools to help management personnel and developers alike track their progress towards realising a quality project. In this section, we discuss some of the more popular platforms and frameworks that have emerged throughout the decades.

1995 saw the publication of Watts Humphrey's seminal text on software engineering measurement, '*A Discipline for Software Engineering*'. Within, he preached the importance of metrics analysis, and advocated for each software developer to manually track their own progress as part of the '*Personal Software Process*' (PSP). The process sought to "improve project estimation and quality assurance … by collecting size, time, and defect data".[12] Prior to this, some companies collected aggregate development data, however the PSP shifted the responsibility from management to programmer. Developers would naturally become acquainted with areas they needed to improve with respect to productivity, and it provided a natural incentive for improvement over time. The PSP was also flexible and did not emphasise a particular template for project measurement, meaning it could be tailored on a per-project basis.

However, the limitations of a manual approach soon became abundantly apparent. Developers felt the PSP's high degree of metric granularity rendered it too cumbersome to be sustainable. The PSP could contain as many as twelve separate forms that must be maintained over the entire development process, including a project plan summary, a time-recording log, a defect-recording log, a process improvement proposal, a size estimation template, a time estimation template, a design checklist, and a code checklist.[11] Every compiler error, every interruption, every file edit was tracked manually. Furthermore, the PSP produced a significant mental tax on developers via '*context-switching*' between programming and paperwork. Furthermore, manual data was vulnerable to human error and bias. Despite the PSP being surprisingly effective when used in analysis, it did not prove popular with developers outside of contexts in which its use was compelled. In one study, 72 of 78 students introduce to the PSP did not continue the process into general programming endeavours as "it would impose an excessively strict process on them and the extra work would not pay off." [12]

Name: Jill Fonson **Program:** Analyze.java

Date	No.	Type	Inject	Remove	Fix time	Fix defect no.	Description
9/2	1	50	Code	Com	1	1	Forgot import
9/3	2	20	Code	Com	1	2	Forgot ;
9/3	3	80	Code	Com	1	3	Void in constructor

Figure 2: Example of onerous compiler defect recording in PSP, courtesy of Johnson [11]

Automatic data collection tools provided answers, and found their first popular iteration in the program *HackyStat* in 2001. An open-source development project led by researchers at the Collaborative Software Development Laboratory (CSDL) in University of Hawaii, *HackyStat* could work in the background of everyday development tools to unobtrusively track development data in real-time. The program could work online or offline, and submitted the data collected to a company-wide data repository. *HackyStat* was not limited to simple data aggregation; it could create composite metrics and present the data in different views. A user may want to see their own development data in the "*Daily* Diary", for example, which detailed activity otherwise captured via PSP at five-minute intervals.

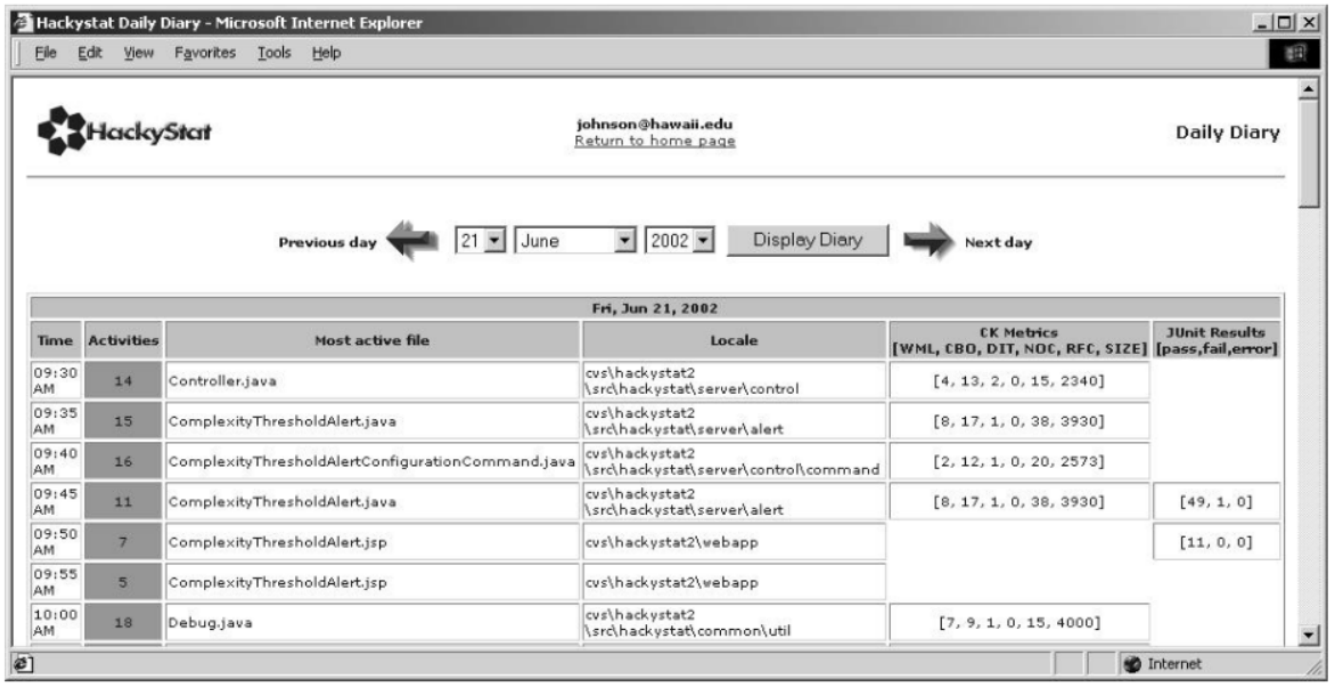
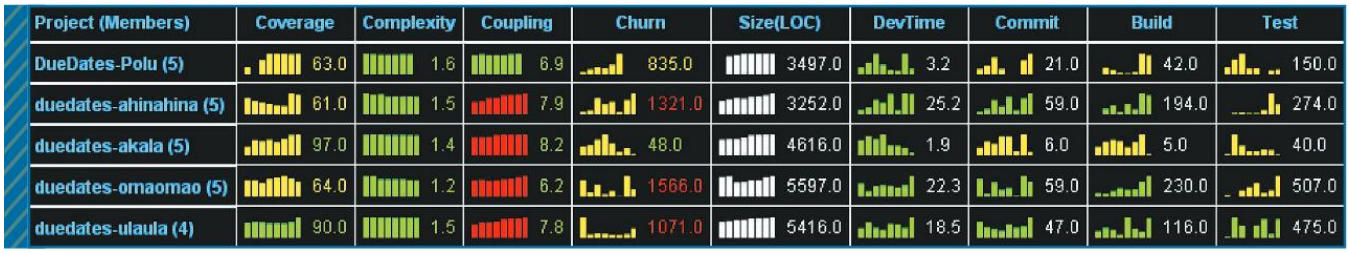


Figure 1: The HackyStat Daily Diary, courtesy of Johnson [12]

Furthermore, an alert system could be enabled when developers exceeded project-specific metric thresholds such as complexity. It also provided first-of-their-kind analysis tools, such as a project-wide *Software ICU* view that could provide analysis of the overall health of the project.



Project (Members)	Coverage	Complexity	Coupling	Churn	Size(LOC)	DevTime	Commit	Build	Test
DueDates-Polu (5)	63.0	1.6	6.9	835.0	3497.0	3.2	21.0	42.0	150.0
duedates-ahinahina (5)	61.0	1.5	7.9	1321.0	3252.0	26.2	59.0	194.0	274.0
duedates-akala (5)	97.0	1.4	8.2	48.0	4616.0	1.9	6.0	5.0	40.0
duedates-omaomao (5)	64.0	1.2	6.2	1566.0	5597.0	22.3	59.0	230.0	507.0
duedates-ulaula (4)	90.0	1.5	7.8	1071.0	5416.0	18.5	47.0	116.0	475.0

Figure 3: Overview of the HackyStat Software ICU window, courtesy of Johnson [11]

Commercial use of *HackyStat* began in 2006. It soon spawned numerous successors that extended its feature set or adapted it to perform more in-depth data gathering and analysis for specific contexts. One such successor system was *Zorro*, which focused on providing teams with a more standardised way to measure and enforce *Test-Driven Development* (TDD).[13] TTD is a software engineering methodology in which tests are written before code, in accordance with the requirements of a system. In this way, requirements are embedded into the programmer before the first line is written, leading to reduced defect rates, higher code coverage and rigorous testing effort. Practical adoption of TTD was not straightforward however, and many wondered whether a one-size fits all approach was even feasible. Zorro achieved flexibility by decomposing TTD's principles into a set of 22 *'episode types*'. Low-level developer actions, such as unit test invocation and compilation events, were then processed for their adherence to one of these episode types. Management could then tailor their analysis of TTD adherence to their own personal standards.

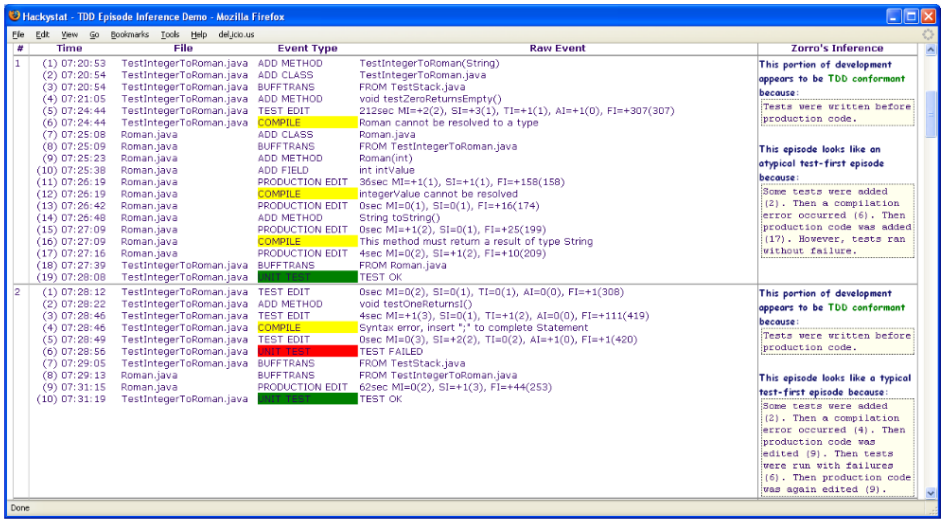


Figure 2: Zorro's TTD episode inference screen, courtesy of Johnson [13].

The advantages of using such automatic data gathering and analysis tools over manual tools like PSP can be seen in the following chart:

Characteristic	**Generation 1** (manual PSP)	**Generation 2** (Leap, PSP Studio, PSP Dashboard)	**Generation 3** (Hackystat)
Collection overhead	High	Medium	None
Analysis overhead	High	Low	None
Context switching	Yes	Yes	No
Metrics changes	Simple	Software edits	Tool dependent
Adoption barriers	Overhead, Context-switching	Context-switching	Privacy, Sensor availability

Figure 3: Comparison of the key adoption factors across the early stages of software measurement tools, courtesy of Johnson [11]

One of the factors that changed the landscape of software engineering measurement tools was the emergence in popularity of version control technologies, GitHub, and related services. With this, massive banks of metadata-rich source code were accessible for use in development of analysis tools. Indeed, GitHub itself offers many metrics out-of-the-box for a given project, such as LOC information, density of contributions per user, *Issues* and their resolution dates, etc.

As projects have scaled over the years, even more sophisticated commercial tools have entered the market allowing for detailed evaluation across a wide variety of product and process metrics. *Flow* from *PluralSight* (previously *GitPrime*), for example, offers analysis of factors specific to the structure of source code repositories and tailored to large-scale software development environments that allow for detailed, personalised evaluations of productivity.[14] These metrics include

- *Impact*: an estimation of the difficulty of a given commit in terms of the mental load on the contributor.
- *Churn*: to what extent code has changed over a given period
- *TT100 Raw*: a measurement of how long it takes a particular software engineer to produce one hundred lines of code including re-writes
- *TT100 Productive*; a measure of how long it takes to write one hundred lines of code *without* re-writes.

Consideration of the latter two metrics in tandem helps create a better overall picture of employee productivity. For example, considering only *TT100 Raw* might be unfair to the meticulous programmer who often writes slowly but correctly, and measuring *TT100 Productive* only may give the impression that a programmer is near-idle when in reality they are working on a particular challenging section of code. *Churn* also helps indicate particularly troublesome sections of code, where the solution is being changed

often. They also provide support for calculating the *risk* of a commit, helping to maximise the benefit teams get from code reviews. In terms of measuring team cohesiveness, metrics are gathered to determine which teammates helped others, how responsive a developer is to tasks they are assigned, and an individual's overall influence and involvement in the project. Each of these metrics can be used to generate a variety of summary reports, giving team leads an overview of the development process, such as the *Daily Update*, *Project Timeline*, (developer) *Proficiency*, *Player Card*, and *Snapshot* report templates.

Figure 4: Example PluralSight Flow report, courtesy of pluralsight.com

A competitor to *PluralSight Flow* is *WayDev*[15]. *WayDev* analyses version control repositories to allow conclusions to be drawn at a glance via detailed visualisations of data. It provides analysis at the commit-level and provides analyses tailored to AGILE development. Other contenders in the software measurement ecosystem, which typically offer an emphasis on a particular niche of metric collection, include:

- *CodeClimate* (pattern visualisation)
- *DevCreek* (for Ruby development)
- *OpenHub* (previously Ohloh. For analysis of open-source projects)
- *Atlassian Jira* (AGILE project management), *Atlassian Trello* (User story and general project management), etc.
- *CAST* (uses computational intelligence techniques to generate unique insights in projects)
- *Parasoft* (automated testing)

Software engineering measurement platforms have transformed dramatically since their inception in the late 1990s. Beginning as cumbersome paper forms into which product and process metrics were manually entered, the industry saw a shift into more and more advanced automated metric gathering and analysis systems that were platform-independent. Nowadays, any competitive software engineering

measurement platform is expected to respect the importance of team productivity and cohesion in their analyses. Modern platforms incorporate advancements in behavioural studies to carefully measure teams whose sizes may number in the thousands. State-of the art data analysis techniques such as computational intelligence solutions are becoming increasingly popular as a means for exposing unobvious patterns. We will further examine the algorithmic means for producing these analyses in the next section.

Algorithmic Approaches to Analysis

Software metrics remain simply data without some technique to extract a meaningful interpretation. While obvious patterns and conclusions can be inferred with the naked eye, the threshold for human understanding is negligible at the scale of the footprint software engineer produces. Processing of this data by an advanced data analysis system is necessary to draw involved, accurate, fine-grained, and unobvious conclusions. Practical application of these techniques by management can tailor the process of monitoring and controlling the development of a product and keep software teams focused where it matters[5].

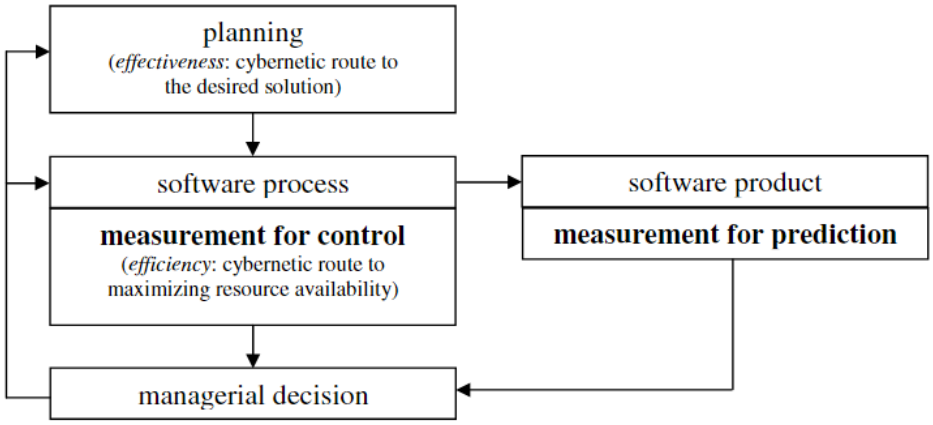


Figure 5: Role of Software Engineering Measurement in Project Management, courtesy of [16]

Software engineering measurement was in limited use in the late 1990s due to the size of industrial scale datasets; indeed, research had "failed almost totally in terms of industrial penetration".[5] The advent of metrics-gathering systems in the early 2000s, as detailed in the previous section, solved this by providing development teams with unobtrusive modules they could attach to their development platforms, aggregating data in real time and performing rudimentary algorithmic analysis. Nowadays, algorithms optimised to specific domains of interest exist to facilitate powerful data processing techniques. We will discuss numerous examples of these in this section, with an emphasis on emerging techniques that address the software measurement process as a socio-technical enterprise.

Perhaps the most influential development in the last decade relating to software engineering measurement is the emergence of Computational Intelligence (CI) paradigm. Traditional algorithmic approaches to data analysis struggle in complex analyses that require the consideration of multiple variables at a time. The most fundamental issue with multivariate analysis is the perennial problem of *correlation vs. causation*; that is, correlations between to measures suggest only simultaneous occurrences

of those measures, and provide no evidence in themselves for a causal relationship between the two.[16] Further, to consider multiple metrics in the same domain, one often needs to work at a higher level of abstraction, leaving data subject to noise and unforeseen influences. This becomes particularly apparent as software measurement concerns itself more and more with metrics inspired by behavioural analysis techniques. CI is an alternative approach to analysis that marries fuzzy logic, neuro-computing and evolutionary computing, to imbue a system with reasoning behaviours of the human mind.[17] Such systems boast pseudo-human inference and greater adaptability than traditional techniques.

A critical component of CI systems is a foundation in what are known as *Bayesian Belief Networks*. These help ground multivariate observations in a more rigorous foundation, by allowing a probabilistic correlation between many variables to be represented as a directed acyclic graph, where each node is a variable and each edge shows a conditional dependency between two variables. Given evidence of the values of one or more such variables, a system can then make probable inferences about the state of other variables to which it is correlated. This highly powerful and non-context specific technique helps train learning and inference in CI systems, and allows for a rigorous representation of uncertainty, cause-and-effect, and forecasting. '*What-if?*' analyses conducted by inputting hypothetical variable values can be used to observe the effect on the system at large[5][16], thus directly addressing many of the key responsibilities of software project managers.

As the number of teams incorporating AGILE methodologies into their management strategies increases, an interest has emerged in advanced techniques for producing accurate cost estimation models. This is another area in which advancements in CI are lucrative. These CI systems typically operate on five families of metrics: *quality*, *quantity*, *time*, *cost*, and *productivity introduced*. For instance, many traditional systems have trouble reckoning with qualitative metrics, such as *quality*, whose value must be reasoned about using a human expert.[18] However, using *fuzzy logic*, we can reason in degrees of truth about the state of a system rather than just via the typical binary (*true* or *false*) representations. A qualitative metric such as 'oldness' can be represented in fuzzy logic as a *linguistic variable* as shown below:

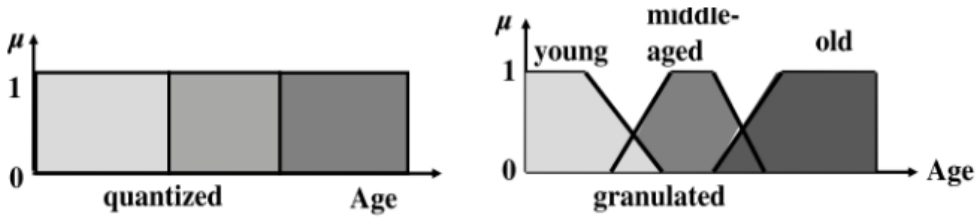


Figure 6: Example of 'fuzzification' of a state variable in a system, courtesy of [18]

Such a system allows a complex qualitative metric such as *quality* to be broken down into a number of qualitative contributory measurements, such as *ease-of-use* and *maintainability*. Using this, one can reason about how adjusting components of the system would affect such qualitative metrics. This allows us to reason about software quality from the same high-level perspective we did at the start of this paper,

except now in computable terms. Further cornerstone techniques in CI have been used for feature weighting and project selection, such as genetic algorithms. These simulate natural evolution, where instead of fitness survival into the next generation is governed by some user-defined fitness function that operates on the current generation of the system. Thus, we can generate increasingly more optimised models that can predict the effort required in person-months to produce a finished product.[18] Effort prediction can also be done via *Artificial Neural Networks* (ANNs), which simulate an organic neuron in terms of memory capacity and the ability to represent related concepts via interconnected neurons. By using a machine learning algorithm called *Back-Propagation*, the system can learn about relationships between the input data (in this case, product and process metrics) and the output (in-person months), by working with a set of training data. For example, the system may be fed a set of data of multiple historical projects and the recorded in-person months it took to complete each project, and the system can then find unobvious patterns in the input data to be able to make a general connections between metrics. We can then apply this model to a current project to obtain and estimation of remaining in-person months, thus making it a general, evidence-based tool for predicting effort.

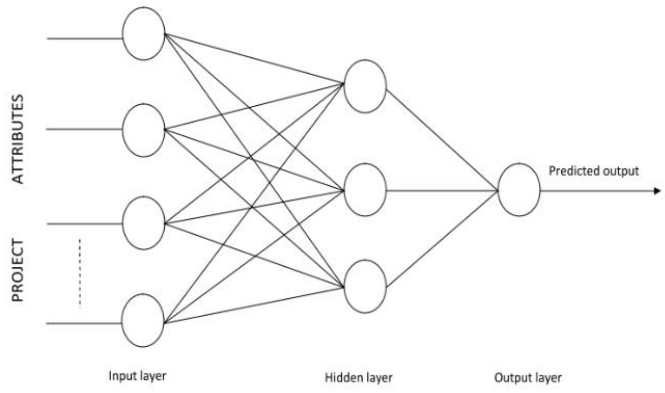


Figure 7: Diagram of an artificial neuron used for the purpose of software engineering measurement, courtesy of [18]

An intriguing generalisation of the Bayesian Belief Network is the *Influence Diagram*, which is of particular interest in modelling social behaviours. Socialisation does not typically lend itself well to precise mathematical modelling; internal states of subjects are immeasurable, behaviour is not guaranteed to be pattern based and may be deliberately altered, actions differ within different contexts, and interactions between actors may not be precisely indicative of the influence one entity has on another. This makes predicting influence-based behaviours difficult to measure analytically. [18]

In the same way as a *Bayesian Belief Network*, the *Influence Diagram* utilises independent time series to predict how much of an impact the state of one actor in a system has on the states of the other actors in the system with which it interacts.[18] These models can also be represented as an *influence matrix*, where the strength of the influence between entities is quantified. Entities need not only represent a single actor (i.e. an employee), but may be used to represent a team, those present at a meeting, a set of

management personnel, etc. The required metrics are easily obtained, by measuring whether the entity is speaking or not speaking, whether it is present or absent in a particular context, etc. Bayesian Belief Networks and Influence Diagrams both suffer from common faults associated with machine learning techniques in that models can only be as accurate as their training data. Nonetheless, sufficient training produces a powerful tool that can predict precipitant state changes among entities that would place other entities into 'desirable states'; those where productivity is improved, training is maximised, etc. Rather than working with expensive process redefinitions, the social dynamics of the business or team can be manipulated to produce benefit.

Management can also gain an overview of interpersonal relationships between staff members using repository mining techniques.[8] As we discussed in section one, metrics for determining knowledgeable team members can help provide productivity increases for new personnel by adjusting team composition accordingly. The knowledgeability of an individual can be estimated based on the number and quality of commits they make to the source code of a project under a number of relevant keywords. Research suggests that software teams are often designated for each new project irrespective of their shared work history[19], ignoring co-workers that have worked together successfully in the past. A solution is to create a network of employees and produce an overview of the common working history of employees for future team assemblage. Using information sourced from the log files of repositories, one can produce a bipartite graph where nodes represent authors and vertices connecting two authors that made changes to the same file. Nodes of high degree suggest core developers for a project. Edges can be weighted to incorporate degrees of collaboration between authors. A low clustering coefficient suggests that developers typically work alone or often with the same teammates, whereas a high clustering coefficient indicates that a team is well acquainted and work together often.

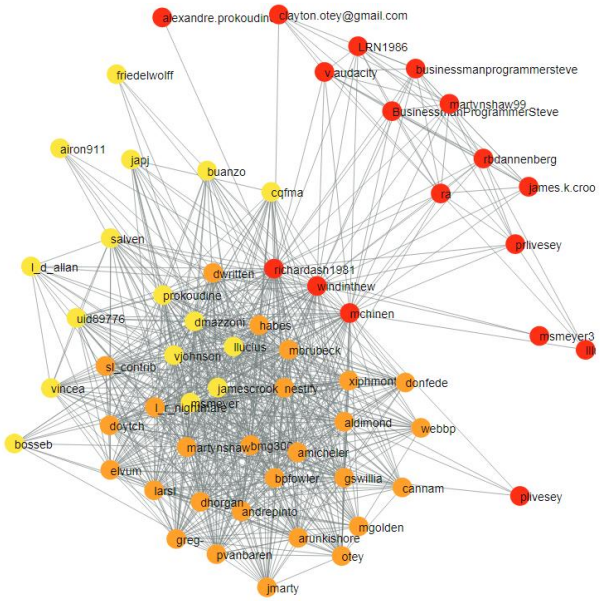


Figure 8: Example graph generated from mining the open source 'Audacity' repository, courtesy of [8]

Algorithmic techniques for processing software engineers' data have evolved considerably since the late 1990's, providing much more powerful insights than simple univariate measurement and allowing for non-analytical techniques in CI to thrive. Additionally, software engineering as a socio-technical discipline has been increasingly recognised. This efficacy has grown in tandem with the rise in general computing power, which has facilitated the practical at-scale application of many of the CI techniques discussed in this section. However, with the increasing power of these techniques and fine-grained measurement of not only employee productivity, but their very behaviour and social-interactions at work, the field of software engineering measurement must engage in a rigorous ethical analysis of the practices in engages in, before allowing these technologies to grow further. This analysis will be the subject of the following final section.

Ethical Considerations

Any new technology with important ramifications on the public's day-to-day quality of life is expected to be subject to scrutiny in the interest of harm-reduction. The '*Big Data Revolution*' has been likened to the Industrial Revolution in terms of its projected societal impact [20], and thus critical ethical issues have emerged regarding how such data should be collected and to what ends it should reasonably be used. These decisions will impact what is considered a 'normal' level privacy compromise for decades to come. However, given the esoteric nature of the subject, it has become harder for lawmakers to exercise sufficient scrutiny to maintain checks and balances. To a concerningly large extent, responsibility has been left to developers and their profit-driven commissioners to ensure data is being used ethically. Thus, software engineers are in the unique position of being both potential perpetrators and victims of unethical data practices which extend to the software engineering measurement processes we have discussed thus far in this essay.

Even in the 1990s, the diffusion of data analytics into software engineering measurement was met with unease from a significant subset of developers. The creators of *HackyStat* originally saw their unobtrusive metrics gathering technology as an asset to developers, but soon accepted that some considered it spyware due to its silent data collection. The program was even referred to as "Hacky-Stalk" in some circles.[11] Privacy concerns have inflated to such an extent in recent years that the modern observer might scoff at the idea that measuring lines of code is 'intrusive'. However, no matter how complex the metric or the domain it stems from, the root cause of the unease is consistent – software is the creation of a developer, and the metric is an emergent phenomenon of the software. Thus, a change to a product or process metric has a human root, and that human can be made accountable.

Indeed, most software development processes are done through a repository system, where every alteration to a piece of code is unambiguously attached to an employee; accountability for mistakes and likewise shortcomings is inescapable. Given the team-based nature of programming, the introduction of such systems may cause developers may push themselves harder in fear of sanctions from not only their employers but from other team members. Studies conducted in supermarkets observed that cashiers work faster when in the vicinity of a more efficient worker in an effort not to appear inferior.[6] This phenomenon may particularly affect minority employees, particularly women, who are more likely than their male peers to have underperformance attributed to lack of ability [22].

Furthermore, the platforms listed in section three boast meticulous employee productivity tracking tools, and the unavoidable data footprint left behind by developers make them perhaps the most vulnerable employee demographic in this regard. Despite the right to privacy being enshrined in the Universal Declaration of Human Rights[23], the intersection of the workplace and privacy is typically not in

the employee's favour. In Ireland an employee's "right to privacy is balanced against the rights of [their] employer to run their business and protect [their] company"[24]. Given the implications for employees of extreme surveillance in the workplace, as is facilitated by technical advancements in the modern day, legislators must seriously consider whether such ambiguous phrasing equally weighs the interest of the employee and the employer. After all, nowadays the software developer is also left with no option to opt-out, meaning they must bargain with their privacy for financial security.

It is important to stress that the tools and platforms deployed in software engineering measurement do not necessitate nor promote their unethical misuse, but nonetheless leave management at liberty to decide to what use they put this data. If profit outweighs human capital, the attempt to maintain ethical integrity is precarious and unballasted. Additional ethical complexity arises since methods are seemingly employed to an impersonal end; indeed, "the raison d'être of software metrics is to improve the way we monitor, control or predict various attributes of software and the commercial software production process".[5] They are used to make easier and more efficient the task of software development, and by extension, the quality of the end product. However, to use this frame of reference is to ignore the fact that the analysis is reached as part of an *on-going process*, whose intermediary metric values are visible to management. As management seeks to make finer and finer tweaks to squeeze additional productivity, it is easy to ignore, deliberately or otherwise, that the direct drivers of these metrics are humans. Furthermore, the arbiters of what constitutes a system 'too unethical' to implement are *themselves* the beneficiaries of such a system. Scepticism in this regard is warranted, as legal incentive has been a driving factor in ethical data handling. The introduction of *General Data Protection Rules* (GDPR) in Europe in 2018 saw 5% more businesses drafting privacy policies, 50% of websites updating their privacy policies, and a majority of websites introducing explicit opt-out mechanisms for data collection.[21] As we have seen time and again, the need to remain competitive in business drives the adoption of unethical practices – and one should not expect software engineering employers are any different.

As management turns the microscope on human behaviour the ethical objections become increasingly incontestable. The first step to performing such analysis is recording data representative of these behaviours. The privacy concerns here are obvious, but one must also consider the issues surrounding application of unsupervised learning to aspects of social measurement that may be used to influence decision making regarding a company's employee base. Any machine being taught to 'think like a human' is vulnerable to incorporating the same cognitive biases that provoke unethical conclusions in the management of human resources. For example, tech hiring platform Gild combs through CVs, social media accounts, GitHub and Stack Overflow interactions to build a model suggestive of traits that are correlated with good programming skills.[25] However, given the training set reflects a strong male bias due to the underrepresentation of women in software development, the profile contains traits that are biased towards stereotypically male interests. For example, demonstrating machine learning's penchant for finding unexpected correlations in data, browsing a particular manga site was found to be a "solid predictor of strong coding". The system is then used to automatically filter applicants to firms such as

Citadel Insurance to find likely candidates, eschewing the need for human oversight which might catch that the omission of such a trait in a female candidates profile was not a reasonable metric to detract from her projected programming ability.

One must also be wary of using these metrics to seemingly ethical ends. We have seen in section one studies showing that improving the happiness of workers can improve their overall productivity, thus create financial incentive for companies to improve employee contentment. However, the inclusion of such a fundamental aspect of human fulfilment as *happiness* being under the jurisdiction of an employer raises concerns. It is important to note that the only reason why data analysis of social data works is because human behavioural patterns are predictable, and because of this management can use techniques to manipulate employee behaviours such that productivity is maximised. We have seen techniques for this in section three relating to influence matrices and social network graphing. However, if happiness becomes just another input factor to a unsupervised learning system optimising for quality or productivity, it may come to the conclusion that happiness is an expendable factor and that much more efficient gains can be made by sacrificing it in place of increasing other metrics that incur productivity.

The ethical issues surrounding software engineering measurement are multiple and complex. What once was questionable with respect to employee privacy has become accepted as employers move towards increasingly nuanced ways of manipulating employee behaviour and their experiences at work to maximise profit. Employees currently lack privacy safeguards, given that there is no hard and fast legal definition of what crosses the line in terms of employee surveillance. As data analytics techniques become more sophisticated, the law lags behind in enforcing a strict ethical standard for businesses practices that enforce norms in the next iteration of the information age, and ensure that such powerful, transformative technologies as data analytics and computational intelligence remain benefits not hinderances to humanity as a whole and software developers by proxy.

Conclusion

This report constitutes a four-part analysis of the field of software engineering measurement. In section one, we examined the importance of wise metric collection, the distinction between product and process metrics, the emergence of social and behavioural measurement, and the relationship between a metric and its source as an imperfect replica.

The second section traced the evolution of software measurement tools and platforms from their inception as a manual data collection process to powerful automated collection, processing, and visualisation tools that are employed by modern companies to produce valuable insights into their development processes.

In the third section, we observed in detail how the huge volumes of data generated in the modern day can be processed by computational intelligence techniques to identify unseen correlations between metrics and provide insights for management purposes that could not be generated by traditional analytical techniques.

In the final section, we examined the nuanced ethical issues that permeate the propagation of the field as a whole. The importance of a legal definition for ethical data usage as a deterrent to unethical business practice was discussed, alongside how the 'blindness' of unsupervised computational intelligence techniques may have an unseen human penalty as social biases are reinforced and human behaviours unethically manipulated.

The field of software engineering measurement is expanding in importance year on year. According to a study conducted by IAG Consulting, an estimated 68% of IT projects fail and that the failure rate is proportional to the skill of the business analysts governing the project.[2] Furthermore, 34% of projects surveyed saw at least two of the following disastrous outcomes:

- The project completion deadline was extended by at least 80%.
- The project satisfied ≤ 70% of the target requirements.
- The overall spending on the project exceeded 160% of the estimated budget.

As IT projects continue to grow in complexity in the modern era, it is only natural to expect that management personnel would be interested in maximising the gains they get from analysing the data left behind by their employees and using it to improve development practices in the future. The field is in the most primitive state it will ever be in, yet advancements are already being made that will revolutionise the way our ever-increasing digital footprints can be analysed to produce quality of life improvements both in our day-to-day lives and in the workplace, but whether this future is to be realised depends on the urgent devising of ethical foundation in data analysis.

References:

[1] Yano, Kazuo, et al. "Measuring happiness using wearable technology." Hitachi Review 64.8 (2015): 517.

[2] Ellis, Keith. "The impact of business requirements on the success of technology projects." Benchmark, IAG Consulting (2008).

[3] Kan, Stephen H. “Metrics and Models in Software Quality Engineering.” Second Edition (2002), p. 25, p. 114

[4] Sillitti, Alberto, et al. "Collecting, integrating and analyzing software metrics and personal software process data." Conference Proceedings of the EUROMICRO, 2003.

[5] Fenton, Norman E., and Martin Neil. "Software metrics: successes, failures and new directions." Journal of Systems and Software 47.2-3 (1999): 149-157.

[6] Lindquist, Matthew J., Jan Sauermann, and Yves Zenou. "Network effects on worker productivity." (2015).

[7] Pan, Wei, et al. "Modeling dynamical influence in human interaction: Using data to make better inferences about influence within social systems." IEEE Signal Processing Magazine 29.2 (2012): 77-86.

[8] Dittrich, Andrew, Mehmet Hadi Gunes, and Sergiu Dascalu. "Network analysis of software repositories: identifying subject matter experts." Complex Networks. Springer, Berlin, Heidelberg, 2013. 187-198.

[9] Achor, Shawn. "Positive intelligence." Harvard Business Review 90.1 (2012): 100-102.

[10] Javdani, Taghi, et al. "On the current measurement practices in agile software development." arXiv preprint arXiv:1301.5964 (2013).

[11] Johnson, Philip M. "Searching under the streetlight for useful software analytics." IEEE software 30.4 (2013): 57-63.

[12] Johnson, Philip M., et al. "Beyond the personal software process: Metrics collection and analysis for the differently disciplined." 25th International Conference on Software Engineering, 2003. Proceedings.. IEEE, 2003.

[13] Johnson, Philip M., and Hongbing Kou. "Automated recognition of test-driven development with zorro." Agile 2007 (AGILE 2007). IEEE, 2007.

[14] Retrieved from https://help.pluralsight.com/help/

[15] Retrieved from https://waydev.co/

[16] Bellini, Carlo Gabriel Porto, Rita De CASsia De Faria Pereira, and Joao Luiz Becker. "Measurement in software engineering: From the roadmap to the crossroads." International Journal of Software Engineering and Knowledge Engineering 18.01 (2008): 37-64.

[17] Siddique, Nazmul, and Hojjat Adeli. Computational intelligence: synergies of fuzzy logic, neural networks and evolutionary computing. John Wiley & Sons, 2013.

[18] Benala, Tirimula Rao, Satchidananda Dehuri, and Rajib Mall. "Computational intelligence in software cost estimation: an emerging paradigm." ACM SIGSOFT Software Engineering Notes 37.3 (2012): 1-7.

[19] Faraj, Samer, and Lee Sproull. "Coordinating expertise in software development teams." Management science 46.12 (2000): 1554-1568.

[20] Richards, Neil M., and Jonathan H. King. "Big data ethics." Wake Forest L. Rev. 49 (2014): 393.

[21] Degeling, Martin, et al. "We value your privacy... now take some cookies: Measuring the GDPR's impact on web privacy." arXiv preprint arXiv:1808.05096 (2018).

[22] Wang, Ming-Te, and Jessica L. Degol. "Gender gap in science, technology, engineering, and mathematics (STEM): Current knowledge, implications for practice, policy, and future directions." Educational psychology review 29.1 (2017): 119-140.

[23] Assembly, UN General. "Universal declaration of human rights." UN General Assembly 302.2 (1948).

[24] Retrieved from https://www.citizensinformation.ie/en/employment/employment_rights_and_conditions/data_protection_at_work/surveillance_of_electronic_communications_in_the_workplace.html

[25] Criado-Perez, Caroline. Invisible Women: Data Bias in a World Designed for Men. New York: Abrams Press, 2019, p. 107.