Date: 20/11/2018
Name: Tobias Hallen
S/N: 16322468
Module: CS3012 (Software Engineering)
Lecturer: Stephen Barrett

Report on the Measurement of Software Engineering:

Specification:

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 ethics concerns surrounding this kind of analytics. 10 pages.

Structure:

This report will be comprised of the above specified sub-headings, namely:

1. Introduction
2. Metrics
3. Overview of Different Algorithmic Approaches
4. Overview of Available Methods/Tools/Platforms
5. Ethical Implications
6. Conclusion

The topics will be discussed in relation to material from the reading list (available at the end of the document), though it will not be referenced/quoted directly often in order to improve readability.

Introduction:

The purpose of this report is to examine the ways in which software engineers and their work may be assessed, methods and tools commonly used to go about this, the effects the constant supervision has on the work environment and productivity of those being monitored, and the ethical implications involved.
The reasons for such monitoring are many, as with the correct approach, one can significantly improve workflow and efficiency across the board on software engineering projects, by isolating and addressing issues found using the data collected.
Monitoring worker efficiency is not a new concept, even in a profession as young as software engineering. The methods used have changed drastically over the years, as with growing software and project complexity, the tools used to gather such data have improved and become increasingly easy and non-invasive to use.
However, detractors of this concept have raised concerns ethical in nature, as the “Big Brother”-esque overwatch of work done and overall efficiency can create unhealthy work environments, or, in the most drastic of cases, border on illegality.

The topic of workplace data collection and measurement is controversial. It is a monumentally complicated task to attempt to measure definitively a developer's worth and contribution to a given project in the workplace. It is easy to cite metrics such as LOC(Lines of Code) or defect rates, but without proper context these data points are worthless in determining actual work done. It is, in fact, this misuse of analytics, among other things, that causes much distrust among the development community surrounding the measurement and collection of efficiency data.

A focal point of this argument is the idea of quality vs. quantity of work. A developer could spend a short amount of time adding a feature using a very hacky method, which would result in an unnecessarily large amount of code. Looking purely at the metrics, this developer's statistics would look very good. High LOC, with (presumably) a low error rate. However, had this method been implemented in a more efficient way, the LOC would be lower.
This can lead to developers gaming the system, if they know what they are being "graded" on. They can focus on those aspects of their work which make them shine statistically, while the actual quality of their work degrades over time.
This is a real concern for data collection and measurement, because without proper handling and understanding of the data, or if it is relied upon too heavily, it will fail. Data collection and measurements should always be accompanied by human review, due to the fact that it is likely impossible to accurately measure the worth of a block of code by algorithm. This creates a need for personal or interpersonal review, such as to avoid this issue.

There is, of course, a more emotional side to the misgivings about this topic. Many people simply do not like to be compared to their peers, or rather, would not like to have their flaws exposed. Also, there are significant privacy concerns when talking about monitoring an employee's every move. Even in the workplace, there should exist a modicum of privacy, as for many people, while they may not have anything to hide, having a company or management look over their shoulder at all times would be a detriment to their work, as it can cause nervousness and a fear of failure or mistakes.

It is important to discuss these concerns before the next section on the different metrics used in order to be able to draw back on arguments which can be made for and against these metrics, mentioned earlier in this section. Because as the tools which are used to collect data have changed, so too have the metrics we are able to collect with them, each with their own benefits and drawbacks.

Metrics

When it comes to actually measuring the work and efficiency of software engineers, many different metrics have been used over the course of software engineering history.
These metrics have improved drastically, and have moved from directly correlating to an engineers performance to rather monitoring the health of a given project instead.
This section will contain a list of some of the more common metrics which have been used to measure the software engineering process.

The first, and simplest metric is LOC. Lines of code has been used since time immemorial, as it is a very easy to gather, simplistic way of looking at a project's progress. Obviously not a particularly accurate measurement, LOC serves to estimate the size of any given project, though not its complexity.
Of course, a great deal of Lines of Code does not equate to a complex project, nor does it equate to a large volume of work done. In fact, use of this metric is likely to inspire some form of "gaming" in the affected developers' work, as it is quite easy to artificially inflate this number by writing poor code.
Though LOC does not bestow an inherent value upon a piece of work, it does serve a purpose in giving you exactly what it says - the amount of lines of code in the program. With the correct experience and context, and alongside some sort of human code review, it can serve as a somewhat useful metric in analysing software engineering, though decidedly not when used alone.

Where LOC seemingly measures the quantity of work produced, though possibly with less-than-accurate results, a metric which estimates the quality of a developer's work is the Error Rate of their submitted code.
Error rate is representative of the exhaustiveness of the developer's testing. The higher the error rate, the clearly lower the quality of testing that the code is put through. Error rate can be gathered from a number of sources, be it before even the testing stage or after the code has been added to the working product.
Depending on where the error rate occurs, it can tell you different things about the state of the project's health - if found at the end stage, then QA isn't being rigorous enough in testing. Also, naturally, error rate is generally not exhaustive, as the metric relies on errors which have actually been found, so the error rate could possibly be much higher than indicated.

Another interesting metric is Code Churn. Code churn is representative of the number of changes made to a specific piece of code over a short period of time. High code churn values are representative of a low quality of code, as it seemingly needs to be revised repeatedly to make it work properly.
Of course, this metric is not necessarily indicative of anything. Much like any other performance metric, it can be seen as more of a smoke signal, which could *possibly* be indicative of a 'fire', or issue.
Churn is likely to be significantly higher during the initial stages of a new, ambitious project - where a team is attempting different iterations of solutions etc. As a project nears its end stages, code churn has been known to reduce, as the core problems become solved.

Task completion speed is another, rather self explanatory metric. It relates to how quickly developers finish any given task. Tools have been developed which are used to collect this data manually (see “Time Trackers” section below). This data point is fairly simple to interpret, even to a layperson.
A long task completion time is indicative of either a challenging, complicated task or of an inefficiency in the work pipeline.

This is just a small sample of the metrics which can be used to measure software engineering. Overall, it is clear that without proper context, none of these metrics is particularly useful, nor do they identify problems when examined in a vacuum.
As mentioned above, these data points are merely smoke signals, which show you where to look to find an issue, rather than identifiers of the root cause of these problems. It is then the responsibility of the user to find the root of these issues and fix them.

These metrics, among many others, are used across the board in the software engineering world, as they grant invaluable insight into potential problems and help users in maintaining a clear overview of a given project, no matter how complex.

Overview of Different Algorithmic Approaches to Measuring Software Engineering:

While designing an algorithm to measure to overall effectiveness of a software engineer is likely to be an impossible task for a long time, there are algorithms available which may be used to analyse their work, which, in turn, reflects on them.
There are a vast number of such algorithms which have been developed over many years, so this will only touch on one of them.

Arguably the most well-known, and simplest of these, is the cyclomatic algorithm, which seeks to measure program complexity.
It is calculated by developing a Control Flow Graph of the program which computes the number of linearly-executable paths through the code. The nodes on the graph correspond to indivisible groups of commands in the program, where a directed edge may connect two nodes if the following node may be executed directly after the first node.
For example, if the code contained no control flow statements (ifs, ands, ors, etc.), then the cyclomatic complexity of that program is 1, as there is only one possible linear path to be taken through the program.
Mathematically, the algorithm appears as such:

$$M = E - N + 2P ,$$

Where:

- M is the complexity
- E is the number of edges within the control flow graph
- N is the number of nodes within the control flow graph
- P is the number of connected components

Cyclomatic complexity is argued to be a poor way of accurately measuring complexity, as certain types of flow control statements are much more difficult to follow when reading through a program, but the algorithm weighs them all more or less equally. This can lead to situations where two functions of vastly different levels of readability express the same cyclomatic complexity value.
However, there is a valid use case for this algorithm - namely to serve as an upper bound for the number of tests which need to be run in order to obtain optimum code coverage. Despite its age and questionable practicality, the cyclomatic algorithm serves as a simple example of the type of algorithmic analysis which may be performed on code, and the information which can be garnered from it.

One may also attempt to algorithmically deduce the estimated cost of a project. In its most general of forms, an algorithm to determine the cost of a given project can be given as:

$$Effort = A * Size^B * M$$

Where:

- A is a constant factor depending on organisational practices and the type of project
- Size is an assessment of the project size/scope
- B reflects the disproportionate effort involved in larger projects, which would not scale linearly
- M is a multiplier to account for project stages

While this sort of equation does not directly yield information about the software engineering process, it is a useful metric to compare to actual costs once the project has had some time put into it or is finished completely.
Comparing the actual costs of completing a project versus the estimated costs, however accurate they may be, can, with analysis of where the money went, give insight into how the funds could have perhaps been used more effectively elsewhere.

As one can see, while there is no direct way of measuring a software engineer's efficiency using an algorithm, it is entirely possible to analyze their work algorithmically, and the comparison between cost estimation and actual cost can yield useful information about the software engineering process and cost, and how to maximize one's performance.

Overview of Available Methods/Tools/Platforms:

Something to bear in mind throughout this section is that there is no method of Data Collection and Code Review without flaw. No method can accurately capture every aspect of the Software Development workflow alone. Each system has its own benefits and drawbacks, and users must be mindful of these in order to tailor them to their specific needs. Additionally, when it comes to data collection, one must be aware of the fact that data collected is just that - data. Data in a void is useless, and the key to extracting use from it is context.

As mentioned above, data measurement has been in the game for a long time for software engineering and development.
To begin with, software engineers were expected to “review” themselves. Essentially, each time they completed a portion of work, they would have to log data concerning their production of that work manually. This is known as self-evaluation. It is a relatively widespread practice throughout many industries. There are a number of issues with this, most simple, of approaches.
Firstly, and perhaps, most importantly, the overhead required for this system to function properly is considerable. For the developer, it interrupts their workflow every time they complete a task, leading to a tedious cycle of *do some work* → *log that work* → etc. This overhead reduces the total amount of work actually completed, because the developer is so caught up in having to record their own work data.
Additionally, this method of course can lead to bias. If the developer logs their own data, there is little stopping them from fudging certain points to make themselves look better, or others look worse, to advance themselves in the workplace.
This bias could, of course, be avoided to a certain extent, by instead performing peer reviewing of code, or pair programming.
However, the scope of the data collected is likely to still be relatively limited, as the aforementioned overhead consumes a large amount of work time.
However, the benefit of this type of review is that the information can be tailored specifically to each employee by themselves. This means that sources of data, which would not be available in a standardized package, are available for collection and study. If, for example, employee A notices that their workflow is impeded by morning chatter amongst colleagues, that data can be recorded and adjustments can be made to improve efficiency and reduce distraction.
The efficacy of self-review, however, relies on the concept that each employee using it has self-improvement and efficiency in mind and will strive towards their own goals. This is, of course, not always the case, which is why standardized data samples are generally more useful, though perhaps not as specifically insightful into individual employees.

As time has gone on and the industry has developed, projects and code have become increasingly complicated. Alongside the growth in complexity of projects, the tools used to measure productivity and efficiency have grown accordingly.
As mentioned above, the overhead required for each employee to perform self-evaluation is significant. However, with the tools which will be covered below, the overhead for those individuals being measured becomes almost non-existent, as all data collection comes passively and automatically, requiring no effort from the user.

As mentioned above, nowadays, there are a plethora of commercial-grade tools available to firms wanting to collect data on their employees work. This section will cover a small handful of popular examples.

Data Collection Tools:

The best (read: only) way of accurately measuring large-scale projects or even individual developers' development process is to gather a large volume of data first, and then analyse it using various tools. Some tools (discussed below) bake the information gathering into some form of utility, such as a timer, whereas others simply collect the data for analysis at a later point in time.

An example of such a data collection tool is Hackystat. Hackystat is an open source data collection, analysis, and visualization software which allows users to non-invasively monitor the software engineering process.
This is typically done by attaching Hackystat 'sensors' to their development tools, which collect data concerning their development process, which is forwarded to a web service for storage and querying.
While highly unobtrusive and intended for personal use only, Hackystat, like many other tools listed here, has the potential to be abused or misused by higher-ups with access to the data, but not the technical know-how to understand it, which would lead to misinterpretation. However, that aside, Hackystat is an enormously useful and fully featured data collection service.

Project Management Tools:

One of the most basic methods of measuring the development process is to first create a tool with which to plan its direction. Knowing the course the development process is likely to take makes it easier to display metrics and data actually pertinent to the project itself. There are a host of such management tools available to the market.

Jira is a tool created by Australian company Atlassian, to be used for bug tracking, issue tracking and project management. It is offered in three different packages, each suited for a different use case:

1. Jira Core, the generic option, used for project management
2. Jira Service Desk, which is tailored towards use in the IT or technical support sphere
3. Jira Software, which features agile project management features.

Jira is, as mentioned, geared towards use in agile development environments. It is very full featured, allowing the users to plan their approach, track their work and then view a report of team performance based on real-time, visual data pertaining to their development process so far.

Trello is a simpler alternative for those who wish to track project development but are not looking for a solution as fully-featured as Jiro. Essentially a to-do app, Trello provides users with a method of tracking which tasks have been completed, and which still remain. A management tool in its most simplest form, but easy to use and lightweight, which lends itself to smaller projects.

Time Trackers:

Another method of development analytics, is the simplest possible concept - the time tracker.

An example of a popular time tracking service is Toggl. Toggl is a multi-platform service which allows you to track what type of work is consuming the greatest amount of your time, and determines how much that time is worth, and whether it may be better spent on other tasks. The premise is very simple - one types in what one is working on, hits start, and then a timer begins; when done, it is as simple as hitting stop to finish the timer.
This is an example of a tool which - while providing a very useful metric - is relatively invasive when compared to many other measurement tools. It serves as a constant reminder that your time is being “used up”, and can put a lot of pressure on developers to get their work done as quickly as possible. Packaged as being user-friendly, the reality is far harsher. Accurate timer results paint an easy target on the back of those developers taking their time, even though it is not always at their own fault.
This results in a massive increase in stress for those being monitored, always fearing that management will misconstrue their results, which can of course lead to behavioural changes and the rushing of work, rather than being diligent.
Nevertheless, toggle provides a very tangible metric for measuring how much money your time is worth, which may be seen as a good or a bad thing, depending on who is measuring whom.

Software Forensics:

Software forensics tools serve to identify issues with your code, by analysing it for patterns, and predicting the direction it will take in the future. As a concept, it promises to identify several different types of issues within your project.

Codescene, for example, is a tool which can be used to analyse github (or other vcs) repositories for a number of different measurements, notably code churn, code age, change frequency and main authoring.
These metrics can in turn be used to generate a comprehensive model of project health, by identifying so-called “hot-spots” in your code, or areas which feature the most active development activity, as they may be leading to issues, or may lead to issues in the future. This is because areas of code which are frequently changed are often incurring so-called “technical debt”. Technical debt is a concept which reflects the extra work which may be incurred in the future by choosing a programming approach which, while simple now, may lead to systemic problems down the road.
Codescene can also be used to identify developers which are being less-than-helpful to project health, e.g. by writing hard code and causing an inordinate number of hotspots, or by writing code which cannot be maintained by another member of the team.
Of course this tool is also open to data misinterpretation, as different team members may be working on different tasks with varying levels of difficulty, but it provides a user interface

which displays a comprehensive view of the data collected during the software engineering process on a project scale.

Ethical Implications:

The ethical concerns of monitoring software engineering as a concept are manyfold. The question must be raised - when is user data private property and when is it owned by whomever is paying for the work?

It is a contentious subject, certainly. On one hand, the tools themselves are not unethical. When used privately, they provide a simple methodology for improving one's own work. Used, however, at the industrial or corporate scale, use of these tools can be seen as an invasion of privacy.

It is understandable of course, why a person would not like to be monitored at work 24/7. It is almost akin to having someone from management standing over one's shoulder at all times, watching one work. It can be unnerving. Additionally, as mentioned throughout this report, it can easily lead to all sorts of 'gaming' if not used with care.

Of course, if the developer is not made explicitly aware of the data that is being collected from them, then one can say that the information gathering is immoral, as the party being collected from cannot consent to it without knowledge of its existence.

Additionally, it is entirely possible that, even if certain forms of data collection are consented to, inferences can be made from those metrics which were not consented to explicitly. This is an example not of the unethicality of data collection, but rather how it is made use of and treated afterwards.

However, when a person signs on to a project, they generally sign away the right to privacy when it comes to their user data pertaining to work on said project, or are made aware of it in some other fashion. Is that moral? The mantra "Shouldn't have anything to hide" rings true here. Certainly, when working on the employer's time, on the employer's project, the information gathered should rightfully be theirs, no? What is essentially a contract between consenting adults cannot be considered unethical. Though some may disagree, the rest of this section will accept that to be generally true.

If the information gathering itself is moral (assuming consenting parties), then what are the ethical implications of handling said data? Depending on what is collected, and the decisions and actions resulting from the analysis of that data, much injustice could be done.
Say the tools used by company A collect data on the efficiency of software developers when working on a given project, and it is found that there is a negative correlation between the quality and efficiency of work done by a software developer and their given age group. Is it then moral for the company to pursue action against all their employees of that age group? Obviously not.

However, say that company was instead to find positive correlations between certain pairs of employees working together and their work, which lead them to favouring certain pairings and groups of people when deciding whom to assign certain projects. Is that inherently immoral? Unlikely, barring any other factors.
Therefore, I posit that it is not with the collection of data that ethical quandaries lie, but rather with the actions that follow, based on the information gathered. As long as the information gathered is consented to, there is no ethical issue, as both parties assent to the arrangement, barring a leak or sale of said information to a third party, which was not consented to.
No, it is the actions which are made based on the information gathered that potentially have ethical implications. The sheer mass of data gathered can be used in a host of malicious, unethical ways, though this is not the fault of the process of information gathering, but rather, of the party who owns the information gathered.

Closing Thoughts:

Clearly, software engineering data collection is a complicated topic. Data collection is a stable in most types of industry, so why is it different for software engineering? Perhaps because it is difficult to accurately quantify the engineering process, and directly compare it to another person, or piece of work.

Perhaps this is due, in part, to the variance and scope of the field, that the differences between two pieces of work can simply be too large to compare them to each other in any meaningful way, and that it is therefore difficult to paint a complete picture of the software engineering process. As the industry grows, this will ostensibly become easier, as more and more firms will create larger databases of information for more specific aspects of the development process, that will be more easily compared to one another.

Perhaps it is due, in part, to the fact that often software engineering can be likened to art rather than science. Give to software engineers the same issue, and you are unlikely to get the same result out of both. There is a certain measure of creative license associated with software development, and while there is usually an empirically ‘correct’ way of doing things, establishing what that method is is an integral part of the development process itself.

Regardless of how difficult it may be to draw comparisons in the realms of software engineering analytics, it is undeniably an essential part of improving one’s own work. The metrics provided are invaluable in finding issues in one’s process and correcting them, or refocusing on aspects of work which need it.

Despite the difficulty in interpretation, and the potential for misuse, it has to be said that the potential negatives are outweighed by the likely huge benefits software engineering analytics and the tools provided will have on the industry in the long run. As time goes on, tools will only become more and more sophisticated, and in general, software engineering and development will become even more refined than it already is.

Bibliography:

(Note: this is the recommended reading list we were supplied with to study for this report. It is not an exhaustive list of all resources consulted, but gives an idea of the type of information used)

- https://www.youtube.com/watch?v=Dp5_1QPLps0
- http://www.citeulike.org/group/3370/article/12458067
- http://2016.msrconf.org/
- http://www.nextlearning.nl/wp-content/uploads/sites/11/2015/02/McKinsey-on-Impact-social-technologies.pdf
- http://s3.amazonaws.com/academia.edu.documents/35963610/2014_American_Behavioral_Scientist-2014-Chen-0002764214556808_networked_worker.pdf?AWSAccessKeyId=AKIAJ56TQJRTWSMTNPEA&Expires=1479122144&Signature=pO6ZkNbSZgbNTqHMd7xyIdMV5iE%3D&response-content-disposition=inline%3B%20filename%3D2014_Do_Networked_Workers_Have_More_Con.pdf
- https://www.researchgate.net/profile/Jan_Sauermann2/publication/285356496_Network_Effects_on_Worker_Productivity/links/565d91c508ae4988a7bc7397.pdf
- http://www.hitachi.com/rev/pdf/2015/r2015_08_116.pdf
- http://patentimages.storage.googleapis.com/pdfs/US20130275187.pdf
- Fenton, N. E., and Martin, N. (1999) "Software metrics: successes, failures and new directions." Journal of Systems and Software 47.2 pp. 149-157.
- Stephen H. Kan. 2002. Metrics and Models in Software Quality Engineering (2nd ed.). Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA.
- Grambow, G., Oberhauser, R. and Reichert, M (2013) Automated Software Engineering Process Assessment: Supporting Diverse Models using an Ontology. Int'l Journal on Advances in Software, 6 (1 & 2). pp. 213-224.
- Dittrich, Andrew, Mehmet Hadi Gunes, and Sergiu Dascalu. (2013) "Network Analysis of Software Repositories: Identifying Subject Matter Experts." Complex Networks. Springer Berlin Heidelberg. pp. 187-198.
- W. Pan, W. Dong, M. Cebrian, T. Kim, J. H. Fowler and A. S. Pentland, "Modeling Dynamical Influence in Human Interaction: Using data to make better inferences about influence within social systems," in IEEE Signal Processing Magazine, vol. 29, no. 2, pp. 77-86, March 2012.
- P.M. Johnson et al., “Beyond the Personal Software Process: Metrics Collection and Analysis for the Differently Disciplined,” Proc. 25th Int’l Conf. Software Eng. (ICSE 03), IEEE CS, 2003, pp. 641–646.
- Johnson, Philip M., and Hongbing Kou. "Automated recognition of test-driven development with Zorro." Agile Conference (AGILE), 2007. IEEE, 2007.
- Johnson, Philip M., et al. "Improving software development management through software project telemetry." IEEE software 22.4 (2005): 76-85.
- Grambow, G., Oberhauser, R. and Reichert, M. (2013) Automated Software Engineering Process Assessment: Supporting Diverse Models using an Ontology. Int'l Journal on Advances in Software, 6 (1 & 2). pp. 213-224.
- Taghi Javdani , Hazura Zulzalil, Abd. Azim Abd. Ghani, Abubakar Md. Sultan, On the current measurement practices in agile software development, International Journal of Computer Science Issues, 2012, Vol. 9, Issue 4, No. 3, pp. 127-133.
- W. Snipes, V. Augustine, A. R. Nair and E. Murphy-Hill, "Towards recognizing and rewarding efficient developer work patterns," 2013 35th International Conference on Software Engineering (ICSE), San Francisco, CA, 2013, pp. 1277-1280.
- Martin P. Robillard, Wesley Coelho, and Gail C. Murphy. 2004. How Effective Developers Investigate Source Code: An Exploratory Study. IEEE Trans. Softw. Eng. 30, 12 (December 2004), 889-903.
- G. C. Murphy, M. Kersten, and L. Findlater, “How are Java software developers using the Elipse IDE?” IEEE Software, vol. 23, no. 4, pp. 76–83, Jul. 2006.

- P. M. Johnson, H. Kou, J. Agustin, C. Chan, C. Moore, J. Miglani, S. Zhen, and W. E. J. Doane, “Beyond the personal software process: Metrics collection and analysis for the differently disciplined,” in Proceedings of the 25th international Conference on Software Engineering. IEEE Computer Society, 2003.
- A. Sillitti, A. Janes, G. Succi, and T. Vernazza, “Collecting, integrating and analyzing software metrics and personal software process data,” in Proceedings of the 29th Euromicro Conference. IEEE, 2003, pp. 336– 342.
- E. B. Passos, D. B. Medeiros, P. A. S. Neto and E. W. G. Clua, (2011) "Turning Real-World Software Development into a Game," Games and Digital Entertainment (SBGAMES), 2011 Brazilian Symposium on, Salvador, 2011, pp. 260-269.,
- L. Singer and K. Schneider, "It was a bit of a race: Gamification of version control," Games and Software Engineering (GAS), 2012 2nd International Workshop on, Zurich, 2012, pp. 5-8.
- Silverman, Rachel (Nov 2, 2011). "Latest Game Theory: Mixing Work and Play — Companies Adopt gaming Techniques to Motivate Employees". Wallstreet Journal.
- Hassan, A.E. and T. Xie, Software intelligence: the future of mining software engineering data, in Proceedings of the FSE/SDP workshop on Future of software engineering research2010, ACM: Santa Fe, New Mexico, USA. p. 161-166.
- Di Penta, M. (2012) Mining developers' communication to assess software quality: Promises, challenges, perils. In Emerging Trends in Software Metrics (WETSoM), 2012 3rd International Workshop on. IEEE.
- R. Want, A. Hopper, a. Veronica Falc and J. Gibbons. The active badge location system. ACM Trans. Inf. Syst., 10(1):91–102, 1992.
- Pentland, A. (2014). Social physics: how good ideas spread-the lessons from a new science. Penguin.
- E. Murphy-Hill and G. C. Murphy, “Peer interaction effectively, yet infrequently, enables programmers to discover new tools,” in Proceedings of the ACM 2011 Conference on Computer supported cooperative work, CSCW ’11. ACM, 2011, pp. 405–414.