Sean Hassett

CS3012

Software Engineering Measurement Report

Motivation

Before talking about methods of measuring software engineering, I want to briefly talk about the motivations for measuring at all. In what ways does it benefit employers or engineers and in what ways does it harm them?

The intended benefits for employers are plain enough. Having measurement data on their engineers helps them check that they are hiring productive people and are not paying people for low output. It also helps to predict deadlines and measure how close to the schedule a project is progressing. The engineers get to see data on their own performance. By scrutinising the time spent on projects, including the time spent fixing errors, they can see whether or not they are working efficiently or whether a certain practice that they have is hindering them.

The possible harm of this data collection is also plain enough. Engineers can feel extra pressure under the knowledge that their work is being analysed on a daily, maybe even hourly basis. It can lead to a stressful work environment where downtime might be frowned upon. It can also lead to unhealthy competition between employees where a week of lower productivity might lead to shame within the team. The detriment to employers is maybe losing good engineers who are unhappy working in such an environment, and a less attractive image when hiring.

So the goal seems to be finding methods of measuring engineering such that employers and employees both benefit while avoiding the potential pitfalls associated with such measurement. Finding a balance between overly invasive measurement and measurement which provides no useful insight.

Methods of Measurement

Personal Software Process

The Personal Software Process (PSP) is a system of measurement designed to increase the productivity of individual engineers, without taking into consideration the performance of the team or the success of a project. Individual engineers can excel even if their team is lacking or their project is failing. The PSP is solely concerned with the individual. It was designed by Watts Humphrey who took the principles of his Capability Maturity Model, which was designed to measure development processes, and applied them to measuring an individual.

The PSP is highly data-oriented and requires the individual engineer in question to manually record data about their performance. It's intention is to improve their ability to estimate deadlines for their work and reduce the number of defects. There are four process stages to the PSP:

1. Personal Measurement, wherein the engineer gathers data on the time they spend and the defects they find. There are three stages to this measurement: planning, development and postmortem.
2. Personal Planning, wherein the engineer uses the data they have gathered to make estimations on development time for new projects.
3. Personal Quality, wherein the engineers uses the data they have gathered to creates checklists for design and code review.
4. Scaling Up, wherein the engineer combines processes in a cyclic fashion using an iterative enhancement approach to work on large scale projects.

The data gathered using the PSP gives an engineer measurements on development time, i.e. how long it takes them to complete tasks, and on defects, i.e. changes that they had to make to their original design or code in order to successfully complete a task. These measurements help engineers to gauge how long it will take the to complete future tasks and may help in preventing future defects.

There are criticisms of the PSP and chief among them is the amount of development overhead added to tasks by having engineers manually recording data.

Some also point out that the large amount of manual data collection inevitably leads to errors and consequently, inaccurate results.

Agile Measurement

Measuring software engineering in agile software development is a different task to measuring traditional approaches to software development. Agile focuses on releasing quickly and releasing often with little comprehensive documentation. The desired result is fast delivery of product and adapting to changing user requirements, with a focus on simplicity and iteration leading to continual improvement. User requirements are defined as user stories and there are different approaches to estimating the effort it will take to satisfy the requirements.

Subjective estimation is a popular approach, where the developers will use their own judgment to estimate the effort. Some agile teams will also use previous iterations in order to estimate the remaining effort required, however this data is generally only useful within the scope of the same user story. Another popular method is the scrum, in which teams meet for a short time on a regular basis to give updates on progress and thus estimate the effort remaining.

Velocity is a term in agile development which can be seen as an analogue to productivity. Velocity is defined as the number of completed user stories per iteration and can also be used for estimating effort. Velocity is sensitive to some factors which are subject to change. For example, changes to the composition of a team will render the team's velocity measurement as invalid. A good measurement of velocity is obtained by measuring for several successful iterations. Even still, velocity is seen as an indication of past performance and not a guarantee of future results.

A burndown chart (see Fig. 1) is a tool used to plan and monitor the progress of a project. A line is drawn indicating the estimated time until project completion, while the actual time it takes to complete tasks is superimposed onto the graph as tasks get completed. This gives the team an idea of how closely they are following their estimated time, and whether they need to speed up or cut some of the planned features in order to meet the deadline.

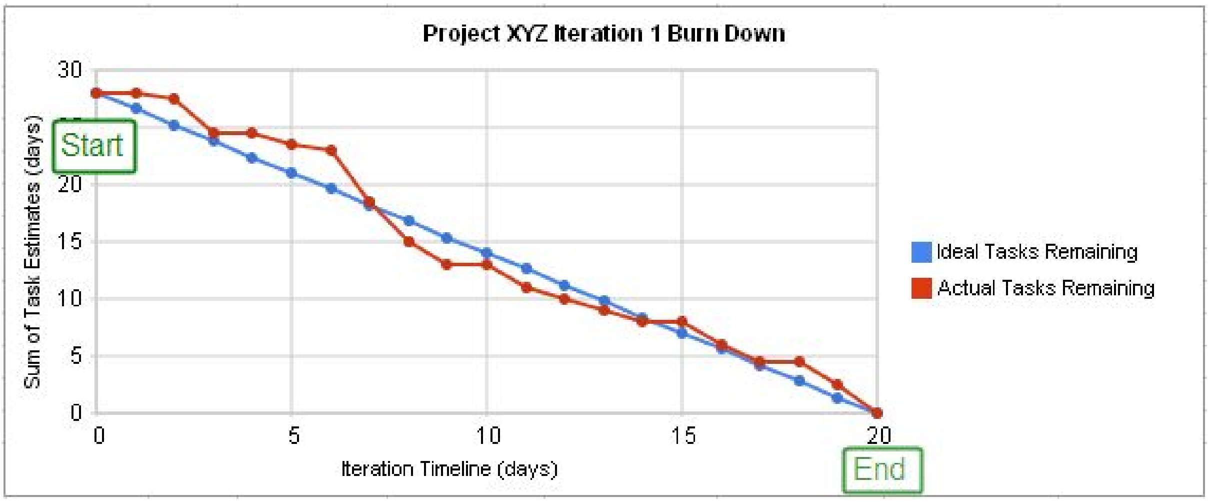

Fig 1. Burndown chart

A Cumulative Flow Diagram is another chart that was intended as more informative replacement for the burndown chart. It works in a similar manner to the burndown chart but separates tasks into more states than just “finished” or “not finished”. In Fig. 2, tasks are separated into five states, “Not Started”, “Design”, “Coding”, “Testing” and “Complete”. This visualisation allows the team to identify the division of time spend on each phase of task completion and check for bottlenecks. For example, it would be easy to see if too much time was being spent on Design vs Testing and it helps the team to increase throughput and meet deadlines with greater ease.

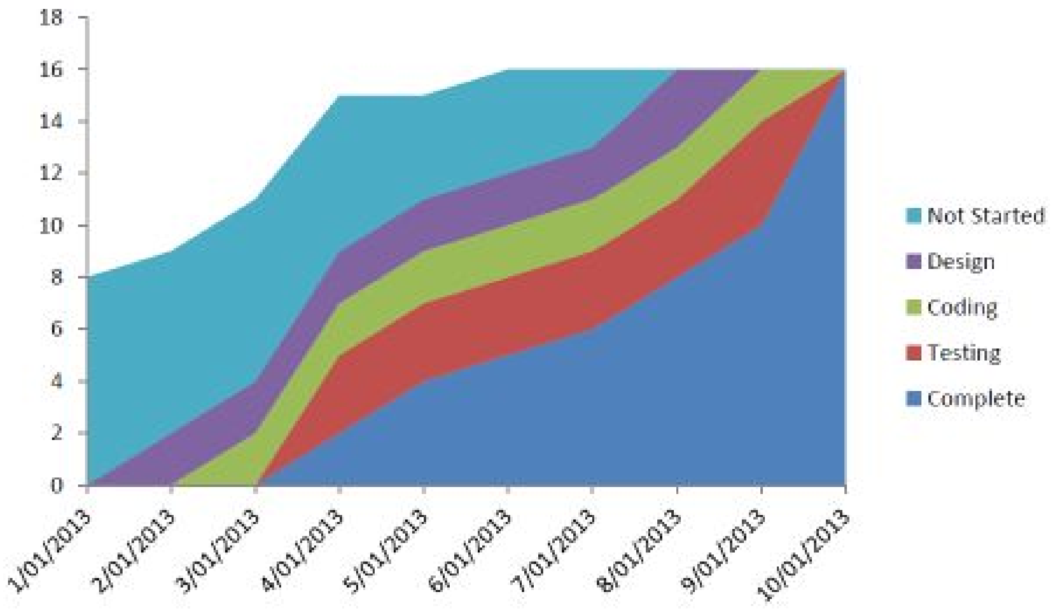

Fig. 2 Cumulative Flow Diagram

Computational Platforms

Leap

Leap is a toolkit which attempts to compensate for the problems of manual entry in PSP by automating some of the data analysis. It was designed by the Collaborative Software Development Laboratory (CSDL) The developer still manually collects the data regarding their performance but the Leap toolkit lightens the load somewhat by attempting to perform some of the analysis automatically once it is given the data. Leap operates in the spirit of the PSP. It maintains a repository of data and analysis that pertains to the individual developer. This developer is anonymised by removing reference to their name within the data and the repository is portable, allowing the developer to carry it over to new projects and new workplaces.

Leap is a toolkit designed to aid the individual developer in following the Personal Software Process. It did not solve all of the problems inherent in the PSP, namely the amount of time a developer had to spend manually inputting data. The CSDL began to see that fully automating the PSP was maybe an impossible task, and that the overhead of manual data entry might not be worth the results provided by the PSP, especially in the world of agile development. To that end, they began work on a new project.

Hackystat

Hackystat's early goals were to depart from the time-consuming manual data collection of the PSP and explore what kind of metrics and analysis could be obtained by automating both the data collection and analysis aspects of measuring development. The CDSL claim that Hackystat was built with four important design features:

1. Client and Server-side data collection. This lets Hackystat collect data from development on local workstations as well as server-based activities.
2. Unobtrusive data collection. A big contrast to the manual PSP, the CDSL claim that with Hackystat, "users shouldn't notice that data is being collected".

3. Fine-grained data collection. Hackystat can collect data on a minute-by-minute or even second-by-second basis. This allows Hackystat to “track a developer as he or she edits a method, constructs a test case for that method, and invokes the test, yielding insight into real-world test-driven development”.
4. Personal and group-based development. In another departure from the PSP, Hackystat allows the option to define a project and collect data on how team members work together on the project.

Hackystat collects metrics on things like development time, which it estimates by monitoring how long a developer spends in an IDE interacting with files; commits by measuring the amount of commits a developer makes to the project repository and how many lines of code is in each commit; built time by monitoring how often a developer builds the project and how many builds are successful and finally testing by monitoring how often tests are run on the system and how often they complete successfully. These metrics together allow overseers of projects to see how the developers adhere to the company’s policy. For example, under the agile methodology, you would expect to see many commits and many successful builds.

Zorro

Studies in the benefits of Test-Driven Development had lead to mixed conclusions, with some studies showing it had a positive effect on the software produced and others saying it had a negative effect. Johnson and Kou posit that the reason for this was difficulty in getting software developers to both understand exactly how TDD is supposed to work and getting them to comply with the practice properly during development. Zorro is designed to automatically monitor whether developers are complying with an operational definition of Test-Driven Development.

Zorro is built on top of Hackystat which it uses to automatically collect the low-level activities of the developer. It also uses an application of Hackystat called Software Development Stream Analysis (SDSA) which takes the data from Hackystat and organises it into a chronological sequence of “episodes” which denotes the development stream. Episodes consist of “events”, examples of which include calling a unit test of compiling a file.

The Zorro model consists of 22 episode types divided into 8 categories. This makes it more comprehensive than the simple red/green/refactor model, where the developer introduces tests that fail initially, changes the code to pass the test and then refactors the code while keeping the test passable. The Zorro model also indicates the complexity in trying to monitor TDD in active developers. One of the important aspects of Zorro is that it is flexible enough that users of Zorro can configure their own operational definition of TDD and tweak the system to monitor for this definition, and this helps to remedy the issue of conflicting definitions encountered in TDD research. Since it utilises Hackystat for the base data collection, it carries the benefits (and drawbacks) of that system, primarily the quick and automatic collection of data.

Algorithmic Approaches

COCOMO

The Constructive Cost Model (COCOMO) is a procedural software cost estimation model developed by Barry W. Boehm. The model parameters are derived from fitting a regression formula using data from historical projects (61 projects for COCOMO 81 and 163 projects for COCOMO II).

There are three levels to the COCOMO model: Basic, Intermediate and Detailed. On top of that, there are three modes of the model: Organic, Embedded and Semi-Detached. The levels determine the accuracy of the model's prediction. Basic requires less work to calculate but gives a less accurate time prediction. Detailed requires a lot of factors but gives a more accurate prediction. The mode is determined by the scale of the project. Organic is used for small projects, Semi-Detached for medium projects and Embedded for large projects.

In the Basic COCOMO model, the formulae used for predictions are straightforward. If we take E to be the effort required to complete a project measured in person-months and S to be the size of the project in thousands of lines of code, then the relationship between E and S in each mode of the model is:

Organic:	$E = 2.4 * (S^{1.05})$
Semi-Detached:	$E = 3.0 * (S^{1.12})$
Embedded:	$E = 3.6 * (S^{1.20})$

The duration of the project in calendar months (TDEV) can then be determined with the following formulae:

Organic:	$TDEV = 2.5 * (E^{0.38})$
Semi-Detached:	$TDEV = 2.5 * (E^{0.35})$
Embedded:	$TDEV = 2.5 * (E^{0.32})$

The Basic model of COCOMO uses only one predictor variable, S. This means that the only influence in the prediction of the required effort and duration is the size of the project. The Intermediate model uses fifteen more predictor variables to make the prediction. These are:

1. Required software reliability
2. Database size
3. Product complexity
4. Execution time constraint
5. Main storage constraint
6. Virtual machine volatility
7. Computer turnaround time
8. Analyst capability
9. Applications experience
10. Programmer capability
11. Virtual machine experience
12. Programming language experience
13. Use of modern programming practices
14. Use of software tools
15. Required development schedule

The manager of the project determines the magnitude of each of these factors in a range of six values from “Very Low” to “Extremely High”. Each value in the range has a numerical value associated with it and the product of all fifteen of these values is taken as the variable I. There are then more formulae to determine a nominal effort required (ENOM):

Organic:	$ENOM = 3.2 * (S^{1.05})$
Semi-Detached:	$ENOM = 3.0 * (S^{1.12})$
Embedded:	$ENOM = 2.8 * (S^{1.20})$

With ENOM, E can then be determined with:

$$E = ENOM * I$$

With the value for E determined, the rest of the model proceeds the same way as the Basic model. The accuracy of the prediction lies mostly in the ability of the manager to correctly determine the values for the fifteen predictor variables as well as predicting the size of the project. The Detailed model of COCOMO involves carrying out the Intermediate model at every phase of the project.

Ethics Concerns

Developer Reservations

Computational platforms such as Hackystat are great at automating the tedious data collection that makes many development metrics work, however they are not universally welcomed. Hackystat has faced some objections, particularly from developers who's employers make use of the system. The CDSL acknowledge three major complaints:

1. Although Hackystat was designed to collect data unobtrusively, some developers actually single out that feature as a problem. They are uncomfortable with software on their machines collecting data with no notification to the developer of what is happening.
2. The fine-grained data collection can cause issues within teams as it becomes easy for team members to see detailed information on what the other team members are doing. Some developers claim Hackystat offers too much transparency on the individual development styles of group members.
3. Developers are uncomfortable with their managers having access to highly detailed information on their working styles. Promises from management not to misuse the data has not stopped many developers from complaining about having to give managers access to Hackystat's data.

As one could probably expect, highly detailed data collection is not popular with everyone, especially when the data is made available to so many others. The CDSL argue however that for Hackystat to give the best insights it can, it is necessary for it to collect fine-grained information and to directly compare members of a team. These complaints could apply to any platform which makes use of automatic data collection and so the onus is on employers to make developers aware of any collection that happens and to make sure everyone is comfortable with the level of data collection.

Bibliography

Overview of the Personal Software Process
http://www-public.imtbs-tsp.eu/~gibson/Teaching/CSC5524/CSC5524-PSP.pdf

Overview of Leap and Hackystat
https://raw.githubusercontent.com/csdl/techreports/master/techreports/2012/12-11/12-11.pdf

“Automated recognition of test-driven development with Zorro” (appears to be corrupted pdf but worked when downloaded directly)
http://citeseerx.ist.psu.edu/viewdoc/download?doi=10.1.1.69.4525&rep=rep1&type=pdf

“On the current measurement practices in agile software development”
https://www.researchgate.net/profile/Hazura_Zulzalil/publication/235009023_On_the_Current_Measurement_Practices_in_Agile_Software_Development/links/0deec52799054e8cfe000000/On-the-Current-Measurement-Practices-in-Agile-Software-Development.pdf

UMD Lecture slides on COCOMO
http://www.cs.umd.edu/~mvz/cmsc435-s09/pdf/slides16.pdf