Measuring Engineering - A Report

Neil Simon

15323212

Trinity College Dublin

Measuring Software Engineering

When measuring software development, it is important to choose what metric you use as development isn't the same as other professions in the way that you can't measure the number of correctly made products, that you could if you were a manufacturer. There isn't one metric to rule them all and generally many are used to find problems or if changes are required.

One of the ways of measuring productivity for developers is code simplicity as almost all development problems occur due to incorrectly implementing key programing principles or practices.
Lines of code is a good baseline metric to use and can be used as a baseline for the right project, as some projects could be code intensive, while others can be debugger intensive.
Commits are another way of measuring productivity as if people establish a predictable cadence then it can be used to assess work done.
Version Control System History(VC) is good to give a sense of past productivity and if a code base is old enough, people can use it look at VC history and identify

‘hotspots’ in the code where teams can break up the code and solve issues.

We can also use agile process metrics such as: Lead time, which is how long it takes from taking an idea and turning it into software, Cycle time, which is how long it takes to ship any changes you make into production, Team velocity, which is how much software a team creates in an iteration, and Open/close rates, which is how many bugs and reported an fixed within a time period. Though using these metrics to measure success is inappropriate, it tells you the general trend and if you should be worried if the group is going in an alarming direction.

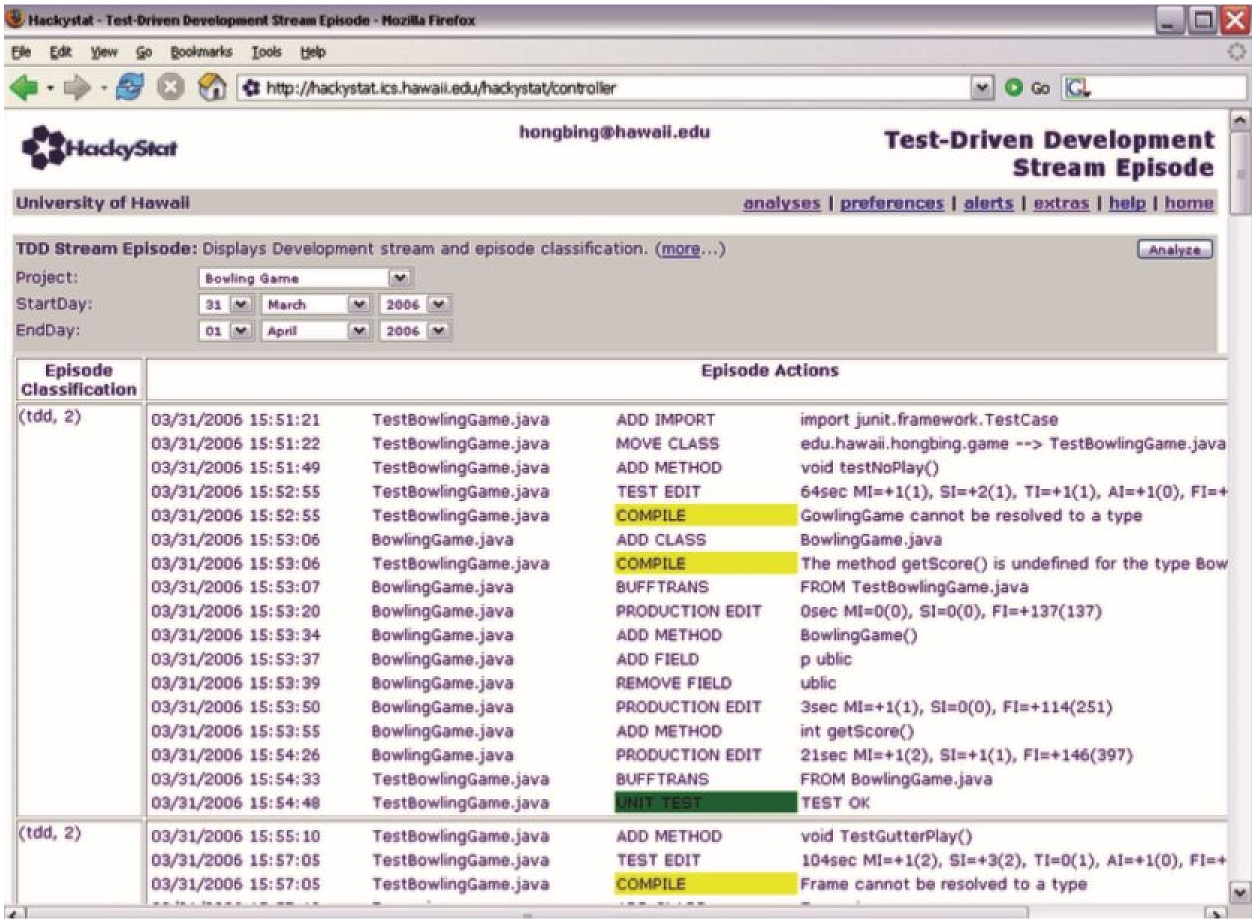
Overview of the computational platforms

The problem with measuring productivity in software engineering is outlined nicely in a quote by Philip M. Johnson “**the easier an analytic is to collect and the less controversial it is to use, the more limited its usefulness and generality**”.
An example of this is it's easy to collect data you need from a configuration management repository. The repository is also public so there is little outcry from

developers when you try to analyse this data. But any information you get from this is just a very small chunk of the work from the developers. But if you try to get more concise data, such as from the original version of the Personal Software Process(PSP), the costs are staggering and developers won't be as agreeable to such an intrusive study. Developers had to fill out 12 different forms in one version of PSP, which consisted of things such as a project plan, a time estimation template and a code checklist. This was a very expensive task as developers had to calculate over 500 distinct values. This method of measuring productivity is more powerful than getting public data from a repository as this also allows developers to investigate problems that they believe impacts their productivity such as interruptions. This method is not without its own host of problems such as due to its manual nature, incorrect process conclusions are more common.

The Hackystat project was introduced to reduce development overhead as the returns usually wasn't worth the cost and it abandoned supporting PSP analyses. It focuses on collecting data with very little overhead for developers and then achieve high-level software goal by studying and analysing such data. It was implemented using 4 features, the first of which was sensors attached to development tools which send data to a server which analysis it. The second one was data collection in a way that developers aren’t disturbed from their work as the tools collected the data and uploaded it. The third feature was collecting data in a steady but fast stream so that any data they get is relatively fresh and up to date. The final feature was tracking group based work, such a keeping a record of which developers with shared access to a resource, edited a certain file. Though the project was a success and has developed tech that can support unobstructive software development, some problems arose when developers didn’t want software that collects data without their permission and when people complained that Hackystat didn’t allow much privacy as it collected a lot of data and people weren’t happy with management having access to such fine data.

Many other projects have become recently available and have grown in use as they have 2 many strengths. The first of which is that data collection is almost automated which reduces overhead for developers and the second of which is that data collected focuses on the product rather than the developer and their way of programming.


Hackystat - Test-Driven Development Stream Episode - Mozilla Firefox
File Edit View Go Bookmarks Tools Help
http://hackystat.ics.hawaii.edu/hackystat/controller
Go
HackyStat
analyses | preferences | alerts | extras | help | home
Analyze

hongbing@hawaii.edu

Test-Driven Development Stream Episode

University of Hawaii

TDD Stream Episode: Displays Development stream and episode classification. (more...)

Project:	Bowling Game
StartDay:	31 March 2006
EndDay:	01 April 2006

Episode Classification	Episode Actions			
(tdd, 2)	03/31/2006 15:51:21	TestBowlingGame.java	ADD IMPORT	import junit.framework.TestCase
	03/31/2006 15:51:22	TestBowlingGame.java	MOVE CLASS	edu.hawaii.hongbing.game --> TestBowlingGame.java
	03/31/2006 15:51:49	TestBowlingGame.java	ADD METHOD	void testNoPlay()
	03/31/2006 15:52:55	TestBowlingGame.java	TEST EDIT	64sec MI=+1(1), SI=+2(1), TI=+1(1), AI=+1(0), FI=+
	03/31/2006 15:52:55	TestBowlingGame.java	COMPILE	GowlingGame cannot be resolved to a type
	03/31/2006 15:53:06	BowlingGame.java	ADD CLASS	BowlingGame.java
	03/31/2006 15:53:06	TestBowlingGame.java	COMPILE	The method getScore() is undefined for the type Bow
	03/31/2006 15:53:07	BowlingGame.java	BUFFTRANS	FROM TestBowlingGame.java
	03/31/2006 15:53:20	BowlingGame.java	PRODUCTION EDIT	0sec MI=0(0), SI=0(0), FI=+137(137)
	03/31/2006 15:53:34	BowlingGame.java	ADD METHOD	BowlingGame()
	03/31/2006 15:53:37	BowlingGame.java	ADD FIELD	p ublic
	03/31/2006 15:53:39	BowlingGame.java	REMOVE FIELD	ublic
	03/31/2006 15:53:50	BowlingGame.java	PRODUCTION EDIT	3sec MI=+1(1), SI=0(0), FI=+114(251)
	03/31/2006 15:53:55	BowlingGame.java	ADD METHOD	int getScore()
	03/31/2006 15:54:26	BowlingGame.java	PRODUCTION EDIT	21sec MI=+1(2), SI=+1(1), FI=+146(397)
	03/31/2006 15:54:33	TestBowlingGame.java	BUFFTRANS	FROM BowlingGame.java
	03/31/2006 15:54:48	TestBowlingGame.java	UNIT TEST	TEST OK
(tdd, 2)	03/31/2006 15:55:10	TestBowlingGame.java	ADD METHOD	void TestGutterPlay()
	03/31/2006 15:57:05	TestBowlingGame.java	TEST EDIT	104sec MI=+1(2), SI=+3(2), TI=0(1), AI=+1(0), FI=+
	03/31/2006 15:57:05	TestBowlingGame.java	COMPILE	Frame cannot be resolved to a type

Hackystat example.

All projects that analyse software development have three main trades offs: the less overhead for developers, the greater the automation needed to obtain good analysis, the barrier faced when adopting new technology or technique and finally the focus of the technology or technique that can be maintained while adhering to all characteristics.

Algorithmic Approaches

Of the many algorithmic approaches to measure software development, the one I found the most clear and useful is COCOMO. COCOMO is used in the industry to estimate the size, effort and duration based on the cost of a software. COCOMO has 3 model levels: Basic, which is used for quick estimates of software cost and schedule, but is limited due to undetailed data. Intermediate, which is used for stronger estimates of the cost and schedule because it takes the software project environment into account. And Detailed, which is used for the most accurate estimates, because it accounts for the impact of the environment to all of the phases of a project.
The COCOMO model has the following modes, Organic, which is intended for small projects. Embedded,which is used for large project that require innovation, a complex software interface and are faced with time constraints. Finally Semi-detached mode, which is used for anything in between.
COCOMO follows a basic formula for each individual mode, the effort formula is below:

Organic	***E = 2.4*(S^1.05)***
Semi-detached	***E = 3.0*(S^1.12)***

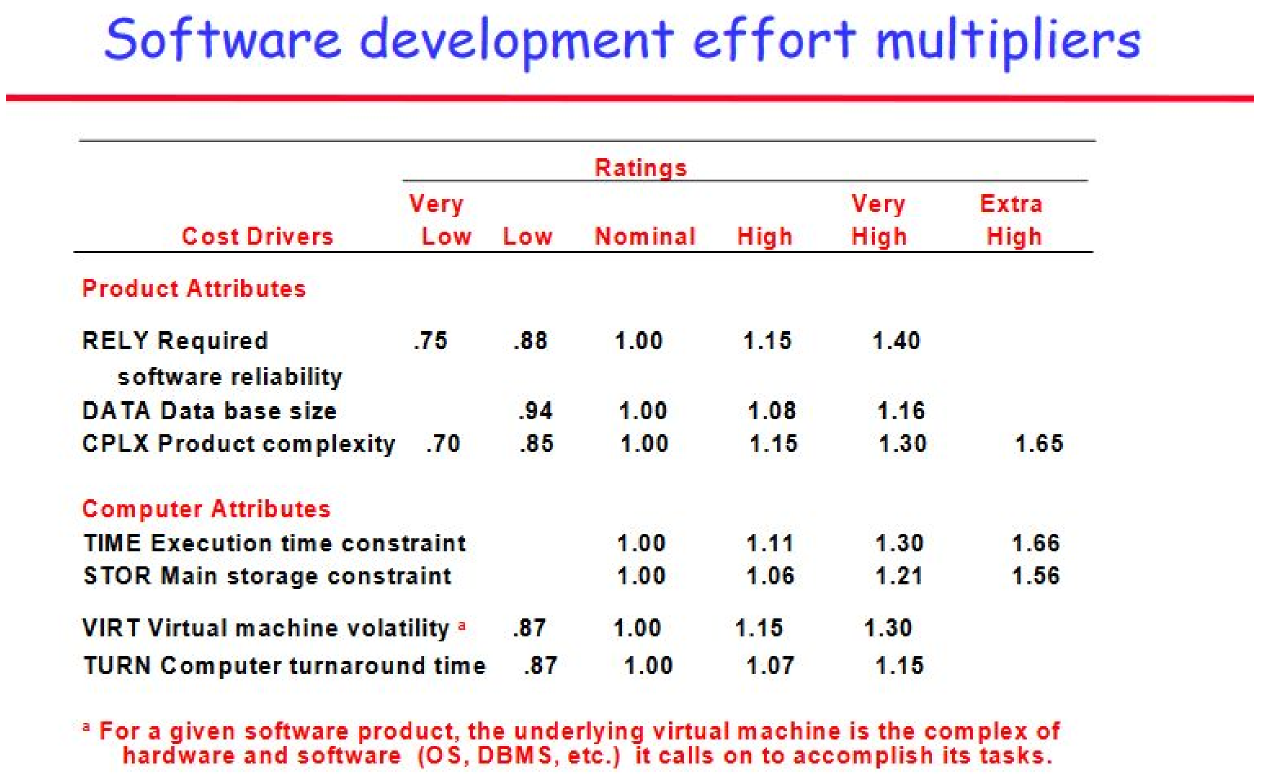
Embedded	$E = 3.6*(S^{1.20})$

where E is required effort in person months to develop the required software system as a function of S which is the source size. The project duration formula is also shown below:

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

Where TDEV is a function of effort expressed in calendar months and E in person months.

The intermediate COCOMO model is just an extended version of the basic version. This model uses 15 more cost drivers, where the manager assigns a value to each driver from a range of values such as ‘very low’ to ‘extra high’ where each of the values corresponds to a number that vary with cost drivers.



Software development effort multipliers

Cost Drivers	Ratings					
	Very Low	Low	Nominal	High	Very High	Extra High
Product Attributes						
RELY Required software reliability	.75	.88	1.00	1.15	1.40	
DATA Data base size		.94	1.00	1.08	1.16	
CPLX Product complexity	.70	.85	1.00	1.15	1.30	1.65
Computer Attributes						
TIME Execution time constraint			1.00	1.11	1.30	1.66
STOR Main storage constraint			1.00	1.06	1.21	1.56
VIRT Virtual machine volatility [a]		.87	1.00	1.15	1.30	
TURN Computer turnaround time		.87	1.00	1.07	1.15	

[a] For a given software product, the underlying virtual machine is the complex of hardware and software (OS, DBMS, etc.) it calls on to accomplish its tasks.

Example of effort multipliers

TDEV starts when a project enters its project design phase and it ends at the finishing of the software tests. The required effort covers the management and documentation efforts though it doesn’t count activities such as training. The requirement specification is assumed to not have changed substantially after the requirements phase when using COCOMO. A person-month can be changed to a person-day by multiplying by 19, and can be changed to person-hours by multiplying by 152.

Ethical Concerns

As seen above in ‘Overview of computational platforms’, ethical concerns play a huge role in measuring developer productivity. Incorrectly used methods to measure productivity can easily backfire and cause decreasing productivity and create a toxic work environment. For example if a developer feels pressured by any new changes when measuring productivity, such as frequently filling in forms about his work, he could be overwhelmed and not leave enough time to do his job correctly and his work could suffer. Another example is if a developers peer gets ahold of their statistics and he has been underperforming recently, strife between workers can easily take place as people might resent him for it. Not only this developers could easily try to ‘game’ the system if they find out on what metrics they are judged on, such as if its lines of code, they could just just write horribly inefficient code just to take up more lines while appearing to have high productivity.

Too much overhead is liable to cause stress and negatively impact work and all data handled should be completely confidentially and only really affect people if their work deteriorates into something alarming. And if developers get extra restrictions added to their work, resentment would grow and motivations would run low.

Many such problems exist and it's important that the management takes all this into account before deciding on how to measure their team's development productivity.