Alex Groce (agroce@gmail.com), Northern Arizona University

Alan Schwarz's *The Numbers Game: Baseball's Lifelong Fascination with Statistics* is this column's topic, but I'm going to postpone talking about it until I've digressed into a little, seemingly unrelated, software engineering history/biography.

Harlan D. Mills (1919-1996) is the namesake of the Harlan D. Mills Award given by the IEEE Computer Society to recognize "researchers and practitioners who have demonstrated long-standing, sustained, and meaningful contributions to the theory and practice of the information sciences, focusing on contributions to the practice of software engineering through the application of sound theory." First given in 1999, recipients include such luminaries as David Parnas (the first winner), Elaine Weyuker, Bertrand Meyer, the Cousots, Gerard Holzmann, Pamela Zave, Gail Murphy, Mark Harman, Matt Dwyer, and David Harel. Mills' idea of the Chief Programmer Team influenced Brooks' proposals for how to avoid software disaster in *The Mythical Man-Month*; his Cleanroom conception informs every approach that is focused on defect-prevention (from static analysis to aggressive unit testing to mutation testing) rather than removing defects found in the field; Mills was also a major force in introducing both statistics (viewing tests as statistical experiments, which arguably also is a foundational insight for all of random testing and fuzzing) and formal specification (thus a contributor to the lines of thought that have given the field Alloy, TLA+, Dafny, and other treasures). In short, he was a heckuva big shot in our field, though (other than having one of the top awards in the field named after him) I think he's one of the least talked-about major visionaries (one name for the Mills Award, confusingly implied as being started in 1996, and perhaps by ICSE/the ACM, in Wikipedia, is the "Harlan Mills Practical Visionary Prize") we have.

I know the rough outlines of the above details because I am a software engineering history buff. I know that Mills was a command pilot for the US Army Air Corps in WWII, and rewrote the B-24 pilot training program because he thought it was garbage, and that he worked at the Institute for Advanced Study alongside Einstein, Godel, and von Neumann, because Harlan Mills and his brother Eldon are the most prominent characters in Chapter 4 of the book this column is about. Because, you see, besides a passion for applied mathematics and operational research and software engineering, Harlan Mills had a passion for baseball. Harlan and Eldon Mills, in 1968, founded a company called Computer Research in Sports. Their goal was to go beyond the (much-beloved: one could argue that batting average, which we'll talk about below, drove the country crazy a few times in the years since baseball became America's pastime) statistics baseball fans already knew to statistics that truly measured how a player contributed to winning a game. They were among many high-powered intellects (including multiple Nobel Prize winners and the popular Harvard evolutionary-biology maverick, public intellectual, and superb writer Stephen Jay Gould) pioneering an effort to truly understand baseball that is documented in Schwarz's engaging, in-depth, look at baseball's love of statistics and how those statistics have developed from the earliest, alien-looking, box-scores.

The connection between baseball statistics and software engineering, however, goes beyond the curiosity that Mills was a pioneer in both software engineering and baseball stats-savvy.

The obvious connection is that the stats revolution in baseball was in substantial part tied to the computer revolution; nothing like https://www.retrosheet.org/ could plausibly exist in a non-computerized world, and certainly looking for trends or even just finding individual facts in such a database (play-by-play accounts of over 100,000 pro baseball games) would be impossible even if a warehouse of filing cabinets had the relevant data. *The Numbers Game* shows how computers were essential to the growth of the modern baseball stats world, and takes us back to when baseball work was done by "stealing" off-time on hulking company and government mainframes. The deeper connection, however, that this book may help the software engineering reader ponder, is that baseball was, for a very long time, tied to extremely traditional, and beloved statistics, but even stodgy, change-averse professional baseball front and back offices eventually realized the limitations of these statistics and moved on to more sophisticated, nuanced, statistics that truly reflected *how to win the game*.

To develop the theme of the similarities (data is easy to gather, hard to understand; a lot of people who like futzing with spreadsheets and coding up formulas are involved; there's a lot of money and a lot of (to stay polite, this is a family column) "BS" involved) and differences (baseball is an extraordinarily structured thing, and the binary outcome of almost every event (did someone make an out? did someone score? did the team win?) is obvious, while in software engineering figuring out if a computer program works and was well-made turns out to be extremely annoyingly hard unless you wait twenty years and see if anyone is using it and how much money it made) between baseball and software engineering would require more space than I can reasonably use in a Passages column. I think I'll write an Onward! Essay on the topic sometime, and exhaust the lengthier patience available there, or perhaps even a whole book on the topic. But perhaps we can quickly see what I mean by the move from traditional stats to ones that *mean more* by looking at a simple example from software testing (we could also compare LOC produced to modern software productivity measures that are less simpleminded, like Change Failure Rate).

One way to determine if a program is well-tested is to take the test suite for the program and look at the *statement coverage*: the percent of all statements in the whole program that are executed at least once by the test suite. Obviously if a test suite fails to execute most of the code in the program it is ostensibly testing, it is likely a pretty bad test suite. Similarly, if a test suite runs 90% of the code in the program, it's presumably at least doing *something* non-trivial right.

In baseball, for a very very long time, the king of statistics was *batting average*. What is batting average? Even if you know little about baseball, even if you're a European or from Mars (the best kid's book on baseball is Walter R. Brooks' (Brooks is also the guy who invented Mr. Ed) *Freddy and the Baseball Team from Mars*, by the way), you may well know that in baseball, a batter is thrown balls by the pitcher and tries to hit those balls, and then (when the ball is hit) run to a series of three bases, before returning to the place from which the ball is initially hit (home base). Batting average looks at how often a player hits the ball to advance to first, second, third, or home base. Batting average is not nonsense! The most sophisticated baseball

statisticians tend to agree players celebrated for their batting averages, such as Ty Cobb, Rogers Hornsby, and Ted Williams, are superb, top-of-the-line players. So it, like code coverage, is a meaningful number.

However, there are some problems with statement coverage, and code coverage in general. For one thing, maybe you only run a really important line of code once in the whole test suite, and that in a rather boring and uninteresting situation where the things the line of code is supposed to do are not seriously stressed (it's a complex multiplication, but one factor is zero, for example). Or, worse yet, you may run lots of code lots of times, but not check anything much about whether the program *did what it was supposed to do* other than "not crash." There are lots of ways a program can go wrong other than outright crashing. *Mutation score* asks "what do we really care about in testing? We want to find bugs. Statement coverage is an awful roundabout way to measure finding bugs, since bugs aren't even mentioned here. Let's see how a test suite does at *finding bugs*." In mutation testing, a large number of essentially random *bugs* are actually injected into a program, and the score is the percent of the *fake bugs* the test suite was able to find. It's not perfect, but if you think about it, it completely removes the two obvious problems with statement coverage I mentioned.

Similarly, the way batting average is defined, it doesn't look at every time a player comes up to home plate to try to hit the ball. The numerator is "times the player hit the ball and got to a base" but the denominator is "times the player came up to bat, minus times the pitcher did such a bad job the player was allowed to take a base without hitting the ball, and the batter took proper advantage of this by not swinging at bad pitches (or times the pitcher feared the player so much this was *intentionally* done to keep the bat off the ball), times the pitcher hit the player with the ball, times the catcher did something weird, times the player intentionally hit the ball in a way that tends not to let you reach base but has another game-winning-relevant purpose, etc. etc. etc.). *On-base-percentage*, a revolution in the 1980s in understanding how to value hitters, counted most everything, and looked (essentially) at "of times this player came up to bat, how often did this player *end up at least standing on a base*." Like mutation score, it focused on the more essential (not getting out before reaching a base, finding bugs) rather than the easily and obviously measured (hitting the ball, running the code). You'll understand this insight and perhaps think more intelligently about the software engineering statistics you gather and use, after reading this enjoyable book.

A final note; a portion of the territory this not-so-well-known book covers is covered in more depth and with a lot more flavor in Michael Lewis' very famous book *Moneyball*. *The Numbers Game* is less popular; there will be no movie starring Brad Pitt as Harlan Mills. Baseball fans reading this column may understand the difference, and why the Schwarz book is a more important read for software engineers, from the following simple statistic: the phrase "Strat-O-Matic" appears almost 40 times, scattered across many chapters, in *The Numbers Game*. It does not appear once in Lewis' otherwise extremely enjoyable, but slightly superficial, book.