Inferring Skill from Tests of Programming Performance: Combining Time and Quality

Gunnar R. Bergersen
Department of Informatics,
University of Oslo and Simula
Research Laboratory, Norway
gunnab@ifi.uio.no

Jo E. Hannay
Simula Research Laboratory,
P.O. Box 134, NO-1325 Lysaker,
Norway
johannay@simula.no

Dag I. K. Sjøberg
Department of Informatics,
University of Oslo, P.O. Box 1080,
NO-0316 Oslo, Norway
dagsj@ifi.uio.no

Tore Dybå
Department of Informatics,
University of Oslo and
SINTEF, Norway
tore.dyba@sintef.no

Amela Karahasanović
SINTEF and Department of Informatics,
University of Oslo
P.O. Box 124 Blindern, NO-0314 Oslo, Norway
amela@sintef.no

***Abstract*—The skills of software developers are crucial to the success of software projects. Also, controlling for individual differences is important when studying the general effect of a method or a tool. However, the way skill is determined in industry and research settings is often *ad hoc* or based on unvalidated methods. According to established test theory, validated tests of skill should infer skill levels from well-defined performance measures on multiple small, representative tasks. We show how time and quality can be meaningfully combined to a well-defined measure of small-task performance, and hence a measure of programming skill. Our results show significant and positive correlations between our proposed measures of skill and variables such as seniority or self-evaluated expertise. These methods for combining time and quality are a promising first step to measuring programming skill in industry and research settings.**

Programming; skill; performance; time; quality; productivity

I. INTRODUCTION

The skills of individual software developers may have a dramatic impact on the success of software projects. The large differences in programming performance, reported in the late 1960s, indicated performance differences of orders of magnitude and more. Although more recent analysis [32] and studies [13] indicate somewhat more conservative estimates, companies that succeed in hiring the best people will nevertheless achieve great economic and competitive benefits [17,34,37].

Individual differences in skill also affect the outcome of empirical studies in software engineering. When evaluating alternative processes, methods, tools or other technologies, skill levels may temper with the relative effect of using a specific alternative. For example, in an experiment on the effect of a centralized versus delegated control style, the purportedly most skilled developers performed better on the delegated control style than on the centralized one, while the less skilled developers performed better on the centralized one than on the delegated one [5]. In another experiment, skill levels had a moderating effect on the benefits of pair programming [4].

However, determining the skill level of software developers is far from trivial. In the work life, there are common-sense guidelines from experienced practitioners on how to distinguish the good from the bad [37]. But there seems to be consensus that this crucial human resource management task remains hard. Often, job recruitment personnel use tests that purport to measure a variety of traits such as general cognitive abilities (intelligence), work and life values, interests, as well as personality [20] to predict job performance [11]. Research has, however, established that work sample tests in combination with General Mental Ability (GMA) are among the best predictors of job performance [34]. GMA is a general aspect of intelligence and is best suited for predicting performance on entry-level jobs or job-training situations. This stands in contrast to work sample tests that are task specific and are integrated in the concept of job skill [15]. Although the predictive validity of standardized work samples exceed that of GMA alone [11], these predictors seem to yield the best results when they are combined [34].

In the context of empirical studies in software engineering, the notion of programming skill is generally not well founded. This has led to studies that failed in adequately correcting for bias in quasi-experimental studies [23]. Often the more general concept of programming expertise is used, with little validation. For example, in a recent study [20], we conceptualized programming expertise as the level of seniority (junior, intermediate, senior) of the individual programmer as set by their superior manager. While bearing some relevance to the consultancy market, this conceptualization is not sufficient to capture the skill of individual programmers. The concepts of expertise and skill are operationalized in vicarious ways in also other domains; see [22] for a survey of operationalizations in IT management.

Whether the purpose of determining programming skill is to recruit the best developers or to assess the usefulness of a software engineering technology relative to levels of skill, the most viable approach seems to be that candidates solve a collection of small programming tasks. We used a single task for this purpose in [5], but it was uncertain to what degree bias may be present as a result of only using one task. Generally, it seems non-trivial to identify the appropriate collection of tasks from which one can infer a reasonably accurate level of programming skill in an acceptable amount of time. We are working on this challenge at present, but this is not the focus of this paper.

The focus of this paper is as follows. Given a small set of programming tasks, how does one infer the candidates' programming skill from both the quality of the task solutions and the time (effort) spent performing the tasks? It is well recognized that the combination of quality and time task is essential to define skill [15,16], but how to combine them in practice is challenging. For example, how does one rank programmers who deliver high quality slowly relative to those who deliver lesser quality quicker? This paper addresses such challenges and proposes a method for combining quality and time for a task solution into a single ordinal score of performance (i.e., low, medium, high). Multiple performance scores are then aggregated to form a ordinal approximation of programming skill. The method is demonstrated by using data from two existing experiments.

Section 2 gives the theoretical and analytical background for skill as a subdomain of expertise. Section 3 discusses how quality and time are dealt with at present and describes how to combine them when measuring performance. Section 4 reanalyzes existing data sets according to the arguments given in the previous sections. Sections 5 discuss the results and Section 6 concludes.

II. Background

A. *Expertise*

Expertise is one of the classic concepts of social and behavioral science. Expertise is usually related to specific tasks within a given domain and does not in general transfer across domains or tasks [15]. Expertise has several aspects; we present five of these in Fig. 1 (a). The aspects are all related. For example, in the usual descriptions of skill acquisition [1,14,16], which is a subdomain of expertise, a person starts by acquiring declarative knowledge which for experts is *qualitatively different in representation and organization* compared to novices [15,38]. Further, through practice, declarative knowledge is transformed into procedural skill, which at first is slow and error prone [16]. However, though *extended experience*, performance improves and experts should converge on their understanding of the domain for which they are an expert as well [36] (i.e., *consensual agreement*). Experts should also regard themselves as being experts, for example, through the use of *self-assessments*. Overall, the desired effect of expertise is superior performance on the tasks on which one is an expert. In our context, this is performance on real-world programming tasks, i.e., job tasks. It is, however, unreliable and inefficient to predict future job performance by observing actual job performance [11]. This is why it is desirable to design quick tests based on how well an individual *reliably performs on representative tasks* [15].

B. *Skill*

It is in the aspect of performance on small representative tasks that we generally understand skill. Note that inferring skill from a reliable level of performance on representative tasks is not the same as *defining* it in terms of performance on the job. Representative tasks in our context denote smaller tasks which represent real-world tasks, and for which there are well-defined measures of performance [15]. The inference from performance on small representative tasks to performance on the job requires an understanding of key mechanisms at play shared between tasks in the two settings. This is theory-driven generalization [33], based on the economy of artificiality [21]. In the absence of, or as a complement to strong theory, it is useful to seek confirmation in how well skill measures coincide with other aspects of expertise. This is relevant for skill in programming.

Programming skill was investigated by Anderson et al. [1,2] from a psychological perspective. They reported that coding time and as well as the number of programming errors decreased as skill improved. Further, programming in LISP required the learning of approximately 500 if-then rules. The acquisition of these rules followed a power-law learning curve; the improvement in performance was largest at first and then decelerated until an asymptote was reached. Thus, the relationship between amount of practice (extended experience) and performance was non-linear. However, if

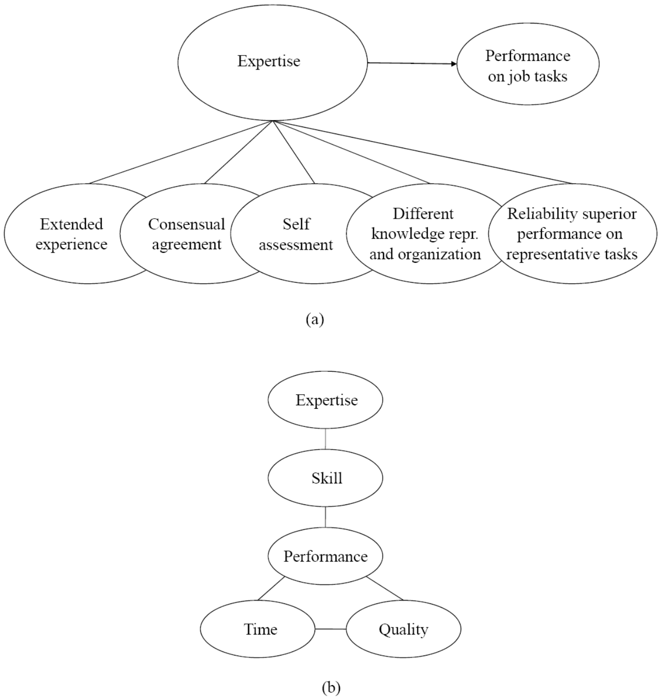

Figure 1 Expertise (a) and skill as one aspect of expertise (b). The desired effect of expertise is superior job performance.

amount of practice and performance were log-transformed, an approximate linear trend was observed. This phenomenon is widely observed and is therefore often referred to as the log-log law of practice [31].

Fits and Posner [16] has extensively studied skill acquisition. Within many different domains of expertise, they found that with increased skill, the amount of errors in performance decreases and the speed of which a task is executed improves. Regarding measures of skill, they state: "[t]he measure should take into account the length of time taken to perform a skill as well as the accuracy with which it is performed" [16, p. 85]. Therefore, the time and the more general term for accuracy—namely quality—is intimately linked to skill. At the same time, the term "performance" is linked to all three concepts. Because skill, by its very definition, affects performance, we can hierarchically structure the five concepts expertise, skill, performance, time and quality as in Fig. 1 (b). From the top, expertise, which should affect job performance, is a generalization of skill. Further, skill is inferred from multiple performances; as already stated, *reliably* superior performance on representative tasks is a requirement. At the bottom, time and quality in combination dictate whether we regard performance overall as, for example, high or low.

C. *Measures of programming performamce*

It is common in empirical software engineering to deal with quality and time separately when analyzing results; i.e., one studies performance in terms of quality and then in terms of time, often under the assumption that the solution meets some kind of criterion for correctness (see [4,5,7] for examples). We acknowledge that for many studies, this is acceptable. However, when the purpose is to characterize individual differences, problems may occur.

Time is a ratio variable with an inverse relation to performance (i.e., little time implies good performance). Quality, on the other hand, may consist of plethora of variables where each one may have complex relations to each other and where all often cannot be optimized simultaneously (see, e.g., [30]). Further, depending on how quality is operationalized, these variables may have different scale properties (i.e., nominal, ordinal, interval or ratio) or other different propensities. Therefore, when aiming to characterize individual differences, one is (a) forced to disregard quality and report differences in time spent or (b) only analyze time for observations that surpass some specific level of quality (often correctness), thereby adhering to the basic principle delineated by Thorndike and others in the 1920s: "the more quickly a person produces the *correct* response, the greater is his [ability]" [12, p. 440, emphasis added]. It is also possible to (c) devise acceptance tests that forces everyone to work until an acceptable solution is achieve. Generally, we regard this as, perhaps, the most viable approach today, because individual variability is expressed through time spent in total. However, by using (b) or (c), large portions of the dataset may be excluded from analysis, in particular when the proportion of correct solutions is low.

At the most fundamental level of the time/quality tradeoff problem, it is not given how to place programmers who deliver high quality slowly relative to those who deliver lesser quality quicker. In the datasets that are available to us, correctness and time are often negatively correlated. This indicates that the longer it takes to submit a task, the lower is the likelihood of the solution being correct. Although this may seem contrary to what may be expected (i.e., higher quality requires more time, lower quality requires less time), there are two important distinctions to be made: First, there is a difference between quality in general and correctness specifically. Second, there is also a difference between *within-subject* and *between-subject* interpretations; when a correct solution can be identified, a highly skilled individual can arrive at this solution in less time and with higher quality than a less capable individual (an *inter*-individual interpretation). But given more time, a single individual can generally improve an existing solution (*intra*-individual interpretation).

Another challenge is to what degree an individual's performance in a study is *reliable* at a specific level or *incidentally* high or low from time to time. One way to address such concerns it to use multiple indicators of performance [6,18]. Based on the same principles for combining time and quality as performance which is delineated in this article, we have already advanced the measurement of skill using multiple indicators of performance [9]. But, a detailed discussion on the principles involved is needed and we will illustrate the approach using larger datasets here.

D. *Using the Guttman structure for time and quality*

The two-by-two matrix in Fig. 2 has two possible values for quality (low, high) and two possible values for time (slow, fast). It should be easy to agree that in this simplified example, "high performance" is represented by the upper right quadrant (fast and high quality) whereas "low performance" is represented by the lower left quadrant (slow and low quality). Further, it should also be possible to agree that the two remaining quadrants lie somewhere between these two extremes, say, "medium performance". However, which one of the two alternatives one would rate as the better one (or whether they should be deemed equal), is a value judgment: In some instances, "fast and low quality" may be

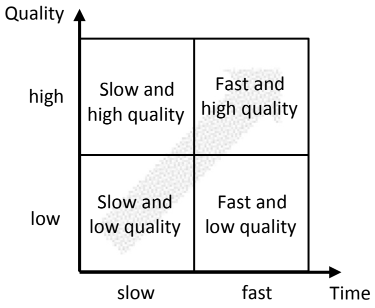

Figure 2. Examples of scoring based on value judgments favoring time and quality for "medium performances".

deemed superior performance compared to “slow and high quality”. To address how performance should relate to different values for time and quality, we propose to use principles as delineated by Louis Guttman.

The Guttman scale was originally developed to determine whether a set of attitude statements is unidimensional [19]. In Guttman’s sense, a perfect scale exists if a respondent who agrees with a certain statement also agrees with milder statements of the same attitude. The Guttman *structured scoring rules* we propose, utilize the same underlying principle as the Guttman *scale*, albeit at a lower level of abstraction (a scale is an aggregation of indicators, whereas the structure we employ refers to the indicators themselves). The approach utilizes general principles as delineated by others [3], but which have only somewhat informally been addressed by us so far [8]; overall the benefits of using the Guttman structure is fundamental for some modern measurement models which we address in Section 5 C as further work.

With a Guttman structure it is possible to rank combinations of quality and time relatively to each other as well as being explicit about how different tradeoffs in time and quality are scored. Performance on a programming task is thus determined by a series of thresholds that are rank ordered. Combined, these thresholds constitute a set of ordered response categories (i.e., an ordinal variable). Surpassing a given threshold implies all thresholds below has been passed as well. This implies that for a score of, say 3 (of 5 possible), the thresholds for obtaining scores of 0, 1, and 2 must have been passed, while the threshold for obtaining score 4 has failed.

One may, further, express performance in terms of quality and time by adding and adjusting score categories. For example, a task that differentiates more on quality aspects may be scored on multiple quality categories and a task that also differentiates more on time aspects may have more time categories. Conversely, one may deliberately emphasize quality over time (or vice versa) by adjusting score categories accordingly.

III. Research Methods

In this section we describe how we combined time and quality using multiple indicators; each indicator is the performance as time and quality combined on a single task. We show how we operationalized and reanalyzed two different data sets, using different principles for score operationalizations.

A. Data set 1

The first data set we reanalyzed is from a controlled quasi-experiment [5]. In a one-day experiment, 99 consultants from eight different software consultancy companies and 59 undergraduate and graduate students were paid to participate. The treatment in the experiment was the control style of the code (centralized versus delegated). Five programming tasks were presented in succession to the subjects during the experiment, which lasted one day. The first task i_1 (pretest) was identical for both treatment conditions. The four next tasks, i_2–i_5 contained the treatment. We analyze only the first four tasks here due to challenges present in using the last task for our purpose (see Section 5 B).

For a Guttman-structured scoring rule, we used the following approach for each task i_1–i_4: Let *Q1*, *T1*, *T2* and *T3* be dichotomous variables, scored as requirement not met = 0, requirement met = 1. Let *Q1* be functional correctness (as reported by the original authors), scored as incorrect = 0 or correct = 1. Let *T3* be time < 3rd quartile, *T2* be time < median, *T1* be time < 1st quartile. A Guttman structure for an ordinal performance score for a single task that combines quality and time is then defined by the Cartesian product *Q1*×*T3*×*T2*×*T1* as follows (x denotes either of 0, 1):

$(0,x,x,x) = 0$ (i.e., incorrect, time is irrelevant)
$(1,0,x,x) = 1$ (i.e., correct and very slow)
$(1,1,0,x) = 2$ (i.e., correct and slow)
$(1,1,1,0) = 3$ (i.e., correct and fast)
$(1,1,1,1) = 4$ (i.e., correct and very fast)

The matrix representation of this scoring rule is illustrated in Table 1 (a). By using this structure, a solution must be correct before time is taken into consideration. Increasing scores for time are, further, only passed in order (*T3* before *T2* and *T2* before *T1*). Hence, for a single task i, time and quality combined define an ordered response category of performance. And the precedence of quality in this type of scoring rule reflects the view that, for this study, we do not consider a non-working solution to reflect high or medium performance, even when it is developed quickly.

We also constructed two alternative Guttman-based scoring rules to *Q1*×*T3*×*T2*×*T1* that differentiate less on time, but that is still based on the same *Q1* as above: *Q1*×*T2*×*T1* uses three categories for time based on the 33rd (*T2*) or 67th (*T1*) percentile; *Q1*×*T1* only uses two categories for time, which is above versus below the median. The range of the overall performance score in all instances is equal to the number of dichotomous score variables plus one, e.g., *Q1*×*T3*×*T2*×*T1* has one variable for quality and three for time, implying a total of five well ordered performance score categories with a range of 0–4.

The procedure we have described so far for just described was repeated for all four tasks. Because of different time distribution for each task, the quartiles and medians for time is calculated on a task-by-task basis. The resulting score vector consist of four Guttman-structured score variables and the sum of these, the sum score, is the ordinal skill scale.

TABLE I. Score According to Time and Quality Thresholds

Score	*T3=0*	*T3=1*	*T2=1*	*T1=1*
Q1=1	1	2	3	4
Q1=0	0	0	0	0

(a) Dataset 1

Score	*T2=0*	*T2=1*	*T1=1*
Q2=1	2	3	4
Q1=1	1	1	1
Q1=0	0	0	0

(b) Dataset 2

For comparison, we also devised two alternative scoring rules that combined quality and time for tasks by addition (*additive scoring rules*). On a task–by-task basis, we standardized quality and time (mean 0 and standard deviation 1) before adding the standardized variables as a composite score of performance. This was a manifestation of treating "slow and high quality" as roughly equal to "fast and low quality" (in Fig. 2), but where the continuous properties of time is not forced into discrete categories. We name these scoring rules *Q+T* and *Q+lnT*. Here, time was negated in both instances and for the latter variable, time was log transformed as well before negation. Finally, we constructed scoring rules on the four quality variables alone (*Q*) and the four time variables alone (*T*). (See Section 5 B for other alternatives that we choose not to report).

It should be noted that the relation between the performance score vector, i.e., the score of on each task, and skill score, i.e., sum of performance scores, is a many-to-one (surjective) function. For example, when using *Q1×T3×T2×T1*, an individual with correct but very slow solutions for all four tasks receives the sum score of 4. One correct solution with very fast time and the other three tasks incorrect, also receive the same sum score. But it is problematic to talk about the latter instance as "reliably (superior) performance" because such a response structure exhibits superior performance on only a single task.

B. Data set 2

The second data set stems from three different quasi-experiments which all used the same programming tasks. During of one day, the subjects were required to perform three different change tasks in a 3600 LOC library application system that contains 26 Java classes. Two of the studies used students as subjects; one study used professionals. The study in [25] investigated the effects of different comprehension strategies using 38 subjects; the study in [24] compared feedback collection and think-aloud methods for 34 subjects; and the study in [27] studied the effects of expertise and strategies on program comprehension for 19 subjects. Additionally, the same pretest task as in Dataset 1 (i_1) was used. However, one of the studies had missing data for the last change task, thereby reducing the number of available tasks for analysis from four to three. Human graders scored the quality of each task on a five-point scale using the following scheme:

0: nothing done on the task (no code changes)

1: failure, does not compile or no discernible functional progress toward solution

2: functional anomalies, one or more subtasks are achieved

3: functionally correct, major visual anomalies

4: functionally correct, only minor cosmetic anomalies

5: functionally correct, visually correct, i.e. "perfect solution"'

We defined a Guttman structure *Q1*×*Q2*×*Q3* for quality dimension as follows: We decided that the original categories 0 and 1 should be collapsed into a single category, because neither might be preferred over the other. Thus, variable *Q1* was defined as "one or more subtasks achieved" (category 2 above). Next, the *Q2* variable was "functionally correct, but with major visual anomalies allowed" (category 3), and *Q3* was functionally correct with only minor visual anomalies allowed (categories 4 and 5). For the time dimension, we used *T1×T2* to partition the time for those individuals who passed *Q3* into three groups. The matrix representation of this scoring rule, denoted *Q1×Q2×Q3×T1×T2*, is provided in Table 1 (b).

We also devised scoring rules using one and two dichotomous quality variables as well: *Q1×Q2×T2×T1* does not separate between major and minor visual anomalies that are otherwise correct. Further, *Q1×T2×T1* only separate between functionally correct solutions with major (or better) visual anomalies from those that are not functionally correct. Finally, we devised scoring rules for *Q+T*, *Q+lnT*, *Q* and *T* using the same procedure as in Dataset 1, but using three tasks instead of four.

C. Analysis method and handling of missing data

The analysis method for the two data sets, each using six different score operationalizations, included the same four basic steps. All time variables were negated (for *T*) or log transformed and then negated (for *Q+lnT*, *QlnT*) in order to increase interpretability so that high values indicate high performance:

1) Use exploratory factor analysis.

We extracted the main signal in the data for each scoring rule by Principal Component Analysis (PCA) using the analysis software PASW^{TM} 18.0. We used listwise deletion of missing variables, regression for calculating the factor score, and an unrotated (orthogonal) factor solution to maximize interpretability of each factor. All factors with eigenvalues above 1 are reported.

2) Inspect external and internal results

Operationalizations of the scoring rules were compared with several experience variables. We report non-parametric correlations (Spearman's ρ, "rho") unless otherwise noted. We assumed that a valid scoring rule should correlate moderately and positively with relevant background variables such as developer category or and length of experience. Because such variables are not influenced by our investigated score operationalizations, we refer to this analysis as *external results*.

Conversely, all the reported *internal results* are influenced by how each scoring rule was constructed. For internal results, we used the proportion of explained variance for the first Principal Component (PC), which is analogue to the sum score, as the signal-to-noise ratio for each scoring rule. Cronbach's α was used as a reliability coefficient which expresses the internal consistency of the scores. To ascertain the applicability of each score operationalization, we used Root Mean Square Error of Approximation (RMSEA) as reported in the analysis software Amos^{TM} 18.0. This is a parsimony-adjusted index in the sense that it favors models with fewer parameters in a confirmatory model. We used a tau-equivalent reflective measurement model with multiple indicators [29]. This implies that all tasks receive the same weight when calculating the sum score. All scoring rules are further regarded as ordinal scale approximations of skill.

3) Handling of missing data

Each dataset contain some missing data. For solutions that were not submitted, we applied the same basic principle as the authors of Dataset 2: “non-working solutions or no improvements in code” were equated with “nothing submitted at all” and scored as incorrect. Additionally Dataset 2 had some missing values for time. We did the same as the owners of this dataset and removed these observations altogether. Missing data pose a threat to validity if data are not missing at random. We therefore analyzed our results using data imputation as well. However, because the same substantive results apply with or without data imputation, we report results without imputation.

IV. Results

In this section, we first report the correlations between the investigated scoring rules and the subjects’ background experience variables. Next, we report several indices that must be inspected together, such as explained variance, internal consistency and how well the scoring rules fit confirmatory factor analysis. Finally, we highlight some selected details about the scoring rules investigated.

A. External correlations

Table 2 shows correlations between experience variables and the proposed score operationalizations for both datasets. Developer category was only available for Dataset 1. In the initial classification scheme (i.e., undergraduate = 1, intermediate = 2, junior = 3, intermediate = 4, expert = 5) insignificant and low correlations were present between developer category and different scoring alternatives (rho = 0.05–0.12). However, because many graduate students performed at levels comparable to seniors, it doubtful that and this operationalization of expertise is a monotonically increasing function of performance. When removing the two student categories (1 and 2) from the analysis, the company-assigned developer category complied to some extent with individual results; all correlations were significant and positive around 0.3.

The other experience variables were self assessed. Years of programming experience (lnProfExp) is also an aspect of extended experience. In general, the correlations for this variable were low and insignificant for all scoring alternatives, but were slightly improved having been log transformed (a justifiable transformation given the log-log law of practice discussed earlier). Java programming expertise (SEJavaExp) is a single self-assessed variable ranging from novice = 1 to expert = 5. This variable was significantly and positively correlated around 0.3 with all of the scoring alternatives for Dataset 1. However, for Dataset 2, the correlations were lower and less systematic, but caution should be shown when interpreting this result due to low *n*. Nevertheless, parametric correlations for this variable were lower than the non-parametric correlations for both datasets. Overall, self-assessed Java programming expertise seems to have a non-linear but monotonically increasing relation to the proposed score operationalizations. Self-estimated Lines Of Code (lnLOCJava) in Java has positive skew and kurtosis, but approximates a normal distribution after log transformation. All scoring operationalizations were significantly and positively correlated with LOC (around rho = 0.3) with two exceptions: *Q* in Dataset 1 and *T* Dataset 2.

Nevertheless, even though the proposed scoring alternatives are positively correlated with relevant experience variables, only one of several required but not sufficient hurdles may have been passed. The reason why correlations alone can mainly provide negative (and not positive) evidence for validity, is that it is uncertain what the “true” correlation should be between a test score and a background variable (see, e.g., [10]).

TABLE II. Correlations, Factors, Explained variance, Reliablity and Confirmatory fit of scoring alternatives

	***Non-parametric Correlations* rho (*n*)**				***Fit indices***			
Dataset 1	***Developer Category***	***lnProfExp***	***SEJavaExp***	***lnLOCJava***	***#f***	***%E***	***α***	***RMSEA [lo90, hi90]***
Q	(99) 0.26**	(157) 0.08	(158) 0.25**	(158) 0.12	2	33.3	0.45	0.145 [0.086, 0.211]
T	(93) 0.33**	(152) 0.16*	(152) 0.31**	(152) 0.38**	1	47.9	0.54	0.189 [0.131, 0.253]
Q+T	(93) 0.34**	(152) 0.14	(152) 0.30**	(152) 0.29**	1	48.7	0.65	0.096 [0.027, 0.166]
Q+lnT	(93) 0.35**	(152) 0.15	(152) 0.30**	(152) 0.29**	1	52.6	0.70	0.093 [0.021, 0.163]
Q1×T1	(99) 0.31**	(157) 0.07	(158) 0.33**	(158) 0.29**	1	45.0	0.58	0.094 [0.023, 0.164]
Q1×T2×T1	(99) 0.33**	(157) 0.11	(158) 0.31**	(158) 0.29**	1	47.7	0.63	0.076 [0.000, 0.149]
Q1×T3×T2×T1	(99) 0.35**	(157) 0.11	(158) 0.31**	(158) 0.30**	1	49.4	0.65	0.074 [0.000, 0.147]
Dataset 2								
Q	NA	(89) 0.12	(19) 0.14	(89) 0.36**	1	52.5	0.54	0.109 [0.000, 0.261]
T	NA	(89) –0.15	(19) –0.02	(89) 0.19	1	47.6	0.41	0.019 [0.000, 0.212]
Q+T	NA	(89) –0.01	(19) 0.10	(89) 0.35**	1	59.9	0.66	0.137 [0.000, 0.284]
Q+lnT	NA	(89) –0.02	(19) 0.10	(89) 0.34**	1	62.9	0.70	0.095 [0.000, 0.250]
Q1×T2×T1	NA	(89) 0.01	(19) 0.03	(89) 0.30**	1	52.6	0.55	0.000 [0.000, 0.179]
Q1×Q2×T2×T1	NA	(89) 0.05	(19) 0.23	(89) 0.34**	1	54.9	0.59	0.000 [0.000, 0.103]
Q1×Q2×Q3×T2×T1	NA	(89) 0.09	(19) 0.22	(89) 0.33**	1	55.7	0.60	0.000 [0.000, 0.153]

n is the number of observations, ccategory is junior (3), intermediate (4) or senior (5) , lnProfExp is the log-transformed number of years of professional programming experience where part time experience is counted as 25% of full time experience, JavaExp is the number of months of experience with the Java programming language, SEJavaExp is self-evaluated Java programming expertise on a scale from novice (1) to expert (5), #f is the number of suggested factors by PCA, %E is percent total variance Explained by the first PC, α is Cronbach’s alpha, RMSEA is the Root Mean Square Error of Approximation with 90% low (lo90) and hi (hi90) confidence intervals. Data not available for analysis are marked NA. Correlations significant at the 0.05 level (two-tailed) are marked *, correlations significant at the 0.01 level are marked **.

B. *Internal fit indices*

Table 2 shows the fit indices of the investigated score operationalizations. We used PCA to extract the main signal in the data, as represented by the first PC. The number of factors (#f) suggested by PCA indicates to what degree our expectations are present empirically. Q in Dataset 1 indicates a problem, because two factors are indicated by PCA.

The proportion of explained variance by the first PC (%E), indicate the signal to noise ratio for each score operationalization. The additive scoring rules ($Q+T$, $Q+lnT$) which employ standardized values have the highest proportion of explained variance; log transformation before standardization of time results in additional explained variance.

Internal consistency is one way to investigate whether an individual's performance is stable over multiple tasks. High values for Cronbach's α are better than low values (0.60 for inferring group differences and 0.85 for inferring individual differences sometimes used). The two additive scoring rules do well in this respect, followed by Guttman-structured scoring rules. Further, Q in Dataset 1 and T in Dataset 2 have lower α than other alternatives.

Finally, we report how the different scoring alternatives fit according to confirmatory factor analysis. Lower RMSEA values signify better fit; values less than or equal to 0.05 indicates close approximate fit, values between 0.05 and 0.08 indicates reasonable error of approximation, and values above 0.1 suggest poor fit [26]. For Dataset 1, $Q1\times T2\times T1$ and $Q1\times T3\times T2\times T1$ display reasonable model fit (< 0.08) whereas T, and Q has poor fit. $Q+T$ alternatives lie somewhere in between, and the log transformed version for time has better overall fit than the untransformed version. Q for Dataset 2 has poor fit as well, even though this dataset has better overall confirmatory fit, despite the lower statistical power as can be seen by wider 90% Confidence Intervals (CI). Nevertheless, by inspecting the lower CI of Dataset 1, there is sufficient power to state that T fits poorly. However, as is evident by the upper CI of all alternatives, there is not sufficient statistical power to claim support for close model fit for any of these alternatives either; $Q1\times Q2\times T2\times T1$ is overall the best fitting alternative with upper CI slightly above 0.10.

A potential objection to our confirmatory analysis using tau-equivalence is that relative weights for each task may be allowed in the equation for the calculation of the sum score. We investigated RMSEA from such a perspective as well (i.e., a congeneric factor model). Although slightly better fit could be obtained for all scoring alternatives, the same substantive results as above nevertheless apply when comparing alternative ways to combine time and quality.

In summary, an analysis of Q and T separately seems problematic in terms of some correlations, relatively low explained variance and somewhat problematic confirmatory fit in three out of four instances. The additive scoring rules show the highest levels of explained variance and internal consistency, but they display some problems with confirmatory model fit. Overall we see best fitting score operationalizations as $Q1\times T3\times T2\times T1$ for Dataset1 and $Q1\times Q2\times T2\times T1$ for Dataset 2.

C. *Details for factors in Dataset 1and 2*

Table 3 shows the correlations between all but one of the investigated scoring rules ($Q+lnT$ was found to be a better alternative than $Q+T$ and the latter is therefore not reported). Correlations below the diagonal are in terms of (non-parametric) Spearman's rho which does not assume linearity between factors and therefore be used. Parametric correlations (Pearson's r) are given above the diagonal for comparison. All correlations are significant at the 0.01 level (two-tailed).

T and Q have the lowest correlation with each other in both datasets. For all Guttman-structured alternatives in Dataset 1, one may further observe how these scoring rules migrate from closeness with Q to closeness to T when additional time categories are added. For Dataset 2, the Guttman-structured alternatives, similarly, mitigate from closeness to T to closeness to Q when additional quality categories are added. All proposed scoring alternatives also have more shared variance with T and Q separately, than Q and T have with each other. Further, the additive and Guttman-structured scoring alternatives are also somewhat similar in their rank ordering of individuals for Dataset 2 (rho > 0.6). For Dataset 1, however, they are highly similar in their rank ordering of individuals (rho > 0.9).

To verify that the scoring rules predict performance on other programming tasks, we used the $Q1\times T2\times T1$ scoring

TABLE III. CORRELATIONS FOR DATASET 1 AND 2

Scoring rule	***(1)***	***(2)***	***(3)***	***(4)***	***(5)***	***(6)***
Q (1)		0.36	0.76	0.82	0.75	0.71
T (2)	0.33		0.82	0.64	0.73	0.74
Q+lnT (3)	0.72	0.85		0.92	0.95	0.94
Q1×T1 (4)	0.80	0.70	0.92		0.96	0.96
Q1×T2×T1 (5)	0.75	0.78	0.96	0.95		0.98
Q1×T3×T2×T1 (6)	0.72	0.81	0.97	0.96	0.98	

(a) Dataset 1

Scoring rule	***(1)***	***(2)***	***(3)***	***(4)***	***(5)***	***(6)***
Q (1)		0.42	0.83	0.68	0.85	0.95
T (2)	0.41		0.83	0.53	0.56	0.50
Q+lnT (3)	0.84	0.80		0.70	0.83	0.85
Q1× T2×T1 (4)	0.64	0.49	0.62		0.86	0.75
Q1×Q2×T2×T1 (5)	0.81	0.52	0.79	0.83		0.89
Q1×Q2×Q3×T2×T1 (6)	0.96	0.45	0.84	0.70	0.87	

(b) Dataset 2