Construction and Validation of an Instrument for Measuring Programming Skill

Gunnar R. Bergersen, Dag I. K. Sjøberg, *Member, IEEE,* and Tore Dybå, *Member, IEEE*

Abstract—Skilled workers are crucial to the success of software development. The current practice in research and industry for assessing programming skills is mostly to use proxy variables of skill, such as education, experience, and multiple-choice knowledge tests. There is as yet no valid and efficient way to measure programming skill. The aim of this research is to develop a valid instrument that measures programming skill by inferring skill directly from the performance on programming tasks. Over two days, 65 professional developers from eight countries solved 19 Java programming tasks. Based on the developers' performance, the Rasch measurement model was used to construct the instrument. The instrument was found to have satisfactory (internal) psychometric properties and correlated with external variables in compliance with theoretical expectations. Such an instrument has many implications for practice, for example, in job recruitment and project allocation.

Index Terms—skill, programming, performance, instrument, measurement

1 Introduction

Software engineering folklore states that the skill of programmers is crucial to the success of software projects [27], [42]. Consequently, being able to measure skill would be of great interest in such work as job recruitment, job training, project personnel allocation, and software experimentation. In such contexts, an individual's capacity for programming performance is usually evaluated through inspection of their education and experience on CVs and through interviews. Sometimes standardized tests of intelligence, knowledge, and personality are also used. Even though such methods may indicate an individual's level of skill, they do not *measure* skill per se.

Skill is one of three factors that *directly* affect the performance of an individual [32]. The two other factors are motivation and knowledge. Motivation is the willingness to perform. An overview of studies on motivation of software developers can be found in [14]. Knowledge is the possession of facts about how to perform. Much research on programming knowledge can be found in the novice-expert literature of the 1980s [119], [127]. Other factors, such as experience, education, and personality, also *indirectly* affect individual performance through their influence on motivation, knowledge, and skill [105], [126]. In contrast, we are interested in how skill can be measured directly from programming performance. Consequently, our research question is, *to what extent is it possible to construct a valid instrument for measuring programming skill*?

In accordance with the most commonly used definition of skill, from the field of psychology [62], we define programming skill as the ability to use one's knowledge effectively and readily in execution or performance of programming tasks. Consistent with this definition, we constructed and validated an instrument for measuring programming skill by adhering to the principles given in [92], [95], [97]. The implicit assumption was that the level of performance a programmer can reliably show across many tasks is a good indication of skill level. This approach is also commonly used within research on expertise [53]. In the construction of the instrument, we sampled 19 programming tasks of varying degrees of difficulty, taken from prior experiments or developed by ourselves. To determine the difficulty of the tasks, we hired 65 developers from eight countries to solve the tasks.

The construction and validation of the instrument has not been reported before. However, the instrument has already been used to investigate whether a psychological theory of cognitive abilities can be applied to programmers [16] and to investigate how skill moderates the benefit of software technologies and methods [18]. It has also been used to select programmers as research subjects in a multiple-case study [115]. Moreover, the instrument is at present used in commercial pilot setting to measure the skill of employees and candidates from outsourcing vendors.

This research concerns an instrument for measuring programming skill. However, the article may also

- *Gunnar. R. Bergersen is with the Department of Informatics, University of Oslo, PO Box 1080 Blindern, NO-0316, Oslo, Norway. E-mail: gunnab@ifi.uio.no.*
- *Dag I.K. Sjøberg is with the Department of Informatics, University of Oslo, Norway. E-mail: dagsj@ifi.uio.no.*
- *Tore Dybå is with the Department of Informatics, University of Oslo and SINTEF, Norway. E-mail: tore.dyba@sintef.no.*

guide the construction and validation of instruments for measuring other aspects of software engineering.

The remainder of this article is structured as follows. Section 2 describes the theoretical background and fundamental concepts. Section 3 describes the steps involved in the construction of the instrument. Sections 4 and 5 describe, respectively, the internal and external validation of the instrument. Section 6 discusses the answer to the research question, contributions to research, implications for practice, limitations, and future work. Section 7 concludes.

2 Fundamental Concepts

This section describes the theory of skill, models for measurement, operationalizations of performance, and instrument validity. Fig. 1 shows how the fundamental concepts involved are related. The skill measure is indicated by the performance of an individual on a set of tasks. Each task is thus an *indicator* [40], which in turn is *defined* by a scoring rule that is applied to the time and quality of task solutions (i.e., a "response" in the figure). The arrows show the direction of reading and causality. The part of the model with arrows pointing downwards constitutes a *reflective model*. The part with arrows pointing upwards constitutes a *formative model* [50].

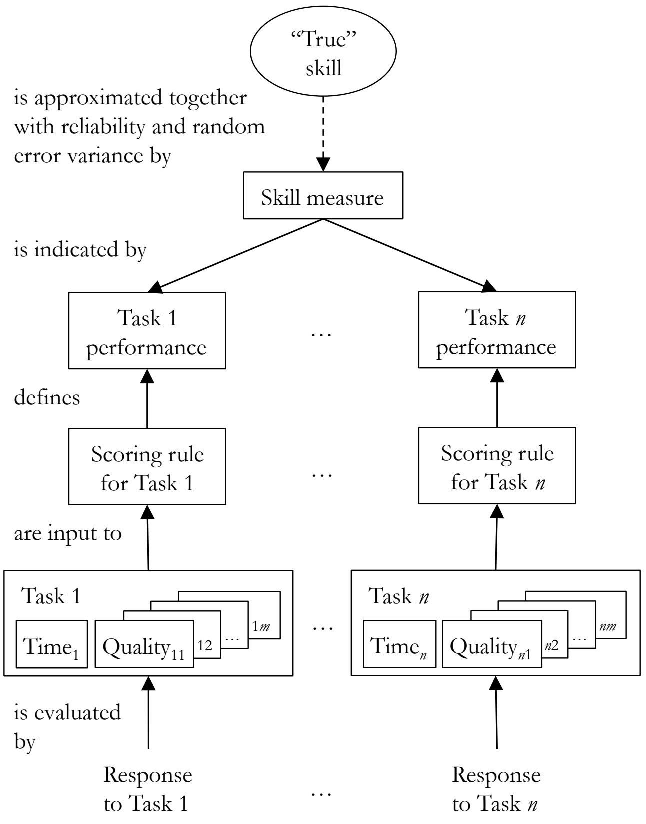

Fig. 1. The relations between variables of skill, task performance, and time and quality

2.1 Theory of Skill

In this work, skill is considered as a specific type of ability, albeit with some distinguishing features. Generally, all human abilities are "defined in terms of some kind of performance, or potential for performance" [33, p. 4]. "The term ability . . . may refer to measures of . . . an underlying latent variable, which is presumed to be a continuous monotonic increasing function of the observed measures of performance" [60, p. 122]. Thus, skill has—together with concepts such as aptitude, achievement, capacity, competence, expertise, and proficiency—a monotonic relation to performance.

This positive relation is also an assumption in research on expertise, where reliably superior performance on representative tasks is one of several extended aspects of expertise [55]. According to Ericsson, "[a]s long as experts are given representative tasks that capture essential aspects of their expertise, they can rely on existing skill and will exhibit the same stable performance as they do in everyday life" [52, p. 52].

Unlike some abilities, skill is a psychological variable that can be defined theoretically. Over 80 years ago, Pear [96] recommended using the term for higher levels of performance and then only in conjunction with well-adjusted performance. According to Fitts and Posner [62], the acquisition of skill consists of three overlapping phases. During the initial, *cognitive* phase, an individual uses controlled processing of information to acquire facts on how to perform a task successfully. This phase is sometimes referred to as the knowledge acquisition phase, where declarative facts (i.e., knowledge) "concerned with the properties of objects, persons, events and their relationships" [102, p. 88] are acquired. In the second, *associative* phase, facts and performance components become interconnected and performance improves, with respect to both number of errors and time. In the third, *autonomous* phase, tasks are accomplished fast and precisely with less need for cognitive control.

Although much of the earlier research on skill was conducted on *motor* skills, Anderson and other researchers devoted much attention to the research on *cognitive* skills in general during the 1980s (e.g., [4]) and *programming skills* in particular (e.g., [5], [6], [111]) using Fitts and Postner's [62] early work. Anderson [5] noted that the errors associated with solving one set of programming problems was the best predictor of the number of errors on other programming problems. We now examine how such insights can be used in the development of a model for measuring skill.

2.2 Model for Measurement

A *model for measurement* explicates how measurement is conceptualized in a specific context [58], [91]. The choice of a measurement model also exposes the assumptions that underlie one's effort to measure something [22].

It is common in software engineering to use the term "measurement" according to Stevens' broad definition from 1946: "the assignment of numerals to objects or events according to rules" [120, p. 667]; see, for example, the early paper by Curtis [41]. However, Stevens' definition of measurement is irreconcilable with scientific measurement as defined in physics: "Scientific measurement is ... *the estimation or discovery of the ratio of some magnitude of a quantitative attribute to a unit of the same attribute*" [93, p. 358]. More generally, Stevens' definition is not universally accepted [97] because even meaningless rules can yield measurements, according to this definition [21], also see [58].

A challenge is that Stevens' definition is commonly used in software engineering, while scholars [58], [59] advocate that measurement in software engineering should adhere to the scientific principles of measurement [79], [81]. This call for increased rigor was answered by calls for pragmatism; if the stricter definition of measurement was applied, it "would represent a substantial hindrance to the progress of empirical research in software engineering" [26, p. 61]. Consequently, the term "measurement" is used according to varying levels of rigor in software engineering. At one extreme, a researcher merely asserts that measurement is achieved, or else the researcher is silent about this issue altogether. At the other extreme, a researcher may rigorously test whether a quantitative measure has been obtained for a phenomenon, for example, through testing whether the data conforms to the requirements of additive conjoint measurement [87].

In this work, we chose the Rasch measurement model, which resembles additive conjoint measurement, albeit from a probabilistic viewpoint [24]. Although this viewpoint is still being debated [83], the use of probability is central to science in general [67] and experimentation in particular [108]. Nevertheless, for the present work, it suffices to point out that the Rasch model allows more and better tests of whether measurements are achieved according to a rigorous definition of measurement.

2.3 Rasch Measurement Model

Many types of models are available to assess psychological abilities such as skill. These models often present questions or tasks (called items) to an individual and then an estimate (preferably, a measure) of an individual's ability can be calculated from the sum-score of the item responses. In item response theory (IRT) models, estimates of item difficulty and consistency of responses across people and items are central.

The original Rasch model [101] is a type of IRT model by which skill can be measured. The ability of a person j is denoted β_j, and the difficulty of an item i is denoted δ_i. X_{ij} is a random variable with values 0 and 1 such that $X_{ij} = 1$ if the response is correct and $X_{ij} = 0$ if the response is incorrect when person j solves item i. The probability of a correct response is:

$$Pr\left(X_{ij} = 1 \mid \beta_j, \delta_i\right) = \frac{e^{\beta_j - \delta_i}}{1 + e^{\beta_j - \delta_i}}. \qquad (1)$$

The Rasch model typically uses some form of maximum likelihood function when estimating β and δ. The model uses an interval-logit scale as the unit of measurement. A logit is the logarithmic transformation of the odds. Humphry and Andrich [70] discuss the use of this unit of measurement in the context of the Rasch model.

The original Rasch model is classified as a unidimensional model; that is, ability is measured along only one dimension. Furthermore, the model is called the *dichotomous* Rasch model because only two score categories are available (e.g., incorrect and correct).

Andrich derived the *polytomous* Rasch model [7] as a generalization of the dichotomous model. The polytomous model permits multiple score categories $0, \ldots, M_i$, where M_i is the maximum score for an item i. Each higher score category indicates a higher ability (and therefore also an increased difficulty in solving correctly), which enables evaluations of partially correct solutions. This is an attractive feature for our work and we therefore used the polytomous Rasch model.

A requirement of the polytomous, unidimensional Rasch model is that score categories must be structured according to a Guttman-structured response subspace [8]. For example, a response awarded a score of "2" for an item i indicates that the requirements for scores 0, 1, and 2 are met and that the requirements for scores 3 to M_i are not met.

The Rasch model has been used in many large-scale educational testing programmes, such as OECD's Programme for International Student Assessment [20]. The Rasch model has also been used to measure programming ability in C [128], Lisp [98], and Pascal [122], and to explain software engineering practices that are based on CMM [44].

2.4 Operationalization of Performance

When inferring skill, only performance that is under the complete control of the individual is of interest [32]. Performance may be evaluated with respect to time and quality. A challenge in software engineering is that software quality is not a unitary concept. For example, McCall [89] lists 11 software quality factors along with their expected relationships. Thus, to answer which of two different solutions are of higher quality, one must know which quality factors should be optimized given the purpose of the task. To illustrate, a calculator that supports division is

TABLE 1
Addressing validity aspects recommended by the APA guidelines

Aspect	Description	Addressed in section(s)
1: Task content	Do the tasks as a whole span the dimension of the thing being measured?	3.1, 3.2, 6.1
2: Response process	Are the mental processes involved when solving the tasks representative of the psychological variable being measured?	2.1, 3.2, 6.1
3: Internal structure	Does the structure of the response data (in dimensionality and reliability) conform to expectations?	2.5, 4.1–4.5, 6.1
4: Correlations with other variables	Does the measure yield patterns in correlations with other variables that are consistent with what is expected from theory or previous research?	5.1, 5.2, 6.1

APA also includes ´´consequences of testing,´´ which addresses social policies of testing. This aspect is beyond the scope of this article.

of higher quality than one that does not. Further, a solution that gracefully handles an exception for division by zero is better than a solution that crashes when such an exception occurs.

In addition to dealing with different levels of quality when evaluating performance, it is a challenge to deal with the tradeoff between quality of the solution and the time spent on implementing the solution. Generally, for two solutions of equal quality, the solution that took the least time to implement denotes higher performance. It is also trivial to classify an incorrect solution that took a long time to implement as lower performance than a correct solution that took a short time. However, whether a high-quality solution that took a long time to implement is of higher performance than a low-quality solution that took a short time is not a simple question. In general, there are two main strategies for combining time and quality to define performance [17]:

- *Time fixed, quality = performance*: Use a brief time limit and let subjects solve tasks in predetermined incremental steps; the number of successful steps within the time limit defines performance.
- *Quality fixed, negated time = performance*: Use a relaxed time limit with a high, expected chance of a correct solution; the less time used, the higher the performance.

A mix of the two main strategies is also possible. In [17], we reanalyzed previously published experiments and combined time and quality as performance using a Guttman structure. Higher scores on task performance were first assigned according to the *time fixed* description described above. Correct solutions were additionally assigned higher scores according to the *quality fixed* description given above.

In addition to tasks that require a single solution, the Guttman structure can also be used for testlets, that is, for tasks where solutions are solved in multiple steps and where each step builds upon the previous step. A core issue in this article is the extent to which programming performance on a set of tasks that are scored using a Guttman structure can be used to measure programming skill using the Rasch model.

2.5 Instrument Validity

According to Shadish, Cook, and Campbell, the use of measurement theory is one of several ways to make generalized causal inferences: "[r]esearchers regularly use a small number of items to represent more general constructs that they think those items measure, and their selection of those items is rarely random" [108, p. 349]. Therefore, it is important to address when and how performance on a combined set of programming tasks can be regarded as a valid measure of programming skill. We use Borsboom, Mellenbergh, and Van Heerden's definition of validity: "[a] test is valid for measuring an attribute if and only if (a) the attribute exists and (b) variations in the attribute causally produce variations in the outcomes of the measurement procedure" [23, p. 1].

We distinguish validity from the *process* of evaluating validity, that is, validation [23]. According to the American Psychology Association (APA), support for or evidence against validity may be addressed according to the aspects shown in Table 1 [2].

When multiple observations of task performance are used as indicators of skill (Fig. 1), it is possible to address what is shared across observations. After the *common variance* for skill is extracted from the task performance data, what remains is error variance (Fig. 2). This error, or residual, variance can be divided in two: *Random error variance* is noise, which should be reduced to the largest possible extent, but does not in itself invalidate a measure. However, *systematic error*

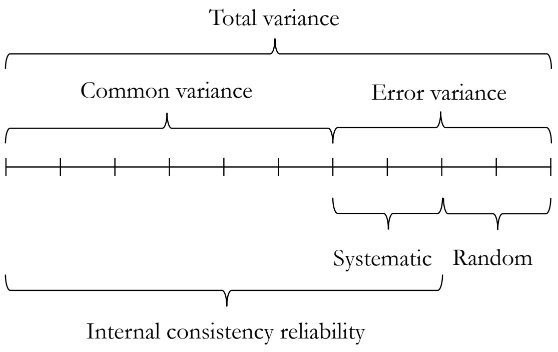

Fig. 2. Variance components (adapted from [103])

TABLE 2
Activities of the construction phase

Activity	Description	Purpose
Definition of programming and scope of instrument	Defined and explained the area that the concept of programming skill was meant to capture.	Clarify the area in which the instrument should be used.
Task sampling and construction	Obtained 19 tasks with heterogeneous operationalizations across dimensions.	Obtain a set of tasks that span the intended scope of the instrument.
Scoring rules for tasks	Developed preliminary scoring rules based on different quality attributes for the 19 tasks.	Decide how combinations of time and quality as performance should be scored.
Subject sampling	Hired 65 subjects from nine companies in eight countries to participate.	Obtain a sample of industrial programmers with widely different backgrounds.
Data collection	The subjects solved the tasks over two days using individual randomized task order.	Obtain programming performance data to be used in the evaluation of scoring rules.
Data splitting	Split the subject data into one construction (n = 44) and one validation (n = 21) data set.	Establish two independent data sets for instrument construction and validation.
Determining criterion for evaluating scoring rules	Used as the criterion the fit of task performance data to the Rasch model.	Evaluate scoring rules that combine time and quality as task performance.
Constructing and adjusting scoring rules using Rasch analysis	Combined time and quality using the scoring rules for 17 tasks. Scoring rules for two tasks could not be obtained and these tasks were removed.	Obtain a well-defined measure of task performance.

variance indicates systematic patterns in the variance that are not a part of the intended measure.

According to Messick [91], systematic error variance is one of the two major threats to (construct) validity. This threat occurs when something other than the variable being measured systematically influences observations in unintended ways; it is therefore called construct-irrelevant variance.

The second major threat to validity is construct underrepresentation, which occurs when central aspects of the thing being measured are not captured [92]. For example, if only one type of programming task is represented in the instrument, mono-operation bias may occur; that is, other categories of programming tasks are not captured. Similarly, if only type of evaluation of task quality is used (e.g., unit test cases), mono-method bias may occur; that is, other categories of evaluation are not captured. Thus, even if the tasks share a large proportion of common variance, they may still not fully represent what one intends to measure.

Reliability may be calculated in several ways [95], but we will only focus on internal consistency reliability, of which Cronbach's coefficient alpha (α) is one instance where reliability is represented as the average inter-correlations of tasks. As shown in Fig. 2, internal consistency reliability comprises both common variance and systematic error variance. A high α is therefore insufficient to conclude with respect to validity, because the magnitude of the systematic error variance is unknown.

3 Instrument Construction

The activities that we conducted to construct the measurement instrument are shown in Table 2. Each activity occurred chronologically according to the subsection structure.

3.1 Definition of Programming and Scope of Instrument

We define programming as the activites of writing code from scratch, and modifying and debugging code. In particular, the last two activities also code comprehension as a central activity. Although other life-cycle activities such as analysis, design, testing, and deployment are important, programming is the dominant activity of software development. In a study of four software projects [3], the proportion of programming constituted 44 to 49 percent of all the development activities, which is similar to what was found in a survey of software developers conducted at a programming-related forum (50%, $n = 1490$).[1]

Programming skill is related to performance on programming tasks. The universe of tasks consists of many dimensions, such as application domains, technologies, and programming languages. To increase generalizability, we limited the scope of the instrument to tasks that belonged neither to any particular application domain nor to any particular software technology. However, we are not aware of how to measure programming skill independent of a programming language. Therefore, we limited the scope of our instrument to the widely used programming language, Java.

3.2 Task Sampling and Construction

To obtain generalizable results, one would ideally use random sampling of tasks [108]. However, there is no such thing as a pool of all programming tasks that

1. www.aboutprogrammers.org

have been conducted in the last, say, five years from which we could randomly sample tasks.

Alternatively, one could construct or sample tasks that varied across all relevant dimensions and characteristics. A programming task may have many characteristics, including the comprehensiveness and complexity of the task, the required quality of the solution, and the processes, methods, and tools used to support the fulfillment of the task. Additionally, characteristics of the system on which the task is performed will also affect the performance, for example, size and complexity of the system, code quality, and accompanying artifacts (requirement specifications, design documents, bug reports, configuration management repositories, etc.). However, the number of possible combinations of such characteristics is almost infinite [49]. Therefore, one has to use convenience sampling.

To help achieve generalizability into an industrial programming setting, we *purposively sampled typical instances* [108]. Thus, the set of tasks were selected or constructed to capture a range of aspects of industrial programming tasks to increase realism [112].

We also included some "toy tasks" to measure low-skilled subjects. Another purpose was to investigate whether the use of such tasks yields the same measure of skill as the one yielded by using industrial tasks. See Kane et al. [74] for a discussion on the use of tasks with various degrees of realism in educational measurement.

More generally, whether tasks with different characteristics yield the same measure of skill, is an open question. In our case, we varied task origin, lifecycle, time limit, presence of subtasks, and evaluation type to reduce their potential confounding effect as follows:

- *Task origin* was varied across previous experiments (verbatim or modified), problems formulated in books, general programming problems, or tailored new tasks (we paid two professionals to develop new tasks).
- *Lifecycle* was varied across write, maintain and debug phases.
- *Time limit* was varied across a mix of short ($\sim$10 minutes), medium ($\sim$25 minutes), and long tasks ($\sim$45 minutes).
- *Subtasks*, which require multiple submissions (i.e., testlet structure; see Section 3.3), were used for some of the tasks.
- *Evaluation type* was automatic, manual, or a combination of automatic and manual (e.g., automatic regression and functional testing combined with manual evaluations of code quality).

Table 7 in Appendix A summarizes all characteristics of the 19 tasks that were sampled or constructed for the instrument.

3.3 Scoring Rules for Tasks

The decision on how to combine time and quality into a single variable of task performance for a specific task is operationalized in terms of scoring rules [17]. Each scoring rule is uniquely associated with a specific task. An example of a scoring rule that we constructed is shown in Table 3. Three subtasks extend the functionality of the Library Application described in Table 7: add an e-mail field to "create new" book lender (Subtask A), allow an entry for e-mail and make it persistent (Subtask B), and update all other dialogue boxes in the application correspondingly (Subtask C). The three subtasks were to be implemented incrementally, where a correct solution for Subtask B required a correct solution for Subtask A, and a correct solution for Subtask C required correct solutions for Subtasks A and B.

Quality was operationalized as correct ($Q = 1$) or incorrect ($Q = 0$) implementation of each of the subtasks. Incorrect solutions for Subtask A ($Q_A = 0$) or solutions submitted after the time limit of 38 minutes ($T_{38} = 0$) received a score of "0". Solutions submitted within the time limit ($T_{38} = 1$) received a score of "1" if only Subtask A was correct ($Q_A = 1$), "2" if both Subtasks A and B were correct ($Q_B = 1$), and "3" if Subtasks A, B, and C ($Q_C = 1$) were correct. Additionally, if Subtasks A, B, and C were correct, the score was "4" if time was below 31 minutes ($T_{31} = 1$) and "5" if time was below 25 minutes ($T_{25} = 1$).

For most of the tasks, functional correctness was the main quality attribute, which was evaluated automatically using test cases in JUnit and FitNesse. For five of the tasks, the quality attributes were manually evaluated to some extent. Examples of such attributes were code readability, good use of object-oriented principles, and redundancy of code. (See Table 7 for more details.) A challenge with manual evaluation is that it may be hard to perform consistently. Therefore, we refrained from using subjective evaluations of quality, such as "poor" or "good." Instead, we used scoring rubrics where each score was uniquely associated with a requirement that could be evaluated consistently, for example, "is an abstract base class used in X?" Using scoring rubrics this way helps achieve objective assessments given that the rater has sufficient knowledge in the area. In the example

	Time			
Correctness	$T_{38} = 0$	$T_{38} = 1$	$T_{31} = 1$	$T_{25} = 1$
$Q_C = 1$	0	3	4	5
$Q_B = 1$	0	2	2	2
$Q_A = 1$	0	1	1	1
$Q_A = 0$	0	0	0	0

TABLE 3
The scoring rule for the Library Application task

above, the rater must know how to identify an abstract base class.

For all tasks, the time limits used were either based on empirical data from earlier studies (see Table 7) or results from pilot tests. Some strategies for deciding on time limits in the scoring of performance are provided in our previous work [17].

Each task description specified which quality focus should be emphasized to help reduce potential confounding effects of having subjects working towards different perceived goals. All the task descriptions also stated that a solution was required to be correct, or of acceptable quality, in order for a more quickly submitted solution to be scored as being of higher performance.

3.4 Subject Sampling

We contacted 19 companies in 12 countries with a request to submit quotes for participation. We hired 65 developers from nine of the companies for two full days. The companies were located in Belarus, Czech Republic, Italy, Lithuania, Moldova, Norway, Poland, and Russia. Company size was a mix of small (less than 50 developers) and medium (less than 250 developers). According to the categorization by the companies themselves, there were 27 senior, 19 intermediate, and 19 junior developers. We requested a fixed hourly price for each developer and paid each company additionally for six hours of project management, including recruiting subjects and setting up infrastructure. The total cost for hiring the subjects was approximately 40,000 euros.

We requested that individuals volunteer to participate, be allowed to terminate the participation, be proficient in English, and have experience with programming in Java for the last six months. All the subjects and companies were guaranteed anonymity and none were given results. Therefore, no clear motivation for individual cheating or company selection bias (e.g., by selecting the most skilled developers) was present.

3.5 Data Collection

A support environment [12] for the experiment was used to administer questionnaires, download task descriptions and code, and upload solutions. Additionally, we developed a tool to run automatic and semi-automatic test cases for quality and to apply the scoring rules. A pilot test was conducted on the task materials. All descriptions, tasks, and code were written in English.

The subjects filled in questionnaires both before beginning the programming tasks and upon completion. After solving an initial practice task, each subject received the 19 tasks in an individually randomized order. The subjects used their regular integrated development environment (IDE) to solve the tasks. Those who finished all the tasks early were allowed to leave, which ensured some time pressure [10], [11]. Without time pressure, it is difficult to distinguish among the performance of the developers along the time dimension.

To reduce the confounding effect of the subjects' reading speed on their programming skill, they were given 5 to 10 minutes to familiarize themselves with the task description and documentation prior to downloading code. The time used for the analysis began when the code was downloaded and ended when the solution was submitted.

3.6 Data Splitting

The true test of any model is not whether most of the variance in the data can be accounted for but whether the model fits equally well when new data becomes available [61]. Overfitting occurs when adjustable parameters of a model are tweaked to better account for idiosyncrasies in the data that may not represent the population studied [36]. Because the ways to combine time and quality variables into a single performance variable are potentially infinite [17], a concern was whether we would overfit the scoring rules during instrument construction.

A common strategy to account for this problem is to construct the model using one data set and subsequently use another data set to test hypotheses [43] or other claims. Using generalizability theory [109], we first investigated the magnitude of different sources of variance for the experiment reported in [11]. This analysis confirmed that variability between persons and tasks was large, which was also reported in [45], [100]. Therefore, we decided to use about two-thirds of the available data (44 persons, 19 tasks) to construct the instrument and the remaining one-third to check for potential overfitting (21 persons, 19 tasks). We randomly sampled two-thirds of the developers within each company for the instrument construction data set. The remaining one-third of the data was saved for (and not looked at before) the validation phase (Section 4).

3.7 Determining the Criterion for Evaluating Scoring Rules

We measure skill from programming performance on multiple tasks. As mentioned in Section 2.3, the polytomous Rasch model uses multiple score categories for each task. The scores for each task (item) i are determined as a function X_i on the set of individuals, that is, for an individual j, $X_{ij} = X_i(j)$. The function rule for X_i is determined by the time used to solve a specific task T_i and m quality variables $Q_{ik}, k = 1, \ldots, m_i$ that describe the task solution (see Fig. 1). This rule is called a scoring rule for the item i:

$$X_i = scoringrule_i(T_i, Q_{i1}, \ldots, Q_{im}). \quad (2)$$

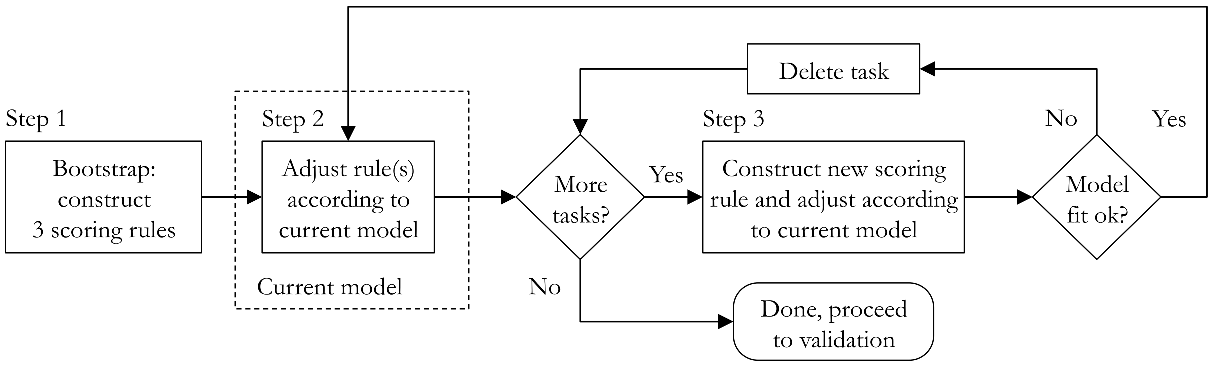
Step 1
Bootstrap: construct 3 scoring rules
Step 2
Adjust rule(s) according to current model
Current model
More tasks?
Yes
No
Step 3
Construct new scoring rule and adjust according to current model
Model fit ok?
No
Yes
Delete task
Done, proceed to validation

Fig. 3. Constructing and adjusting scoring rules prior to instrument validation

To measure skill, one must determine both the maximum allowed score and a sensible scoring rule for each of the tasks. However, which criteria should be used to make such a decision? In some situations, external criteria may be used. For example, the parameters in a model for measuring skill in sales can be adjusted according to the actual net sales (external criterion) [32]. However, within most other fields, it is difficult to obtain a suitable external criterion [95]. For example, it is problematic to use a supervisor's rating of overall performance as *the* external criterion for the job performance of individuals [31]. More generally, the idea of using an "ultimate criterion" has also been described as "an idea that has severely retarded personnel research" [30, p. 713] and as "truly one of the most serious mistakes ever made in the theory of psychological measurement" [23, p. 1065].

Given the lack of a valid external criterion for determining programming skill, we used the fit of task performance data to the Rasch measurement model as an internal criterion to determine the maximum number of score points for any task and to evaluate each scoring rule. Rasch analysis can determine whether performance on each of the combinations of available tasks and persons is mutually consistent. For all *n* tasks, the question is whether the set of $n-1$ other tasks are consistent with the current task. Each task is thus evaluated analogously to Neurath's bootstrap process:

> ...in science we are like sailors who must repair a rotting ship while it is afloat at sea. We depend on the relative soundness of all of the other planks while we replace a particular weak one. Each of the planks we now depend on we will in turn have to replace. No one of them is a foundation, nor a point of certainty, no one of them is incorrigible [28, p. 43].

3.8 Constructing and Adjusting Scoring Rules Using Rasch Analysis

Fig. 3 shows the steps involved in constructing and adjusting scoring rules (cf. Neurath's metaphor). In Step 1, an initial set of scoring rules must be established using bootstrapping. This set serves as the basis for the evaluation of subsequent rules. To identify the initial set of scoring rules in our case, we used Tasks 8, 9, and 17 (Table 7), for which we already had extensive programming performance data available [10], [11], [77], [78], [80]. We had determined a set of scoring rules based on this data for these three tasks [17], which became the initial set of scoring rules in the construction of the instrument.

In Step 2, scoring rules are adjusted relative to each other so that the pattern of performance of available tasks is consistent according to Rasch analysis. In our case, we adjusted the scoring rules to achieve good overall model fit, as indicated by Pearson's chi-square test. We then had to check that each new task increased measurement precision, that no task displayed a misfit to the model, and that other model fit indices were acceptable (see Section 4). A frequent reason for misfit was too few or too many score categories for a task (i.e., parameter M_i in Equation 2). Space limitations prevent us from describing the details of the analyses we performed using the Rumm2020 software [9]. (See, for example, [20], [129], [130] for an introduction to Rasch analysis.)

In Step 3, the current model (i.e., the minimally "floating ship") is built upon by including one additional task at a time. Each new task, with its corresponding scoring rule, has to yield consistent empirical results with the current model, similar to Step 2. If an acceptable fit to the current model can be obtained for the new task, it is imported into the current model, and Step 2 is repeated. Otherwise, the task is deleted.

The process is repeated until no more tasks are available. In our case, the two tasks that involved concurrent programming (Tasks 18 and 19) were excluded because we could not identify well-fitting scoring rules, leaving 17 tasks with acceptable model fit for the construction data set. The two excluded tasks were originally included to contrast measures of programming skill based on tasks that involved concurrent programming with tasks that did not involve concurrent programming. However, a problem

TABLE 4
Activities of the internal validation phase

Activity	Description	Purpose
Test for overfitting	Compared model fit with task difficulty parameters of the two data sets. Two tasks were excluded.	Identify whether the scoring rules for the construction data set were overfitted.
Test for unidimensionality	Compared the two maximally different subsets of the tasks to check for unidimensionality. Three tasks were excluded.	Determine whether various subsets of the tasks yield the same measure of skill.
Test for task model fit	Investigated the residual variance and match between model values and empirical data.	Determine whether each task displays consistent and well-fitting performance across subjects.
Test for person model fit	Investigated residual variance and determined whether subjects' response patterns across tasks are too random or too deterministic.	Determine whether each subject displays consistent and well-fitting performance across tasks.
Check psychometric properties	Checked for reliability and targeting.	Determine the final version of the instrument.

we encountered was that the two concurrent tasks were both difficult to solve and difficult to score consistently. With only two tasks constituting a single sub dimension, it is difficult to know whether they could not be integrated into the instrument due to problems arising from the difficulty of the tasks, or problems with the scoring rules or task descriptions. Therefore, at present, the relation between concurrent programming and the existing tasks of the instrument should be considered an unanswered question.

4 Internal Instrument Validation

We investigate aspects that may indicate a lack of instrument validity according to the activities in Table 4.

4.1 Test for Overfitting

We investigate whether a model based on the task performance scores of one group of subjects fits equally well the performance scores of another group of subjects. The instrument was constructed using data from 44 subjects. Another set of data from 21 subjects was used for validation. If there were no overfitting of the scoring rules created on the basis of the task performance of the subjects in the construction data set, the task performance scores of the validation subjects would fit the Rasch model equally well. Tasks for which there are differences between the two groups of subjects should be removed from the instrument to reduce the risk of model overfitting.

First, we verified that the two sets of subjects had similar programming skill as measured by the instrument: only negligible differences in mean skill ($\Delta\beta = 0.02$ logits) and distribution ($\Delta SD\beta = 0.10$ logits) were found. This indicates that the random allocation of subjects to the two data sets was successful.

Second, a differential item functioning (DIF) analysis [95] was performed to investigate measurement bias. Task 17 (see Table 7) showed statistically significant DIF for the two data sets. By removing this task, the overall model fit was significantly better ($\Delta\chi_2 = 5.42$, $\Delta df = 1$, $p = 0.02$), as indicated by a test of nested model differences [86]. Consequently, the task was removed from the instrument.

Third, Task 13, the "Hello World" task, also contributed negatively to model fit. In the debriefing session after data had been collected, several subjects expressed confusion about why this task had been included. Even highly skilled individuals used up to 10 minutes to understand what "the trick" was with this task. It was therefore removed from the instrument, thereby leaving 15 tasks to be validated.

We have now removed all the tasks that either indicated overfitting during the construction process (Section 3) or contained other problems that we became aware of when we compared model fit for the construction and validation data sets. To increase statistical power in the subsequent validation activities, we joined the two datasets.

4.2 Test of Unidimensionality

Many different processes and factors are usually involved when an individual performs a task. From the perspective of the Rasch measurement model, unidimensionality refers to whether it may still be sufficient to only use one variable, programming skill, for each individual to account for all non-random error variance in the performance data (see Fig. 2).

It is a challenge to decide on an operational test for determining unidimensionality. In the physical sciences, two valid rulers must yield the same measure of length within the standard error of measurement. In contrast, the large standard errors of measurement associated with single tasks makes it implacable to use this approach. A solution is therefore to first combine tasks into subsets of tasks to reduce the standard error (by increasing the common variance in Fig. 2), and then evaluate whether the subsets of tasks measure the same (i.e., are unidimensional). However, how does one determine the allocation of tasks into the distinct subsets?

Smith [118] proposed a test that uses principal component analysis (PCA) on item residuals (here: task residuals) to determine the two sets of subtasks that will yield the most different estimates of ability. A task residual is the difference between the actual (observed) and expected task performance score. For example, an individual may score "2" on one task, whereas the expected score from the Rasch model would be "2.34" based on the individuals performance on the other tasks. Smith's test is based on each task's loading on the first residual principal component. In terms of unexplained variance, all the tasks that explain the largest part of the residual variance in one direction comprise one subset used to measure skill; all the tasks that explain the largest part in the opposite direction comprise the contrasting subset to measure skill. If the difference between the two measures of skill deviates significantly from what one would expect from a normal distribution, the test is not unidimensional.

The result of Smith's test for the 15 tasks showed that the instrument was not unidimensional. The correlations between the task residuals indicated that the three debugging tasks (Tasks 14–16 in Table 7) contributed negatively to instrument unidimensionality. By removing these three tasks, Smith's test indicated acceptable unidimensionality ($p = 0.046$, low 95% CI = 0.00). The visual representation of this result is included as supplementary material to be available at http://ieeexplore.ieee.org.

Even though unidimensionality was acceptable, the tasks loading on the first residual principal component revealed that this component contained some systematic error variance. The most different estimates of skill by the two subsets were obtained by assigning six relatively easy tasks ("Easy") and six relatively difficult tasks ("Difficult") to the two subsets. This indicates that a slight difficulty factor is the source of the systematic error variance. The two subsets of tasks are indicated in Table 8 in Appendix B, which reports the performance scores of all 65 subjects.

4.3 Test of Task Model Fit

In Section 4.1, we used overall model fit to determine which tasks to remove. In this section, we inspect that part of overall model fit which relates to tasks and their residuals. Similar to the two types of error variance, residuals can display systematic patterns or be random noise. Ideally, there should be no systematic patterns, and the residuals should approximate a random normal distribution [117]. We now describe three standard Rasch analyses of the residuals.

First, if the residuals for two tasks are both random noise, they should not covary. By convention in the Rasch community,[2] correlations larger than ± 0.3 may be problematic. For the 66 correlations investigated (those below the diagonal of the 12×12 correlation matrix of tasks) we found only four correlations outside the acceptable region. We investigated the corresponding tasks and found no substantive reason for the residual correlations. We also ran five simulations with data simulated to perfectly fit the model and found a similar low frequency of unacceptable correlations for all 65 persons and 12 tasks. Therefore, we did not regard residual correlations as a major threat for the instrument.

Second, to detect whether residual variance was shared between tasks, we analyzed the residuals using PCA. For the 12 tasks, 12 factors were extracted using Varimax rotation (unlike Smith's test, which uses a solution where all the factors are orthogonal to each other). For all the tasks, we found that the residual variance loaded above ± 0.91 on one, and only one, factor and that no task loaded higher than ± 0.31 upon factors that were unique to other tasks. Consequently, the residual variance was mostly unique for each task, which in turn indicated independence among the tasks.

Third, we investigated the extent to which there was a match between the *expected* performance (according to the Rasch model) on a task given a certain skill level and the *actual* performance of an individual (or group) with this skill level. The Rasch model calculates estimates of person skill and task difficulty using the available task performance data (see Table 8). Based on the estimated task difficulty, the expected task performance score for any skill level can be calculated (e.g., if $\beta = \delta = 1$ in Equation 1, the expected task performance score is 0.50).

The actual performance on a task is calculated using individuals that are divided into two (or more) groups based on their skill level as calculated on the basis of their performance on all the *other* tasks. The mean task performance of such groups, for example, below-average versus above-average skill, are then contrasted with what is expected from the Rasch model given the same skill levels as those of the two groups, respectively.

A task residual is the difference between the expected and actual performance of all subjects on a specific task. Using Rumm2020, positive task residuals indicate under-discrimination; that is, below-average skill subjects perform better than expected and above-average skilled subjects perform worse than expected. Negative task residuals indicate over-discrimination, which is the reverse of under-discrimination. Fig. 4 shows the standardized task residual and the estimated difficulty for all the tasks. The size of the bubbles indicates the standard error of measurement of

2. Online resources, such as www.rasch.org, can provide insight of such conventions and how they are applied. Nevertheless, many conventions lack a documented rationale.

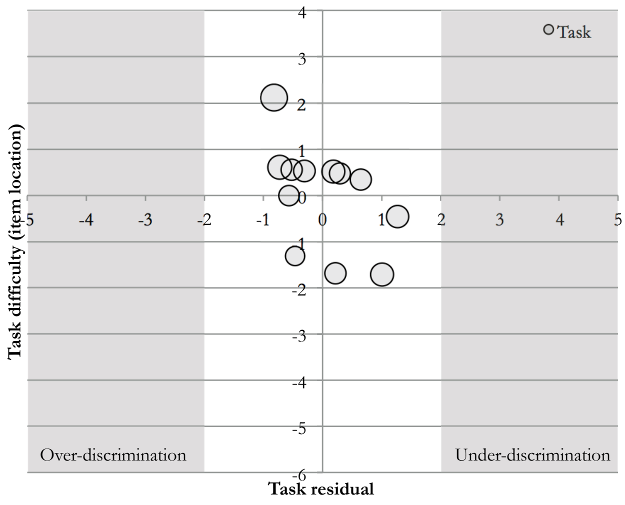

Fig. 4. Task fit to the model

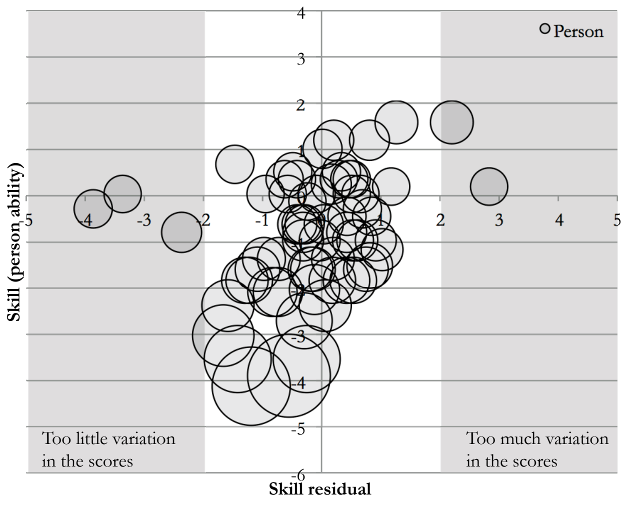

Fig. 5. Person fit to the model

each estimate. By convention in the Rasch community, task residuals between -2.0 and 2.0 are acceptable (i.e., ±2 SD or roughly a 95% confidence interval) and all the task residuals had acceptable, non-significant values. Overall, this indicates a reasonable match between the expected and actual performance on the tasks.

4.4 Test of Person Model Fit

Similar to task model fit, the analysis of person model fit also involves the inspection of standardized residuals. The Rasch model assumes that people perform according to a normal distribution around their true level of skill (i.e., some randomness is assumed). Using Rumm2020, negative person residuals (here: skill residuals) indicate *too little* variation in the performance scores, whereas positive skill residuals indicate *too much* variation [20].

Fig. 5 shows that the individual's response pattern in general fits the model; 5 of the 65 subjects have unacceptable skill residuals, which is close to the proportion of acceptable values by chance (3.25 persons) given the sample size. The bubble size indicates the standard error of measurement for the skill estimate of each individual. Less skilled individuals have higher standard errors of measurement than the more skilled ones, because the measurement precision of the Rasch model is not uniform; it is the smallest when the difficulty of items matches the ability of the subjects [51]. Fig. 5 also shows that, on average, less skilled subjects are also more associated with negative residuals than more skilled subjects who, to some extent, are more associated with positive skill residuals. An explanation is that it is more likely that a highly skilled person completely fails a task by accident than a lower-skilled person achieves the highest possible score by accident (see [88] generally).

Another concern is whether the subjects increased their performance throughout the two days they solved the tasks. Fig. 6 shows a box plot of the skill-task residuals (actual minus expected performance for each task for each individual) according to task order; that is, the first box plot shows the distribution of the first task solved by the subjects, the second box plot shows the second task, etc.[3]

The subjects received the tasks in individual randomized order (Section 3.5). Therefore, if a systematic learning effect [108] or other improvements in performance [52] were present, negative skill-task residuals

3. There are 19 locations for task order because 19 tasks were originally given to the subjects, even though seven tasks were removed later.

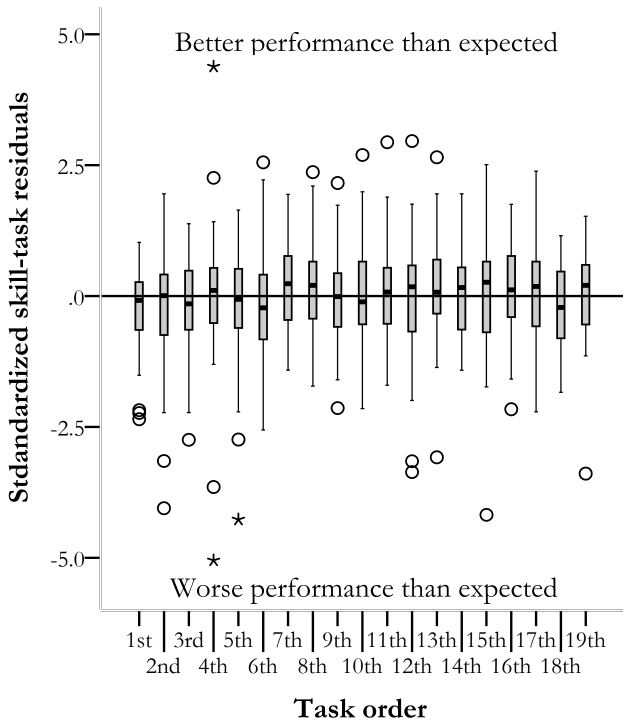

Fig. 6. Skill-task residuals depending on task order

TABLE 5
Descriptive statistics for Java skill and external variables

Variable type	Variable	N	Mean	SD	Min	Max	Unit
Test based (objective)	Java skill	65	−0.8	1.3	–4.1	1.6	Logit
	Java knowledge	60	21.7	4.8	7	29	Sum of correct answers
	Working memory	27	0.1	2.2	–3.4	3.6	Sum of three standardized tests[a]
Test based (subjective)	Technical skills	65	11.5	1.9	5	15	Sum of 3 questions[b]
	Managerial skills	65	25.9	4.2	17	34	Sum of 8 questions[b]
	People skills	65	2.2	2.9	16	29	Sum of 6 questions[b]
Reported by others	Developer category	65	2.1	0.8	1	3	Junior, intermediate or senior[c]
Self-reported	Experience						
	Recent Java	65	26.6	21.4	0	75	Months
	Java	65	40.0	25.2	2	130	Months
	Programming	65	45.9	37.8	4	160	Months
	Work	65	63.6	65.3	4	360	Months
	Expertise						
	Java	65	3.5	0.9	1	5	5-category Likert scale[d]
	Programming	65	3.7	0.7	2	5	5-category Likert scale[d]
	Java LOC	64[e]	135k	253k	0.5k	1000k	Lines of code
	Motivation	65	8.4	1.1	6	10	10-category Likert scale[f]
	Learned new things	65	3.5	0.9	1	5	5-category Likert scale[g]

[a] The sums of perfectly recalled sets for each of the three tests were standardized and added. [b] Unsatisfactory = 1; marginal = 2; average = 3; good = 4; excellent = 5. [c] Scored 1–3; assigned by closest supervisor or project manager. [d] Novice = 1; expert = 5. [e] Maximum = 10 (minimum = 1 is implied) [e] One observation was removed as an extreme outlier according to Grubbs' test. [g] Strongly disagree = 1; disagree = 2; neither = 3; agree = 4; strongly agree = 5.

would be overrepresented during the first tasks, and positive skill-task residuals would be overrepresented during the final tasks. There is a slight tendency toward more negative skill-task residuals during the first three tasks, which may be due to a few negative outliers and no positive outliers. A plausible explanation for the negative outliers is that developers are more likely to make mistakes when they are new to the programming environment.

Still, this effect appears to be small. When comparing these results with simulated data, the effect size of this "warm-up" was estimated to be 0.5 logits, at maximum, which translates to an odds ratio of $e^{0.5} = 1.65$. A standardized effect size is a scale-free estimate that enables relative comparisons of effect size estimates based on different representations (e.g., correlation, mean differences, and odds). By using a formula [38] for converting logits into a standardized effect size combined with software engineering conventions for "small", "medium", and "large" effect sizes [72], the warm-up effect can be classified as a small effect size.

4.5 Psychometric Properties of the Instrument

The internal consistency reliability (Section 2.2) of the instrument was calculated using the person separation index (PSI) [121]. PSI expresses the ratio of the "true" variance to the observed variance and can be calculated even with missing data. PSI was 0.86. Cronbach's α was 0.85 for the subjects that had no missing data for tasks ($n = 61$). Values for α above 0.70 are usually considered as sufficient for use [106].

The targeting of an instrument expresses to what extent there is a good match between the difficulty of the tasks and the skill of the developers who were used to construct the instrument. The targeting can be inspected by comparing the distribution of the task difficulty with that of skill. The mean task difficulty is set at 0 logits by Rumm2020. The standard deviation was 1.12 logits. In contrast, the mean skill of the subjects was -0.83 logits with a standard deviation of 1.30 logits, which is much larger than that found in a study of students (SD = 0.65 logits) [122]. That the mean skill of the subjects is lower than the mean difficulty of the tasks implies that the tasks at present are too difficult for a low-skilled developer. Therefore, the existing tasks of the instrument are at present best suited to measure skill for medium to highly skilled subjects.

5 External Instrument Validation

Section 4 showed that the instrument has desirable *internal* psychometric properties. This section compares and contrasts programming skill, as measured by the instrument, with variables *external* to the instrument.

5.1 Correlations Between Programming Skill and External Variables

Convergent validity is that variables that from theory are meant to assess the same construct, should correlate in practice [29]. Conversely, divergent validity is that variables that, in theory, are not meant to assess the same construct, should not correlate in practice.

TABLE 6
Cross correlations between Java skill and external variables

	(1)	(2)	(3)	(4)	(5)	(6)	(7)	(8)	(9)	(10)	(11)[a]	(12)	(13)	(14)[a]	(15)[a]	(16)
Java skill (1)	—	60	65	65	65	65	27	65	64	65	65	65	65	65	65	65
Java knowledge (2)	**0.64****	—	60	60	60	60	24	60	59	60	60	60	60	60	60	60
Java expertise (3)	**0.46****	0.30**	—	65	65	65	27	65	64	65	65	65	65	65	65	65
Recent Java exp. (4)	**0.37****	0.27*	0.54**	—	65	65	27	65	64	65	65	65	65	65	65	65
Prog. expertise (5)	**0.36****	0.27*	0.57**	0.22*	—	65	27	65	64	65	65	65	65	65	65	65
Java experience (6)	**0.34****	0.26*	0.62**	0.59**	0.39**	—	27	65	64	65	65	65	65	65	65	65
Working memory (7)	**0.34***	0.41*	0.00	-0.13	-0.16	0.03	—	27	27	27	27	27	27	27	27	27
Dev. category (8)	**0.32****	0.22*	0.48**	0.34**	0.39**	0.70**	0.03	—	64	65	65	65	65	65	65	65
Java LOC (9)	**0.29***	0.43**	0.47**	0.38**	0.24*	0.43**	0.44*	0.28*	—	64	64	64	64	64	64	64
Technical skills (10)	**0.28***	0.27*	0.43**	0.32**	0.56**	0.51**	-0.26	0.38**	0.26*	—	65	65	65	65	65	65
People skills[a] (11)	**0.16**	0.27*	0.15	0.14	0.20	0.25*	-0.07	0.14	0.15	0.44**	—	65	65	65	65	65
Prog. experience (12)	**0.15**	0.09	0.24*	0.13	0.43**	0.58**	0.08	0.62**	0.28*	0.36**	-0.04	—	65	65	65	65
Work experience (13)	**0.09**	-0.02	0.20	0.02	0.36**	0.53**	0.19	0.58**	0.19	0.36**	0.05	0.90**	—	65	65	65
Managerial skills[a] (14)	**0.06**	0.24	0.35**	0.33**	0.34**	0.20	-0.28	0.13	0.22	0.58**	0.42**	0.07	0.09	—	65	65
Motivation[a] (15)	**0.05**	0.09	-0.04	0.01	0.03	0.25*	0.03	0.22	0.10	0.15	0.25*	0.21*	0.26*	0.03	—	65
Learned things (16)	**-0.40****	-0.31**	-0.36**	-0.22*	-0.13	-0.24*	-0.28	-0.18	-0.15	-0.32**	-0.23	-0.01	0.00	-0.27*	-0.06	—

Correlations using Spearman's ρ are below the diagonal and the number of subjects (N) for each variable is above the diagonal.
* Significant at $p < 0.05$; ** significant at $p < 0.01$. [a] Using two-tailed tests of significance.

Table 5 shows the descriptive statistics for skill and the external variables that we investigated.[4] Our main concept, programming skill, was operationalized in the instrument using Java as the programming language; the variable is denoted *javaSkill*. The operationalization of the other, external variables is either described throughout this section or is apparent from the questionnaires to be available as supplementary material at http://ieeexplore.ieee.org. The four experience variables and lines of code use ratio scale. *JavaSkill* use interval scale. The remaining variables of the table are all ordinal scale. The operationalization of each variable is either described throughout this section or included as supplementary material to be available at http://ieeexplore.ieee.org.

Table 6 shows the Spearman correlation ρ between *javaSkill* and the external variables, sorted in descending order. For variables where no theory or prior research has established in what direction the correlations with skill should go, we used two-tailed tests of significance. For the other variables, we used one-tailed tests because the variables were investigated in a confirmatory manner.

A commercially available test of Java knowledge (*javaKnowledge*) was purchased from an international test vendor for $ 7,000 and administered to 60 of the 65 developers 1 to 4 months after they solved the programming tasks. From this test, we sampled 30 multiple-choice questions that covered the same domain as the tasks used in the skill instrument. Table 6 shows that *javaKnowledge* was the variable with the highest correlation with *javaSkill*. Because knowledge and skill should be close to unity for developers currently learning how to program, but should diverge as skill evolves through the second and third phase of skill acquisition (Section 2.1), we split the individuals into two groups. For those with *javaSkill* below the mean (see Table 5), *javaKnowledge* and *javaSkill* were highly correlated ($\rho = 0.52$, $p = 0.003$, $n = 30$). For those with *javaSkill* above the mean, there was no correlation ($\rho = 0.002$, $p = 0.990$, $n = 30$). Thus, the relation between programming skill and knowledge was best represented using a cubic trend line, as shown in Fig. 7. Overall, this result is consistent with theoretical expectations and implies that the instrument captures something other than *javaKnowledge* as operationalized by the multiple-choice test, thus demonstrating convergent and divergent validity for the two groups, respectively.

Working memory is a system of the human brain that temporarily stores and manages information. In general, working memory capacity is central to theories of skill acquisition [4], [35]. In particular, working memory has been found to predict technical skill acquisition [82] and programming skill acquisition to a large extent [111]. In our study, three tests of working memory were acquired from Unsworth et al. [123]. In the tests, the developers were required to memorize letters or locations while being distracted by math (Ospan), symmetry (Sspan), or English reading (Rspan) questions [123]. The tests were administered to 29 of the developers using the E-prime software (*workingMemory*). The reason for the low N for this variable is that the software was not available for the first half of the companies visited. Furthermore, the software crashed for two of the developers, which reduced N to 27. Table 6 shows

4. Researchers interested in obtaining the raw data or using the instrument may contact the first author. Agreements adhering to the protocol described in Basili et al. [13] will be required.

that *workingMemory* was significantly and positively correlated with *javaSkill*, as expected. The distribution of *workingMemory* was similar to that of the US student population reported by Unsworth et al. [124].

In this study, experience was represented using four variables. Total Java experience (*javaExperience*) is the amount of time for which an individual has been programming in Java. Recent Java experience (*recentJavaExperience*) is the amount of time for which an individual has been programming in Java consecutively up until present. Both variables correlated significantly with *javaSkill*. Recent practice is not commonly included as a variable in software engineering experiments, but it should nevertheless be central to performance because skills decrease over time when they are not practiced. Table 6 also shows a small correlation of 0.15 between *javaSkill* and general programming experience (*programmingExperience*), which may include exposure to languages other than Java.This is consistent with the correlations between programming experience and performance found in two other large datasets, 0.11 and 0.05, respectively [17]. General work experience (*workExperience*), which may not involve programming, had only 0.09 correlation with *javaSkill*. Consequently, the order of the correlations for these four experience variables with *javaSkill* is consistent with their closeness to Java programming. Because *javaExperience* and *recentJavaExperience* are specializations of *programmingExperience*, which in turn is a specialization of *workExperience*, not obtaining this order of correlations would have indicated validity problems.

Lines of code (LOC) written in Java (*javaLOC*), which is another experience-related variable, was also significantly correlated with *javaSkill* ($\rho = 0.29$). This result is consistent with the correlations between LOC and programming performance found for two other large datasets, 0.29 and 0.34, respectively [17].

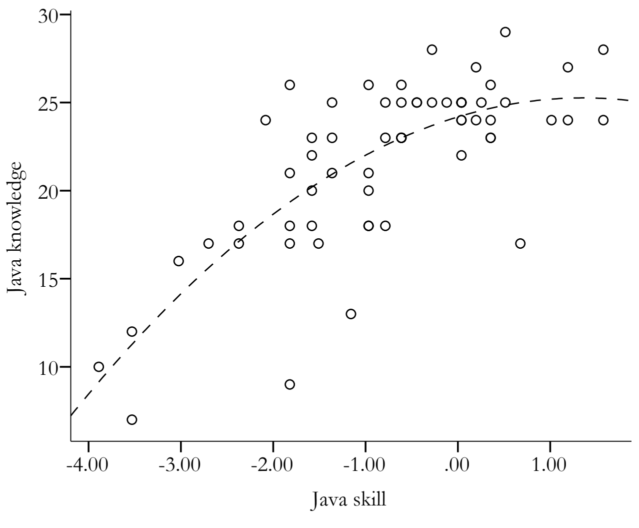

Fig. 7. The relation between Java skill and knowledge

Among the self-reported variables in Table 6, Java expertise (*javaExpertise*), which uses a 5-point Likert scale from "novice" = 1 to "expert" = 5 (see Table 5), had the highest, significant correlation of 0.46 with *javaSkill*. This is similar to the correlation reported in [1] between self estimates and objective measures for math ($r = 0.50$) and spatial abilities ($r = 0.52$). General programming expertise (*programmingExpertise*), which in this context is non-Java specific, was also self-reported and used the same scale as did *javaExpertise*. The correlation between *programmingExpertise* and *javaSkill* was 0.36. This indicates that the two correlations were also well ordered with respect to their closeness to Java programming, similarly as for the four experience variables.

The mean within-company correlation between *javaExpertise* and *javaSkill* was 0.67 (range 0.36–1.00, $n = 9$). This indicates that comparative self-rating of expertise is better than absolute ratings, which is consistent with people's ability to judge in comparative versus absolute terms in general [95].

We also administered a questionnaire, published in [37], for rating the behavior of information technology personnel within the three dimensions of technical skills (*technicalSkills*), people skills (*peopleSkills*), and managerial skills (*managerialSkills*). This questionnaire has previously been used by managers to rate employees, but we adapted the questions to be applicable in ratings of self. Table 6 shows that only *technicalSkills* was significantly correlated with *javaSkill*.

An individual's motivation to spend as much effort and energy during the study was self-reported using a 10-point Likert scale (*motivation*). Table 6 shows an insignificant, low correlation between *motivation* and *javaSkill* (0.05). A strong positive correlation would have been detrimental to validity because this would have indicated that motivation is confounded with the measure of skill. Nevertheless, those with high skill are still more adversely affected by low motivation [75] because an individual with high skill and low motivation would be measured at low skill (i.e., a large difference), whereas an individual with low skill and low motivation would still be measured at low skill (i.e., a small difference). Therefore, motivation continues to be a confounding factor in *javaSkill*, although this limitation is not unique to us because most empirical research is based on the assumption of cooperative subjects.

Finally, the subjects were asked about the extent to which they learned new things while solving the 19 tasks (*learnedNewThings*). Table 6 shows a statistically significant negative correlation between *learnedNewThings* and *javaSkill*. This demonstrates divergent validity, because people with high skill will likely not learn new things when carrying out well-practiced behavior.

5.2 Predicting Programming Performance

Predictive validity is traditionally regarded as an integral part of instrument validation [95]. We investigated how well the instrument predicted programming performance on a set of tasks compared with alternative predictors, such as the external variables reported in the previous section. To reduce bias in the comparison, the tasks of that being predicted must be independent from the instrument. For convenience, we used the performance data from four of the seven tasks that were removed from the instrument (Tasks 14–17 in Table 7). The tasks were selected because they were easy to score with respect to correctness and the subjects' solutions varied in both quality and in time (a variable that contain no variance cannot be predicted). The remaining three tasks either had little variance to be predicted (Task 13 "Hello World") or would have required scoring rules to be available (Tasks 18 and 19 both used subtasks with quality attributes that varied in multiple dimensions).

One may question why tasks that were previously excluded from the instrument can be used in the validation process. As we have described, there are strict requirements for a task to be included in an instrument for measuring programming skill. Prediction, on the other hand, only requires that solving the task should involve some degree of programming skill.

Fig. 8 shows the correlation between the investigated predictors and task performance with respect to correctness and time on the four tasks, yielding a total of eight correlations (circles) for each predictor. Correctness was analyzed using point-biserial correlation and time for correct solutions was analyzed using Spearman's ρ. The vertical lines divide between small (S), medium (M), and large (L) correlations according to the guidelines stated in [73]. The trend line shows the mean of the correlations for each predictor and confirms that instrument (i.e., *javaSkill*) was the best predictor, ahead of *javaKnowledge*.

Fig. 8 also shows that the correlation between task performance and the four experience variables was small. A similar result was also found in an early study of programming productivity across three organizations [71]. That study found no association between performance and experience for two of the organizations, which employed developers with one to nine years of programming experience. However, in the third organization, which consisted of developers with only one to three years of experience, performance and experience were related. Based on these findings the authors conjectured that either developers "learn their craft in a year and from thereon additional experience makes little difference [or] large individual differences in programming skill [exist] but these are not related to number of years

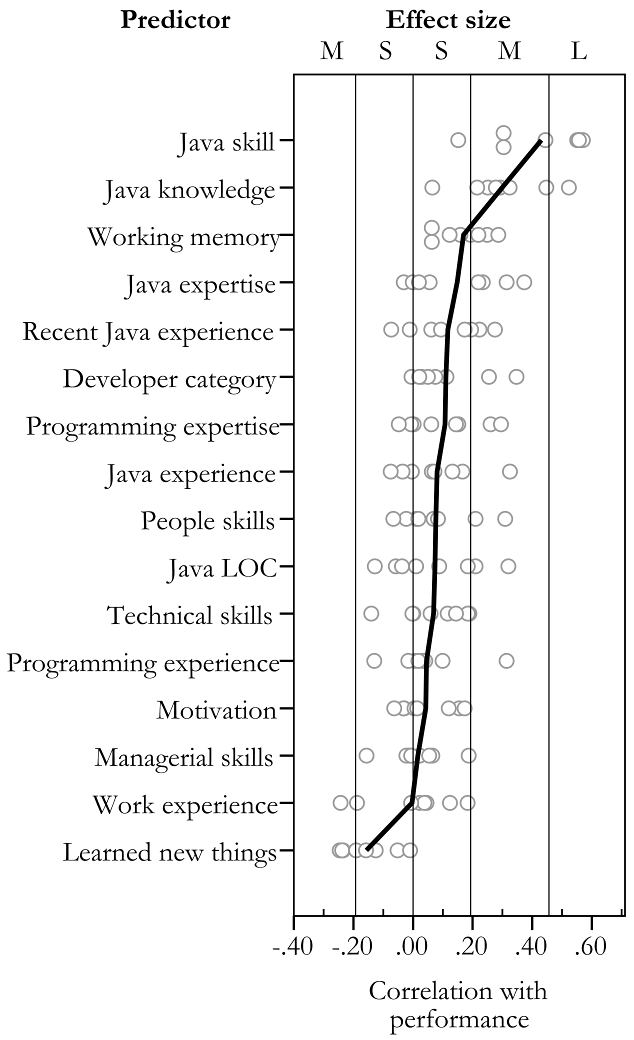

Fig. 8. Java skill and alternative predictors of task performance

of experience" [71, p. 376]. We found a similar result (not shown): The correlation between experience and skill was largest during the first year of experience, but then gradually diminished and disappeared completely after about four years. That these two variables display an increasing but deaccelerating relationship is expected from the log-log law of practice [94], as well as research on expertise [54].

6 Discussion

This section discusses the answer to the research question, contributions to research, implications for practice, limitations, and future work.

6.1 Measuring Programming Skill

Our research question was "to what extent is it possible to construct a valid instrument for measuring programming skill?" We now discuss the validity of the instrument according to the aspects of Table 1.

Task content regards the extent to which the 12 tasks of the final instrument adequately represent the scope we defined in Section 3.1. Only a few of the tasks required the developer to optimize software quality aspects other than functional correctness. For example, many of the quality aspects in ISO 2196/25010 are underrepresented. We focused on functional correctness because it is a prerequisite for the other quality aspects. For example, it is difficult to evaluate the efficiency of two task solutions if they are not functionally equivalent.

Nevertheless, the tasks combined are more comprehensive than in most experiments on programmers. Both sample size and study duration are large compared with experiments in software engineering in general [114]. Compared with [122] and [128], who also used the Rasch model to study "programming ability", our tasks are also more realistic—but also time consuming—in the sense that developers must submit code as their solution. Furthermore, our tasks were structured around a programming problem that may involve many programming concepts simultaneously, whereas [122] and [128] focused on "narrow" problems, where one programming concept is evaluated per question. Thus, answering the question of whether the tasks as a whole span the dimension that one is trying to measure is difficult. One may argue that adding yet another task (ad infinitum) would better span the dimension one is aiming to measure. There is no stop criterion; the choice of when to stop is subjective. The universe of potential programming tasks is infinite [49].

Response process concerns whether the mental processes involved when solving the tasks are representative of programming skill. The processes involved during software development are clearly more demanding than selecting (or guessing) the correct answer to a short programming problem in multiple-choice questions, such as in [122]. The open response format (e.g., used in [128]) alleviates this problem, but we regard questions such as "What kind of data structure can be stored with this definition?" as akin to assessing programming *knowledge*. In contrast, many of the tasks were selected or constructed to capture a range of aspects of industrial programming tasks. For example, the tasks were solved in the developers' regular programming environment, and many of the tasks contained code that was too extensive to understand in full. This increased the likelihood that the developers used response processes similar to those that they use in their daily work.

The *internal structure* of the data concerns the dimensionality and reliability of the measure of programming skill (Section 4). Fundamental to statements such as "developer A is *more/less* skilled than developer B" is the assumption that one dimension exists along which one can be more or less skilled. Although programming has many facets, we found that programming skill could be represented as a unidimensional, interval-scale variable for the majority of the programming tasks we investigated. That performance on different programming problems may essentially be regarded as a single dimension was also found in a study of students in a C++ course [65]. This indicates that programming skill accounts for the major proportion of the large differences observed in programming performance (i.e., the "common variance" in Fig. 2) reported elsewhere [41], [45], [66], [100]. However, there may be other explanations. Therefore, we investigated other potential sources of construct-irrelevant variance, but found only a slight warm-up and task-difficulty effect. Furthermore, the ratio of random error variance to common variance plus systematic error variance (i.e., internal consistency reliability) was found to be satisfactory. Compared with [122], who used factor analysis to investigate unidimensionality, the eigenvalue of our first factor was larger (4.76) than the eigenvalue of their first factor (2.81). This implies a greater proportion of common variance to error variance in our study.

Programming skill, as measured by the instrument, *correlated with external variables* in accordance with theoretical expectations. More specifically, as shown in Section 5.1, programming skill and programming knowledge appeared to be strongly related for low to medium skill levels, whereas they were unrelated for medium to high skill levels. We also found that experience and expertise variables were both well ordered with respect to their closeness to Java programming. Convergent validity was found for variables such as developer category, lines of code, and technical skills, where divergent validity was present for managerial and people skills, as well as motivation. Moreover, as we have previously reported [16], we found that five of the variables in Table 5 display a pattern in the correlations that is consistent with Cattell's investment theory, see [34]. This psychological theory describes how the effect of intelligence (in our context, working memory) and experience on skill is mediated through knowledge. Previous work by Anderson [5] showed that the best predictor of programming errors on tasks was the amount of error on other programming problems. Similarly, we showed in Section 5.2 that performance on a set of programming tasks was best predicted by performance on another set of programming tasks, that is, the instrument.

The APA [2] also regards *validity generalization* as related to "correlations with other variables." From an analytical perspective, the generalizability of the instrument is based on its connection to theory about *essential features* [85], in which the concept of transfer [60] is central when generalizing between instrument and industry tasks. For example, Anderson et al. used a software-based tutor that trained students in 500 productions (i.e., "if-then" rules) that comprise the components of programing skill in LISP. They found a "transfer from other programming experience to the extent that this programming experience involves the same productions" [6, p. 467]. Thus, when a programming language such as C# is semantically similar to Java on many accounts, one would expect that skill in either language would transfer to a large extent to the other language. We believe that the principle of transfer also informs the generalizability of tasks of the instrument, because these tasks involve many concepts central to programming that are also present in real-world tasks.

Concerning the generalizability across populations, one would ideally randomly sample from the world's population of professional developers. In practice, this is impossible. However, we managed to sample professional developers from multiple countries and companies. The extent to which the results generalize to *other* populations of professionals (e.g., different countries or types of companies) is an empirical question that must be addressed in follow-up studies.

Overall, our investigation of validity indicates that our instrument is a valid measure of programming skill, even though a single study cannot answer this conclusively. This inability to make conclusions is similar to the challenge of confirming a theory. A theory cannot be proved. Instead, it is only strengthened by its ability to escape genuine attempts at falsification [99].

6.2 Contributions to Research

Theory-driven investigations are rare in empirical software engineering [69], even though theory is often required to interpret and test results [99], [108]. In [113], we described how theories can enter software engineering: unmodified, adapted, or built from scratch. We applied an unmodified version of the theory of skill and interpreted and tested expectations from this theory "as is", using professional software developers (most other researchers use students). We also applied the Rasch model, which can be regarded as a non-substantive theory of how item difficulty and person abilities interact, without modification. However, to use programming performance data as input to the Rasch model, we adapted the scoring principles described in [68] to account for the time-quality problems when scoring performance on programming tasks [17].

Scoring rules are rarely justified or evaluated. In [46], we justified, but did not evaluate, the use of a five-point Likert scale for each indicator of key factors of success in software process improvement. In contrast, through the use of the Rasch model, we have shown in this article how to evaluate the number of score points and the scoring rule for each indicator.

We demonstrated methods for internal validation through tests of overfitting, unidimensionality, and person and task fit to the measurement model. For example, we investigated whether practice effects were a confounding factor [110] by analyzing residual variance. Moreover, we demonstrated that by requiring that residual variance be uncorrelated, the testability of the proposed model is enhanced. In [46], we used PCA to identify the factor structure of multiple scales, but we did not investigate whether residual variance between indicators for each factor was uncorrelated.

In [48], we showed that the meaning of a construct can easily change as a result of variations in operationalizations. In the present study, we extended this work to include *empirical testing* of whether operationalizations internally are mutually consistent and derivable expectations from theory are met. By using a convergent-divergent perspective [29], we showed that the closer the variables were to programming skill, the higher was the correlation.

The validity of empirical studies in software engineering may be improved by using the instrument to select subjects for a study, assign subjects to treatments, and analyze the results. When selecting subjects for a study, one should take into account that the usefulness of a technology may depend on the skill of the user [18]. For example, representativity is a problem when students are used in studies for which one wishes to generalize the results to (a category of) professional developers [112]. The instrument can be used to select a sample with certain skill levels. For example, the instrument was used to select developers with medium to high programming skills in a multiple-case study [115].

When assigning subjects to treatments, a challenge is to ensure that the treatment groups are equal or similar with respect to skill. A threat to internal validity is present when skill level is confounded with the effect of the treatments. In experiments with a large sample size, one typically uses random allocation to achieve similar skill groups. However, in software engineering experiments, the average sample size of subjects is 30 [114] and the variability is usually large. Even in an experiment with random allocation of 65 subjects, we found an effect (although small) in the difference in skill [18]. By using the instrument for assigning subjects to equally skilled pairs (instead of groups), more statistically powerful tests can be used, which in turn reduces threats to statistical conclusion validity [47], [108].

Quasi-experiments are experiments without random allocation of subjects to treatments. Randomization is not always desirable or possible; for example, "the costs of teaching professionals all the treatment conditions (different technologies) so that they can apply them in a meaningful way may be prohibitive" [73, p. 72]. To adjust for possible differences in skill level between treatments groups, and thus to reduce threats to internal validity, a measure of skill provided by the instrument may be used as a covariate in the *analysis of the results.*

Similar to controlling for the level of skill, the instrument may also be used to control for task difficulty in software engineering experiments. Task difficulty may both be a confounding factor and a factor across which it may be difficult to generalize the results. For example, in an experiment on pair programming with junior, intermediate and senior Java consultants [10], pair programming was beneficial for the intermediate consultants on the difficult tasks. On the easy tasks, there was no positive effect.

6.3 Implications for Practice

According to [32], job performance consists of eight major components. One of them concerns *job-specific task proficiency*, which is "the degree to which the individual can perform the core substantive or technical tasks that are central to the job" [32, p. 46]. In a meta-analysis with over 32,000 employees [104], work sample tests had the highest correlation with job performance (0.54), followed by tests of intelligence (0.51), job knowledge (0.48), and job experience (0.18). A benefit of work sample tests is that they possess a high degree of realism and thus appear more valid to the individual taking the test, see generally [25]. However, they are more costly to develop and score and more time-consuming to administer [74]. Like a work sample test, our instrument uses actual performance on tasks as the basis for inferring job-specific task proficiency in the area of programming and, consequently, would be useful for recruiting or project allocation.

Work samples and our instrument may complement each other. Work sample tests may include programming tasks that are tailored for a highly specific job. The result an individual receives on a work sample test may be a composite of many factors, such as domain-specific or system-specific knowledge. In contrast to most work-sample tests, as well as other practical programming tests used in-house in a recruiting situation, our instrument aims to provide a measure of programming skill based on a scientific definition of measurement, that is, the claim that "something is measured" can be falsified. Furthermore, the construction of the instrument is theory-driven and the validation has been performed according to the aspects as reported above.

Many other criteria than correlations are involved when comparing alternative predictors of job-specific task proficiency. For example, work sample tests may require relevant work experience to be applicable in a concrete setting [104]. Time is also an important factor: Grades from education or work experience can be inspected within minutes, standardized tests of intelligence or programming knowledge may be administered within an hour, and the use of standardized work samples, or our instrument, may require a day. For example, exploratory work on a model for assessing programming experience based on a questionnaire that can be quickly administered is outlined in [57].

If we had had only one hour available, time would allow the use of a couple of tasks that fit the model well and (combined) have a good span in task difficulty. We chose Tasks 9 and 12 in Table 7 to be used as a one-hour version of the instrument. Although the measurement precision of the instrument is greatly reduced by having only 2 tasks to measure skill instead of 12, the validity of the instrument should be unaffected because all the tasks still measure "the same." When calculating programming skill based solely on those two tasks, the instrument was still as good as the knowledge test (which took approximately one hour to complete) in predicting programming performance (cf. Fig. 8). Consequently, the instrument requires more time to predict programming performance better than the alternatives. Therefore, future work includes ways to retain the reliability and validity of the instrument while reducing the time needed to administer it.

As determined by the scope we defined, the instrument's measure of programming skill is independent of knowledge of a specific application domain, software technology, and the concrete implementation of a system. A developer with extensive knowledge in any of these areas may perform better on a new task within any of these areas than a developer with higher programming skill but with less knowledge in these areas. Creating a tailored version of the instrument that combines programming skill with specific knowledge within one or more of these areas would require access to experts within each field that must assist in the construction of new tasks for the instrument. A pragmatic alternative to creating such a tailored instrument, which must follow the steps outlined in this paper, is to use our instrument for measuring programming skill and combine it with knowledge tests for a given domain, technology or implementation.

Furthermore, the instrument appears to be best for testing medium to highly skilled developers. To make the instrument more suitable for less skilled developers, one would need easier tasks. However, it is a challenge to create an easy task that at the same time resembles an industrial problem. In an industrial system, even an apparently easy change of code may have unintended consequences. Thus, making a small change may require an understanding of a wider part of the system, which in turn makes the task more difficult to solve than originally indented.

The description of the tasks of the present instrument is language independent. The program code for each task is written in Java but can easily be translated into other object-oriented languages. Tailoring the instrument to non-object-oriented languages would be more challenging, because what is considered a high-quality solution might differ between language paradigms. Concerning the test infrastructure, automatic test cases would generally be easy to translate into new programming languages, even though it would be easier to modify the instrument to support languages that can use the Java virtual machine. Note that any major changes to the instrument due to tailoring will require a new sample of developers to be used to calibrate new task difficulty parameters. We also recommend that difficulty parameters are verified even though only *minor* changes to the instrument are present, for example, if the tasks are translated into another object-oriented language.

6.4 Limitations

The sample size of 65 subjects in this study is low. An ideal sample size for the polytomous Rasch model is around 250 subjects [84], even though authoritative work on Rasch modelling has previously been conducted on a sample size similar to ours (see [130]). An increased sample size would have resulted in lower standard errors of measurement in the estimated skill and difficulty parameters (the parameters are shown in Figs. 4 and 5). Increased measurement precision due to larger sample size would have enabled the detection of more cases of statistically significant differences in skill level between developers.

Four of the twelve tasks in the final instrument required manual evaluation of quality, which was performed only by the first author. To reduce the likelihood of bias, we used non-subjective scoring rubrics (see Section 3.3). Still, multiple raters would have increased confidence in results.

In the validation process, the three debugging tasks were excluded because they contributed negatively to unidimensionality, even though the contribution was small. We do not know whether the negative contribution to unidimensionality is because debugging represents something slightly different than "programming", as we defined it, or because these three tasks were atypical. For example, all the tasks were small, had short time limits, and represented an "insight problem" [107]; that is, one struggles for some time until one obtains the insight needed to solve the problem. In practice, however, there are virtually no differences: The correlation between programming skill as measured by the instrument with the debugging tasks present (15 tasks) and programming skill as measured by the instrument without the debugging tasks (12 tasks) present was $r = 0.995$.

Finally, we do not know to what extent the response processes used when solving the tasks of the instrument were representative of real-world programming. This limitation could have been addressed by comparing think-aloud protocols [2] from industry programming tasks with our instrument tasks. However, we have previously found that such methods are intrusive [76] and therefore would have been a serious threat to the internal validity of the instrument if used during instrument construction.

6.5 Future Work

In addition to addressing the limitations just described, this work yields several directions for future work. One topic is how much developers differ in their programming performance [41], [45], [66], [100]. The standard deviation of skill in our sample of developers was 1.30 logits. To illustrate this variability, if two developers are drawn from this sample at random, one of the developers would display better programming performance than the other one in almost 4 out of 5 programming tasks on average.[5] The instrument is used at present in an industrial context, which gives us an opportunity for studying variability in programming skill across various populations of subjects and tasks.

We would also like to increase the understanding of the conditions that are required to achieve high skill levels. For example, to what extent is experience important to achieve a high skill level? In our sample, skill and experience covaried for the first four years of experience. Additional experience was not associated with higher skill level on average. However, the variability in skill level increased for those with extensive experience. A deeper analysis of these data is needed. In particular, we would like to contrast our data with the 10,000 hours of deliberate practice required to reach the highest levels of expert performance, as stated in [54].

The use of the instrument in research and industry will make the tasks known to a wider audience over time, which, in turn, will reduce the usefulness of the instrument. Therefore, it is important that new tasks are continuously being developed and calibrated to be included in the instrument. Thus, in the future, new tasks will be used to measure skill the same way as do the 12 existing tasks today.

To make the instrument more attractive for industrial use, we aim to reduce the time needed to measure skill while retaining precision. A benefit of the Rasch model is that it facilitates computer adaptive testing, which means that the difficulty of the next task given to the subject depends on the score of the previous task. This procedure maximizes measurement precision, thereby reducing the number of tasks required.

The use of our instrument in an industrial setting also gives us an opportunity for investigating how measures of programming skill complement experience, education, peer-ratings, and other indicators as predictors of job performance.

7 Conclusion

We constructed an instrument that measures skill by using performance on a set of programming

5. Our observed variability in skill, 1.3 logits, equals an odds ratio of $e^{1.3}$ = 3.7; that is, the more skilled developer of the pair would perform better with odds of 3.7:1, which is 3.7 out of 4.7 tasks.

tasks. From a theoretical perspective, the combination of theory-driven research and a strict definition of measurement enabled rigorous empirical testing of the validity of the instrument. From a practical perspective, the instrument is useful for identifying professional programmers who have the capacity to develop systems of high quality in a short time. This instrument for measuring programming skill is already being used as the basis for new prototypes and for further data collection, in collaboration with industry.

Appendix A

See Table 7.

Appendix B

See Table 8.

Acknowledgments

This work was supported by Simula Research Laboratory and the Research Council of Norway through grants no. 182567, 193236/I40, and 217594/O30. We thank the anonymous referees for valuable comments, Erik Arisholm and Jo Hannay for useful discussions and feedback on an earlier version of this article, Steinar Haugen, Gunnar Carelius, Arne Salicath, and Linda Sørlie for technical assistance, Steinar Haugen, Sindre Mehus, Arne Salicath, Aleksey Udovydchenko, and Alexander Ottesen for developing new tasks, Magdalena Ivanovska for assistance with mathematical notation, Lenore Hietkamp for copyediting, and the companies and developers who participated in the study.

References

[1] P. L. Ackerman and S. D. Wolman, “Determinants and validity of self-estimates of ability and self-concept measures,” *Journal of Experimental Psychology: Applied*, vol. 13, no. 2, pp. 57–78, 2007.

[2] American Educational Research Association and American Psychological Association and National Council on Measurement in Education and Joint Committee on Standards for Educational and Psychological Testing, *Standards for educational and psychological testing*. Washington, DC: American Educational Research Association, 1999.

[3] B. C. D. Anda, D. I. K. Sjøberg, and A. Mockus, “Variability and reproducibility in software engineering: A study of four companies that developed the same system,” *IEEE Transactions on Software Engineering*, vol. 35, no. 3, pp. 407–429, 2009.

[4] J. R. Anderson, “Acquisition of cognitive skill,” *Psychological Review*, vol. 89, no. 4, pp. 369–406, 1982.

[5] ——, “Skill acquisition: Compilation of weak-method problem solutions,” *Psychological Review*, vol. 94, no. 2, pp. 192–210, 1987.

[6] J. R. Anderson, F. G. Conrad, and A. T. Corbett, “Skill acquisition and the LISP Tutor,” *Cognitive Science*, vol. 13, no. 4, pp. 467–505, 1989.

[7] D. Andrich, “A rating formulation for ordered response categories,” *Psychometrika*, vol. 43, no. 4, pp. 561–573, 1978.

[8] ——, “Understanding the response structure and process in the polytomous Rasch model,” in *Handbook of polytomous item response theory models: Developments and applications*, M. L. Nering and R. Ostini, Eds. New York: Routledge, 2010, pp. 123–152.

[9] D. Andrich, B. Sheridan, and G. Luo, *RUMM2020 [computer software]*, RUMM Laboratory, Perth, 2006.

[10] E. Arisholm, H. Gallis, T. Dybå, and D. I. K. Sjøberg, “Evaluating pair programming with respect to system complexity and programmer expertise,” *IEEE Transactions on Software Engineering*, vol. 33, no. 2, pp. 65–86, 2007.

[11] E. Arisholm and D. I. K. Sjøberg, “Evaluating the effect of a delegated versus centralized control style on the maintainability of object-oriented software,” *IEEE Transactions on Software Engineering*, vol. 30, no. 8, pp. 521–534, 2004.

[12] E. Arisholm, D. I. K. Sjøberg, G. J. Carelius, and Y. Lindsjørn, “A web-based support environment for software engineering experiments,” *Nordic Journal of Computing*, vol. 9, no. 3, pp. 231–247, 2002.

[13] V. R. Basili, M. V. Zelkowitz, D. I. K. Sjøberg, P. Johnson, and A. J. Cowling, “Protocols in the use of empirical software engineering artifacts,” *Empirical Software Engineering*, vol. 12, no. 1, pp. 107–119, 2007.

[14] S. Beecham, N. Baddoo, T. Hall, H. Robinson, and H. Sharp, “Motivation in software engineering: A systematic literature review,” *Information and Software Technology*, vol. 50, no. 9–10, pp. 860–878, 2008.

[15] A. C. Benander, B. A. Benander, and J. Sang, “An empirical analysis of debugging performance—differences between iterative and recursive constructs,” *Journal of Systems and Software*, vol. 54, no. 1, pp. 17–28, 2000.

[16] G. R. Bergersen and J.-E. Gustafsson, “Programming skill, knowledge and working memory capacity among professional software developers from an investment theory perspective,” *Journal of Individual Differences*, vol. 32, no. 4, pp. 201–209, 2011.

[17] G. R. Bergersen, J. E. Hannay, D. I. K. Sjøberg, T. Dybå, and A. Karahasanović, “Inferring skill from tests of programming performance: Combining time and quality,” in *Proceedings of the International Symposium on Empirical Software Engineering and Measurement*. IEEE, 2011, pp. 305–314.

[18] G. R. Bergersen and D. I. K. Sjøberg, “Evaluating methods and technologies in software engineering with respect to developer’s skill level,” in *Proceedings of the International Conference on Evaluation & Assessment in Software Engineering*. IET, 2012, pp. 101–110.

[19] J. Bloch, *Effective Java programming language guide*. Mountain View, CA: Sun Microsystems, 2001.

[20] T. G. Bond and C. M. Fox, *Applying the Rasch model: Fundamental measurement in the human sciences*. Mahwah, NJ: Erlbaum, 2001.

[21] D. Borsboom, *Measuring the mind: Conceptual issues in contemporary psychometrics*. New York: Cambridge University Press, 2005.

[22] D. Borsboom, G. J. Mellenbergh, and J. van Heerden, “The theoretical status of latent variables,” *Psychological Review*, vol. 110, no. 2, pp. 203–219, 2003.

[23] ——, “The concept of validity,” *Psychological Review*, vol. 111, no. 4, pp. 1061–1071, 2004.

[24] D. Borsboom and A. Z. Scholten, “The Rasch model and conjoint measurement theory from the perspective of psychometrics,” *Theory & Psychology*, vol. 18, no. 1, pp. 111–117, 2008.

[25] H. I. Braun, R. E. Bennett, D. Frye, and E. Soloway, “Scoring constructed responses using expert systems,” *Journal of Educational Measurement*, vol. 27, no. 2, pp. 93–108, 1990.

[26] L. Briand, K. El Emam, and S. Morasca, “On the application of measurement theory in software engineering,” *Empirical Software Engineering*, vol. 1, no. 1, pp. 61–88, 1996.

[27] F. P. Brooks, “No silver bullet: Essence and accidents of software engineering,” *IEEE Computer*, vol. 20, no. 4, pp. 10–19, 1987.

[28] D. T. Campbell, “A phenomenology of the other one: Corrigible, hypothetical, and critial,” in *Human action: Conceptual and empirical issues*, T. Mischel, Ed. New York: Academic Press, 1969, pp. 41–69.

[29] D. T. Campbell and D. W. Fiske, "Convergent and discriminant validity by the multitrait-multimethod matrix," *Psychological Bulletin*, vol. 56, no. 2, pp. 81–105, 1959.

[30] J. P. Campbell, "Modeling the performance prediction problem in industrial and organizational psychology," in *Handbook of industrial and organizational psychology*, 2nd ed., M. D. Dunnette and L. M. Hough, Eds. Palo Alto, CA: Consulting Psychologists Press, 1990, vol. 1, pp. 687–732.

[31] J. P. Campbell, M. B. Gasser, and F. L. Oswald, "The substantive nature of job performance variability," in *Individual differences and behavior in organizations*, K. R. Murphy, Ed. San Francisco, CA: Jossey-Bass, 1996, pp. 258–299.

[32] J. P. Campbell, R. A. McCloy, S. H. Oppler, and C. E. Sager, "A theory of performance," in *Personnel selection in organizations*, N. Schmitt and W. C. Borman, Eds. San Francisco, CA: Jossey-Bass, 1993, pp. 35–70.

[33] J. B. Carroll, *Human cognitive abilities: A survey of factor-analytic studies*. Cambridge: Cambridge University Press, 1993.

[34] R. B. Cattell, *Abilities: Their structure, growth, and action*. Boston, MD: Houghton-Mifflin, 1971/1987.

[35] W. G. Chase and K. A. Ericsson, "Skill and working memory," *The Psychology of Learning and Motivation*, vol. 16, pp. 1–58, 1982.

[36] C. Chatfield, "Model uncertainty, data mining and statistical inference," *Journal of the Royal Statistical Society, Series A*, vol. 158, no. 3, pp. 419–466, 1995.

[37] M. A. Chilton and B. C. Hardgrave, "Assessing information technology personnel: Towards a behavioral rating scale," *DATA BASE for Advances in Information Systems*, vol. 35, no. 3, pp. 88–104, 2004.

[38] S. Chinn, "A simple method for converting an odds ratio to effect size for use in meta-analysis," *Statistics in Medicine*, vol. 19, no. 22, pp. 3127–3131, 2000.

[39] A. Cockburn, "The Coffee Machine design problem: Part 1 & 2," *C/C++ Users Journal*, May/June 1998.

[40] B. P. Cohen, *Developing sociological knowledge: Theory and method*, 2nd ed. Chicago: Nelson-Hall, 1989.

[41] B. Curtis, "Measurement and experimentation in software engineering," *Proceedings of the IEEE*, vol. 68, no. 9, pp. 1144–1157, 1980.

[42] ——, "Fifteen years of psychology in software engineering: Individual differences and cognitive science," in *Proceedings of the 7th International Conference on Software Engineering*. IEEE, 1984, pp. 97–106.

[43] F. A. Dahl, M. Grotle, J. Š. Benth, and B. Natvig, "Data splitting as a countermeasure against hypothesis fishing: With a case study of predictors for low back pain," *European Journal of Epidemiology*, vol. 23, no. 4, pp. 237–242, 2008.

[44] S. Dekleva and D. Drehmer, "Measuring software engineering evolution: A Rasch calibration," *Information Systems Research*, vol. 8, no. 1, pp. 95–104, 1997.

[45] T. DeMarco and T. Lister, *Peopleware: Productive projects and teams*, 2nd ed. New York: Dorset House Publishing Company, 1999.

[46] T. Dybå, "An instrument for measuring the key factors of success in software process improvement," *Empirical Software Engineering*, vol. 5, no. 4, pp. 357–390, 2000.

[47] T. Dybå, V. B. Kampenes, and D. I. K. Sjøberg, "A systematic review of statistical power in software engineering experiments," *Information and Software Technology*, vol. 48, no. 8, pp. 745–755, 2006.

[48] T. Dybå, N. B. Moe, and E. Arisholm, "Measuring software methodology usage: Challenges of conceptualization and operationalization," in *Proceedings of the International Symposium on Empirical Software Engineering*. IEEE, 2005, pp. 447–457.

[49] T. Dybå, D. I. K. Sjøberg, and D. S. Cruzes, "What works for whom, where, when, and why? On the role of context in empirical software engineering," in *Proceedings of the International Symposium on Empirical Software Engineering and Measurement*. ACM-IEEE, 2012, pp. 19–28.

[50] J. R. Edwards and R. P. Bagozzi, "On the nature and direction of relationships between constructs and measures," *Psychological Methods*, vol. 5, no. 2, pp. 155–174, 2000.

[51] S. E. Embretson, "The new rules of measurement," *Psychological Assessment*, vol. 8, no. 4, pp. 341–349, 1996.

[52] K. A. Ericsson, "The acquisition of expert performance as problem solving: Construction and modification of mediating mechanisms through deliberate practice," in *The psychology of problem solving*, J. E. Davidson and R. J. Sternberg, Eds. Cambridge: Cambridge University Press, 2003, pp. 31–83.

[53] K. A. Ericsson, N. Charness, P. J. Feltovich, and R. R. Hoffman, Eds., *The Cambridge handbook of expertise and expert performance*. Cambridge: Cambridge University Press, 2006.

[54] K. A. Ericsson, R. T. Krampe, and C. Tesch-Römer, "The role of deliberate practice in the acquisition of expert performance," *Psychological Review*, vol. 100, no. 3, pp. 363–406, 1993.

[55] K. A. Ericsson and J. Smith, "Prospects and limits of the empirical study of expertise: An introduction," in *Towards and general theory of expertise*, K. A. Ericsson and J. Smith, Eds. New York: Cambridge University Press, 1991, pp. 1–38.

[56] H.-E. Eriksson and M. Penker, *UML toolkit*. New York: John Wiley & Sons, 1997.

[57] J. Feigenspan, C. Kastner, J. Liebig, , S. Apel, and S. Hanenberg, "Measuring programming experience," in *IEEE 20th International Conference on Program Comprehension*. IEEE, 2012, pp. 73–82.

[58] N. Fenton, "Software measurement: A necessary scientific basis," *IEEE Transactions on Software Engineering*, vol. 20, no. 3, pp. 199–206, 1994.

[59] N. Fenton and B. Kitchenham, "Validating software measures," *Journal of Software Testing, Verification, and Reliability*, vol. 1, no. 2, pp. 27–42, 1991.

[60] G. A. Ferguson, "On transfer and the abilities of man," *Canadian Journal of Psychology*, vol. 10, no. 3, pp. 121–131, 1956.

[61] R. P. Feynman, *The meaning of it all: Thoughts of a citizen-scientist*. New York: Basic Books, 1998.

[62] P. M. Fitts and M. I. Posner, *Human performance*. Belmont, CA: Brooks/Cole, 1967.

[63] A. E. Fleury, "Programming in Java: Student-constructed rules," *ACM SIGCSE Bulletin*, vol. 32, no. 1, pp. 197–201, 2000.

[64] R. W. Floyd, "The paradigms of programming," *Communications of the ACM*, vol. 22, no. 8, pp. 455–460, 1979.

[65] R. Freedman, "Relationships between categories of test items in a C++ CS1 course," *Journal of Computing Sciences in Colleges*, vol. 29, no. 2, pp. 26–32, 2013.

[66] E. E. Grant and H. Sackman, "An exploratory investigation of programmer performance under on-line and off-line conditions," *IEEE Transactions on Human Factors in Electronics*, vol. HFE-8, no. 1, pp. 33–48, 1967.

[67] I. Hacking, *The taming of chance*. Cambridge: Cambridge University Press, 1990.

[68] B. Hands, B. Sheridan, and D. Larkin, "Creating performance categories from continuous motor skill data using a Rasch measurement model," *Journal of Outcome Measurement*, vol. 3, no. 3, pp. 216–232, 1999.

[69] J. E. Hannay, D. I. K. Sjøberg, and T. Dybå, "A systematic review of theory use in software engineering experiments," *IEEE Transactions on Software Engineering*, vol. 33, no. 2, pp. 87–107, 2007.

[70] S. M. Humphry and D. Andrich, "Understanding the unit in the Rasch model," *Journal of Applied Measurement*, vol. 9, no. 3, pp. 249–264, 2008.

[71] D. R. Jeffery and M. J. Lawrence, "An inter-organisational comparison of programming productivity," in *Proceedings of the 4th International Conference on Software Engineering*. IEEE Press, 1979, pp. 369–377.

[72] V. B. Kampenes, T. Dybå, J. E. Hannay, and D. I. K. Sjøberg, "A systematic review of effect size in software engineering experiments," *Information and Software Technology*, vol. 49, no. 11, pp. 1073–1086, 2007.

[73] ——, "A systematic review of quasi-experiments in software engineering," *Information and Software Technology*, vol. 51, no. 1, pp. 71–82, 2009.

[74] M. Kane, T. Crooks, and A. Cohen, "Validating measures of performance," *Educational Measurement: Issues and Practice*, vol. 18, no. 2, pp. 5–17, 1999.

[75] R. Kanfer and P. L. Ackerman, "Motivation and cognitive abilities: An integrative/aptitude-treatment interaction approach to skill acquisition," *Journal of Applied Psychology*, vol. 74, no. 4, pp. 657–690, 1989.

[76] A. Karahasanović, U. N. Hinkel, D. I. K. Sjøberg, and R. Thomas, "Comparing of feedback-collection and think-aloud methods in program comprehension studies," *Behaviour & Information Technology*, vol. 28, no. 2, pp. 139–164, 2009.

[77] A. Karahasanović, A. K. Levine, and R. C. Thomas, "Comprehension strategies and difficulties in maintaining object-oriented systems: An explorative study," *Journal of Systems and Software*, vol. 80, no. 9, pp. 1541–1559, 2007.

[78] A. Karahasanović and R. C. Thomas, "Difficulties experienced by students in maintaining object-oriented systems: An empirical study," in *Proceedings of the Australasian Computing Education Conference*. Australian Computer Society, 2007, pp. 81–87.

[79] D. H. Krantz, R. D. Luce, P. Suppes, and A. Tversky, *Foundations of measurement*. New York: Academic Press, 1971, vol. 1.

[80] K. Kværn, "Effects of expertise and strategies on program comprehension in maintenance of object-oriented systems: A controlled experiment with professional developers," Master's thesis, Department of Informatics, University of Oslo, 2006.

[81] H. E. Kyburg Jr., *Theory and measurement*. Cambridge: Cambridge University Press, 1984.

[82] P. C. Kyllonen and D. L. Stephens, "Cognitive abilities as determinants of success in acquiring logic skill," *Learning and individual differences*, vol. 2, no. 2, pp. 129–160, 1990.

[83] A. Kyngdon, "The Rasch model from the perspective of the representational theory of measurement," *Theory & Psychology*, vol. 18, no. 1, pp. 89–109, 2008.

[84] J. M. Linacre, "Sample size and item calibration stability," *Rasch Measurement Transactions*, vol. 7, no. 4, p. 328, 1994.

[85] E. A. Locke, "Generalizing from laboratory to field: Ecological validity or abstraction of essential elements?" in *Generalizing from laboratory to field setting: Research findings from industrial-organizational psychology, organizational behavior, and human resource management*, E. A. Locke, Ed. Lexington, MA: Lexington Books, 1986, pp. 3–42.

[86] J. C. Loehlin, *Latent variable models: An introduction to factor, path, and structural equation analysis*, 4th ed. Mahwah, NJ: Lawrence Erlbaum, 2004.

[87] R. D. Luce and J. W. Tukey, "Simultaneous conjoint measurement: A new type of fundamental measurement," *Journal of Mathematical Psychology*, vol. 1, no. 1, pp. 1–27, 1964.

[88] D. G. MacKay, "The problems of flexibility, fluency, and speed-accuracy trade-off in skilled behavior," *Psychological Review*, vol. 89, no. 5, pp. 483–506, 1982.

[89] J. McCall, "Quality factors," in *Encyclopedia of software engineering*, J. J. Marciniak, Ed. Wiley-Interscience, 1994, vol. 2, pp. 958–969.

[90] M. McCracken, V. Almstrum, D. Diaz, M. Guzdial, D. Hagan, Y. B.-D. Kolikant, C. Laxer, L. Thomas, I. Utting, and T. Wilusz, "A multi-national, multi-institutional study of assessment of programming skills of first-year CS students," *ACM SIGCSE Bulletin*, vol. 33, no. 4, pp. 125–140, 2001.

[91] S. Messick, "Validity," in *Educational measurement*, 3rd ed., R. L. Linn, Ed. New York: American Council on Education/Macmillan, 1989, pp. 12–103.

[92] ——, "The interplay of evidence and consequences in the validation of performance assessments," *Educational Researcher*, vol. 23, no. 2, pp. 13–23, 1994.

[93] J. Michell, "Quantitative science and the definition of measurement in psychology," *British Journal of Psychology*, vol. 88, no. 3, pp. 355–383, 1997.

[94] A. Newell and P. Rosenbloom, "Mechanisms of skill acquisition and the law of practice," in *Cognitive skills and their acquisition*, J. R. Anderson, Ed. Hillsdale, NJ: Erlbaum, 1981, pp. 1–56.

[95] J. C. Nunnally and I. H. Bernstein, *Psychometric theory*, 3rd ed. New York: McGraw-Hill, 1994.

[96] T. H. Pear, "The nature of skill," *Nature*, vol. 122, no. 3077, pp. 611–614, 1928.

[97] E. J. Pedhazur and L. P. Schmelkin, *Measurement, design, and analysis: An integrated approach*. Hillsdale, NJ: Lawrence Erlbaum, 1991.

[98] P. Pirolli and M. Wilson, "A theory of the measurement of knowledge content, access, and learning," *Psychological Review*, vol. 105, no. 1, pp. 58–82, 1998.

[99] K. Popper, *Conjectures and refutations*. New York: Harper & Row, 1968.

[100] L. Prechelt, "The 28:1 Grant/Sackman legend is misleading, or: How large is interpersonal variation really?" University of Karlsruhe, Tech. Rep. 18, 1999.

[101] G. Rasch, *Probabilistic models for some intelligence and achievement tests*. Copenhagen: Danish Institute for Educational Research, 1960.

[102] P. N. Robillard, "The role of knowledge in software development," *Communications of the ACM*, vol. 42, no. 1, pp. 87–92, 1999.

[103] R. J. Rummel, *Applied factor analysis*. Evanston, IL: Northwestern University Press, 1970.

[104] F. L. Schmidt and J. E. Hunter, "The validity and utility of selection methods in personnel psychology: Practical and theoretical implications of 85 years of research findings," *Psychological Bulletin*, vol. 124, no. 2, pp. 262–274, 1998.

[105] F. L. Schmidt, J. E. Hunter, and A. N. Outerbridge, "Impact of job experience and ability on job knowledge, work sample performance, and supervisory ratings of job performance," *Journal of Applied Psychology*, vol. 71, no. 3, pp. 432–439, 1986.

[106] N. Schmitt, "Uses and abuses of coefficient alpha," *Psychological Assessment*, vol. 8, no. 4, pp. 350–353, 1996.

[107] J. W. Schooler, S. Ohlsson, and K. Brooks, "Thoughts beyond words: When language overshadows insight," *Journal of Experimental Psychology: General*, vol. 122, no. 2, pp. 166–183, 1993.

[108] W. R. Shadish, T. D. Cook, and D. T. Campbell, *Experimental and quasi-experimental designs for generalized causal inference*. Boston: Houghton Mifflin, 2002.

[109] R. J. Shavelson and N. M. Webb, *Generalizability theory: A primer*. Thousand Oaks, CA: Sage Publications, 1991.

[110] B. A. Sheil, "The psychological study of programming," *ACM Computing Surveys*, vol. 13, no. 1, pp. 101–120, 1981.

[111] V. J. Shute, "Who is likely to acquire programming skills?" *Journal of Educational Computing Research*, vol. 7, no. 1, pp. 1–24, 1991.

[112] D. I. K. Sjøberg, B. Anda, E. Arisholm, T. Dybå, M. Jørgensen, A. Karahasanović, E. F. Koren, and M. Vokáč, "Conducting realistic experiments in software engineering," in *Proceedings of the International Symposium Empirical Software Engineering*, 2002, pp. 17–26.

[113] D. I. K. Sjøberg, T. Dybå, B. C. D. Anda, and J. E. Hannay, "Building theories in software engineering," in *Guide to advanced empirical software engineering*, F. Shull, J. Singer, and D. I. K. Sjøberg, Eds. London: Springer-Verlag, 2008, pp. 312–336.

[114] D. I. K. Sjøberg, J. E. Hannay, O. Hansen, V. B. Kampenes, A. Karahasanovic, N.-K. Liborg, and A. C. Rekdal, "A survey of controlled experiments in software engineering," *IEEE Transactions on Software Engineering*, vol. 31, no. 9, pp. 733–753, 2005.

[115] D. I. K. Sjøberg, A. Yamashita, B. Anda, A. Mockus, and T. Dybå, "Quantifying the effect of code smells on maintenance effort," *IEEE Transactions on Software Engineering*, vol. 39, no. 8, pp. 1144–1156, 2013.

[116] S. S. Skiena and M. A. Revilla, *Programming challenges: The programming contest training manual*. New York: Springer, 2003.

[117] R. M. Smith, "The distributional properties of Rasch standardized residuals," *Educational and Psychological Measurement*, vol. 48, no. 3, pp. 657–667, 1988.

[118] E. V. Smith Jr., "Detecting and evaluating the impact of multidimensionality using item fit statistics and principal component analysis of residuals," *Journal of Applied Measurement*, vol. 3, no. 2, pp. 205–231, 2002.

[119] E. Soloway and K. Ehrlich, "Empirical studies of programming knowledge," *IEEE Transactions on Software Engineering*, vol. SE-10, no. 5, pp. 595–609, 1984.

[120] S. S. Stevens, "On the theory of scales of measurement," *Science*, vol. 103, no. 2684, pp. 677–680, 1946.

[121] D. L. Streiner, *Health measurement scales*. Oxford: Oxford University Press, 1995.

[122] A. Syang and N. B. Dale, "Computerized adaptive testing in computer science: Assessing student programming abilities," *ACM SIGCSE Bulletin*, vol. 25, no. 1, pp. 53–56, 1993.

[123] N. Unsworth, R. P. Heitz, J. C. Schrock, and R. W. Engle, "An automated version of the operation span task," *Behavior Research Methods*, vol. 3, no. 37, pp. 498–505, 2005.

[124] N. Unsworth, T. Redick, R. P. Heitz, J. M. Broadway, and R. W. Engle, "Complex working memory span tasks and higher-order cognition: A latent-variable analysis of the relationship between processing and storage," *Memory*, vol. 17, no. 6, pp. 635–654, 2009.

[125] M. Vokáč, W. Tichy, D. I. K. Sjøberg, E. Arisholm, and M. Aldrin, "A controlled experiment comparing the maintainability of programs designed with and without design patterns—a replication in a real programming environment," *Empirical Software Engineering*, vol. 9, no. 3, pp. 149–195, 2004.

[126] D. A. Waldman and W. D. Spangler, "Putting together the pieces: A closer look at the determinants of job performance," *Human Performance*, vol. 2, no. 1, pp. 29–59, 1989.

[127] S. Wiedenbeck, "Novice/expert differences in programming skills," *International Journal of Man-Machine Studies*, vol. 23, no. 4, pp. 383–390, 1985.

[128] D. Wilking, D. Schilli, and S. Kowalewski, "Measuring the human factor with the Rasch model," in *Balancing Agility and Formalism in Software Engineering*, ser. Lecture Notes in Computer Science, B. Meyer, J. R. Nawrocki, and B. Walter, Eds. Berlin: Springer, 2008, vol. 5082, pp. 157–168.

[129] M. Wilson, *Constructing measures: An item response modeling approach.* Mahwah, NJ: Lawrence Erlbaum Associates, 2005.

[130] B. D. Wright and G. N. Masters, *Rating scale analysis*. Chicago: Mesa Press, 1979.

TABLE 7
Tasks sampled or constructed for the instrument

Task ID	Name	Origin	Lifecycle	Time limit[a]	Sub-tasks (#)	Evaluation type	Description (Subtask)
1	Postfix Calculator	Article [90]	Write	5, 35	Yes (3)	Automatic	(A) Implement +, – and * operators. (B) Implement support for exception handling. (C) Implement /. Pseudo code for the solution was provided.
2	Mine-sweeper	Book [116]	Write	5, 40	Yes (3)	Automatic	(A) Implement algorithm for locating mines on a small field. (B) Improve and (C) extend this implementation. Code that handled basic reading of text files was provided.
3	Crosswords	Construct-ed	Write	5, 45	No	Automatic	Implement a set of classes with support defining and solving crosswords. A code skeleton with test cases was provided. Written by a professional programmer.
4	Numeric Display	Article [63]	Maintain	5, 25	No	Manual and automatic	Extend the functionality of a timekeeping device from using minutes and seconds to include hours and milliseconds. Use of good object-oriented principles was evaluated manually. Used verbatim.
5	Line Printer	Article [64]	Write	5, 20	No	Automatic	Read lines of text and print these 30 characters to a line without breaking the words in between. Required standard Java library knowledge. The problem formulation was used verbatim.
6	Array Sorting	Construct-ed	Write	5, 25	Yes (3)	Automatic	(A) Find the smallest integer in an array. (B) Calculate the average of elements in an array. (C) Sort even elements before odd elements in an array. A master student wrote the task with accompanying code.
7	Comm Channel	Article [125]	Maintain	10, 35	Yes (2)	Automatic	(A) Add a new class for Hammingcodes based on existing classes. (B) Use Decorator pattern to set up a communication channel between a sender and receiver. Rewritten from C++.
8	Minibank	Article [11]	Maintain	0, 45	No	Automatic	Implement functionality for a bank statement for an ATM machine. No reading time and a generous time limit was used for comparability with previous studies. Used verbatim.
9	Library Application	Article [56]	Maintain	5, 35	Yes (3)	Manual	(A) Change window size and add a text field to a GUI application. (B) Implement persistence, following the standards in the provided code. (C) Update affected dialogue boxes with email functionality.
10	Absolute Space	Construct-ed[b]	Maintain	5, 20	Yes (3)	Manual	(A–B) Modify laser cannon behavior and (C) spaceship movement in a GUI-based game. The original code was available online and only the problems were formulated.
11	Geometric Calculator	Construct-ed	Maintain	5, 40	No	Manual and automatic	Refactor poor, but functionally correct, code for a calculator for geometric shapes. Good use of object-oriented principles and Java constructs was evaluated manually.
12	Laser Controller	Con-structed	Maintain	5, 15	No	Automatic	Identify and correct a problem with an industrial laser system. The task was based on a real industrial problem encountered by the professional programmer who wrote the task.
13	Hello World	Legacy	Write	0, 10	No	Automatic	Probably the easiest and most well-known programming tasks of all time. The task was included in an attempt to establish how much more difficult the other tasks were, relative to this task.
14	Memory Leak	Book [19]	Debug	5, 10	No	Automatic	Locate and fix a small bug in an implementation of a stack routine that consumes too much memory. The problem was formulated based on the original code.
15	Node List Find	Article [15]	Debug	5, 10	No	Automatic	Locate and fix a bug in a method that should find an element in a linked list. The task was rewritten in Java from a C implementation. Two different implementations (recursive, iterative) were used.
16	Node List Copy	Article [15]	Debug	5, 10	No	Automatic	Locate and fix a bug in a method that should copy an element in a linked list. The task was rewritten in Java from a C implementation. Two different implementations (recursive, iterative) were used.
17	Coffee Machine	Article [39]	Maintain	5, 30	No	Automatic	Extend the functionality of a coffee machine implementation to dispense bouillon. A sequence diagram for the existing system was provided. The recursive version of the second task was used verbatim.
18	Watering System	Construct-ed	Maintain	10, 40	Yes (3)	Automatic	(A) Implement a controller class for a watering system and (B–C) extend this functionality. Written by a professional programmer and requires concurrent programming.
19	Traffic Lights	Construct-ed	Debug	5, 45	Yes (4)	Manual	(A) Fix bugs related to cars, (B) light sensors and (C–D) new sensor placements in a traffic control system. Written by a professional programmer and requires concurrent programming.

Tasks 13–19 were excluded from the instrument. [a] Reading time began when the task description was downloaded (before downloading code), coding time began after code was downloaded.
[b] By permission of A. Udovydchenko, Absolute Space, 2000, http://www.onasch.de/games/absolute.source.zip.

TABLE 8
Task performance scores for the subjects in the final instrument

Subject			Task ID											
ID	Dataset	Skill	6	10	9	4	1	8	3	12	11	7	2	5
1	C	-4.12	1	0	1	0	0	0	0	0	0	0	0	0
2	V	-3.89	1	—	0	1	0	0	0	0	0	0	0	0
3	C	-3.53	0	1	1	1	0	0	0	0	0	0	0	0
4	C	-3.53	1	1	1	0	0	0	0	0	0	0	0	0
5	V	-3.03	1	1	1	1	0	—	0	0	0	0	0	0
6	C	-2.70	1	2	1	0	1	0	0	0	0	0	0	0
7	C	-2.37	1	2	2	1	0	0	0	0	0	0	0	0
8	C	-2.37	2	1	1	0	1	1	0	0	0	0	0	0
9	V	-2.08	1	1	2	1	1	0	0	0	0	1	0	0
10	V	-2.08	1	1	2	2	0	1	0	0	0	0	0	0
11	V	-2.03	1	1	3	0	1	0	—	0	0	1	0	0
12	V	-1.82	1	2	3	1	0	1	0	0	0	0	0	0
13	C	-1.82	2	1	3	0	0	1	0	1	0	0	0	0
14	C	-1.82	1	1	1	2	0	1	0	1	0	1	0	0
15	C	-1.82	1	1	3	1	1	1	0	0	0	0	0	0
16	V	-1.82	1	1	2	1	0	2	1	0	0	0	0	0
17	C	-1.58	2	2	1	2	0	1	0	0	1	0	0	0
18	V	-1.58	1	3	3	2	0	0	0	0	0	0	0	0
19	C	-1.58	2	0	1	1	0	1	1	0	1	1	1	0
20	C	-1.58	1	2	3	1	0	1	1	0	0	0	0	0
21	C	-1.51	2	1	2	3	0	0	—	0	0	0	1	0
22	C	-1.36	1	2	2	1	0	1	2	0	1	0	0	0
23	C	-1.36	1	1	3	1	1	1	0	0	1	0	1	0
24	C	-1.36	2	2	2	1	1	1	0	1	0	0	0	0
25	V	-1.16	1	2	3	1	0	1	1	2	0	0	0	0
26	V	-0.97	2	2	3	1	1	0	2	0	0	1	0	0
27	C	-0.97	2	2	2	1	1	0	1	0	0	2	1	0
28	C	-0.97	2	2	1	2	1	0	1	0	1	1	0	1
29	C	-0.97	3	2	4	1	0	0	1	0	0	1	0	0
30	C	-0.97	2	1	3	2	0	0	1	0	1	0	2	0
31	C	-0.79	2	2	3	1	1	1	1	0	1	0	1	0
32	C	-0.79	1	1	3	2	2	1	0	0	2	0	1	0
33	C	-0.79	1	2	3	2	2	2	0	0	0	1	0	0
34	C	-0.61	2	2	2	1	0	2	1	1	1	1	0	1
35	C	-0.61	2	2	2	2	1	0	2	1	1	1	0	0
36	V	-0.61	2	2	4	1	0	1	2	0	1	0	1	0
37	C	-0.61	2	2	3	2	0	0	1	0	1	2	1	0
38	V	-0.45	2	3	3	2	3	0	0	0	1	0	0	1
39	C	-0.45	2	3	3	2	0	2	0	1	2	0	0	0
40	C	-0.28	2	2	3	2	1	1	1	1	1	1	1	0
41	V	-0.28	2	2	3	1	3	1	0	1	2	0	0	1
42	C	-0.12	1	2	3	1	2	2	1	0	2	1	1	1
43	V	0.04	2	3	3	2	1	1	3	0	1	1	1	0
44	V	0.04	2	2	4	1	2	0	0	2	1	2	1	1
45	C	0.04	2	2	4	2	2	1	0	0	1	2	1	1
46	C	0.04	1	3	4	2	2	2	0	1	1	2	0	0
47	C	0.04	2	2	4	2	2	1	1	1	1	1	1	0
48	C	0.04	3	3	3	2	1	2	2	0	0	2	0	0
49	C	0.04	2	3	4	2	1	2	1	2	1	0	0	0
50	C	0.20	2	3	5	3	2	0	0	0	1	1	1	1
51	V	0.20	1	0	4	1	2	2	0	2	2	2	2	1
52	C	0.25	3	2	3	2	2	0	—	1	2	1	1	1
53	C	0.35	3	3	3	1	2	1	1	1	2	1	1	1
54	C	0.35	2	3	4	2	2	1	0	0	2	2	1	1
55	V	0.35	2	3	5	2	2	0	1	2	1	1	1	0
56	C	0.35	1	3	4	2	1	2	3	1	2	0	1	0
57	V	0.35	1	3	4	1	2	1	2	1	3	1	1	0
58	C	0.51	2	3	5	2	1	2	1	0	3	1	1	0
59	C	0.51	3	3	4	2	1	2	2	1	0	1	1	1
60	V	0.68	3	3	3	2	2	2	2	1	1	2	1	0
61	C	1.01	2	3	3	2	2	2	2	0	2	3	2	1
62	V	1.19	1	3	4	2	3	3	1	2	1	3	1	1
63	V	1.19	1	3	5	2	3	2	2	1	1	2	2	1
64	C	1.58	3	2	3	3	3	3	2	1	1	3	2	1
65	C	1.58	2	3	5	0	3	3	3	2	1	2	1	2

Missing observations are denoted as —. C is the instrument construction and V the instrument validation data set; see Section 3.6. Tasks are sorted in increasing order of difficulty. Tasks 3, 4, 6, and 8–10 is the Easy subset described in Section 4.2; the remaining tasks is from the Difficult subset.