Measuring Programming Skill

Construction and Validation of an Instrument for Evaluating Java Developers

Gunnar Rye Bergersen

Thesis submitted for partial fulfillment of
the requirements for the degree of Philosophiae Doctor

Department of Informatics
Faculty of Mathematics and Natural Sciences
University of Oslo

May 2015

Abstract

Skilled developers are important to the software industry. In empirical studies in software engineering, knowing the skill level of the participants is also important for correct interpretation of results. The current practice in industry and research 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. Consequently, this thesis aimed to construct a valid instrument for measuring programming skill, where skill is inferred from performance on programming tasks.

The Rasch measurement model was used to construct the instrument. Sixty-five professional developers from eight countries participated in validating the instrument, solving 19 Java programming tasks over two days. The validity of the instrument was theoretically investigated through commercial and research-based tests. Programming skill, as measured by the instrument, was also investigated in terms of experience and other background variables.

The instrument was found to have desirable psychometric properties, and the overall results appear well aligned with theoretical expectations. This work has shown that acceptable measures of programming skill may be obtained with less than one day of testing. Further work should be directed at reducing the time needed to measure programming skill without affecting the validity of the instrument. The results of the research have already been transferred to the industry through a commercial prototype.

Acknowledgments

First and foremost, I am profoundly thankful to my principal supervisor Dag Sjøberg. He has provided me with unwavering support during my research for this thesis, often at odd hours and in strange places. I feel privileged to have had the opportunity to occupy so much of Dag's time over the years. The making of this thesis has been a truly rewarding experience, both professionally and personally, because of him. I am also grateful to my second supervisor, Tore Dybå, for interesting discussions and for inspiring me to look to other research fields for solutions to my research problem.

Other researchers have also been important at various stages. I thank Erik Arisholm for his guidance and flexibility during the early years of my work. A special thanks goes to Jo Hannay and Magne Jørgensen, who are highly inspiring researchers. I am also grateful to Jan-Eric Gustafsson for his interest in my work.

This thesis would have been impossible without financial support. Simula Research Laboratory gave me the opportunity to carry out the research and Simula Innovation and Simula School of Research and Education provided support as well. The FORNY program at the Norwegian Research Council funded parts of my work, while the University of Oslo allowed me to finish the thesis at a pace that was rewarding for me.

This thesis would similarly have been impossible without technical support. Early on, Steinar Haugen shared his passion for programming with me and has since become an invaluable colleague. To get anything complex done these days, one needs a highly skilled programmer; I am glad that programmer is Steinar. I also thank Gunnar Carelius for his many hours of technical support during data collection.

I thank Erik Arisholm, Amela Karahasanović, James Dzidek, Marek Vokáč, and Kaja Kværn for allowing me to analyze their data sets; Hans Gallis, James Dzidek, Kristin Børte, and Viktoria Stray for being such good office mates; Hans Christian Benestad, Stein Grimstad, Vigdis By Kampenes, Aiko Yamashita, Simen Hagen, and Rolf Vegar Olsen, whose encouragement helped me continue; Bjarne Johannessen for proofreading; and Lenore Hietkamp for copy editing far beyond the call of duty.

I am grateful to my friends and family for their support and for listening; sometimes it helps to talk about challenging issues with those who know you well. A special thanks goes to my mother, Elisabeth, who always urges me to keep learning and who unselfishly offers her assistance in countless ways. My thoughts are also with my father, Geir, who enjoyed listening to me talk about my work but never saw me finish it.

Finally, I am especially grateful to my lovely wife, Gina, for her continuous encouragement and patience, and for putting up with a husband who often is physically, but not mentally, present. A special debt of gratitude goes to Hedvig and Erle for being such fun and loving daughters; when they are older, I hope they both forgive the minuscule experiments they unknowingly participate in from time to time.

List of Papers

The following papers are included in this thesis:

(I) **Inferring skill from tests of programming performance: combining time and quality**
Gunnar R. Bergersen, Jo E. Hannay, Dag I. K. Sjøberg, Tore Dybå, and Amela Karahasanović
In Proceedings of the 5th International Symposium on Empirical Software Engineering and Measurement, 2011, pp. 305–314.

(II) **Construction and validation of an instrument for measuring programming skill**
Gunnar R. Bergersen, Dag I. K. Sjøberg, and Tore Dybå
In *IEEE Transactions on Software Engineering*, Vol. 40, No. 12, pp. 1163–1184, 2014.

(III) **Programming skill, knowledge, and working memory among professional developers from an investment theory perspective**
Gunnar R. Bergersen and Jan-Eric Gustafsson
In *Journal of Individual Differences*, Vol. 32, No. 4, pp. 201–209, 2011.

(IV) **Evaluating methods and technologies in software engineering with respect to developers' skill level**
Gunnar R. Bergersen and Dag I. K. Sjøberg
In Proceedings of the 16th International Symposium on on Evaluation and Assessment in Software Engineering, 2012, pp. 101–110.

For all the four papers, I was responsible for the design, analysis, and writing. Dag Sjøberg and Erik Arisholm provided the overall research problem of measuring programming skill. My supervisors, Dag Sjøberg and Tore Dybå, contributed with general advice and suggestions for implementation of the research and were involved in most parts of the writing. Erik Arisholm and Jo Hannay also provided much appreciated feedback on many parts of the work. Data collection for Paper I was conducted by Erik Arisholm, Amela Karahasanović, and Kaja Kværn. For Papers II, III, and IV, which are all based on the same data, I was responsible for the data collection. Additionally, I also wrote the application for funding that was required to hire the professional software developers

in the same three papers. For Paper III, Jan-Eric Gustafsson was partly involved in the design of the study.

In 2011, I presented two extended abstracts at a conference on the Rasch measurement model. Even though they resulted in valuable feedback on the reported work, they are not considered publications and are therefore not included as separate parts of the thesis:

Combining time and correctness in the scoring of performance on items
Gunnar R. Bergersen
In Probabilistic Models for Measurement in Education, Psychology, Social Science and Health, J. Brodersen, T. Nielsen, and S. Kreiner (Eds.). Copenhagen: Copenhagen Business School and the University of Copenhagen, June 2011, pp. 43–44.

Detecting learning and fatigue effects by inspection of person-item residuals
Gunnar R. Bergersen and Jo E. Hannay
In Probabilistic Models for Measurement in Education, Psychology, Social Science and Health, J. Brodersen, T. Nielsen, and S. Kreiner (Eds.). Copenhagen: Copenhagen Business School and the University of Copenhagen, June 2011, pp. pp. 56–57.

Contents

List of Figures

List of Tables

Summary

Paper I

Paper II

Paper III

Paper IV

Summary

1 Introduction

Software engineering involves the development, improvement, and understanding of technologies, processes, and resources that constitute software development. In the modern society, which is driven by software, one key resource is the software developer or programmer. One way to satisfy society's ever-increasing demand for greater productivity is to educate and train developers to become highly skilled and, thus, productive. This thesis investigates the extent to which programming skill can be measured.

1.1 Programming Skill Differences

As in most human activities, individual performance in the development of software varies considerably. The purpose of studying individual differences is to understand, predict, and represent such differences. Variation in performance can be attributed to the different capacities for performance that individuals possess. Such capacities, or capabilities, which have a potential for performance, are usually expressed as abilities.

Most generally, "an ability is a [human] trait defined by what an individual can do" (Ferguson, 1956, p. 122). While skill falls under the broad category of human "abilities", it is a specialized type of ability, one that improves with practice and is well adjusted (Pear, 1928), well organized, and goal oriented (Fitts & Posner, 1967). Many factors affect the performance of an individual, albeit indirectly, but there are only three direct determinants of performance—knowledge, skill, and motivation (Campbell, McCloy, Oppler, & Sager, 1993).

Even though skill is inferred from performance (Fitts & Posner, 1967), skill cannot be *equated* with performance. For example, if one observes that an individual performs well on a specific task, one may state the obvious: "this is good performance." However, if one

states that "the individual performed well *because of* his high level of skill", one is making a generalization, which in turn requires justification (Messick, 1994). High performance is, in many occasions, most likely due to high skill. However, high performance may also be due to luck.

When stating that someone is highly skilled, the expectation is that the person would perform well over time for a wide range of tasks, as in a job context. In the taxonomy of eight major components of job performance suggested by Campbell et al. (1993), one such component is "technical skill". In the software industry, several surveys and studies have ranked technical skill as the most important skill for a software developer or programmer (see, e.g., Bailey & Mitchell, 2006; Hawk et al., 2012; Lethbridge, 2000; McGill, 2008). A recent analysis of nearly 800,000 projects or tasks at an outsourcing provider concluded that "the client may substantially reduce the risk of project failure by emphasizing good provider skills rather than low price" (Jørgensen, 2014, p. 19). Moreover, according to the US Bureau of Labor Statistics, the description of a software developer is someone who usually holds "a bachelor's degree in computer science and strong computer programming skills" (2014). The programming skill level of software developers is therefore important during staffing decisions, such as hiring a new employee or consultant, or assigning existing employees to a software project. At present, such decisions are based on more or less well-founded perceptions of skill level.

Individual skill is also central to how well teams of software developers perform. Although team performance is more complex to understand than individual performance due to the many components that may interact in a team (see, e.g., Baker & Salas, 1992; Volmer, 2006), individual skill or expertise is nevertheless a central component in team performance (Land, Wong, & Jeffery, 2003). Because team skill may be a function of individual skill plus interactions between individuals, measuring team skill (e.g. Beaver & Schiavone, 2006) partially depends on the understanding and measurement of individual skill.

The technical skill of an individual is also central to research on job performance in general; according to Campbell, Gasser, and Oswald, "a full model of the causal mechanisms linking ability, personality, training, experience, and so on with [job] performance will require valid measurement of . . . job skill" (1996, p. 276).

Variability between developers creates problems in experiments in software engineering (Tichy, 1998). For example, in randomized experiments in software engineering, where groups of developers are presented with different treatments to determine their effect, researchers sometimes assume that no other differences exist between the two groups that can affect the outcome of the dependent variable. Differences in the skill level of the individuals in each of the groups may nevertheless exist, and can thereby confound the interpretation of the results (see generally Shadish, Cook, & Campbell, 2002). In quasi-experiments in particular, which are experiments without random assignment to treatment, the problem of having groups with unequal skill is presumed to be a pervasive confounding factor (Kampenes, Dybå, Hannay, & Sjøberg, 2009).

Education is another area where programming skill differences may play an important role. For example, the goal of curriculum in software programming courses is to teach students relevant knowledge of software development as well as the *application* of this

knowledge. Particularly for vocational educations, the degree to which a newly graduated student can immediately contribute positively to a project may be an important criterion of whether students are sufficiently skilled to be hired for a job. The development of skill is also a major component in the evolution of the profession of software engineering, for example, as expressed in multi-institutional initiatives such as SWEBOK2004 (Abran, Moore, Bourque, Dupuis, & Tripp, 2004) or SE2004 (Lethbridge, LeBlanc, Sobel, Hilburn, & Diaz-Herrera, 2006). In addition to ensuring that students are taught relevant skills (see, e.g., Gallivan, Truex, & Kvasny, 2004; Surakka, 2007), the level of programming skill that groups of students acquire during their training may also indicate the quality of an educational system.

Differences in programming skill are substantial in industry, research, and education (see, e.g., Boehm, 1981; Bryan, 1994; Card, Mc Garry, & Page, 1987; Curtis, 1981; DeMarco & Lister, 1985, 1999). In an early experiment on programming performance, large differences in performance were found between professional developers who solved the same tasks. Consequently, Grant and Sackman recommended that "[t]echniques measuring individual programming skills should be vigorously pursued, tested and evaluated, and developed on a broad front for the growing variety of programming jobs" (1967, p. 46). Even though there is disagreement regarding the magnitude of these differences (Dickey, 1981; Prechelt, 1999a), the differences appear to be considerable, and measures of skill therefore should be developed (Kampenes et al., 2009; Prechelt, 1999a).

1.2 Scientific versus Common Measurement

An indication of the maturity of a discipline is the extent to which standardized measurement instruments are available (Ebert, Dumke, Bundschuh, & Schmietendorf, 2005). The practices involved when measuring something may vary considerably. It is therefore important to explicate how such practices differ.

The gold standard in terms of rigor is that of *scientific measurement* used within the physical sciences (see, e.g., Krantz, Luce, Suppes, & Tversky, 1971). According to Michell, measurement is defined as "the estimation or discovery of the ratio of some magnitude of a quantitative attribute to a unit of the same attribute" (1997, p. 358). Measures of attributes may be fundamental or indirect. For example, measurement of "length" is considered fundamental because it does not require the measurement of other attributes, whereas measurement of "density" is indirect because it requires the measurement of both mass and volume (Krantz et al., 1971). Nevertheless, both fundamental and indirect measures can be obtained through measurement instruments, which are high-precision devices or tools used to obtain measures.

In contrast, perhaps the most commonly used definition of measurement in psychology originates from a 1940 report that addressed whether the intensity of sensory events were measurable. Although the members of the committee "found themselves unable to agree on the meaning of such terms as 'measurement' or 'quantitative estimate'" (Ferguson et al., 1940, p. 332), Stevens paraphrased one of the committee member's views and defined measurement "in the broadest sense ... as the assignment of numerals to objects or events according to rules" (1946, p. 677).

Software engineering is a multi-disciplinary field and uses both the scientific and the more informal definition of measurement. For example, "time" is a quantitative variable that can be scientifically and fundamentally measured. Conversely, variables such as software quality (Ebert et al., 2005) and key factors of success in software process improvement (Dybå, 2000) are also variables that are indirectly measured using quite different methods. Thus, "measurement" can refer to two factually different practices. I will refer to the more specific scientific definition as "scientific measurement", in contrast to the common, more informal definition of "measurement" throughout this summary.

For the present context, the most important distinction between scientific and common measurement is that the two practices differ according to their level of rigor. Scientific measurement involves a unit and a continuous, quantitative attribute, and it is possible to empirically test whether a purported measurement instrument actually yields scientific measures of quantitative attributes (Michell, 1997). In contrast, the rigor involved in common measurement varies. In some instances, whether a variable is quantitative and thus measurable is not tested. This is problematic because measurement then becomes a truism where nothing informative is gained by the assertion that something is being measured. However, in other instances, partial requirements for scientific measures are met (see Borsboom & Mellenbergh, 2004). Thus, one may surmise, first, that from the perspective of scientific measurement, the attribute being measured must be quantitative (Markus & Borsboom, 2012), and second, that according to Stevens' version of measurement, there must exist laws about the attribute that is being measured that are empirically testable (Luce, 1997).

Another distinction between the two views on measurement is the scales for which the measure of some attribute is represented. Common measurement uses four scale categories: nominal, ordinal, interval, and ratio (Stevens, 1946). However, because the nominal scale is merely a category where no two objects are assigned the same value, only three scales are of primary interest for representing differences in skill. In increasing order of precision, the remaining three scales can state whether a developer A differs from a developer B with respect to being (say)

- better than (ordinal scale, i.e., greater or less),
- a certain magnitude better than (interval scale, i.e., equality of differences), or
- twice as good (ratio scale, i.e., equality of ratios).

For scientific measurement, only the interval and ratio scales are properly used in connection with the term measurement.

The problem of having two different practices involved in the validation of measures has previously been pointed out by researchers within software engineering (see Fenton, 1994; Fenton & Kitchenham, 1991). This challenge appears to have been answered by calling for pragmatism; if scientific definition of measurement were to be used, "it would represent a substantial hindrance to the progress of empirical research in software engineering" (Briand, El Emam, & Morasca, 1996, p. 61). Generally, there is nothing wrong with being pragmatic as long as shortcomings are acknowledged. However, it is easy to misinterpret this pragmatism as an indication that software engineering has somehow

resolved the problems associated with measuring central concepts such as programming skill.

1.3 Research Problem and Research Questions

A valid scientific measure of programming skill that is easily administered, scored, and interpreted will have a wide range of applications in industry, research, and education. However, it is uncertain whether such a measure is possible to attain. Thus, the overall research problem in this thesis is *the extent to which programming skill can be validly measured from programming performance*, using a scientific definition of measurement.

One may ask why measures of skill are required when one can use fundamental and scientific measures of "programming performance" directly. For example, the time needed to obtain a correct solution on a programming task can be scientifically measured, allowing ratio comparisons to be made directly (e.g., "developer A is twice as fast as developer B"). There are several problems associated with such a solution. First, there is no easy way to compare the time for those problems with incorrect solutions with the time for those with correct solutions. For example, if developer B is unable to solve a problem that developer A solved in one hour, the ratio of performance between developers A and B is unknown. Another problem is that programming performance not only involves time but also software quality, which consists of many sub dimensions (McCall, 1994) that can be difficult to measure (see, for example, Jones, 1978 for an early paper). Researchers in software engineering have encountered problems with the conceptualization and measures of software quality (see Fenton & Kitchenham, 1991; Kitchenham & Pfleeger, 1996). Thus, there are no "trivial" solutions to the overall research problem. Instead, three research questions (RQ) will therefore be investigated in this thesis:

RQ1: How can time and quality of a task be combined as programming performance?

RQ2: How can programming skill be measured from programming performance?

RQ3: How can measures of programming skill be validated?

Concerning RQ1, the relation between time, quality, and performance may be formulated as follows: First, assume that other variables remain fixed. Then, to define an individual's level of programming performance as high or good, quality should be as high as possible, and time spent should be as low as possible. However, a problem is that time and quality do not operate using the same units, and the tradeoff for one with the other is therefore often unknown. Time and quality may also use different scales, depending on what factor of quality is intended. For example, computing efficiency, which is the amount of resources a computer uses during a calculation, may be measured using a ratio scale (e.g., computer clock cycles or CPU time). Furthermore, correctness may use an ordinal scale (e.g., "incorrect", "partially correct", or "correct"). Thus, one must investigate how programming tasks can be used to define performance, when time, quality, or both time and quality may vary.

With respect to RQ2, it should be clear that performance is a characteristic of the actions of an individual, but it is not an *attribute* of the individual; that is, performance is "something we do", not "something we are" (see Messick, 1994). For example, for an airplane passenger, the difference between landing safely (good flying performance) and dying in a crash (the worst possible flying performance) is probably infinitely large. However, it is clearly wrong to infer that there are infinitely large differences in the piloting skill of the commercial airline pilots because airplanes do crash from time to time. Thus, whereas performance is central to infer skill, performance cannot be equated with skill.

One particular issue that informs the distinction between performance and skill is the measurement unit used to represent the two concepts. Whereas performance may use units such as time, degree of correctness, efficacy, reliability, etc., none of these units are meaningful to characterize skill. For example, Ackerman (1992) studied differences in skill in an air traffic control simulator, where the subjects were required to manage the arrival and departure of 28 airplanes. Successful completion of the task, and thus an indicator of the highest skill level, actually involved multiple tasks, since the airplanes must not crash, violate flight safely regulations, or otherwise depart from their scheduled flight plan. Within programming, Anderson, Farrell, and Sauers (1984) investigated programming skill in LISP and used the number and type of errors that the student made during programming to infer skill. Thus, even though an individual's performance on a task may be well defined in terms of capturing both time and quality in relation to skill, no single instance of performance has been identified that can capture all aspects of a skill (Fitts & Posner, 1967). Therefore, some way to measure programming skill from performance is needed.

Finally, RQ3 asks why detailed distinctions in the definition, theory, and, to some extent, the philosophy concerning measurement are important. Why is it important to investigate the validity of instruments that purport to measure skills in programming? The most general answer is that added precision enables well-informed decisions. A prerequisite for well-informed decisions is accurate knowledge. One way such knowledge is sometimes acquired is through theories. However, as Popper states: "measurements presuppose theories" (1968, p. 62). Thus, accurate knowledge about skill requires that we are able to measure it. It is also difficult to begin thinking about a theory or a problem without any idea of how elements that constitute a part of the theory are actually measured (see Kyburg, 1984). Moreover, in some situations, it is not even possible to begin to ask the right research questions until more fundamental issues are resolved, such as whether something can be measured (Michell, 1997). Borsboom states:

> "Thinking about the relation between a psychological attribute and the data patterns that are supposed to measure it forces a deeper investigation into the nature of the attribute and the way the measurement instrument is supposed to work. It requires one to spell out, at least at a very coarse level, why one is justified in treating the data patterns as measurements; i.e., it gives one the beginnings of an argument for the validity of the measurement instrument used" (2008, p. 50).

1.4 Thesis Statement

My thesis statement is as follows:

> For a large proportion of programming tasks in a specific programming language, programming skill can be measured because developers display a reasonably stable level of performance across tasks in a way that is consistent with the theory of skill and, to some extent, scientific measurement.

That only *some* programming tasks can be used to measure programming skill is a delimiting factor. Nevertheless, it will be shown that a valid measure of programming skill can predict programming performance on tasks that cannot be used to measure programming skill. Programming skill is also specific to a programming language. Although many programming concepts are the same across programming languages (i.e., same semantics, but different syntax), technical challenges arise when evaluating these without using a specific programming language.

The aim is to develop *scientific* measures of programming skill. However, the extent to which scientific measures of programming skill can be achieved remains an open question. It will likely remain unresolved for some time to come. Nevertheless, what is important is not this end goal, but rather *the partial results arising from the progress* towards this goal. Many testable consequences can be derived from the theory of skill and the theory of measurement. By systematic, empirical testing of such consequences as well as related assumptions, it may be possible to resolve issues one at a time, thereby yielding cumulative knowledge about the measurement of programming skill.

Finally, the level of abstraction one uses when referring to the term "programming skill" is important. At a low level, programming skill may be both categorical and multidimensional, in the same way that solving addition and solving multiplication problems should be considered distinct skills (see van der Maas, Molenaar, Maris, Kievit, & Borsboom, 2011). Thus, representing programming skill along a single continuous dimension for this low level of abstraction may therefore be inappropriate. However, the intended level of abstraction for this thesis is that of "which of two developers A and B should I assign to project X with a complexity of Y?" At this higher level of abstraction, an interval scale variable may still adequately represent the actual observed differences in programming performance of the two developers.

1.5 Claimed Contribution

From the perspective of methodology in empirical software engineering, this thesis contributes to

- an increased understanding of how time and quality of the solutions to programming tasks can be analyzed as a combined variable (i.e., "programming performance") in a consistent manner,
- a demonstration of alternative ways to conceptualize the measurement of programming skill using instruments where skill is inferred from programming performance,

- an explication of how to validate instruments that claim to "measure programming skill" according to generally accepted scientific standards, and

- insights from conducting a large and comprehensive study on professional programmers in a realistic industrial setting.[1]

This thesis also has research and industry applications. The measurement instrument of programming skill that was developed in this thesis has already been used in empirical studies to select, describe, and analyze the programming skill levels of software developers. An industrial version of the instrument is currently undergoing evaluation in a commercial pilot setting.

1.6 Thesis Structure

This thesis is organized in a summary and a collection of papers:

The *Summary* introduces the papers of the thesis. Section 2 describes the background to the research problem of measuring programming skill and provides an overview of the related literature and fundamental concepts. Section 3 describes the research method. Section 4 summarizes the results of the research questions. Section 5 discusses the overall research problem of measuring programming skill, implication for research, and the use of the measurement instrument. Section 6 concludes.

The *collection of papers* consists of four published papers. Each paper has its own references. Figure 1 shows how Papers I to III address different aspects of the thesis, discussed in detail below. Paper IV demonstrates the use of the instrument in a replicated experiment. Thus, the numbering of the papers is not in chronological order of publication. Instead, the numbering follows a bottom-up approach to understanding the work that ends in a demonstration.

Paper I, "Inferring skill from tests of programming performance: combining time and quality", shows an initial attempt, called the "Pre Study" throughout this thesis, at reanalyzing performance data from four previous programming experiments reported in (Arisholm & Sjøberg, 2004; Karahasanović, Levine, & Thomas, 2007; Karahasanović & Thomas, 2007; Kværn, 2006). The paper was co-authored with Jo Hannay, Dag Sjøberg, Tore Dybå, and Amela Karahasanović and was published in the proceedings of the 5th International Symposium on Empirical Software Engineering and Measurement (Bergersen, Hannay, Sjøberg, Dybå, & Karahasanović, 2011). The main challenge addressed was how to combine time and quality as performance in a way that yielded consistent results across all the tasks that each individual solved. The paper concerns the score aggregation model in Figure 1, where task performance is defined by the variable time and one or more qual-

[1] In a survey of controlled experiments published in 12 leading software engineering journal and conferences between 1993 and 2002, only 19% used professionals as subjects and only 3% of the experiments used payment as reward for participation (Sjøberg et al., 2005). Moreover, with respect to study duration and the number of subjects involved, the research reported here is "large" according to both these classifications used in the survey.

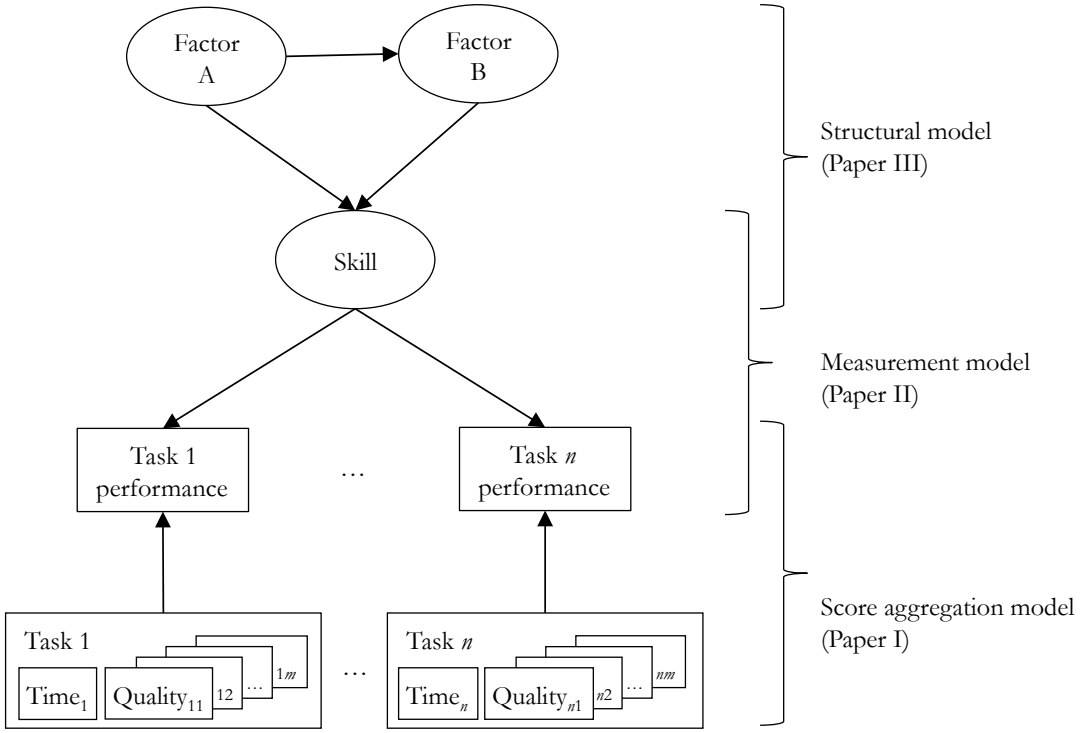


Figure 1: Papers I, II, and III cover different aspects, using three models, whereas Paper IV (not shown) provides a demonstration of how the instrument for measuring programming skill could be used in a replicated experiment.

ity variables using different ways to aggregate these variables. This abstract summarizes the published paper:

> The skills of software developers are important to the success of software projects. Also, when studying the general effect of a tool or method, it is important to control for individual differences in skill. However, the way skill is assessed is often ad hoc, or based on unvalidated methods. According to established test theory, validated tests of skill should infer skill levels from well-defined performance measures on multiple, small, representative tasks. In this respect, we show how time and quality, which are often analyzed separately, can be combined as task performance and subsequently be aggregated as an approximation of skill. Our results show significant positive correlations between our proposed measures of skill and other variables, such as seniority, lines of code written, and self-evaluated expertise. The method for combining time and quality is a promising first step to measuring programming skill in both industry and research settings.

The "Main Study" of this thesis is presented in Papers II, III, and IV. Paper II, "Construction and validation of an instrument for measuring programing skill", takes the main insights from Paper I and uses a new data set with mostly new programming tasks (three tasks from Paper I were reused verbatim to allow comparisons across the data sets).

The paper was co-authored with Dag Sjøberg and Tore Dybå and was published in the *IEEE Transactions on Software Engineering* (Bergersen, Sjøberg, & Dybå, 2014). The paper addresses the measurement model in Figure 1 where an individual's performance over multiple task is used to measure skill. The abstract shows how a measurement instrument of programming skill was constructed and validated:

> 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.

Paper III is entitled "Programming skill, knowledge, and working memory among professional software developers from an investment theory perspective", and was co-authored with Jan-Eric Gustafsson. The paper was published in the *Journal of Individual Differences* (Bergersen & Gustafsson, 2011). The paper extends the validation of the instrument for measuring programming skill in Paper II and investigates the overall results according to Cattell's investment theory (1971/1987). The paper relates to the structural model in Figure 1, where the relation between programming skill and other related variables such as experience and knowledge is investigated. The abstract states:

> This study investigates the role of working memory and experience in the development of programming knowledge and programming skill. An instrument for assessing programming skill—where skill is inferred from programming performance—was administered along with tests of working memory and programming knowledge. We recruited 65 professional software developers from nine companies in eight European countries to participate in a two-day study. Results indicate that the effect of working memory and experience on programming skill is mediated through programming knowledge. Programming knowledge was further found to explain individual differences in programming skill to a large extent. The overall findings support Cattell's investment theory. Further, we discuss how this study, which currently serves a pilot function, can be extended in future studies. Although low statistical power is a concern for some of the results reported, this work contributes to research on individual differences in high-realism work settings with professionals as subjects.

Paper IV, "Evaluating methods and technologies in software engineering with respect to developers' skill level", demonstrates an application of the instrument from the Main

Study in a replicated experiment. The paper was co-authored with Dag Sjøberg and was published in the proceedings of the 5th International Symposium on Evaluation and Assessment in Software Engineering (Bergersen & Sjøberg, 2012). As reflected in the abstract, the paper emphasizes the importance of having measures of programming skill as part of statistical and descriptive analysis when evaluating the benefit of a technology or method:

> It is trivial that the usefulness of a technology depends on the skill of the user. Several studies have reported an interaction between skill levels and different technologies, but the effect of skill is, for the most part, ignored in empirical, human-centric studies in software engineering. This paper investigates the usefulness of a technology as a function of skill. An experiment that used students as subjects found recursive implementations to be easier to debug correctly than iterative implementations. We replicated the experiment by hiring 65 professional developers from nine companies in eight countries. In addition to the debugging tasks, performance on 17 other programming tasks was collected and analyzed using a measurement model that expressed the effect of treatment as a function of skill. The hypotheses of the original study were confirmed only for the low-skilled subjects in our replication. Conversely, the high-skilled subjects correctly debugged the iterative implementations faster than the recursive ones, while the difference between correct and incorrect solutions for both treatments was negligible. We also found that the effect of skill (odds ratio = 9.4) was much larger than the effect of the treatment (odds ratio = 1.5). Claiming that a technology is better than another is problematic without taking skill levels into account. Better ways to assess skills as an integral part of technology evaluation are required.

All four papers present the details of the research related to the overall research problem. Each paper has, thus, a narrow focus which informs the broader discussion provided in this summary.

2 General Background and Fundamental Concepts

This section provides a general background to fundamental concepts related to programming, such as “performance”, “skill”, and “measurement”. However, all these concepts are widely used in everyday life as well as in research literature, often with different meanings and implications, thus presenting a challenge for the present discussion. Synonyms for these concepts are also abundant, which in turn implies that a wide range of other related concepts deserves to be discussed together with the key concepts of this thesis. The specific and actual details on which concepts are investigated in this thesis are provided in Section 3.

Figure 2 shows three philosophical viewpoints on measurement in the context of programming skill: the realist, empiricist, and pragmatist viewpoints. Section 2.1 provides

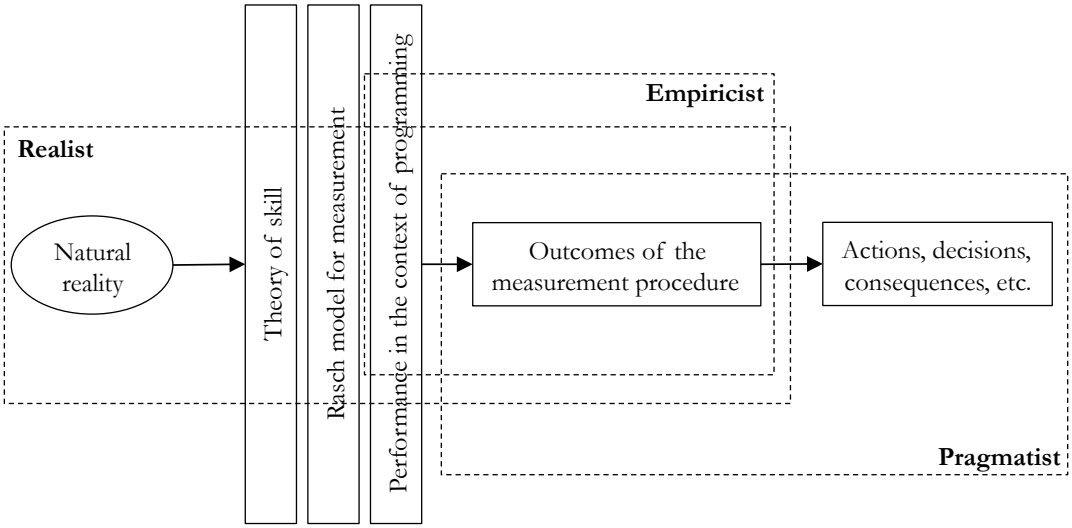


Figure 2: The philosophical view on measurement (adapted from Maul et al., 2013).

a background for central concepts according to the empiricist viewpoint, where the main focus concerns specific issues related to *performance in the context of programming* and *outcomes of the measurement procedure*. Section 2.2 introduces the *theory of skill*, which describes the *natural reality* of skilled behavior across many applied fields (i.e., realist viewpoint). According to this viewpoint, the substantive theory of skill acts as an epistemic layer, which expresses the relation between observable (empirical) outcomes and the natural reality, which may not be observed directly. Next, in Section 2.3, research on programming skill is provided specifically in the context of the (general) theory of skill. Finally, in Section 2.4, the last of the three epistemic layers of the realist view is discussed; the *Rasch model for measurement*.[2] The pragmatist viewpoint, which emphasizes the *actions, decisions, and consequences* of measurement outcomes, is not discussed.

2.1 Research on Programmers and Their Performance

Available research on programmers and their performance is a broad topic that covers many research fields, including particular concepts and research traditions. At the onset, there are challenges to provide a lucid and clearly structured overview of this related work because the terminology in published work is often inconsistent, conflicting, and not well defined. Nevertheless, I will provide a brief overview of related work for the terms ability, aptitude, personality, competency, knowledge, motivation, and expertise (including novice-expert distinctions), as well as growth-based classification of various capabilities.

In this thesis, I use the term capability as a generic term denoting an unspecified capacity for performance. I discuss factors that influence the growth of various capabilities, such as experience, education, and intelligence. I will also introduce performance and pro-

[2]Background theory, focal theory, and data theory are three types of theories that are often used in a thesis. These theories are addressed in, respectively, Sections 2.1, 2.2–2.3, and 2.4 in this thesis.

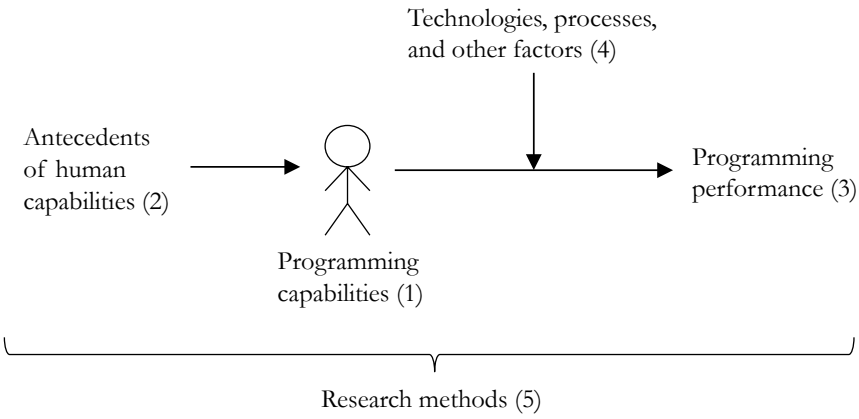


Figure 3: Capabilities of programmers and their performance.

ductivity, which is often discussed in relation to the terms above. Moreover, technologies, processes, and other factors will be addressed together with the research methods and technical assessment frameworks that are often required to study many of the involved concepts.[3]

Research on programmers and their capabilities began soon after the first digital electronic computers were constructed. In the 1950s, card punch operators (McNamara & Hughes, 1955) and the psychological traits of computer programmers (Rowan, 1957) were investigated. In the 1960s, the use of tests in selection and training (Perry & Cantley, 1965) as well as in the general evaluation of computer programmers (Berger & Wilson, 1965; Mayer & Stalnaker, 1968; Oyer, 1969) also received increased research attention (see Simpson, 1973 for an early overview).

Attention to the underlying psychology of computer programming came in focus in the 1970s. Exploratory experiments were conducted on the behavior of programmers (Shneiderman, 1976), and cognitive frameworks describing the skills and knowledge involved in understanding, writing, and maintaining program code (Shneiderman & Mayer, 1979) were suggested. Theories concerning the cognitive processes (Brooks, 1977) and comprehension of computer programs (Brooks, 1983) were also formulated and summarized in books during this time by Shneiderman (1980) and Weinberg (1971).

Looking back at this field, which is more than half a century old, it is clear that research on programmers and their performance covers many different concepts and perspectives. Figure 3 shows one way of structuring some fundamental concepts that are often involved when discussing programmers and their performance. In the center is the programmer who can be characterized according to many different kinds of (1) general psychological capabilities (e.g., intelligence) or more specific capabilities directly related to programming (e.g., knowledge of a specific programming language). Each capability may, in turn, be affected by different (2) antecedents (e.g., programming experience or education may affect the acquisition of programming knowledge). The capabilities of individuals affect (3) programming performance, which in turn may be defined narrowly (e.g., only refer-

[3]The provided exposition is neither complete nor systematic. Instead, I emphasize research that has influenced this thesis with a priority on publications that are based on empirical data.

ring to, say, sorting algorithms) or broadly (e.g., encompassing not only technical, but also process-related and administrative aspects of programming). The relation between capabilities and performance is further affected by (4) technologies, processes, and other factors that may moderate the relationship. Moreover, the (5) research methods used to study the concepts involved may also influence our knowledge about effects that are present in the causal chain leading to programming performance. Each of these concepts is now addressed.

Concerning psychological capabilities (1), a number of terms may be used to characterize the capabilities of an individual in addition to skill. A commonly used classification is to distinguish those that are considered stable (or fixed) from those that are expected to change over time—that is, they are malleable.[4]

Among those capabilities that are considered fixed (or relatively stable) is *general cognitive (mental) ability*, which has been studied using various operationalizations of cognitive ability (see, e.g., Cegielski & Hall, 2006; Linn & Dalbey, 1989; Mayer, Dyck, & Vilberg, 1986). Moreover, research on programming ability has also been conducted, for example, in relation to lab practica (Chamillard & Braun, 2000; Chamillard & Joiner, 2001) and peer ratings (Carver, Hochstein, & Oslin, 2011; see Daly & Waldron, 2004 for an overview.) However, "programming ability" in these works is not theoretically defined but only indicates an unspecified capability for programming.

Programming *aptitude* is a term that is often used to identify the reason that some developers are dispositionally better at becoming good programmers. Tests of programming aptitude were popular in the 1970s (see, e.g., Alspaugh, 1972; Mazlack, 1980), but they appear to have not been used to much extent later due to lack of good results from these tests (Bornat, Dehnadi, & Simon, 2008; Curtis, 1991; Mayer & Stalnaker, 1968). For example, in a sample of over 3,500 students, no incremental validity was found for specialized aptitude tests for programmers over the more general aptitude tests used by the United States Air Force (Besetsny, Ree, & Earles, 1993). Thus, when new tests of programming aptitude are proposed (e.g., Dehnadi, 2006; Harris, 2014; Tukiainen & Mönkkönen, 2002), the main challenge is not whether such tests predict the success in learning to program, but rather that such tests should be better than other tests that are already available for use and, furthermore, may be used in many more situations besides programming.

Personality is yet another relatively stable variable that may explain differences in performance (Acuña, Gómez, & Juristo, 2009; Cegielski & Hall, 2006; Turley & Bieman, 1995; Whipkey, 1984; see Pocius, 1991 for an early review). At the same time, negligible or somewhat inconsistent associations between performance and personality have also been reported (Bell, Hall, Hannay, Pfahl, & Acuña, 2010; Evans & Simkin, 1989; Hannay, Arisholm, Engvik, & Sjøberg, 2010). Nevertheless, support for that conscientiousness, which has in many other areas been found to predict performance, is still somewhat predictive of highly capable software developers (Clark, Walz, & Wynekoop, 2003). There are indications, however, that the personality profile of programmers may not be typical of national norms (Hannay et al., 2010; Pocius, 1991; Whipkey, 1984). This may, in turn, pose a challenge in using personality as an important predictor of performance, for

[4]Within psychology, a fixed capability is sometimes called a "trait", whereas the a malleable one is called a "state" (see, e.g., Carroll, 1993).

example, in a job situation.

In contrast to the fixed variables just discussed, many malleable capabilities have been investigated. A challenge with classifying these capabilities is that synonyms are often used interchangeably. Furthermore, the underlying theoretical framework for the terms used is seldom explicated, making it difficult to ascertain what kind of capability is referred to.[5] For example, Evans and Simkin (1989) provide a summary of studies attempting to predict student "computer proficiency" between 1972 and 1987, but they motivate their work in relation to "programming aptitude". Further discussions on previous work on programming "ability" and "skill" are then provided.

Another example where programming abilities and skills are evaluated empirically is within the evaluation of *competency* (see, e.g., McNamara, 2004). Dijkstra, an authority within computer science, also referred to competence several times in his Turing Award Lecture (1972). However, for many, competence is a "fuzzy" concept, and it therefore does not appear to receive much research attention due to its overlap with concepts such as knowledge, skill, and ability (see generally Le Deist & Winterton, 2005; Stoof, Martens, van Merriënboer, & Bastiaens, 2002, but see Turley & Bieman, 1995 for a study on the identification of important competencies for software engineers). Nevertheless, there are several studies available about the skills and competencies needed by programmers in industrial jobs (Bailey & Mitchell, 2006; Surakka, 2007; Turley & Bieman, 1995).

Programming *knowledge* is another malleable capability that is critical to the successful completion of many software activities. Programming knowledge has been empirically investigated in many different contexts, such as in relation to Bloom's taxonomy (Buckley & Exton, 2003), and with respect to programming courses at universities (see, e.g., Chatzopoulou & Economides, 2010). Nevertheless, programming knowledge is often omitted, intentionally or unintentionally, when addressing what capabilities are important for programmers. An exception is theoretically oriented studies on programming comprehension, where programming knowledge is central (discussed further in Section 2.2).

Motivation is an important capability because human performance cannot be studied if people are not sufficiently motivated to perform. Motivation can also be highly variable. In a systematic literature on motivation in software engineering (Beecham, Baddoo, Hall, Robinson, & Sharp, 2008), 21 different motivators were identified from the literature that positively and negatively affect programming performance to various degrees (see, generally, Latham & Pinder, 2005). In open source projects, increased status, and not payment, may act as an important reinforcing factor of intrinsic motivation (Roberts, Hann, & Slaughter, 2006). From the general literature in psychology it is known that differences in motivation may also be a confounding variable. For example, in their study on intelligence testing, Duckworth, Quinn, Lynam, Loeber, and Stouthamer-Loeber (2011) reported that incentives yielded an average increase of more than half a standard deviation.

Another type of malleable capability is *expertise,* which is typically classified according to the dichotomy *novice versus expert* differences. Although such studies use the same terminology as theoretically oriented studies on expertise (discussed in Section 2.2), the studies mentioned here sometimes differ significantly with respect to how an expert is de-

[5] If a theory of "human capabilities" existed, I would be subject to the same criticism when discussing the term "capability" in this section.

fined. For example, when comparing developers with no prior experience in programming (i.e., a novice), an expert may be defined as anything that deviates from a novice (i.e., a graduate student or an individual with some experience in programming). An example of a study on novice-expert differences is provided by Doane, Pellegrino, and Klatzky (1990) who found that experts (i.e., those with a course in Unix and more than 3 years of experience) performed better than intermediates (i.e., students in upper division computer science courses) who, in turn, were better than novices (i.e., students who just started education in computer science). Other studies comparing students and professionals instead have found that the two groups differ in how they externalize information (Davies, 1993) and that professionals are generally better at focusing on the kernel of programming tasks (Holt, Boehm-Davis, & Schultz, 1987), use exception handling (Shah, Görg, & Harrold, 2010) and use better (and different) comprehension strategies (Burkhardt, Détienne, & Wiedenbeck, 2002).

Growth-based *classifications* describe and explain the phases an individual may move through when a capability increases. Previous work has applied Bloom's taxonomy to describe the stages in the computer science curriculum(Buck & Stucki, 2001) as well as Dreyfus and Dreyfus' (1986) five-step model of skill acquisition in a programming context (see Campbell, Brown, & DiBello, 1992). Related to such work is the *personal software process* (PSP), which describes process elements that a professional programmer should apply (Humphrey, 1996). PSP has also been the target for empirical evaluations (see, e.g., Chen, Hsueh, & Lee, 2011; Prechelt & Unger, 2000). Moreover, attempts have also been made at making a People Capability Model (Curtis, Hefley, & Miller, 1995), similar to the more well-known Capability Maturity Model.

Turning to (2) in Figure 3, a typical antecedent that may affect the acquisition of many programming capabilities is *experience*. In general, experience with a specific software domain, technology, or programming language is more often an advantage when programming than no prior experience. Both breadth and duration of experience are of relevance (Stanislaw, Hesketh, Kanavaros, Hesketh, & Robinson, 1994); for example, ten years of experience with one programming language affects programming capabilities differently than one year of experience in ten different programming languages affects. In addition, a programming problem may also be construed and solved quite differently by experienced rather than inexperienced developers; see, for example, (Adelson & Soloway, 1985) for an early study on the effect of domain experience in software design. Some studies have demonstrated a positive effect of having experience (e.g., Agarwal, Sinha, & Tanniru, 1996; Arisholm & Sjøberg, 2004). Other studies have reported no effect of experience beyond the first few years (e.g., Jeffery & Lawrence, 1979) or no effect of experience (e.g., Jørgensen, 1995; Wohlin, 2002, 2004). Overall, using experience to predict programming performance for students has resulted in mixed results (for a review, see Feigenspan, Kästner, Liebig, Apel, & Hanenberg, 2012). For a professional setting, similar mixed results have also been found. For example, in (Arisholm & Sjøberg, 2004; Zhou & Mockus, 2010), performance increased with experience while in (Sonnentag, 1998) it did not. One study shows that programmers also take on more complex tasks with increased experience (Zhou & Mockus, 2010), thereby making it more difficult to detect the effect of experience on performance when analyzing software repositories. Nevertheless, there are

good theoretical reasons why experience should positively affect the acquisition of some programmer capabilities, even though this effect may not be found in all studies.

Education is a closely related variable to experience. Typically, the goal of education in programming is the acquisition of knowledge of programming, as well as, to some extent, the development of programming skills. Investigations concerning education have been conducted along the lines of general academic performance (Bergin & Reilly, 2005; Butcher & Muth, 1985; Byrne & Lyons, 2001), effect of cognitive, behavioral, and attitudinal factors on learning outcomes (Fincher et al., 2005), knowledge of other programming languages prior to starting education (Hagan & Markham, 2000; Holden & Weeden, 2004), gender (Goold & Rimmer, 2000; Pioro, 2006), academic background (Pioro, 2006), and ability to trace (and explain) code (Lister, Fidge, & Teague, 2009), as well as using the exam scores of friends to predict the score of each student (Fire, Katz, Elovici, Shapira, & Rokach, 2012). Generally, the correlation between job performance and academic grades appear to be modest. A large meta study (Roth, BeVier, Switzer, & Schippmann, 1996) reported an observed correlation of 0.16, which increased to 0.30 after correction for research artifacts (i.e., restrictions of range and criterion unreliability). Nevertheless, an important moderator was found to be the time between graduation and performance measurement. Shortly after the graduation, the correlation between grades and job performance was much higher (1 year, $r = 0.23$, $n = 1{,}288$) than a long time after the graduation (6 years, $r = 0.05$, $n = 866$).

Related to education, one may also investigate broader antecedents to programming capabilities, such as general mental abilities or intelligence. Often, such studies are framed as *predictors* of programmer capacity or programming performance using, for example, grades, SAT, and personality (Whipkey, 1984). Thus, the placement of a capability (such as intelligence) in Figure 3 may sometimes be as an antecedent and sometimes as a capability, depending on the context of a research study. In some situations, feedback loops are also present (see, generally, Waldman & Spangler, 1989). For example, Mayer et al. (1986) investigated whether learning programming skills also improved general intellectual skills, but found little or no effect.

Turning to (3) in Figure 3, performance in general (see, generally, Campbell et al., 1993; Sonnentag & Frese, 2002) and *programming performance* specifically have been the focus of research for many decades. Discussions on the variability in the performance of programmers began late in the 1960s. In what may be the first study on programming performance variability, Grant and Sackman (1967) reported a 1:28 ratio between the highest and lowest performer with respect to the time used to correctly debug a problem (also see Sackman, Erikson, & Grant, 1968). Although concerns were raised regarding both the validity of the study (Lampson, 1967) and the reported ratio (Dickey, 1981; Prechelt, 1999a), the claim that individual differences are one order of magnitude, or more, spread through the research literature (e.g., in McConnell, 1998; Glass, 1980, 2001). Also, in Brooks's seminal article, "No silver bullet", Grant and Sackman are cited when the difference between an average and great designer "approach an order of magnitude" (1987, p. 18). Moreover, according to Glass, "[i]ndividual differences between programmers are immense" (1980, p. 48) and Soloway is quoted as "people matter BIGTIME in programming" (Freeman, 1992, p. 19).

Others also supported the claim of substantial programming performance differences with additional data (Curtis, 1981; DeMarco & Lister, 1985, 1999), reviews of data (Trendowicz & Münch, 2009), and reanalysis of previously published experiments (Prechelt, 1999a). Overall, there were indications that the variability in individual performance was large, and it was also larger than the variability caused by many other technologies or methods that are used to increase productivity. For example, in the book "Software Economics", Boehm (1981) reported that the "people factor" was the largest of all investigated factors in the success of software projects and therefore concluded that developer attributes are by far the best opportunity for improving software productivity. Card et al., who investigated the effect of using different technologies for 22 projects, found a similar result and concluded, "Use experienced, capable personnel. They are a major factor in the productivity and reliability [of the investigated software development projects]" (1987, p. 849). Later, Prechelt compared the variability of programmers with the variability of the effect of using different programming languages and reported that, on average, individual variability was as large, or larger, than the variability among the languages (1999b, 2000). Moreover, Prechelt also confirmed in a more realistic, two-day experiment that teams of developers using different programming languages displayed some degree of variability with respect to solution completeness, product size, robustness, and security (2011).

A concept closely related to performance is *productivity*, which addresses input to a process in addition to its output (i.e., performance). During the 1970s and 1980s, much attention was devoted to programming productivity in general (Jones, 1978; Mills, 1983), as well as determinants of high programming productivity (Boehm, 1981; Chrysler, 1978). The effect of organizational factors (Jeffery & Lawrence, 1985) and properties of program code (e.g., control complexity, see Chen, 1978) were also studied in order to increase productivity. Work focused on issues concerning the measurement of productivity (e.g., Jones, 1997; Walston & Felix, 1977) was often based on counting lines of code that are produced within a given time unit. Nevertheless, it appears that a valid measure of programming productivity has been elusive as there is no readily available way to compare the performance of individuals across different systems. More generally, defining job performance has also been problematic in other fields as well (Campbell, 1990).

Concerning (4) in Figure 3, many *technologies, processes*, and *other factors* moderate the relation between programming capabilities and performance. In contrast to the present work, most studies in software engineering focus on technical aspects and consider human capabilities as a moderating effect of the relationship between a technology, process, or other factor and programming performance. For example, research on topics as different as programming plans (Davies, 1989), structured programming and performance (Lucas & Kaplan, 1976; Sheppard, Curtis, Milliman, & Love, 1979), the role of beacons (Crosby, Scholtz, & Wiedenbeck, 2002), and goal setting (Weinberg & Schulman, 1974) have been studied in relation to factors such as the workplace (DeMarco & Lister, 1985). Also, the role of programming variables represent in the improvement of programming skills (Byckling & Sajaniemi, 2006) and how different ways to implement system control (i.e., centralized versus delegated) affect programming performance (Arisholm & Sjøberg, 2004) have been studied. Moreover, the use of different processes during software development

has been of much interest to researchers. For example, the benefit of pair programming (as opposed to individual programming) have been studied from the perspective of forming the pair based on different levels of expertise (Lui & Chan, 2006) or seniority (Arisholm, Gallis, Dybå, & Sjøberg, 2007). A systematic literature review on pair programming found that, among students, this practice was most beneficial when the pair was comprised of individuals with a similar level of programming skill (Salleh, Mendes, & Grundy, 2011).

Finally, regarding (5) in Figure 3, the *research methods* used to study the concepts described above are sometimes also the focus of investigations. An implicit assumption in studies involving human participants is that individual differences in developer capacities do not bias the results (see, e.g., Kampenes et al., 2009 for a systematic review of current practices in quasi-experiments). For example, the use of self-ratings (as opposed to more objective standardized tests) has been investigated. Using students, it has been found that self-evaluations were on par, or slightly better, than university marks or pre tests (Kleinschmager & Hanenberg, 2011). Ratings of self may work better when people compare themselves relative to each other (which requires them to know each other beforehand) rather than when providing absolute ratings. For example, according to Kruger and Dunning (1999), incompetence in a field may lead to highly inflated self-assessments, typically because accurate metacognition about oneself requires a certain level of skill. Also, Rasch and Tosi (1992) reported a high correlation between self-reported intellectual ability and programming performance. Other methods that have been used are the assessment of, for example, programming skill using multiple-choice knowledge questions (Clark, 2004) and performance on programming tasks in a pretest (Arisholm et al., 2007; Arisholm & Sjøberg, 2004; see Feigenspan et al., 2012 for a review).

Many *technical support frameworks* are related to the research methods involved in the study of various programming capabilities. In such frameworks, a programmer may implement and submit a solution to a programming problem. The framework then automatically calculates one or several numerals that characterize the individual's solution according to some predefined criteria. To make the tests that are supported by the frameworks shorter, some of these frameworks have also been made computer adaptive (see, e.g., Conejo et al., 2004).

In recent years, programming competitions have become popular where individuals or teams compete against each other for prizes or bragging rights (see, e.g., Dagienė & Skūpienė, 2004). In such competitions, technical support frameworks are important because they can evaluate each submitted solution consistently and objectively on multiple quality criteria (e.g., code effectiveness, completeness, etc.) and produce the results almost immediately. However, such automated assessment frameworks have been around for educational purposes for at least five decades (see Hollingsworth, 1960 for early work and Ala-Mutka, 2005; Douce, Livingstone, & Orwell, 2005; Seppälä, 2012 for reviews and overviews).

In the educational context, the purpose of such frameworks is often to provide students with consistent grading (e.g., in an exam situation, see Watson, Li, & Godwin, 2013; Califf & Goodwin, 2002) or feedback (e.g., when learning to write programs; see Seppälä, 2012). Many technical support frameworks have been developed for use in computer science education (see Cheang, Kurnia, Lim, & Oon, 2003; Ellsworth, Fenwick, & Kurtz, 2004;

English, 2002 for examples and Ala-Mutka, 2005; Daly & Waldron, 2004; Joy, Griffiths, & Boyatt, 2005; McCracken et al., 2001 for reviews). In particular, frameworks have been developed to support students in learning test-driven development (see, e.g., Edwards, 2004). Overall, there are many technical challenges related to the automatic evaluation of programming task solutions (see, e.g., Allowatt & Edwards, 2005; Saikkonen, Malmi, & Korhonen, 2001; Truong, Roe, & Bancroft, 2004). Nevertheless, the benefits of using such frameworks are clear, both with respect to reducing student stress and at maintaining student motivation (Woit & Mason, 2003).

In summary, the research on programmers and their performance consists of a multitude of capabilities. In much of the research, the focus has been on programming performance as a dependent variable, rather than the capabilities themselves. Furthermore, for a majority of the studies, the capabilities involved were not theoretically defined. I now turn to one such capability that is of particular interest because it can be theoretically defined, namely skill.

2.2 Theory of Skill

The concept of skill has ancient etymological origins. Skilled behavior can be traced back to Aristotle (1999), who differentiated between epistêmê (i.e., knowledge) and technê (i.e., crafts, art). According to the Merriam-Webster dictionary, the term comes from Old Norse/Middle English where the word *skil/skilen* means to separate, discern, or to distinguish between something; further, skill is defined as

- the ability to use one's knowledge effectively and readily in execution or performance,
- dexterity or coordination, especially in the execution of learned physical tasks, or
- a learned power of doing something competently: a developed aptitude or ability.

The scientific study of skill is said to have begun in 1820 with a study of individuals' accuracy in the recording of astronomy observations (Welford, 1968). During the 1880s, Darwin's cousin, Francis Galton (1883), documented other behaviors that required skill, such as drawing, the assessment of weights, and blindfolded card playing. Over the next half century, further investigations examined skilled activities, such as the learning of Morse code (Bryan & Harter, 1899), typewriting (Book, 1908), and drawing while looking in a mirror (Snoddy, 1926).

In the 1920s, the nature and definition of skill was explicated in greater detail (Bezanson, 1922; Pear, 1927, 1928). Specifically, Pear defined skill as "*an integration of well-adjusted performances*" (1928, p. 611) that should be distinguished from both "ability" and "capacity". A skill may be highly specific, for example, to an occupation or a sport. An early motivation for studying skill was the obvious benefits of having highly skilled military personnel operate machinery during times of war. Thus, skill was studied in both the First (Bezanson, 1922) and Second World Wars (Welford, 1968).

A theory of skill was initially formulated in the late 1960s by Fitts and Posner (1967), based on Fitts' earlier work on this topic (1964; see Proctor & Dutta, 1995; VanLehn, 1996

for overviews). The theory of skill describes three overlapping phases in the acquisition of psychomotoric skill, that is, bodily movement in relation to mental activities. In the first (cognitive) phase, facts about the domain where the developed skill will later be applied are initially acquired. During this phase, the behavior is error prone and slow. During the second (associative) phase, facts required for successful completion of tasks are connected and rehearsed. In the third (autonomous) phase, behavior is fast and almost effortlessly carried out, thus requiring less devoted attention (Shiffrin, 1988) and controlled processing (Shiffrin & Schneider, 1977). Overall, an individual's level of performance increases monotonically through all the three phases, through the speed of performance (i.e., decreased time required to perform), the accuracy or quality of performance (e.g., fewer errors), or a combination of both time and accuracy (see, e.g., MacKay, 1982; Neves & Anderson, 1981 for more details).

Starting in the mid 1970s, Fitts and Posner's work was extended and elaborated within the context of cognitive skill by Anderson (1981, 1987) and colleagues (Pirolli & Anderson, 1985). Today, the theory of skill is central to the Adaptive Control of Thought (ACT) cognitive architecture initially proposed by Anderson (1976) and steadily refined over multiple iterations: ACT* (Anderson, 1982, 1983), ACT-R (Anderson & Lebiere, 1998), and ACT-R 5.0 (Anderson et al., 2004). The level of detail in this cognitive architecture is high, which makes it possible to test specific predictions. In more recent years, functional magnetic resonance imaging (i.e., "brain scanning") has also been used to better understand which parts of the brain are used during the execution of skilled behavior (see Anderson, Anderson, Ferris, Fincham, & Jung, 2009; Fincham & Anderson, 2006). The architecture proposed by Anderson and colleagues appears to mostly support smaller tasks that can be deconstructed into steps. However, many real-world tasks are large and complex. Therefore, efforts to extend Anderson's architecture to support such tasks have been made (see, e.g., Taatgen & Lee, 2003; Taatgen, Huss, Dickison, & Anderson, 2008).

Other researchers have focused on the more general results that arise from theory of skill in relation to other concepts. For example, Ackerman investigated individual differences (1987), determinants (1988), and predictors (1992) of skill acquisition. From such viewpoints, skill is subsumed into the broader theoretical framework for research on psychological abilities (see, e.g., Neisser et al., 1996 for an overview) or intelligence (see, e.g., Sternberg & Kaufman, 1998 for an overview). Views on skill using a hierarchical structure of abilities have also been offered, where some abilities are broad and affect (or predict) a wide range of phenomena (e.g., intelligence) whereas others may be narrow and only relevant to a limited number of highly specialized situations (see, e.g., Gustafsson, 1984 for an overview).

Research on *expertise* and *expert performance* provides a complementary view on skill (see, e.g., Chi, Glaser, & Farr, 1988; Ericsson & Charness, 1994; Ericsson, Charness, Feltovich, & Hoffman, 2006; Ericsson, Krampe, & Tesch-Römer, 1993; Ericsson & Lehmann, 1996; Ericsson & Smith, 1991). One aspect of expertise concerns reliably superior performance on representative tasks (Ericsson, 2006). Not all types of tasks permit reliably superior performance (Shanteau, 1992). In the study of skilled behavior, Ackerman (1987) distinguishes between consistent and inconsistent tasks, where only

consistent tasks permit improved performance as the result of extended practice. Such consistent tasks can be used in the study of expertise and skill (see Shanteau, Weiss, Thomas, & Pounds, 2002 for a general discussion). For example, in an early study on chess playing (Chase & Simon, 1973), experts had superior recall of the positions of chess pieces in well-structured games (as in a real, on-going game) but not for positions that were random. Central to the field of expertise is the amount of deliberate practice (usually a minimum of ten years) required to achieve expert performance (Ericsson et al., 1993). Thus, the field of expertise emphasizes the "nurture" perspective on human capabilities, in contrast to the "nature" perspective, which is often central to the study of (presumably) stable capabilities (see Ackerman, 2014a, 2014b; Ericsson, 2014; Plomin, Shakeshaft, McMillan, & Trzaskowski, 2014 for a recent debate on the two perspectives).

2.3 Research on Programming Skill

Using the same theoretical conceptualization of skill as described in the previous subsection, research has also been conducted on the *acquisition* of skill in programming. For example, detailed analyses of the problem-solving plans by programmers have been reported from a cognitive perspective (Davies, 1989, 1994; Rist, 1989, 1995). In perhaps the most comprehensive investigation, Anderson, Conrad, and Corbett (1989) found that the acquisition of programming skill in LISP required the learning of about 500 if-then rules, which they call "productions". Each production is a specific piece of knowledge required to solve a programming problem using a defined syntax. They found that the acquisition of these productions follows a power-law learning curve, indicating that the improvement in performance is greatest initially. It then increases in a decelerating manner. This implies that the relationship between amount of practice and performance is not linear. However, a linear trend can be observed if both practice and performance are logarithmically transformed. This phenomenon occurs so widely for improvements in performance across domains that it is often referred to as the log-log law of practice (Newell & Rosenbloom, 1981). With increasing practice, Anderson et al. (1989) found a decrease in both coding time and programming errors, approximately following the log-log law.

Furthermore, in a study of novice LISP programmers, Anderson and Jeffries (1985) also found that "the best predictor of individual subject differences in errors on problems that involved one LISP concept was the number of errors on other problems that involved different concepts" (Anderson, 1987, p. 203). Thus, one may use an individual's current level of programming performance on one set of tasks to predict an individual's future level of performance on another set of tasks. This is similar to the "conventional wisdom [that] 'The best indicator of future performance is past performance'" (Wernimont & Campbell, 1968, p. 372). The implication is that if past (or present) performance can be measured, better informed decisions can be made on who is most likely to perform well in the future.

Within programming, research on conceptualizations similar to expertise can be seen as an indirect way to gain information about an individual's level of skill. Because it is fairly easy to identify a programmer with little skill (just find someone who has never written a program before), it is possible to investigate how behavior, due to skill acquisition, changes for novice programmers (Soloway & Spohrer, 1989; see Allwood, 1986 for an early review)

or during the teaching of programming (Lord, 1997; Soloway, 1986; see Robins, Rountree, & Rountree, 2003 for a review). By comparing developers who are known to be novices with more experienced programmers (i.e., often called "experts" even though the criteria for being an expert may vary), it is further possible to gain additional information of how skills improve. For example, Wiedenbeck (1985) investigated novice-expert differences of programmers from a psychological perspective and found that experts could automate some of the subcomponents required to solve a programming task (cf. the third phase of skill acquisition, Section 2.2). Koubek and Salvendy (1991) studied the differences between "experts" and "super experts" and found no difference between the two groups in terms of automation. Using a novice-expert distinction, programming-related activities have been investigated in, for example, debugging (Chmiel & Loui, 2004) and functional versus object-oriented programming (Wiedenbeck & Ramalingam, 1999). Emphasis on the differences in the cognitive processing (Bateson, Alexander, & Murphy, 1987) and mental models (von Mayrhauser & Vans, 1996) for the two groups are also studied.

Generally, one can say that performance increases with higher expertise. But there are situations where novice programmers have been found to outperform experts (see, e.g., Adelson, 1984; Haerem & Rau, 2007). In those situations, factors other than programming expertise may dominate an individual's performance on a task. For example, implementing a calculation may pose a complex mathematical problem rather than a programming challenge. Thus, the needed expertise is within mathematics rather than programming.

An alternative way to using various definitions of "experts" to contrast programmers of different skill levels is to use programming tasks that by their very nature are more frequently encountered by highly skilled programmers. Central to the field of *programming comprehension* (see, e.g., Fix, Wiedenbeck, & Scholtz, 1993; Wiedenbeck, Fix, & Scholtz, 1993; Wiedenbeck, Ramalingam, Sarasamma, & Corritore, 1999), programming tasks concerning software maintenance are present where (often) large amounts of code must be inspected to understand where and how a change to an existing system should be performed. Thus, maintenance tasks depend to a large extent on how well an existing program is comprehended (Burkhardt et al., 2002; von Mayrhauser & Vans, 1995). There is also evidence that strategies used in the comprehension of code change with increasing skill levels (Koenemann & Robertson, 1991), which also indirectly informs how programming skills are acquired and predicted.

Another central predictor of skill is *knowledge* (Chi et al., 1988). That knowledge is central to skill follows directly from phase one of skill acquisition: Kyllonen and Woltz (1989) name this phase "knowledge acquisition".[6] A distinction between knowledge and skill, however, is that knowledge concern declarative facts, whereas skill typically involves a procedural component (Kyllonen & Stephens, 1990). A similar distinction between knowledge and skill is also central to the theory of job performance (see Campbell et al., 1993; see also Miller, 1990).

Within the context of software development, prior research has found that experts not only possess more knowledge, but the knowledge is also better organized (McKeithen, Reitman, Rueter, & Hirtle, 1981; see, e.g., Robillard, 1999 for an overview). In addition,

[6] Some researchers further regard domain knowledge as a central part of adult intelligence (Ackerman, 2000; Rolfhus & Ackerman, 1999).

the experts' representations of facts are more similar to that of other experts than that the representation of facts for novices relative to other novices (also see Sheetz, 2002). An individual's level of knowledge also has an important bearing on how well code is comprehended and subsequently modified. For example, in an early study on programming knowledge, Soloway and Ehrlich found that "advanced programmers have *strong* expectations about what programs should look like, and when those expectations are violated—in seemingly innocuous ways—their performance drops drastically" (1984, p. 608).

Predictors of programming skill acquisition have also been studied using the theory of skill. In a study of 260 students over seven days, Shute (1991) used a wide battery of cognitive tests that are frequently used in the US armed forces (see (Shute, 1992; Shute & Kyllonen, 1990; Shute & Pena, 1990) for additional details). The results confirmed the central role of working memory capacity as an important predictor of programming skill acquisition (also see, e.g., von Mayrhauser, Vans, & Howe, 1997), thereby supporting earlier (Anderson, 1983; Woltz, 1988) and more recent developments in the understanding and modeling of skilled behavior (Anderson et al., 2004). Generally, working memory is the "temporary storage of information in connection with the performance of other cognitive tasks such as reading, problem-solving or learning" (Baddeley, 1983, p. 311; see also Baddeley, 1992). Furthermore, working memory is fairly well understood in terms of how this psychological variable operates because it can be investigated through various technologies for "brain scanning" (see, e.g., McNab & Klingberg, 2008). Generally, the maximum number of "chunks" of information one can temporarily store is 7 ± 2 (Miller, 1956), where a chunk is "any stimulus that has become familiar, hence recognizable, through experience" (Simon, 1990, p. 16). Because working memory is limited, one would benefit from identifying and representing larger and more meaningful chunks when solving a problem.

Overall, the above accounts of the theory of skill provide many theoretical expectations. According to Levin, "[t]here is nothing so practical as a good theory" (as cited in Sandelands, 1990, p. 235). The reason for having a theory is that many expectations can be derived from the theory and subsequently used in empirical testing of results. However, while much is known about skill in general, and a considerable amount of research on the acquisition of programming skill has been conducted, the aspect of skill measurement has not been studied in as much detail. It is important to distinguish conceptually between *how* skills are acquired from *measuring the skill level* of an individual, see (Kyllonen & Stephens, 1990). Thus, to address the latter problem, I will now turn to what the term "measurement" entails.

2.4 Conceptualizations of and Models for Measurement

In an early paper on measurement and software engineering, Curtis stated that "measurement and experimentation are complementary processes. The results of an experiment can be no more valid than the measurement of ... [that which is] investigated. The development of sound measurement techniques is a prerequisite of good experimentation" (1980, p. 1155).

Yet, as stated in the introduction of this thesis, the definition of "measurement" is not the same from common to scientific measurement. The common definition is based on the view that anything can be measured. For example, the statement "[m]easure what is measurable, and make measurable what is not so" (Leidlmair, 2009, p. 214) is often attributed to Galileo Galilei and emphasizes that everything *is* either measurable or *can be made* measurable. Within psychology, a similar view is held by Cronbach: "[i]f a thing exists, it exists in some amount. If it exists in some amount, it can be measured" (1990, p. 34). Although Stevens regards the ratio and interval scale as the scales most properly associated with measurement, he also acknowledges that different measures using the same scale may not be "equally precise or accurate or useful or 'fundamental"' (1946, p. 680). Thus, there are differences also in the precision and accuracy, and therefore also in the usefulness, of a measure.

Scientific measurement holds that some things *cannot* be measured. That is, some things possess certain essential features that make them measurable, whereas other things do not. This quote, often attributed to Einstein in relation to measurement, appears to neatly sum up the definition: "not everything that can be counted counts, and not everything that counts can be counted" (Cameron, 1963, p. 13). Although the counting of units is a basic procedure in fundamental measurement (Krantz et al., 1971), counts are *discrete* (as opposed to magnitudes of quantitative attributes) and are therefore not scientific measures (Kyngdon, 2011). This also emphasizes the importance of *quantities* as far as scientific measurement is concerned. According to Michell (1997), measurement consists of two tasks: the scientific task is to show that the attribute being measured is quantitative; the instrumental task is to determine or estimate the magnitude of an attribute that is shown to be quantitative (see Kyburg, 1984 for a general introduction). Direct tests of whether a variable is quantitative is, for example, available through the cancellation axioms of conjoint measurement theory (Luce & Tukey, 1964). Furthermore, Michell criticizes proponents of common measurement for failing to "explicitly discuss the *empirical commitments* implicit ... regarding the internal structure of the attributes involved. Nor do they discuss ways in which these commitments can be *tested experimentally*" (1997, p. 361, emphasis added; see Michell, 1999 generally).

The use of a single term to refer to two factually different practices is a fallacy (Kelly, 1927). According to Borsboom, the problem with common measurement (as conceptualized by, for example, Classical Test Theory; see Lord & Novick, 1968) is that "measurement occurs more or less by fiat. Consequently, it is meaningless to ask whether something is 'really' being measured, because the fact that numerals are assigned according to [a] rule is the sole defining feature of measurement" (Borsboom, 2005, p. 93). In contrast, a defining feature of scientific measurement is that it is empirically testable and thus refutable. What is therefore needed is a detailed account of how something is measured in a way that is empirically testable. This will, in turn, inform questions about the rigor that is present in a measurement procedure.

In this thesis, a "model for measurement" refers to the detailed account of both the empirical commitments and empirical tests that one may conduct in the process of measurement (see Borsboom, 2006). Together with the structural model, which reveals the concepts involved and shows how they (purportedly) relate (Figure 1), the measurement

model explains which input (i.e., programming performance on multiple tasks) results in which output (i.e., a measure of programming skill) (see Cattell, 1988 generally).

More than 50 years ago, Rasch (1960) used tests to study the reading performance of students. He formulated what is now known as the Rasch measurement model. Along with with later generalizations (Andrich, 1978) and extensions (see, e.g., Mair & Hatzinger, 2007 for an overview), Rasch models have become a practical way to measure psychological abilities according to some of the criteria associated with scientific measures. (Papers II and IV provide additional details about the Rasch model.)

Briefly, the Rasch model conceptualizes abilities using an interval scale where the unit of measurement is the logarithm of the odds (called a logit; see Humphry & Andrich, 2008 for a discussion on this unit in the context of the Rasch model). Both an individual's ability and the difficulty of an item (e.g., a task or question) are on the same interval scale, using a process with some similarities to Thurstone's (1927) early work on comparative judgment. Furthermore, the relation between ability and difficulty is expressed stochastically in the Rasch model, so that when the level of an individual's ability equals the difficulty of an item, the probability of a correct answer is 50%.[7] Although there is a debate on whether measurement can be conceptualized as a stochastic (probabilistic) process or not (see Borsboom, 2005 and Michell, 1999 for two books on opposing views), most who write about experimentation today appear to embraces theories of probabilistic causation (Shadish et al., 2002). Moreover, there is at present disagreement concerning the extent to which the Rasch model can be regarded as a probabilistic version of scientific measurement (see Borsboom & Scholten, 2008; Kyngdon, 2008 for opposing views).

A distinctive feature of the Rasch model is that it is parsimonious with respect to how many parameters it requires. It is generally easier to fit a model to the data when the model can use additional parameters. However, it may reduce the testability of the model. The principle of Occam's razor also suggests that one should strive for parsimony to prevent, for example, statistical noise to be captured as parameters of a model.

In summary, whether something is scientifically measured or not may be of little interest for many practitioners. Yet many important decisions may be based on measures that are not valid. In this thesis, the measurement of programming skill uses the Rasch model, which resembles that of scientific measurement albeit with some limitations.

3 Research Method

This section describes how measurement and programming skill are defined in this thesis and how two measures of programming skill were constructed and validated. From the perspective of research in computer science, Curtis (1984) recommends the following steps as necessary for the measurement of abilities such as skill:[8]

[7] Carroll (1993) provides a general discussion on why it is better to use an individual's quantified (or typical) level of performance, rather than the individual's maximal (e.g., top 1%) performance.

[8] Practices for constructing measures may vary across disciplines, but they often contain similar elements to the three steps suggested by Curtis; see, for example, the approach used within information systems (MacKenzie, Podsakoff, & Podsakoff, 2011) or sociology (Upshaw, 1968).

- A clear definition of the ability that is measured
- A carefully constructed performance scale
- A sound validation of both the ability being measured and the scale

I begin by defining measurement before turning to the three above steps, each in a separate subsection.

3.1 Definition of Measurement

The practices involved in common measurement as opposed to scientific measurement differ on many accounts (Sections 1.2 and 2.4). In particular, the two types of measurement differ in their rigor. When defining measurement, common and scientific measurement can be seen as two positions on a continuum of measurement rigor; the aim is to achieve progress towards scientific measurement.

As a hypothetical baseline for this continuum, a measure of programming skill can be defined as the aggregate of performance an individual displays on a set of programming tasks. The tasks are chosen by someone who has the authority to define such things within the field of programming (i.e., philosophical operationalism, see Bridgman, 1927). Performance can be the time required to solve the task correctly (implying a ratio scale) or the quality of the solution, for example, by counting the number of test cases that the solution passes (implying an ordinal, interval, or ratio scale). Although this definition may permit the investigation of, for example, internal consistency, the individual responsible for the definition is not limited to choosing tasks that correlate with each other in the definition. As such, the empirical testability may be zero. There are also other (obvious) limitations to this baseline. For example, would a different authority within the field of programming define the measurement of programming skill the same way, using the same tasks and scoring of performance?

Table 1 shows the two increments to the hypothetical baseline that were investigated in this thesis. The Pre Study, reported in Paper I, was the first increment. The Pre Study investigated several approaches to how time and quality can be combined as programming performance. If time and quality can be *combined* as programming performance, programming skill can, in turn, be inferred from performance that is not constrained to be defined solely by either time or quality. The Pre Study also used the (measurement) model fit indices of confirmatory factor analysis (see Jöreskog, 1969) to investigate which of the various ways to combine time and quality fit the data best. Moreover, the role of performance for each task was changed from *defining* to *indicating* programming skill; see Cohen (1989) for how the former constitutes a nominal definition and the latter a denotative definition. Finally, the Pre Study demonstrated how the various ways to combine time and quality as performance could be analyzed in terms of correlations with those external variables with which a measure of programming skill can be expected, from theory, to be positively correlated. Overall, the Pre Study improved the baseline definition of measurement with respect to empirical testability, but some challenges were still unresolved.

Table 1: Conceptualizations of measurement

Aspect	Baseline (hypothetical)	Pre Study (Paper I)	Main Study (Papers II–IV)
Model for measurement	Sum-score	Sum-score using confirmatory factor analysis	Polytomous Rasch model
Empirical testability	None	Low	Medium
Measurement scale	Ordinal	Ordinal	Interval
Task difficulty a parameter in the model?	No	No	Yes
Interpretable unit of measurement?	No	No	Yes, logarithm of the odds (logits)
Missing data	Delete individual or impute	Delete individual or impute	Increase the standard error of measurement
Role of performance	Defines skill	Indicator of skill	Indicator of skill

One challenge to both the baseline and the Pre Study, shown in Table 1, was that both definitions of measurement rely on an uninterpretable unit (i.e., the sum score of performance on all the tasks combined). Furthermore, the unit that is used to represent skill uses an ordinal scale, and the measurement model does not represent differences in the difficulty of programming tasks. Moreover, because skill is inferred from multiple observations of programming performance, any missing observation of performance makes measurement of skill difficult for an individual unless an algorithm is used to guess how the individual would have performed (i.e., imputation).

The Main Study, which is reported in Papers II, III, and IV, represents the second increment. By using the Rasch model, the Main Study acknowledges the non-linear relation between performance scores and programming skill. The Rasch model also represents the measured variable using an interval scale. Differences in the difficulties of the tasks are represented as parameters in the model. The Rasch model has previously been used to measure programming ability in several programming languages (Pirolli & Wilson, 1998; Syang & Dale, 1993; Wilking, Schilli, & Kowalewski, 2008). Details on Rasch analysis are provided in Papers II and IV. Paper II provides a general discussion on measurement in software engineering and Paper IV shows how the level of difficulty of programming tasks is placed on the same scale as programming skill using the Rasch model. Papers II and III provide additional details on the limitation of measurement as conceptualized by the Rasch model.

3.2 Definition and Theoretical Model of Programming Skill

To define "programming skill", what is implied by "skill" and "programming" must be specified separately. When the term "skill" is used informally, it may refer to many types of behavior. Some of these behaviors may be inconsistent with each other or rely on circular definitions. An example of a circular definition would be to define skill as a type of ability and define ability as type of skill. However, when skill is defined theoretically, as

Table 2: Skill defined according to several mutually consistent perspectives

Perspective	Description	Details in
Acquisition of skill	Three phases (Fitts & Posner, 1967): cognitive, associative, and authonomous; power law learning function (Newell & Rosenbloom, 1981)	Papers I–IV
Antecedents of skill	The effect of working memory capacity and experience on programming skill acquisition is mediated by programming knowledge; see generally (Cattell, 1971/1987)	Paper III
Skill according to a hierarchical structure	General mental ability at the apex, with a multitude of broader and then narrower factors below (Gustafsson, 1984)	Paper III
Consequences of skill		
In general	Programming skill affects programming performance directly together with programming knowledge and motivation; see generally (Campbell et al., 1993)	Papers II and III
Expertise	Programming skill as one of four aspects of programming expertise ("reliably superior perforamance on representative tasks"); see generally (Ericsson, 2006)	Paper I and III
Job performance	Programming skill as similar to a hands-on test of job performance (i.e., a work sample); see generally (Campbell, 1990)	Papers II

in Section 2.2, one may identify multiple perspectives on skill that are mutually consistent and where each perspective highlights a different aspect of the theoretical definition.

Table 2 shows multiple theoretical perspectives for skill that are addressed in more detail in all four papers. Theories on *acquisition* of skill explain in detail how skill evolves through different phases. Such theories also explicate what skill is and is not, as well as the ways skill can be inferred from performance. *Antecedents* of skill emphasize that other variables affect the acquisition of skill, such as working memory capacity and experience. Thus, by stating the variables that one would expect to influence skill, the definition of skill is explicated further. The antecedents of skill may also be placed in a *hierarchical structure*. Broad, general factors that are placed at the apex of the hierarchy presumably affect a wide range of situations and outcomes (e.g., intelligence). Narrow, specific factors, such as programming skill in a specific language, affect only a limited set of situations and outcomes. Skill can also be described theoretically in terms of the *consequences* of having a skill. For example, skill is a central variable to the theory of performance, the theory of expertise, and in studies of job performance. Consequently, one would expect individuals to be highly skilled who display a high level of performance, who are experts, and who display superior job performance. Overall, the perspectives of Table 2 add to the testability of results because one can expect that a valid measure of skill is consistent with multiple perspectives.

In contrast to the definition of "skill", I know of no commonly accepted definition of "programming". Although programming, in its most general sense, concerns the process of understanding some kind of computational problem and developing a machine-executable solution, such a definition is wide and therefore difficult to study. For example, Skiena and

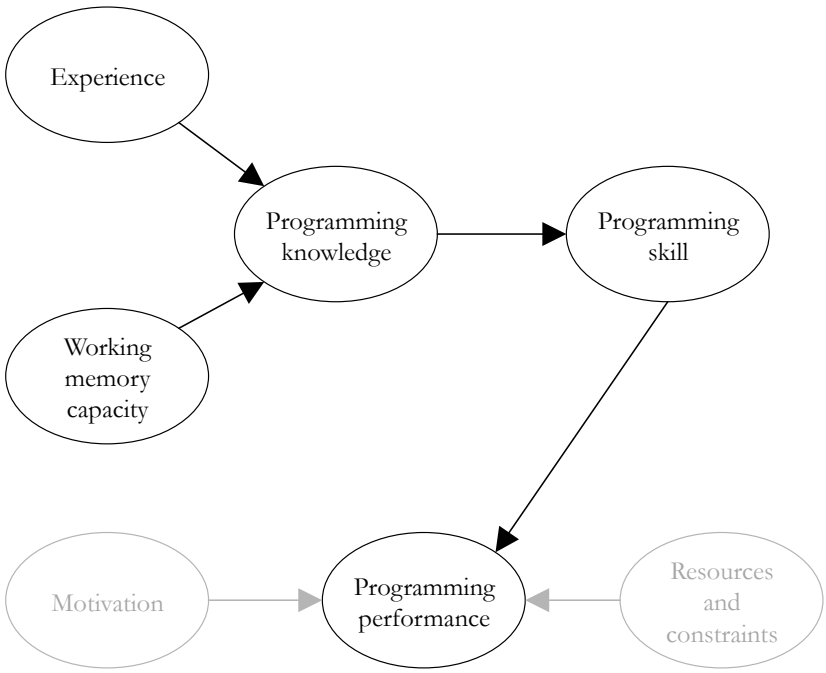


Figure 4: Investigated theoretical model. Concepts not directly investigated are marked in grey.

Revilla (2003) lists 13 main topics and several hundred sub topics of different programming problems. Providing a definition of programming that includes all conceivable activities may therefore be difficult because many widely different practices may be subsumed under such a definition. To study programming in the context of skill, the meaning of the term "programming" was therefore constrained. In Paper II, "programming" was defined as "the activities of writing code from scratch, and modifying and debugging code". It was also specified that the term should refer to one specific programming language, but otherwise be independent of any specific technologies and application domains. In this thesis, the Java programming language was used to study the measurement of programming skills due to the wide usage of this language in both commercial and educational settings.

Figure 4 shows the theoretical model that was investigated. Experience and working memory capacity (a specific type of general mental ability; see Paper III) affect programming knowledge and skills, which in turn affect programming performance. Other abilities, such as working memory capacity and programming knowledge, also affect programming performance. Although motivation, resources, and constraints are important factors for programming performance, controls for these factors were attempted so that they would affect the results to the least possible extent.

The chosen model is based on the theory of performance by Campbell et al. (1993), which has previously been tested using confirmatory factor analysis (McCloy, Campbell, & Cudeck, 1994). In my opinion, the model may be regarded as a simplified version of Waldman and Spangler's (1989) broader model, which outlines the determinants of job performance. However, due to the number of components and the increased complexity of the feedback loops present in their model, a simpler version was investigated; see, for

example, (Locke & Lathham, 1990; Rasch & Tosi, 1992) for alternative models. The investigated model also contains elements from Cattell's investment theory (1971/1987), which has also been investigated empirically (see, e.g., Wittman & Keith, 2004; Wittmann & Süß, 1999).

3.3 Construction of Measures

Instrument construction during the Pre Study used data from four previously published experiments (see Paper I). The construction of the measurement instrument in the Main Study is reported in Paper II. The construction of measures for the Pre Study and Main Study consisted of the four activities shown in Table 3, which are addressed below.

First, the construction and sampling of tasks in the Pre Study was based on previously published tasks. For the Main Study, additional tasks were constructed by two professional developers and by two students. Additional tasks were either taken verbatim from the literature or adapted from the literature. All the tasks differed with respect to origin and time limit, and the methods used to evaluate the quality of solutions also differed. Thus, each task was operationalized differently; see Paper II for an account of the importance of varying such properties of the tasks as well as an overview of the actual tasks that were used. The Main Study also involved software quality aspects that required manual evaluation of quality. The rationale for combining automatic and manual evaluations of quality was that previous work has established that although the two types of evaluations may yield similar results, they also have different strengths and weaknesses that (if possible) should be combined (see Williamson, Bejar, & Hone, 1999). All task materials were pilot tested.

Second, to establish principles to combine time and quality as performance, two data sets were reanalyzed. Data Set 1 was reported in (Arisholm & Sjøberg, 2004) and used only the data from the professionals in the study. Data Set 2 comprises the three data sets in (Arisholm & Sjøberg, 2004; Karahasanović et al., 2007; Karahasanović & Thomas, 2007; Kværn, 2006). Confirmatory factor analysis, as implemented in Amos 18.0, was used to investigate the model fit of various ways to combine time and quality across the programming task of the two data sets.

Third, concerning the subject sampling and data collection, Table 4 shows all the involved subjects and data sets. In addition to Data Sets 1 and 2, another data set (Data Set 3) was used in the Main Study and is discussed in detail in Papers II, III, and IV. As shown in the table, 255 developers from 9 unique countries participated in total, representing 320 person days of programming performance available for analysis. Although some of

Table 3: Activities of the construction phase

Activity	Described in detail (Section)
Construction and sampling of tasks	Paper I (3.1, 3.2) and Paper II (3.2)
Combine time and quality as performance	Paper I (2.4, 3.1, 3.2) and Paper II (3.3)
Subject sampling and data collection	Paper I (3.1, 3.2) and Paper II (3.4, 3.5)
Construction and adjustment of scoring rules	Paper I (3.1, 3.2) and Paper II (3.6, 3.7, 3.8)

Table 4: The investigated data sets

							# Programming tasks	
Data set	*N*	J	I	S	Countries	Study duration	Skill	Treatment
1	99	31	32	36	2	1 day	1	4
2	91	—	—	—	—	1 day	1	3
3	65	19	19	27	8	2 days	12	2
Sum	255	50	51	63	9	320 person days	14	9

Notes. *N* is the number of observations; — is unknown; J = junior, I = intermediate, and S = senior developer (as classified by the company from where the subject was hired to participate); The sum for countries is the number of unique countries; The number of programming tasks that were used are divided into those used to control for skill (e.g., as a pretest) and those who were given as a part of an experimental treatment.

the developers for Data Set 2 were students, the majority of developers were professionals. For the Pre Study, the majority of tasks (4 of 5 for Data Set 1 and 3 of 4 for Data Set 2) contained experimental treatments, whereas only one programming task was initially used for representing differences in programming skill. For the Main Study, this was reversed: the majority of programming tasks (12 of 14) were used to measure programming skill. For all data sets, data (i.e., background questionnaires and all programming tasks) was collected through the Simula Experiment Support Environment (Arisholm, Sjøberg, Carelius, & Lindsjørn, 2002). For all the professionals, the data was collected at the subjects' regular workplace, using their regular tools for software development. In addition, for Data Set 3, about half of the developers ($n = 29$) took a one-hour test of working memory capacity acquired from (Unsworth, Heitz, Schrock, & Engle, 2005). Moreover, most of the developers ($n = 60$) also took a commercially available test of Java programming knowledge purchased from an (anonymous) international test vendor.

Finally, the construction and adjustments of scoring rules according to Table 3 was performed using confirmatory factor analysis (Data Sets 1 and 2) or Rasch analysis (Data Set 3). There were two main requirements when combining time and quality as performance. First, there had to be a correspondence between provided description and the scoring rule for each of the programming tasks. For example, when the description stated that a task had to be functionally correct before additional score points were awarded for spending less time, the scoring rule had to implement this requirement. For Data Sets 1 and 2, the scoring rule was not explicitly stated for any of the tasks. However, the subjects were allowed to leave when the study was completed, thereby imposing some time pressure, which implicitly indicated that one should try to solve the task correctly using as little time as possible. It is nevertheless plausible a lack of correspondence may exist between the scoring rule used and the perceived goal for some of subjects. For example, an individual may have spent more time than required to improve parts of the solution that did not result in a higher score. Conversely, for Data Set 3, all scoring rules were explicitly stated in the task descriptions.

The second requirement when combining time and quality for each of the tasks was that all individuals should display a consistent level of performance across the tasks. The

extent to which this occurred was determined by the fit of the available data to the model of measurement used (see Table 1).

3.4 Internal and External Validation of Measures

The validation of measures proposed in the Pre Study and Main Study was investigated from two perspectives. *Internal validation* concerned the (internal) structure of the programming performance data with respect to each other. For example, is the level of programming performance that an individual shows on one programming task consistent with the level of performance on another task? Conversely, *external validation* inspects correlations, or patterns in correlations, between programming performance data that is aggregated to a measure of programming skill with other, external, variables. The choice of which external variables to include is most frequently derived from theoretical expectations or from prior research. For example, because skills are acquired through practice, one would expect the external variable "amount of practice within programming" to be positively correlated with programming skill.

Table 5 shows an overview of the validation conducted in the Pre Study and Main Study. Because the aim of the Main Study was to improve the results obtained in the Pre Study, the validation in the Main Study was the most comprehensive one. The *psychometric properties* of measurement instruments in software engineering are frequently addressed using scree plots where the signal-to-noise ratio can be visually inspected. Factor loadings and explained variance are also often reported together with internal consistency reliability, such as Cronbach's α, is also common to include.[9] There are, however, several limitations to the use of reliability and what can be deduced from high reliability (Green, Lissitz, & Mulaik, 1977; Schmitt, 1996). The main problem is that high reliability is "required but not sufficient" for validity, while reliability that is too high may indicate that some observations may be redundant. Although one can agree that there should be a discernable "elbow" present in the scree plots, and that more explained variance is better than less, such heuristics are somewhat informal and only provide indirect input into the question of whether something is measured in a valid manner. Other internal validation properties are therefore needed.

A test of *model fit* is one opportunity to more rigorously test whether the data conforms to the expectations of a measurement model. Using the data from the Pre Study, the Root Mean Square Error of Approximation (RMSEA) was used as a test of model fit to evaluate different ways to define programming performance. This approach relies on confirmatory, rather than exploratory, factor analysis from a Structural Equation Modeling perspective; see (Jöreskog, 1969). Additionally, for the Main Study, overall model fit, person model fit (each person's responses over tasks), and task model fit (each task's responses over persons) were investigated, using the indices of model fit available in the Rasch analysis software, Rumm2020, used to analyze the data (Andrich, Sheridan, & Luo, 2006). Yet another way internal validation was conducted in the Main Study was to analyze whether any learning occurred. Learning effects may be detrimental to studies (Shadish et al.,

[9]Concepts that may be complex to measure are also sometimes only represented using a single operationalization, which does not permit the calculations of reliability.

Table 5: Internal and external activities for validation

Type of valiation	What	Details in Pre Study, Paper (Section)	Details in Main Study, Paper (Section)
Internal	Psychometric properties	I (4.2)	II (4.5)
	Model fit	I (4.2)	II (4.3, 4.4)
	Model overfitting		II (4.1)
	Unidimensionality		II (4.2)
External	Correlations with other variables	I (4.1)	II (5.1), III (3.1)
	Prediction		II (5.2), IV (4.1)
	Patterns in correlations		II (5.1), III (3.2)

2002); see (Sheil, 1981) for a detailed discussion on problems with learning effects in the context of programming skill.

The internal validation of the measure developed in the Main Study was further extended by testing whether the tasks that were used to measure programming skill were *unidimensional*. Briefly, the idea of having a set of programming tasks that all measure "the same" (i.e., they are unidimensional) is that subsets of these tasks also measure "the same", albeit with less precision. Unidimensionality was investigated using Smith's test (2002) (see Paper II for details).

Another way to perform an internal validation was to test for *model overfitting*. Briefly, overfitting occurs when the adjustable parameters of a model are tweaked to better account for idiosyncrasies in the data that may not be representative of what is studied (Chatfield, 1995). The ideal is, therefore, to build the model on one set of data and evaluate the model on another set of data (Dahl, Grotle, Benth, & Natvig, 2008; Feynman, 1998). For the Main Study, the data was split into two sets. Two-thirds of the data comprised the construction data set, which was used to construct scoring rules. The remaining third, the validation data set, was used to detect tasks that might have scoring rules that had overfitted the data.

Turning to the external aspects used in validation, the *correlation* between the measure of programming skill and other tests or background variables were investigated for both the Pre Study and the Main Study. Because the Pre Study was based on a reanalysis of data, only a few external variables were available to test whether the measures yielded results in agreement with theoretical expectations. Developer category (i.e., junior, intermediate, senior), self-assessment of programming skills, lines of code (LOC) written, and years of experience are all examples of variables with which one would expect a measure of programming skill to be positively correlated. The Main Study extended the number of external available variables to inspect and included two standardized tests of programming knowledge and working memory capacity. Several variables that should not correlate, or should be negatively correlated, with a measure of programming skill were also included in the external validation of the measure.

The Main Study also investigated how well the measure of skill *predicts* programming performance on other tasks not used to measure programming skill. To make this compar-

ison fair, it was important that the performance on the programming tasks being predicted was independent of any methodical contributions present in the measure of programming skill. Correctness and the time for correct solutions to the four tasks were used in this prediction.

Finally, the Main Study also investigated whether the overall *patterns in correlations* were consistent with theory. First, it was investigated whether variables that are conceptually closer to Java programming skill were also more highly correlated with the measure of programming skill. For example, the variable "Java experience" is a specialization of the variable "general programming experience", which in turn is a specialization of the variable "work experience". One would, therefore, expect these three variables to display a decreasing level of correlation with a measure of Java programming skill. Second, the patterns in correlations were also tested against Cattell's investment theory (1971/1987). According to this theory, one would expect working memory and experience to only *indirectly* influence the acquisition of programming skill through the two variables' direct effects on programming knowledge. Programming knowledge is, thus, a mediator variable between the two variables and programming skill (see Paper III for details).

4 Results

This section describes the results for Research Questions 1–3.

4.1 Combining Time and Quality as Performance

The first research question was, "how can time and quality be combined as performance?" For many typical real-world tasks, performance is a combination of the time spent on the tasks and the quality of the result. Thus, ways to define performance in terms of both time and quality is needed.

Based on a reanalysis of programming performance data, the Pre Study shows that neither quality alone nor time alone yields results that are consistent across tasks. Time and quality must therefore be combined as performance. Combining time and quality were investigated using different ways to combine the two variables. As shown in Paper I, the most consistent results were achieved when performance was first defined in terms of increasing levels of quality, and then in terms of time, when an acceptable level of quality was achieved.

Table 6 shows a hypothetical example of a scoring rule for a task. The rule has four categories for quality, which is operationalized as correctness in this example, and five categories for time. Solutions that are attempted but not submitted, submitted outside the time limit, or clearly do not have any progress towards the quality requirement of the task are scored as zero. Thus, partial scores are not awarded unless the instructions were followed and some progress was made towards the quality requirements. Increasingly higher scores are then awarded until the solution is of "acceptable" quality, irrespective of time. For those solutions that meet this quality requirement, increasingly higher scores are then awarded for those who spent less time.

Table 6: A hypothetical scoring rule for a task

	Time				
Correctness	Limit exceeded	Slow	Medium	Fast	Very fast
Acceptable	0	3	4	5	6
Almost acceptable	0	2	2	2	2
Minimal	0	1	1	1	1
Nothing achived	0	0	0	0	0

The scoring of time and quality according to this structure forces performance to be represented as a categorical, ordinal-scale variable. This reduces the amount of information available in the data, something that is generally not desired. However, there are also clear benefits to the suggested approach: time and quality no longer need to be dealt with as two mutually dependent variables with different scale properties and unknown complex interactions with each other. Additionally, the theory of skill and the scoring of performance are closely related by using this approach. Thus, the use of theory not only guides the development of task materials but also the criteria used in the scoring of tasks (Messick, 1995).

The flexibility of the overall approach by which time and quality are scored as a combined variable of performance was extended in the Main Study. Combinations of computer-based evaluation (e.g., automatic functional and regression tests) and human-based evaluation (e.g., manual inspection of code comprehensibility and use of good object-oriented principles) were applied in the scoring rules, producing a much more diverse set of tasks and scoring rules than those reported in the Pre Study. Moreover, the scoring principle used was also found to be applicable for programming tasks that required both single and multiple submissions (i.e., testlets). In testlet-structured tasks, each submission extends the original solution based on a new, or a slightly more challenging, problem that can be solved incrementally. Because many real-world problems are solved in successive steps, a scoring structure that can support such tasks is beneficial (Millman & Greene, 1989).

The proposed solution is based on using the Guttman scale (2007) in combination with the fundamental principle described by Thorndike and others: "the more quickly a person produces the correct response, the greater is his [ability]" (Carroll, 1993, p. 440). To some extent, the solution is also based on previous work by (Hands, Sheridan, & Larkin, 1999), who investigated the categorization of continuous data for use in the Rasch model. Nevertheless, a challenge with the solution was to determine whether a particular scoring rule was appropriate, because a scoring rule can be defined in many ways. In the Pre Study, the appropriateness of various scoring rules were investigated using the indices of model fit that are available in confirmatory factor analysis, such as the root mean error of approximation. In the Main Study, the Rasch model offered additional possibilities for analyzing the scoring rules for each task, thereby increasing testability further.

In summary, the best results were achieved when time and quality were combined using a Guttman structure. Therefore, higher performance scores should be awarded, first, to

better quality solutions, and, second, to solutions that that use less time. However, to award higher performance scores for time, solution quality must be at an acceptable level, usually operationalized as "correct". A problem with the approach is that there is little guidance for how to decide on the number of score categories that should be used for time and quality. For this, a criterion is needed to evaluate the applicability of a scoring rule. This criterion is addressed in the second research question.

4.2 Measuring Programming Skill from Performance

The second research question was, "how can programming skill be measured from performance?" The results of the Pre Study indicate a reasonably well-fitting model for an ordinal-scale measure of programming skill. Measurements of programming skill using an ordinal scale therefore appear plausible, despite a lack of criteria to use in the validation of the measure. Five other major limitations of the proposed solution for measuring programming skill were also present in the Pre Study:

1. It used an ordinal scale.
2. It was unclear what the measurement unit represent.
3. Reliability was too low to characterize individual differences.
4. Individuals with any missing data had to be removed from analysis.
5. Differences in the difficulty of tasks were not accounted for.

The Main Study aimed to resolve these five limitations. For limitation one, the Main Study conceptualized measurement according to the Rasch model, which uses an interval rather than an ordinal scale. For limitation two, the logarithm of the odds was used as the unit of measurement. For limitation three, additional tasks were included. This improved the internal-consistency reliability of the Main Study. About eight hours' worth of tasks appeared sufficient to characterize individual differences with sufficient reliability to represent individual rather than group differences. For limitation four, the problem of missing data was resolved by the way the Rasch model can handle missing data. When data was missing for an individual in this model, the standard error of measurement increased, but measure of skill could otherwise be regarded as unaffected.[10] Finally, for limitation five, by using the Rasch model, differences in the difficulty of tasks could be accounted for because such differences are represented as parameters in the model.

The Main Study indicated that some types of programming tasks are less suitable for measuring skill than others. Tasks involving concurrent programming (multi-threaded code) could not be successfully integrated into the measure. Further, it was also found that tasks involving debugging (finding errors in code segments) should probably not be used to measure skill, even though such behavior was highly correlated with the measure of skill.

In summary, the Main Study demonstrated a step in the direction of scientific measures of programming skill, where skill is inferred from programming performance. By using

[10] This assumes that the performance on the task in question was not an outlier.

the Rasch model and theory of skill, empirical commitments were made that could subsequently be tested during validation. However, two limitations remained. First, Borsboom and Mellenbergh (2004) point out that successfully fitting data to the Rasch model only indirectly tests whether a psychological variable is quantitative, that is, whether the variable can be scientifically measured or not (see Borsboom & Scholten, 2008; Karabatos, 2001). Second, it is unclear whether one can regard the measure proposed in the Main Study as *valid.* This issue was central to the third research question.

4.3 Validating Measures of Programming Skill

The third research question was, "how can measures of programming skill be validated?" Papers I to III all address aspects of the validation of measures for programming skill. Paper II additionally distinguishes between the term "validation", which is the process used when informing the question of validity, and "validity", which is a property of a measure, see (Borsboom, Mellenbergh, & van Heerden, 2004). APA (1999) lists four aspects of validation as follows:

- Task content
- Response process
- Internal structure
- Correlations with other variables

For the Pre Study, *task content* and *response structure* could not be addressed to any significant extent because the study was based on a reanalysis of available data rather than a theory-driven process, as was the case in the Main Study. Nevertheless, the researchers who designed and implemented the Pre Study probably regarded both the content of the programming tasks and the response process used in solving these tasks relevant to industrial programming to some extent (if they did not, their reported findings would mainly be of academic interest).

For the Main Study, three tasks from the Pre Study were used verbatim. Additionally, 16 new tasks were sampled, constructed, or modified from available literature to further increase the span of the task content. Five of the new tasks in the Main Study were developed by two paid professional programmers, increasing the likelihood that the task content and processes resemble those used when solving industrial problems. Nevertheless, as discussed in Paper II in more detail, to establish that the task content aspect is resolved to satisfaction, a systematic sampling of tasks from a well-defined domain of "industrial programming tasks" should have been present. Regarding the response structure, it is a benefit to have tasks with different origins, authors, and focus represented in the Main Study. However, think-aloud protocols that compared the thought processes used while solving the tasks of the Main Study with those of real industrial tasks should have been available to better inform the validation process.

The *internal structure* of the data was central to both the Pre and Main Study. This structure concerns the internal relationships between test items with respect to, for exam-

ple, dimensionality, reliability, or whether different subgroups of subjects respond differently to subsets of tasks that share a common theme (APA, 1999). The Pre Study investigated the extent to which a coarse aggregation—the sum of task performance scores—for all the tasks in the reported experiments could be treated as a good approximation of skill. The internal structure of the data required that an individual display a consistent level of performance across tasks and, further, that task performance for each task be scored as a combination of time and quality. Unlike the baseline model (Table 1), which regards measurement as "anything goes", the Pre Study used confirmatory factor analysis to investigate the internal structure of the data. According to Jöreskog, confirmatory factor analysis may be a good way to proceed when one "has already obtained a certain amount of knowledge about the variables measured and is therefore in a position to formulate a hypothesis that *specifies* some of the factors involved" (Jöreskog, 1969, p. 183, emphasis added). By using the Root Mean Square Error of Approximation (see, e.g., Loehlin, 2004), the fit of data to the proposed model for measurement could be investigated to indirectly inform the validity of the proposed measure. The overall result for the internal structure of the data yielded somewhat conflicting results: confirmatory model fit, explained variance, and internal consistency reliability favored slightly different approaches when combining time and quality as performance in the Pre Study. Nevertheless, some of the internal indices of fit appeared sufficiently promising to be investigated further in the Main Study.

The Main Study extended the available testable consequences of the internal structure of the data by including

- two data sets, one for construction and one for validation of the proposed measure;
- a measure of programming skill by multiple distinct sets of tasks (unidimensionality);
- inspection of whether noise (residual variance) displayed systematic patterns;
- an investigation of whether performance increases for each new task; and
- other model fit statistics, where individual responses to single and sets of tasks could be investigated.

Overall, no major discrepancies from expectations were detected, even though some minor issues arose. Nevertheless, what was important about the research question was that validation might consist of many perspectives that may all represent "required but not sufficient" conditions in the subsequent validation.

Correlations with other variables in the Pre Study were addressed for such (external) variables as developer category, programming experience, and lines of code (LOC). All the variables confirmed a positive association between the proposed measure of skill and each variable. However, as Paper I illustrates, unless the *absolute* level of correlation is known beforehand between a measure of programming skill and such variables, correlations cannot be used solely to inform the question of measurement validity (see, e.g., Nunnally & Bernstein, 1994).

For the Main Study, Paper II reports how the correlation with programming skill of additional external variables was investigated. For example, programming knowledge is a

concept closely related to programing skill, and it was confirmed that these two variables were highly related. Working memory capacity is another variable that has previously been reported as an important determinant of programming skill acquisition (Shute, 1991), which was also confirmed in the Main Study. Moreover, it was found that the *relative* patterns in the correlation for several of the other variables accorded with their presumed strength correlation with a measure of programming skill. For example, Java experience, programming experience, and (general) work experience were all positively associated with a measure of Java programming skill. Furthermore, Java experience should display the highest correlation with Java programming skill, followed next by programming experience, and then by work experience. This was also confirmed.

Paper III used the patterns in the correlations to test Cattell's (1971/1987) investment theory. This theory describes how experience and general mental abilities affect the performance of individuals indirectly through their mediating effect on the acquisition of knowledge. The results generally accorded with the external correlations of prior research, which adds to the plausibility of the measure. Similarities in results to prior research on job performance were also present, as shown in Paper III.

In summary, multiple instances of support for the validity of the proposed measure of programming skill in the Main Study was found using multiple levels of abstractions (see Figure 1) and perspectives (see Table 2). Combined, one may say that all attempts at validation that do not fail add to the plausibility of the claim that programming skill is measured.

5 Discussion

Programming skills are central to the software industry. Measurement of such skills has therefore obvious and important applications in many situations. However, as indicated in this thesis, it is a challenge to scientifically measure programming skill. This section examines the results in light of the overall research problem of measuring programming skill. Implications for empirical research on programmers are also explored, as is the use of the developed measurement instrument in practice. Limitations and recommendations for further work conclude this section.

5.1 Measuring Programming Skill

I will start with summarizing key points for the three words that compose the main title of this thesis: measuring programming skill.

- *Measuring.* The process of measuring is associated with different practices. In this thesis, I discuss the concept of programming skill according to a scientific meaning, which implies that certain criteria must be either met or acknowledged as limitations. The end goal of having a scientific measure of programming skill was investigated in two increments where each increment provided additional support for the thesis and had fewer limitations.

- *Programming.* The activity of programming consists of a wide range of languages, technologies, and tools that are often supported by various processes and methods used in the development of software. Thus, to investigate the measurement of skill within the specific area of programming, many such combinations were excluded from the definition of programming used in this investigation: "the activities of writing code from scratch, and modifying and debugging code."

- *Skill.* In the lexical definition, used in everyday speech, skill is typically a specialized type of ability that is acquired through experience. Such a dictionary definition is too vague to use in the measurement of programming skill. However, a theoretical definition of skill increases the potential for finding a way to measure skill because the detail of such a definition can be empirically tested. Thus, to measure skill one must agree on what defines skill, even though different research fields may emphasize different aspects of the theory of skill.

The claim of this thesis, that programming skill, to some extent, can be scientifically measured, requires further ontological, epistemological, and methodological considerations. I will address these below.

5.1.1 Ontology

Ontology deals with issues concerning the existence of things independently of our observations (Chalmers, 1999, p. 213). Borsboom et al. (2004) argue that the (ontological) existence of an ability is a prerequisite for the subsequent process of measuring the ability. They further regard measurement and validity as causal concepts: a change in an ability must result in a change in the measure in order for the measure to be valid. Causality is a topic that is often avoided because it may be difficult to demonstrate that a relation between two entities is causal. The problem is that a large number of competing causal models may explain the same set of observations (i.e., the problem of underdetermination). However, there is a difference between a conjecture that a relationship is causal and treating all relationships as non-causal (e.g., by only using correlations). Causal models that provide insight on a topic are, therefore, preferred over non-causal relationships; for example, Porter, Siy, Mockus, and Votta (1998) also make this claim in the area of software inspections. The basis for causality and models supporting causal interpretations have also been improved during more recent years (see, e.g., Pearl, 2000; Glymour, 2001).

Even though the existence of programming skill is granted, there are still uncertainties about what such a skill refers to ontologically. A unidimensional measure of an ability requires that all the variation in the measure must be uniquely determined by the ability (Borsboom, 2008). Thus, it would be tempting to deduce that the ontological origin of this ability is a singleton of some kind (e.g., a single process located in the brain somewhere). However, van der Maas et al. (2011) argue that many psychological variables are based on several distinct (ontological) processes. Because such processes are often practiced together, they appear as unidimensional when analyzing the data. A supporting position for this argument has been made in the study of working memory; Unsworth, Fukuda, Awh, and Vogel state that "working memory tasks require a complex sequence of events

for accurate performance (i.e., encoding processes, maintenance processes, and retrieval and decision processes) [where the] . . . neural activity during the maintenance period alone is a powerful predictor of . . . working memory" (2015, p. 863) (also see van der Maas, Kan, & Borsboom, 2014). The general theory of skill also describes the distinct processes that are involved when skill increases (see e.g.,Anderson, 1982). Thus, I regard it as untenable that the ontological basis for programming skill refers to any single psychological variable or process in the same sense as quantitative variables such as time or length. Still, as shown in this theses, at the abstraction level of programming performance, the measure of these variables and processes appear unidimensional for a subset of Java programming tasks.

A related issue to the ontological basis for programming skill concerns the distinction that Borsboom (2008) makes when he discusses "between-subject" versus "within-subject" differences (also see Borsboom, Mellenbergh, & van Heerden, 2003). The Rasch model expresses differences in skill between individuals. But the instrument that was developed was not devised to, for example, associate a range of skill level with a specific phase of skill acquisition for an individual. As such, no within-subject interpretation of programming skill is offered at present. Moreover, many data patterns in programming performance receive the same skill score in the instrument (for a discussion of this general limitation of the Rasch model, see Borsboom, 2008). Thus, two developers with identical programming skill, as measured by the instrument, can have used different mental processes, strategies, and knowledge *and* at the same time have different patterns in programming performance across tasks (e.g., by one individual displaying a medium level of performance across all tasks whereas the other alternates between high and low performance). Valid methods for identifying such differences in processes, strategies, and programming performance across task should therefore be developed in the future.

5.1.2 Epistemology

The next philosophical consideration concerns epistemology, that is—justified belief that is based on valid knowledge, methods, or scope, as opposed to subjective opinion. In general, one desires validity. Within psychology, validity has traditionally been seen as having different types, such as content validity, criterion validity (predictive and concurrent), and construct validity (Messick, 1989b). Within software engineering, Shadish et al.'s (2002) four types of validity (i.e., construct, internal, external, and statistical conclusion validity) are frequently addressed.

This thesis focused primarily on issues concerning construct validity. According to Shadish et al., construct validity concerns how we can "generalize from a sample of instances and the data patterns associated with them to the particular target constructs they represent" (2002, p. 21). In this thesis, performance on multiple programming tasks are the "sample of instances" and programming skill is the "particular target construct". However, I hesitate to refer to programming skill as a "construct" because of the ambiguity about what a construct is. Borsboom, Cramer, Kievit, Scholten, and Franić has summarized this ambiguity:

> [T]he term 'construct' is used to refer to (a) a theoretical term (i.e., the linguis-

> tic, conceptual, symbolic entity) that we use as a placeholder in our theories ('general intelligence', 'g', 'theta', 'the factor at the apex of this hierarchical factor model', etc.), and (b) the property that we think plays a role in psychological reality and of which we would like to obtain measures (i.e., a linearly ordered property that causes the positive correlations between IQ-tests—assuming, of course, that there is such a property) (2009, p. 150).

In relation to the first understanding of a "construct"—as a placeholder for theoretical terms in theories—it is too easy to believe that because one factor is present in a data set, this factor represents a physical entity (i.e., to reify according to Gould, 1996, which was also discussed in Section 5.1.1). In relation to the second understanding—as an active property of reality that could be measurable—I have argued in this thesis that programming skill has an ontological basis; skills are a part of our physical reality and they do casual work. Consequently, I could use Borsboom et al.'s definition of validity, as stated in Paper II: "A test is valid for measuring an attribute if ... the attribute exists and ... variations in the attribute causally produce variation in the measurement outcomes" (2004, p. 1061).

One consequence of using this definition of validity was to regard each operationalization of programming performance as an indicator of programming skill. Cohen (1989) distinguishes nominal from denotative definitions where the former operationally defines the term (i.e., philosophical operationalism Bridgman, 1927). However, denotative definitions treat operationalizations as examples that partially constrain the meaning of the term. Moreover, such examples are not exhaustive and they are therefore called indicators. Programming skill is therefore not defined solely in terms of the programming tasks of the instrument; new tasks can (and will) be used in the future without changing how programming skill is measured.

Testability is another issue that is related to validity. In general, testability of claims is central to science (Popper, 1968). A further common practice in science is that it the onus is on the person making the claim to provide evidence for a claim. For example, Hitchens formulates the proverb "Quod gratis asseritur, gratis negatur" as "[w]hat can be asserted without evidence can also be dismissed without evidence" (2007, p. 150). Consequently, claims about the measurement of programming skill that does not permit the display of evidence are therefore of limited interest. In my view, many conventional analyses may be too easily satisfied with respect to testability. One achieves increased confidence in the claim that something is actually being measured when one uses a confirmatory factor analysis on a parsimonious measurement model. Confidence can also be increased further, for example, by stating prior to inspection of the data what kind of analysis will be conducted (Wagenmakers, Wetzels, Borsboom, & van der Maas, 2011).

However, to validate measures of programming skill, one must make certain that it is possible to be wrong about the claim that programming skill is measured to begin with. To paraphrase Messick, to be correct about the claim that something is measured, it must also be possible to be wrong about this claim (Messick, 1989b).[11] This answer is certainly

[11] The original quote is: "... one must be an ontological realist in order to be an epistemological fallibilist" (Messick, 1989b, p. 26).

not novel; according to Popper (1968), merit of a scientific claim or theory depends on whether the claim offers testable predictions or not. Thus, being explicit about testable empirical predictions for a measure of programming skill improves the extent to which the measure can be validated. A causal and realist interpretation of the term validity is needed (such as proposed in Borsboom et al., 2004), simply because there are many things one can be wrong about when defining a scientific measure of programming skill.

Despite the research in this thesis, unresolved issues remain. First, no direct tests were conducted to inform the question of whether programming skill is a quantitative variable that permits scientific measurement. Michell's (1997) criticism of the field of psychology, that, in most cases, people fail to address this primary scientific research task, also applies to my work. Nevertheless, more pressing concerns must be resolved before testing whether, when, and how programming skill can be scientifically measured. Bunge captures my overall intent well when he writes about the importance of formulating and challenging theories:

> [P]remature theorizing is likely to be wrong—but not sterile—and . . . a long deferred beginning of theorizing is worse than any number of failures because (1) it encourages the blind accumulation of information that may turn out to be mostly useless, and (2) a large bulk of information may render the beginning of theorizing next to impossible" (1967, p. 384) (as cited in Sjøberg, Dybå, Anda, & Hannay, 2008).

Thus, I would rather be specific but wrong about whether programming skill can be measured than have packaged my thesis so vaguely or made it so untestable that it would be difficult to refute. I concede that additional direct tests that inform the question of the quantitative nature of skill is needed in future work.

A related, unresolved issue concerns whether one is willing to accept scientific measures that are inherently stochastic, rather than deterministic (Borsboom & Mellenbergh, 2004) (see Hacking, 1990 generally). The data of my work shows that developers sometimes fail on tasks when they should have succeeded, given their combined performance on the other tasks. A stochastic definition of measurement fits better in such situations, because a few mistakes (e.g., misunderstanding the task requirements or experiencing a technical problem) do not invalidate the measure. Yet results are more trustworthy when anomalies during a test are reduced to the largest possible extent. Ideally, an individual's skill, as measured by an instrument, should display a deterministic relation to an individual's actual performance.

5.1.3 Methodology

Methodology concerns the system of principles and methods used to conduct research. Software development is a field with rich opportunities for data collection. For example, software repositories may contain historical data on all changes made to a system, all keystrokes made during the development of the system, all accesses made by its users, etc. Software repositories yield massive amounts of data, but the data is usually collected

for a different purpose than to address a specific research question. As a consequence, research is often retrofitted to account for the available data.

An alternative to such data-driven investigations is to use fewer, carefully selected and instrumented variables that are collected according to expectations. Expectations may vary in strength. At one extreme, one has no expectations (e.g., guessing next week's winning lottery number); at the other extreme, one is certain of the outcome (e.g., a stone will fall to the ground when dropped). Between these extremes are many levels. For example, at a low level, one would expect the available data from an investigation to be internally consistent. At a higher level, stronger expectations may arise from prior research or, better, by the use of theory.

The main methodological challenge when data is collected with no clear expectations is as follows: that when faced with "strange" results, one is left with no other good option than to trust one's data. Data may contain errors from many sources and, often, does not fully represent the concepts that a researcher wants to investigate. For example, when studying developer productivity, productivity data based solely on lines of code written per month is associated with many problems (e.g., differences in task complexity or the verbosity of a programming language). Thus, some prior notion of what to expect—preferably from a specific, testable theory—makes it easier to detect situations where data might be problematic, poorly instrumented, or otherwise faulty.

During the instrument *construction*, I used the Rasch model, which can be regarded as a non-substantive theory for the measurement of abilities such as skill. Because this model requires abilities to be measured according to a single dimension, this requirement may be used as an expectation against which one may evaluate the collected data. This expectation was particularly important during the construction of the scoring rules (Section 3, Paper II). The theory of skill also provided expectations during instrument construction, mainly with respect to limiting the possible ways time and quality variables could defensibly be scored as performance. Had no theory been available, the scoring rules would most likely have been overfitted to account for idiosyncrasies in the data.

The theory of skill was also central during instrument *validation*. Overall, I regard the evidence for the theory of skill as sufficiently strong to form a basis for expectations that could be formulated prior to data collection. When there is a match between expectations and subsequent data analyses, the plausibility of of a claim increases. Conversely, when few or no expectations of the data exist, a study may become an exercise in model fitting using a complex model. In such situations, explained variance can usually be increased by reduced parsimony to the point of having an uninterpretable model. I do not argue that a complex model of something, for example, as expressed by a neural network, is not useful for many specific situations. However, when the aim is to express knowledge that can be incrementally developed and generalized across situations, having expectations that can be formulated prior to data collection beneficial.

Another methodological consideration concerns the unit for measuring programming skill suggested in this thesis: the odds that one developer achieves a better solution than another developer. This unit can handle performance that is operationalized as quality given a fixed time, time given a fixed level of quality, or a mix of the two. It is also possible to compare differences between individuals where one developer succeeds at a task and

another developer fails, a comparison that is not meaningful when using time alone. The overall system for measurement that is suggested is not novel. For example, the Elo (1978) rating system of chess players uses a similar approach, where the difference in ratings of two chess players is expressed as probabilities for a win, tie, and loss in each game. In the Elo system, the rated performance in a specific tournament can be contrasted with the individual's rating (i.e., "chess skill"). It is also possible to develop psychometric tests where the goal is to predict the Elo rating of chess skill, such as done by van der Maas and Wagenmakers (2005).

5.2 Implications for Empirical Research on Programmers

This section discusses the implications of generalizability, the use of students versus professionals, and statistical power in empirical studies on programmers.

5.2.1 Generalizability

Generalizability concerns the extent to which inferences drawn from one particular situation hold for other situations that are not studied. An issue in this thesis is whether the results based on the specific populations of developers and programming tasks used in the Pre Study and Main Study can be generalized to the global software industry with its developers and complex software systems. When sampling particulars from a narrow population (e.g., only developers from one particular company, or only tasks relating to one particular system), it may be difficult to generalize to other populations. A way to increase generalizability is therefore to use heterogeneous sampling of particulars (Runkel & McGrath, 1972; Shadish et al., 2002).

In my view, subjects and tasks are two main dimensions of generalizations of particular interest to the constructed instrument. To what extent would the inferences in this thesis hold if different developers and programming tasks had been investigated? Additionally, other aspects may affect generalizability, which I will collect into the dimension context. The subjects of the Pre Study consisted of a mix of professionals from different companies and students from different universities. However, according to the theory of skill as well as research on expertise, one would expect students on average to have less programming skill than professionals. Results obtained for students may therefore not generalize to professionals.

To prevent such a limitation, only paid, professional programmers were used in the Main Study. No attempts were made to sample the subjects as representative of the global population of Java programmers by using, for example, stratified random sampling. Generalizations to the global population of developers are therefore difficult to make based on the data alone. Still, one may argue that the larger the sample size, the higher the likelihood that the sample approximates the target population on important characteristics. Compared to a recent survey on Stack Overflow (2015), one of the leading online forums for programmers globally, the developers of the Main Study had a similar mean age (28 years) as that of the survey (29 years, $N = 26{,}086$). The proportion of subjects with a bachelor of science degree was also similar, with 34% in the Main Study and 38% in the Stack Overflow survey. The largest difference was that 63% of the subjects in the Main

study had a master's of science degree, compared with 18% in the survey.

Similar to the sampling of subjects, neither the Pre Study nor the Main Study sampled representative industrial tasks from strata that previous research had established (I am aware of no research that has identified such strata). Instead, a diverse set of tasks was sampled, because of convenience, from the literature or constructed from scratch. The sampled tasks involved some artificial elements (see, e.g., Hannay & Jørgensen, 2008). This implies that, unlike the sampling of subjects who were all members of the target population, the tasks were not actual industrial ones but rather involved similar cognitive processes as those used in real, industrial tasks. For example, in an experiment on object-oriented programming, a programming task may involve an artificial inheritance problem (e.g., the eating of apples, pears, and other fruits). Although such a task is unrealistic in an industrial setting, the abstraction mechanism used may be the same as the abstract mechanism used in real industrial tasks that involve inheritance. The point is that when the sampled tasks and industrial tasks share characteristics important to the subjects' performance, it may possible to infer an individual's level of performance from the sampled to the industrial tasks. Ferguson (1956) refers to this phenomenon as "transfer". Generalizability is further increased when the sampled tasks are heterogeneous. For example, by only using tasks involving inheritance (i.e., homogeneous sampling), other mechanisms of object orientation are neglected (e.g., encapsulation). It is therefore important to vary task characteristics.

The rationale for such heterogeneity in the sampling of task origin was to increase generalizability: If all tasks met the requirements for measurement in the Rasch model, then there must be a large universe of tasks that could have been sampled. Consequently, generalizations to other tasks would be plausible. If no two tasks met the Rasch model requirements, however, one could probably not generalize across tasks. Overall, the 12 tasks of the final instrument were fairly comprehensive, consisting of 135 files consisting of approximately nine thousand lines of code (of which 7.1k were code and 1.8k were comments).

A large number of contextual factors may affect generalizability, such as technology, system, and organization. Because different combinations of contextual factors may approach infinity (Dybå, Sjøberg, & Cruzes, 2012), the most viable approach is to address how such factors affect generalizability to reduce the effect of the most detrimental factors. For example, the availability of industrial tools may be such a factor. One study on UML found significantly different results when using pen and paper versus a real UML tool (Anda & Sjøberg, 2005). Therefore, subjects of the Pre Study and Main Study used their regular tools for programming (i.e., a computer with an IDE and a web browser) to increase the plausibility that the results generalize to an industrial setting.

One less desirable approach is to argue, with no basis in data or theory, that some specific context factors do not negatively affect generalizability. The problem is that such arguments are speculation. Thus, without data or theory available informing the argumentation, it is more prudent to acknowledge that there are many context factors that one hopes do not negatively affect generalizability. For example, it is unknown whether one can generalize across companies with different sizes and in different sectors (e.g., banking, telecom, e-commerce).

5.2.2 Students Versus Professionals

Students are the typical subjects in empirical studies on programmers. In a survey on experiments in software engineering between 1993 and 2002 involving 5,488 subjects, 87% were students (Sjøberg et al., 2005). A particular problem, however, is the extent to which results that are obtained for students can be generalized to (different groups of) professionals, an issue that concerns external validity (Shadish et al., 2002). The distinction between students and professionals in some cases is marginal (Tichy, 2000). For example, a group of students that are one day short of graduation do not improve their performance over night because they start working as a professional software developer the next day. Neither are students a homogeneous group; large differences may exist within and across universities, countries, and cultures, making generalizations difficult. At the same time, calls for the increased use of professional developers in empirical studies have been around for nearly 30 years (see Curtis, 1986).

Generally, one would expect students as a group to be less skilled than professional programmers on average. This raises two issues. First, is the benefit of a technology or method independent of skill level? If it is, the benefit of a new technology or method would be same for low skill levels as for high skill levels. In Paper IV, an "expertise reversal effect" was reported. The claimed benefit of recursive implementations was only found for low-skilled subjects, whereas the high-skilled subjects had better performance on iterative implementations. Although this result was statistically insignificant, two prior large experiments found similar expertise reversal effects. The experiment reported in (Arisholm & Sjøberg, 2004) found that the benefit of using a "good" object-oriented system was only present for the senior developers, whereas the junior developers displayed better programming performance using the system developed according to "poor" object-oriented principles. The experiment reported in (Arisholm et al., 2007) found that the benefit of using pair programming was present only for the junior developers, whereas the senior developers benefitted most from "solo" programming. If the effect of many processes and technologies are not independent of skill level, this raises major concerns with respect to generalizations.

Second, in some studies students and professionals display equal levels of performance (see, e.g., Höst, Regnell, & Wohlin, 2000; Runeson, 2003; Svahnberg, Aurum, & Wohlin, 2008), for several possible reasons. The students may have acquired some specially required knowledge or skill that helps them excel compared to the professionals (often as a result of taking a course that is taught by the researcher who is responsible for the study). Another possibility is that the professionals are sampled from a company that, on average, has low-skilled employees. The students may also be sampled from a university that, on average, has high-skilled students. Such situations can be identified by using the instrument because the relation between programming skill and the performance on the dependent variable can be inspected. It is also possible that skill or knowledge is not required to display high level of dependent variable performance in the study at all. For example, the opinions of students and professional on a topic may be highly similar, and generalizations may therefore be permissible.

5.2.3 Statistical Power

When conducting experiments, one generally wants to find statistically significant results. Statistical power is a function of three elements (Cohen, 1988): the significance level that is used (usually set at $\alpha = 0.05$), population effect size (the effect of the phenomenon being studied), and sample size (N). Of these, the two first elements are usually fixed. The most obvious way to increase statistical power is, therefore, to increase the sample size. However, a problem is that additional subjects require additional resources (i.e., time, money, effort, etc.) In a systematic review of statistical power in software engineering experiments, Dybå, Kampenes, and Sjøberg (2006) suggest other recommendations for increasing statistical power. The instrument in the Main Study was built specifically to assist with this problem.

One recommendation to increase statistical power is to only investigate relevant variables. In most studies, many other variables besides the effect of treatment may affect the outcome of the study. To obtain a better overall understanding of the phenomenon being studied, such variables should be included. However, how to choose which such variables to include is a problem. In studies where performance is the dependent variable, experience, seniority, and education are often used as proxies for skill. However, such proxies may have low correlations with the dependent variable compared with skill (see Papers II and IV). For example, in a large meta analysis (n = 16,058), McDaniel, Schmidt, and Hunter (1988) reports a moderate correlation between job experience and job performance. The correlation between experience and performance also decreased when experience increased. Overall, the study concluded that "job experience is a better predictor of job performance for low-complexity jobs than for high-complexity jobs" (McDaniel et al., 1988, p. 330).

Another recommendation by (Dybå et al., 2006) is to "reduce measurement error". Measurement error in the dependent variable will reduce statistical power. In Papers I and II, I evaluated one of the tasks that used a score which originally had six categories (scored 0–5). By using the instrument, which allowed an empirical investigation, I found that this task only supported four distinct categories (0–3); "nothing done on the task" and "failure, does not compile" should be collapsed. Similarly, I found no difference between having "minor visual cosmetic anomalies" and a "perfect" solution. Such empirical evaluation of proposed scoring rules of dependent variables in studies can be conducted using the instrument, which in turn may reduce measurement error.

Dybå et al. (2006) also suggests reducing subject heterogeneity to increase statistical power. The instrument can also be used for this purpose by selecting developers with similar skill levels prior to study admission or as a covariate in the study of other variables. For example, the instrument was used to select developers for study on code smells (Sjøberg, Yamashita, Anda, Mockus, & Dybå, 2013). Furthermore, the instrument was used as a covariate in (Kjølberg & Hjorth-Johansen, 2012) to study the moderating effect of education and task complexity on expertise. Moreover, reports of the skill levels of subjects in a study are also useful to present in the descriptive analysis.

A final recommendation by (Dybå et al., 2006) is to "choose powerful statistical tests". Statistical power increases when, for example, matching, blocking, or paired designs (i.e.,

two equally skilled developers are assigned to treatment and control conditions) can be used (Shadish et al., 2002). The instrument may also be used in the design of the study and subsequent statistical analysis of results. According to (Kampenes et al., 2009), in a survey of 113 experiments, less than 5% of the experiments used the results of tests, such as the instrument, in their statistical analysis to increase statistical power.

5.3 Use of the Measurement Instrument in Practice

The use of the instrument can be divided into primary and secondary applications. *Primary applications* concern situations where the goal is to measure differences in programming skill between individuals or group of individuals. For example, outsourcing is increasingly becoming a central factor to the software engineering occupation (Meyer, 2006). During the last three years, a commercial version[12] of the instrument was used by two large Nordic companies to evaluate developers from different outsourcing vendors. The companies also compared the skills of the developers from outsourcing vendors with the skill of their own employees. The results informed which developers were hired and allocated to which projects, by ensuring, for example, that the most complex projects had sufficiently skilled developers.

Other primary applications of the instrument may be for in-house hiring, where the instrument may supplement a technical interview. Another application is for external recruiters who are not necessarily versed in computer science or software engineering. Moreover, the instrument may be used during training to determine which developers need to improve their programming skills prior to beginning a job for a customer. Measures of programming skill may also be requested by individual programmers (e.g., to earn bragging rights) or companies who want justify higher prices for their senior consultants. Finally, the instrument may be used by schools and universities for a variety of assessments. This is an area with a large potential, but a further discussion on this issue is beyond the scope of this thesis.

Secondary applications of the instrument concern situations where one wishes to understand how programming skill affects or interacts with some *other* variable of interest. An example of a secondary application is to better understand how programming skill interacts with software cost estimation. It is also possible to study the extent to which programming skill is a major determinant of team performance in contrast to other teamwork quality variables (e.g., processes), and also to what extent programming skill interacts with the quality of requirements analysis, design, testing, documentation, and other software engineering activities.

For both primary and secondary applications, the utility of the instrument depends on the extent to which it can predict important criteria. For example, when used in an industrial setting, if both current or future job performance can be predicted to a large extent, the utility of the instrument would be high. Conversely, if the instrument

[12]The research version of the instrument was based on several years of design and implementation and consists of approximately 30,000 lines of code. Some early prototype development was carried out by two master's of science students (Salicath, 2008; Sørlie, 2007). The reworked commercial version consists of approximately 110,000 lines of code and 20,000 lines of comments, which took approximately 3.5 years (full-time) to develop.

does not correlate positively with some parts of job performance, the utility would be zero. However, for this to be true, all the programming-related variables that were studied in this thesis would have to be unrelated to actual job performance, which is implausible. Another requirement for utility is that there must be variability in the skill of the developers that are being tested. For example, the utility would be zero if no variability in programming skill exists among a set of candidates for a job or if only one candidate applies for the job (see, e.g., Schmidt, Hunter, McKenzie, & Muldrow, 1979). For the sample of developers in the Main Study, the variability in skill was medium to large, according to conventions of effect size in software engineering (Kampenes, Dybå, Hannay, & Sjøberg, 2007) or the behavioral sciences (Cohen, 1988). Consequently, utility may be present in situations where a sufficient number of candidates or teams are evaluated for a position or project, respectively.

This thesis also suggests that the utility of using measures of programming skill is not uniform. As discussed in more detail in Paper II, low skill levels of developers were relatively well predicted by the programming knowledge test. However, when skill increases, the association between programming skill and knowledge decreases. Thus, the instrument appears to have the highest utility when programming skill is medium or high.

At a more detailed and technical level, the utility of the instrument is also influenced by other factors. In the early years of the psychology discipline, Thurstone (1926) stated several requirements for psychological tests that were regarded unresolved challenges at the time. The challenge of both operationalizing performance in different ways without affecting the measure of skill as well as having a measure that is robust to missing data has already been addressed in RQ1 and RQ2, respectively, and will not be repeated here. Some of Thurstone's additional challenges can be rephrased within the context of the present research problem:

- *Invariant comparison of skill across different tasks.* Even if two individuals are not administered the same tasks, it is still possible to compare their measures of programming skill on the same scale. This ensures that different new tasks may be used to measure programming skill in the future when the existing tasks become well known, irrelevant, or otherwise unusable. By the properties of the Rasch model, the instrument can be updated with new tasks in the future. This would require a gradual introduction of new tasks with unknown difficulty parameters that are solved alongside some of the existing tasks with calibrated difficulty parameters. (An individual's skill in an actual administration of the instrument is, of course, estimated from the tasks with calibrated difficulty parameters.)

- *Increase measurement precision.* In the Rasch model, a programming task that is too easy or too difficult does not provide the same level of information about an individual's level of skill as a moderately difficult task. An individual is also not motivated to solve a task that is too easy or too difficult. By the properties of the Rasch model, the instrument can support computer adaptive testing in the future. In such testing, tasks with an optimal level of difficulty can be presented on an individual basis to optimize measurement precision.

- *Detection of abnormal response patterns in programing performance.* It should be possible to detect when an individual, or groups of individuals, displays patterns in performance that are inconsistent with expectations. Such anomalies should be flagged and investigated, for example, to determine the existence of motivational issues, cheating, or other problems. Anomalies may also be indicative of highly specialized skills for some parts of the software development stack. At present, the instrument can use the person fit residuals from the Rasch model to determine which tasks have worse and which have better performance than expected.

In summary, valid measures of programming skill are useful for a wide range of applications. Having measures of programming skill available will not only increase confidence in results, but will also affect the very research questions that can be asked.

5.4 Limitations

The main limitations of this work are described in the individual papers. A summarized version is provided below.

For the Pre Study, the data sets that were reanalyzed contained sources of unintended systematic error variance due to the treatments of the experiments. Furthermore, the categorization of time during the construction of scoring rules is generally undesired because information is lost. There may also be other undesirable consequences of using cut points for variables such a correctness (see MacCallum, Zhang, Preacher, & Rucker, 2002). Such variables may vary in degree, but are often (operationally) forced into two or more categories. Better ways to combine time and quality may also be available, such as suggested by Maris and van der Maas (2012).

Although the Main Study is large according to the standards for software engineering reported in (Sjøberg et al., 2002), the measurement models employed in this work are demand a great deal of data. Authoritative work using the Rasch model with about 70 subjects has been reported (Wright & Masters, 1979), but a sample size of about 250 subjects would be needed for more conclusive results (Linacre, 1994). Due to the low sample size, I have refrained from absolute claims about whether the data fit the Rasch model or not. For example, no absolute test of fit was reported in Paper II. Instead, I focused on issues that appear to improve or decrease fit, either compared with results that can be obtained by simulated data using (Marais & Andrich, 2012) as an absolute criterion of fit or by comparing alternative versions of the instrument with each other, as a relative criterion of fit.

The extent to which the response processes used by subjects during the solving of the tasks are similar to the response processes used when solving industrial tasks was not investigated. Although this limitation is essential to inform the question of validity according to APA (1999), the construction of measures requires both a different design as well as subject samples (Feynman, 1998) than the validation of measures does. The implications of the validity of the measures of skill developed in this study thus have the same limitations as other single studies, which are strengthened mostly by independent verification in follow-up studies. Other testable elements in the theory of skill were also not investigated in this thesis. For example, one could have designed a study where one

can observe and classify behavior during programming according to the three phases of skill acquisition (Section 2.1), for example, using observation and think-aloud protocols. For a valid measure of skill, one would expect developers with low, medium, and, high skill to display behavior consistent with, respectively, the first, second, and third phase of the theory of skill acquisition.

It should also be clear that measures of skill, such as described in this thesis, are of the maximum capacity an individual can display, not necessarily the level of performance an individual displays, or chooses to display. To illustrate this point, McClelland states that the amount of "beer a person can drink is not related closely to how much he does drink" (1973, p. 11).

A final limitation is that no direct test of whether programming skill is a quantifiable variable was investigated. Although this limitation is shared many other studies in software engineering that employ the term "measurement" in conjunction with a concept that is studied, acknowledging this limitation emphasizes the importance of directing attention to whether the concepts that are studied can be regarded as "measurable" or not. Indeed, as delineated by the representational theory of measurement, the measurement instrument does not meet the requirements for scientific measures, In fact, according to Michell (2008), if programming performance could be considered a quantitative variable, this in itself does not demonstrate that programming skill is a quantitative variable. I therefore reemphasize that unresolved challenges do exist that should receive more research attention; these entail future work.

5.5 Further Work

In addition to improving issues related to the limitations of the previous section, further work can go in several directions. First, attention should be devoted to joint industry-research projects when professionals not only use the instrument for measuring skill but also contribute to the development of new tasks.

Second, more work is needed to investigate the ethical and social responsibilities associated with the development and use of tests (APA, 1999). Using Messick's classification (1989a), this thesis has focused on the *evidential basis* of test interpretation and test use; the, subsequent, *consequential basis* requires more attention. For example, when is a fast and approximately correct solution better than a meticulously developed solution of high quality? Furthermore, what would be the consequences of using the measure of programming skill as part of a grade at a course at a university? Clearly, measures of programming skill are no panacea for all the present challenges in the software industry. However, with careful consideration and appropriate use, a test of programming skill may positively contribute to the improvement of current practices.

A third possibility for future work regards the topic of software complexity. Generally, programming behavior needs to be better understood before software complexity can be measured (Kearney, Sedlmeyer, Thompson, Gray, & Adler, 1986). Software quality is also considered difficult to measure (Kitchenham & Pfleeger, 1996) and attempts have been made at addressing the relation between code complexity and difficulty in maintaining such code (Lanning & Khoshgoftaar, 1994; see Maynard & Hakel, 1997; Wood, 1986 for

important distinctions related to task complexity). Also, if it is true that program size is the main variable in software system complexity (see, e.g., Sjøberg, Anda, & Mockus, 2012), to reduce cost and improve deliverability of a project, the skills of employees should be enhanced while reducing system complexity through an increased focus on reducing system size.

A fourth avenue of investigation is how to decrease the time needed to conduct a test without affecting measurement precision. For example, for a task with a time limit of 30 minutes, no discernable progress may be made towards a solution after 15 minutes. In this instance, it would be more time conserving to terminate this task and present a less difficult task instead. Further research might also investigate ways to increase the number of data points that may be used to infer skill while keeping the time limit constant. For example, in a study of keystroke fluency and programming performance, it was found that for some types of operations, the best programmers are faster on the keyboard (Thomas, Karahasanovic, & Kennedy, 2005). Implementing such multiple perspectives on the same data may therefore yield ways to increase precision of the instrument, as long as the methods can handle that such multiple sources of data collection for a single task are not statistically independent observations.

Finally, it may be possible to extrapolate what constitutes no (zero) programming skill. This may, in turn, make ratio comparisons of skill possible. The present investigation failed to integrate the famous Hello World task, it could be used as an operational definition of zero skill (say, when one has less than 1% probability of solving this task in ten minutes). However, direct tests of the quantitative structure of programming skill (or other skills more generally) are also needed, for example, along the lines proposed by Karabatos (2001). Alternatively, the purported measure of programming skill could be degraded, perhaps using latent class analysis or nonparametric item response theory (e.g., Sijtsma & Molenaar, 2002).

6 Concluding Remarks

This work investigated the extent to which programming skill can be measured. Knowing more about the programming skill level of software developers is important to make well-informed decisions. Increased scientific rigor in the definition, measurement, and validation of psychological variables, such as skill, is therefore required. Using the gold standard of scientific measurement from the natural sciences, this research shows that a valid instrument for measuring programming skill can be achieved for use in both industry and research.

The instrument created in the course of this research can be improved in many ways to facilitate more widespread use. High-quality data from real use of the instrument in the software industry is needed to help identify the most promising opportunities for improvement. A commercial version of the instrument has already been developed and is used in newly started industrial collaborations.

References

Abran, A., Moore, J. W., Bourque, P., Dupuis, R., & Tripp, L. L. (Eds.). (2004). *SWEBOK: Guide to the software engineering body of knowledge*. Los Alamitos, CA: IEEE Computer Society.

Ackerman, P. L. (1987). Within-task intercorrelations of skilled performance: Implications for predicting individual differences (a comment on Henry and Hulin, 1987). *Journal of Applied Psychology*, *74*, 360–364.

Ackerman, P. L. (1988). Determinants of individual differences during skill acquisition: Cognitive abilities and information processing. *Journal of Experimental Psychology: General*, *117*(3), 288–318.

Ackerman, P. L. (1992). Predicting individual differences in complex skill acquisition: Dynamics of ability determinants. *Journal of Applied Psychology*, *77*(5), 598–614.

Ackerman, P. L. (2000). Domain-specific knowledge as the "dark matter" of adult intelligence: Gf/Gc, personality and interest correlates. *Journal of Gerontology*, *55B*, P69–P84.

Ackerman, P. L. (2014a). Facts are stubborn things. *Intelligence*, *45*(July-August), 104–106.

Ackerman, P. L. (2014b). Nonsense, common sense, and science of expert performance: Talent and individual differences. *Intelligence*, *45*(July-August), 6–17.

Acuña, S. T., Gómez, M., & Juristo, N. (2009). How do personality, team processes and task characteristics relate to job satisfaction and software quality? *Information and Software Technology*, *51*(3), 627–639.

Adelson, B. (1984). When novices surpass experts: The difficulty of a task may increase with expertise. *Journal of Experimental Psychology: Learning, Memory, and Cognition*, *10*(3), 483–495.

Adelson, B., & Soloway, E. (1985). The role of domain experience in software design. *IEEE Transactions on Software Engineering*, *SE-11*(11), 1351–1360.

Agarwal, R., Sinha, A. P., & Tanniru, M. (1996). The role of prior experience and task characteristics in object-oriented modeling: An empirical study. *International Journal of Human-Computer Studies*, *45*(6), 639–667.

Ala-Mutka, K. M. (2005). A survey of automated assessment approaches for programming assignments. *Computer Science Education*, *15*(2), 83–102.

Allowatt, A., & Edwards, S. (2005). IDE support for test-driven development and automated grading in both Java and C++. In *Proceedings of the 2005 OOPSLA Workshop on Eclipse Technology eXchange* (pp. 100–104).

Allwood, C. M. (1986). Novices on the computer: A review of the litterature. *International Journal of Man-Machine Studies*, *25*(6), 633–658.

Alspaugh, C. A. (1972). Identification of some components of computer programming aptitude. *Journal for Research in Mathematics Education*, *3*(2), 89–98.

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. (1999). *Standards for educational and psychological testing*. Washington, DC: American Educational Research Associa-

tion.

Anda, B., & Sjøberg, D. I. K. (2005). Investigating the role of use cases in the construction of class diagrams. *Empirical Software Engineering*, *10*(3), 285–309.

Anderson, J. R. (1976). *Language, memory, and thought.* Hillsdale, NJ: Erlbaum Associates.

Anderson, J. R. (Ed.). (1981). *Cognitive skills and their acquisition.* Hillsdale, NJ: Lawrence Erlbaum.

Anderson, J. R. (1982). Acquisition of cognitive skill. *Psychological Review*, *89*(4), 369–406.

Anderson, J. R. (1983). *The architecture of cognition.* Cambridge, MA: Harvard University Press.

Anderson, J. R. (1987). Skill acquisition: Compilation of weak-method problem solutions. *Psychological Review*, *94*(2), 192–210.

Anderson, J. R., Anderson, J. F., Ferris, J. L., Fincham, J. M., & Jung, K.-J. (2009). Lateral inferior prefrontal cortex and anterior cingulate cortex are engaged at different stages in the solution of insight problems. *Proceedings of the National Academy of Sciences of the United States of America*, *106*(26), 10799–10804.

Anderson, J. R., Bothell, D., Byrne, M. D., Douglass, S., Lebiere, C., & Qin, Y. (2004). An integrated theory of the mind. *Psychological Review*, *111*(4), 1036–1060.

Anderson, J. R., Conrad, F. G., & Corbett, A. T. (1989). Skill acquisition and the LISP Tutor. *Cognitive Science*, *13*(4), 467–505.

Anderson, J. R., Farrell, R., & Sauers, R. (1984). Learning to program in LISP. *Cognitive Science*, *8*(2), 87–129.

Anderson, J. R., & Jeffries, R. (1985). Novice LISP errors: Undetected losses of information from working memory. *Human-Computer Interaction*, *1*(2), 107-131.

Anderson, J. R., & Lebiere, C. (1998). *The atomic components of thought.* Mahwah, NJ: Erlbaum.

Andrich, D. (1978). A rating formulation for ordered response categories. *Psychometrika*, *43*(4), 561–573.

Andrich, D., Sheridan, B., & Luo, G. (2006). Rumm2020 [Computer software manual]. Perth.

Arisholm, E., Gallis, H., Dybå, T., & Sjøberg, D. I. K. (2007). Evaluating pair programming with respect to system complexity and programmer expertise. *IEEE Transactions on Software Engineering*, *33*(2), 65–86.

Arisholm, E., & Sjøberg, D. I. K. (2004). Evaluating the effect of a delegated versus centralized control style on the maintainability of object-oriented software. *IEEE Transactions on Software Engineering*, *30*(8), 521–534.

Arisholm, E., Sjøberg, D. I. K., Carelius, G. J., & Lindsjørn, Y. (2002). A web-based support environment for software engineering experiments. *Nordic Journal of Computing*, *9*(3), 231–247.

Aristotle. (1999). *Nicomachean ethics* (Second ed.; T. Irwin, Trans.). Indianapolis: Hackett Publishing.

Baddeley, A. (1992). Working memory. *Science*, *255*(5044), 556-559.

Baddeley, A. D. (1983). Working memory. *Philosophical Transactions of the Royal Society of London. B, Biological Sciences*, *302*(1110), 311–324.

Bailey, J., & Mitchell, R. B. (2006). Industry perceptions of the competencies needed by computer programmers: Technical, business, and soft skills. *Journal of Computer Information Systems*, *27*(2), 28–33.

Baker, D. P., & Salas, E. (1992). Principles for measuring teamwork skills. *Human Factors*, *34*(4), 469–475.

Bateson, A. G., Alexander, R. A., & Murphy, M. D. (1987). Cognitive processing differences between novice and expert computer programmers. *International Journal of Man-Machine Studies*, *26*(6), 649–660.

Beaver, J. M., & Schiavone, G. A. (2006). The effects of development team skill on software product quality. *ACM SIGSOFT Software Engineering Notes*, *31*(3), 1–5.

Beecham, S., Baddoo, N., Hall, T., Robinson, H., & Sharp, H. (2008). Motivation in software engineering: A systematic literature review. *Information and Software Technology*, *50*(9–10), 860–878.

Bell, D., Hall, T., Hannay, J. E., Pfahl, D., & Acuña, S. T. (2010). Software engineering group work: Personality, patterns and performance. In *Proceedings of the 2010 Special Interest Group on Management Information System's 48th Annual Conference on Computer Personnel Research* (pp. 43–47).

Berger, R. M., & Wilson, R. C. (1965). The development of programmer evaluation measures. In *Proceedings of the 3rd Annual Computer Personnel Research Conference* (pp. 6–17). ACM.

Bergersen, G. R., & Gustafsson, J.-E. (2011). Programming skill, knowledge and working memory capacity among professional software developers from an investment theory perspective. *Journal of Individual Differences*, *32*(4), 201–209.

Bergersen, G. R., Hannay, J. E., Sjøberg, D. I. K., Dybå, T., & Karahasanović, A. (2011). Inferring skill from tests of programming performance: Combining time and quality. In *Proceedings of the 5th International Symposium on Empirical Software Engineering and Measurement* (pp. 305–314).

Bergersen, G. R., & Sjøberg, D. I. K. (2012). Evaluating methods and technologies in software engineering with respect to developer's skill level. In *Proceedings of the 16th International Conference on Evaluation & Assessment in Software Engineering* (pp. 101–110). IET.

Bergersen, G. R., Sjøberg, D. I. K., & Dybå, T. (2014). Construction and validation of an instrument for measuring programing skill. *IEEE Transactions on Software Engineering*, *40*(12), 1163–1184.

Bergin, S., & Reilly, R. (2005). Programming: Factors that influence success. *ACM SIGCSE Bulletin*, *37*(1), 411–415.

Besetsny, L. K., Ree, M. J., & Earles, J. A. (1993). Special test for computer programmers? not needed: The predictive efficiency of the Electronic Data Processing Test for a sample of Air Force recruits. *Educational and Psychological Measurement*, *53*(2), 507–511.

Bezanson, A. (1922). Skill. *The Quarterly Journal of Economics*, *36*(4), 626–645.

Boehm, B. W. (1981). *Software engineering economics*. Upper Saddle River, NJ: Prentice-

Hall.

Book, W. F. (1908). *The psychology of skill with special reference to its acquisition in typewriting*. Missoula, MT: University of Montana.

Bornat, R., Dehnadi, S., & Simon. (2008). Mental models, consistency and programming aptitude. In *Proceedings of the 10th Australasian Computing Education Conference* (pp. 53–61).

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

Borsboom, D. (2006). When does measurement invariance matter? *Medical Care*, *44*, S176–S181.

Borsboom, D. (2008). Latent variable theory. *Measurement*, *6*(1), 25-53.

Borsboom, D., Cramer, A. O. J., Kievit, R. A., Scholten, A. Z., & Franić, S. (2009). The end of construct validity. In R. W. Lissitz (Ed.), *The concept of validity: Revisions, new directions, and applications* (pp. 135–170). Charlotte, NC: Information Age Publishing.

Borsboom, D., & Mellenbergh, G. J. (2004). Why psychometrics is not pathological: A comment on Michell. *Theory & Psychology*, *14*(1), 105–120.

Borsboom, D., Mellenbergh, G. J., & van Heerden, J. (2003). The theoretical status of latent variables. *Psychological Review*, *110*(2), 203–219.

Borsboom, D., Mellenbergh, G. J., & van Heerden, J. (2004). The concept of validity. *Psychological Review*, *111*(4), 1061–1071.

Borsboom, D., & Scholten, A. Z. (2008). The Rasch model and conjoint measurement theory from the perspective of psychometrics. *Theory & Psychology*, *18*(1), 111-117.

Briand, L., El Emam, K., & Morasca, S. (1996). On the application of measurement theory in software engineering. *Empirical Software Engineering*, *1*(1), 61–88.

Bridgman, P. W. (1927). *The logic of modern physics*. New York: Macmillan.

Brooks, F. P., Jr. (1987). No silver bullet: Essence and accidents of software engineering. *IEEE Computer*, *20*(4), 10–19.

Brooks, R. (1977). Towards a theory of the cognitive processes in computer programming. *International Journal of Man-Machine Studies*, *9*(6), 737–751.

Brooks, R. (1983). Towards a theory of the comprehension of computer programs. *International Journal of Man-Machine Studies*, *18*(6), 543–554.

Bryan, G. E. (1994). Not all programmers are created equal. In *Proceedings of the IEEE Aerospace Applications Conference* (p. 66-62).

Bryan, W. L., & Harter, N. (1899). Studies on the telegraphic language: The acquisition of a hierarchy of habits. *Psychological Review*, *6*(4), 345–375.

Buck, D., & Stucki, D. J. (2001). JKarelRobot: A case study in supporting levels of cognitive development in the computer science curriculum. *ACM SIGCSE Bulletin*, *33*(1), 16–20.

Buckley, J., & Exton, C. (2003). Bloom's taxonomy: A framework for assessing programmers' knowledge of software systems. In *11th IEEE International Workshop on Program Comprehension* (pp. 165–174).

Bunge, M. (1967). *Scientific research: The search for a system*. New York: Springer-Verlag.

Burkhardt, J.-M., Détienne, F., & Wiedenbeck, S. (2002). Object-oriented program comprehension: Effect of expertise, task and phase. *Empirical Software Engineering*, *7*(2), 115–156.

Butcher, D. F., & Muth, W. A. (1985). Predicting performance in an introductory computer science course. *Communications of the ACM*, *28*(3), 263–268.

Byckling, P., & Sajaniemi, J. (2006). Roles of variables and programming skills improvement. *ACM SIGCSE Bulletin*, *38*(1), 413–417.

Byrne, P., & Lyons, G. (2001). The effect of student attributes on success in programming. *ACM SIGCSE Bulletin*, *33*(33), 49–52.

Califf, M. E., & Goodwin, M. (2002). Testing skills and knowledge: Introducing a laboratory exam in CS1. *ACM SIGCSE Bulletin*, *34*(1), 217–221.

Cameron, W. B. (1963). *Informal sociology: A casual introduction to sociological thinking.* New York: Random House.

Campbell, J. P. (1990). Modeling the performance prediction problem in industrial and organizational psychology. In M. D. Dunnette & L. M. Hough (Eds.), *Handbook of industrial and organizational psychology* (Second ed., Vol. 1, pp. 687–732). Palo Alto, CA: Consulting Psychologists Press.

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

Campbell, J. P., McCloy, R. A., Oppler, S. H., & Sager, C. E. (1993). A theory of performance. In N. Schmitt & W. C. Borman (Eds.), *Personnel selection in organizations* (p. 35-70). San Francisco, CA: Jossey-Bass.

Campbell, R. L., Brown, N. R., & DiBello, L. A. (1992). The programmer's burden: Developing expertise in programming. In R. R. Hoffman (Ed.), *The psychology of expertise: Cognitive research and empirical AI* (p. 269-294). New York: Springer-Verlag.

Card, D. N., Mc Garry, F. E., & Page, G. T. (1987). Evaluating software engineering technologies. *IEEE Transactions on Software Engineering*, *SE-13*(7), 845–851.

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

Carver, J. C., Hochstein, L., & Oslin, J. (2011). *Programming ability: Do we know it when we see it? an empirical study of peer evaluation* (Tech. Rep. No. SERG-2011-06). Tuscaloosa, Alabama: University of Alabama.

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

Cattell, R. B. (1988). The principles of experimental design and analysis in relation to theory building. In J. R. Nesselroade & R. B. Cattell (Eds.), *Handbook of multivariate experimental psychology* (Second ed., p. 21-130). New York: Plenum Press.

Cegielski, C. G., & Hall, D. J. (2006). What makes a good programmer? *Communications of the ACM*, *49*(10), 73–75.

Chalmers, A. F. (1999). *What is this thing called science?* (Third ed.). Indianapolis: Hackett Publishing Company.

Chamillard, A. T., & Braun, K. A. (2000). Evaluating programming ability in an intro-

ductory computer science course. *Communications of the ACM*, *32*(1), 212–216.

Chamillard, A. T., & Joiner, J. K. (2001). Using lab practica to evaluate programming ability. *ACM SIGCSE Bulletin*, *33*(1), 159–163.

Chase, W. G., & Simon, H. A. (1973). The mind's eye in chess. In W. G. Chase (Ed.), *Visual information processing* (pp. 215–281). San Diego: Academic Press.

Chatfield, C. (1995). Model uncertainty, data mining and statistical inference. *Journal of the Royal Statistical Society, Series A*, *158*(3), 419–466.

Chatzopoulou, D. I., & Economides, A. A. (2010). Adaptive assessment of student's knowledge in programming courses. *Journal of Computer Assisted Learning*, *26*(4), 258–269.

Cheang, B., Kurnia, A., Lim, A., & Oon, W.-C. (2003). On automated grading of programming assignments in an academic institution. *Computers & Education*, *41*(2), 121–131.

Chen, E. T. (1978). Program complexity and programmer productivity. *IEEE Transactions on Software Engineering*, *SE-4*(3), 187–194.

Chen, W.-H., Hsueh, N.-L., & Lee, W.-M. (2011). Assessing PSP effect in training disciplined software development: A Plan–Track–Review model. *Information and Software Technology*, *53*(2), 137–148.

Chi, M. T. H., Glaser, R., & Farr, M. J. (Eds.). (1988). *The nature of expertise*. Mahwah, NJ: Lawrence Erlbaum.

Chmiel, R., & Loui, M. C. (2004). Debugging: from novice to expert. *ACM SIGCSE Bulletin*, *36*(1), 17–21.

Chrysler, E. (1978). Some basic determinants of computer programming productivity. *Communications of the ACM*, *21*(6), 472–483.

Clark, D. (2004). Testing programming skills with multiple choice questions. *Informatics in Education*, *3*(2), 161–178.

Clark, J. G., Walz, D. B., & Wynekoop, J. L. (2003). Identifying exceptional application software developers: A comparison of students and professionals. *Communications of the Association for Information Systems*, *11*(1), 1–31.

Cohen, B. P. (1989). *Developing sociological knowledge: Theory and method* (Second ed.). Chicago: Nelson-Hall.

Cohen, J. (1988). *Statistical power analysis for the behavioral sciences* (Second ed.). Hillsdale, NJ: Lawrence Erlbaum.

Conejo, R., Guzmá, E., Milán, E., Trella, M., Pérez-De-La-Cruz, J. L., & Ríos, A. (2004). SIETTE: A web-based tool for adaptive testing. *International Journal of Artificial Intelligence in Education*, *14*(1), 29–61.

Cronbach, L. J. (1990). *Essentials of psychological testing* (Fifth ed.). New York: Harper & Row.

Crosby, M. E., Scholtz, J., & Wiedenbeck, S. (2002). The roles beacons play in comprehension for novice and expert programmers. In *14th Workshop of the Psychology of Programming Interest Group* (pp. 58–73).

Curtis, B. (1980). Measurement and experimentation in software engineering. *Proceedings of the IEEE*, *68*(9), 1144–1157.

Curtis, B. (1981). Substantiating programmer variability. *Proceedings of the IEEE*, *69*(7), 846.

Curtis, B. (1984). Fifteen years of psychology in software engineering: Individual differences and cognitive science. In *Proceedings of the 7th International Conference on Software Engineering* (pp. 97–106).

Curtis, B. (1986). By the way, did anyone study any real programmers? In E. Soloway & S. Iyengar (Eds.), *Empirical studies of programmers: First workshop* (pp. 256–262). Norwood, NJ: Ablex Publishing.

Curtis, B. (1991). Techies as non-technological factors in software engineering? In *Proceedings of the 13th International Conference on Software Engineering* (pp. 147–148).

Curtis, B., Hefley, W. E., & Miller, S. (1995). *Overview of the People Capability Maturity Model* (Tech. Rep. No. CMU/SEI-95-MM-01). Pittsburgh, PA: Software Engineering Institute.

Dagienė, V., & Skūpienė, J. (2004). Learning by competitions: olympiads in informatics as a tool for training high-grade skills in programming. In *2nd International Conference on Information Technology: Research and Education* (pp. 79–83).

Dahl, F. A., Grotle, M., Benth, J. Š., & Natvig, B. (2008). Data splitting as a countermeasure against hypothesis fishing: With a case study of predictors for low back pain. *European Journal of Epidemiology*, *23*(4), 237–242.

Daly, C., & Waldron, J. (2004). Assessing the assessment of programming ability. In *ACM SIGCSE Bulletin* (Vol. 36, p. 210-213).

Davies, S. P. (1989). Skill levels and strategic differences in plan comprehension and implementation in programming. In A. Sutcliffe & L. Macaulay (Eds.), *Proceedings of the 5th Conference of the British Computer Society, Human-Computer Interaction Specialist Group on People and Computers V* (pp. 487–502). Press Syndicate of the University of Cambridge.

Davies, S. P. (1993). Externalising information during coding activities: Effects of expertise, environment, and task. In C. R. Cook, J. C. Scholtz, & J. C. Spohrer (Eds.), *Empirical studies of programmers: Fifth workshop* (pp. 42–61). Norwood, NJ: Ablex.

Davies, S. P. (1994). Knowledge restructuring and the acquisition of programming expertise. *International Journal of Man-Machine Studies*, *40*(4), 703–726.

Dehnadi, S. (2006). Testing programming aptitude. In *Proceedings of the 18th Annual Workshop of the Psychology of Programming Interest Group* (pp. 22–37).

DeMarco, T., & Lister, T. (1985). Programmer performance and the effects of the workplace. In *Proceedings of the 8th International Conference on Software Engineering* (pp. 268–272).

DeMarco, T., & Lister, T. (1999). *Peopleware: Productive projects and teams* (Second ed.). New York: Dorset House Publishing Company.

Dickey, T. E. (1981). Programmer variability. *Proceedings of the IEEE*, *69*(7), 844-845.

Dijkstra, E. W. (1972). The humble programmer. *Communications of the ACM*, *15*(10), 859–866.

Doane, S. M., Pellegrino, J. W., & Klatzky, R. L. (1990). Expertise in a computer oper-

ating system: Conceptualization and performance. *Human-Computer Interaction*, *5*(2), 267–304.

Douce, C., Livingstone, D., & Orwell, J. (2005). Automatic test-based assessment of programming: A review. *Educational Resources in Computing*, *5*(3), 1–13.

Dreyfus, H. L., & Dreyfus, S. E. (1986). *Mind over machine: The power of human intuition and expertise in the era of the computer*. New York: The Free Press.

Duckworth, A. L., Quinn, P. D., Lynam, D. R., Loeber, R., & Stouthamer-Loeber, M. (2011). Role of test motivation in intelligence testing. *Proceedings of the National Academy of Sciences of the United States of America*, *108*(19), 7716–7720.

Dybå, T. (2000). An instrument for measuring the key factors of success in software process improvement. *Empirical Software Engineering*, *5*(4), 357–390.

Dybå, T., Kampenes, V. B., & Sjøberg, D. I. K. (2006). A systematic review of statistical power in software engineering experiments. *Information and Software Technology*, *48*(8), 745–755.

Dybå, T., Sjøberg, D. I. K., & Cruzes, D. S. (2012). What works for whom, where, when, and why? On the role of context in empirical software engineering. In *Proceedings of the 6th International Symposium on Empirical Software Engineering and Measurement* (pp. 19–28).

Ebert, C., Dumke, R., Bundschuh, M., & Schmietendorf, A. (2005). *Best practices in software measurement: How to use metrics to improve project and process performance*. Berlin: Springer.

Edwards, S. H. (2004). Using software testing to move students from trial-and-error to reflection-in-action. *SIGCSE Bulletin*, *36*(1), 26-30.

Ellsworth, C. C., Fenwick, J., J. B., & Kurtz, B. L. (2004). The Quiver System. *ACM SIGCSE Bulletin*, *36*(1), 205–209.

Elo, A. (1978). *The rating of chess players, past and present*. London: Batsford.

English, J. (2002). Experience with a computer-assisted formal programming examination. *ACM SIGCSE Bulletin*, *34*(3), 51–54.

Ericsson, K. A. (2006). An introduction to the Cambridge handbook of expertise and expert performance: Its development, organization, and content. In K. A. Ericsson, N. Charness, P. J. Feltovich, & R. R. Hoffman (Eds.), *The Cambridge handbook of expertise and expert performance* (pp. 3–19). New York: Cambridge University Press.

Ericsson, K. A. (2014). Why expert performance is special and cannot be extrapolated from studies of performance in the general population: A response to criticisms. *Intelligence*, *45*(July-August), 81–103.

Ericsson, K. A., & Charness, N. (1994). Expert performance: Its structure and acquisition. *American Psychologist*, *49*(8), 725–747.

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

Ericsson, K. A., Krampe, R. T., & Tesch-Römer, C. (1993). The role of deliberate practice in the acquisition of expert performance. *Psychological Review*, *100*(3), 363–406.

Ericsson, K. A., & Lehmann, A. C. (1996). Expert and exceptional performance: Evidence

of maximal adaptation to task constraints. *Annual Review of Psychology*, *47*(1), 273–305.

Ericsson, K. A., & Smith, J. (1991). Prospects and limits of the empirical study of expertise: An introduction. In K. A. Ericsson & J. Smith (Eds.), *Towards and general theory of expertise* (pp. 1–38). New York: Cambridge University Press.

Evans, G. E., & Simkin, M. G. (1989). What best predicts computer proficiency? *Communications of the ACM*, *32*(11), 1322-1327.

Feigenspan, J., Kästner, C., Liebig, J., Apel, S., & Hanenberg, S. (2012). Measuring programming experience. In *IEEE 20th International Conference on Program Comprehension* (pp. 73–82).

Fenton, N. (1994). Software measurement: A necessary scientific basis. *IEEE Transactions on Software Engineering*, *20*(3), 199–206.

Fenton, N., & Kitchenham, B. (1991). Validating software measures. *Journal of Software Testing, Verification, and Reliability*, *1*(2), 27–42.

Ferguson, A., Myers, C. S., Bartlett, R. J., Banister, H., Bartlett, F. C., Brown, W., ... Tucker, W. S. (1940). Quantitative estimates of sensory events: Final report of the Committee Appointed to Consider and Report upon the Possibility of Quantitative Estimates of Sensory Events. *Advancement of Science*, *1*, 331-349.

Ferguson, G. A. (1956). On transfer and the abilities of man. *Canadian Journal of Psychology*, *10*(3), 121–131.

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

Fincham, J. M., & Anderson, J. R. (2006). Distinct roles of the anterior cingulate and prefrontal cortex in the acquisition and performance of a cognitive skill. *Proceedings of the National Academy of Sciences of the United States of America*, *103*(34), 12941–12946.

Fincher, S., Baker, B., Box, I., Cutts, Q., de Raadt, M., Haden, P., ... Tutty, J. (2005). *Programmed to succeed?: A multi-national, multi-institutional study of introductory programming courses* (Tech. Rep. No. 1-05). Canterbury, Kent, UK: University of Kent.

Fire, M., Katz, G., Elovici, Y., Shapira, B., & Rokach, L. (2012). Predicting student exam's scores by analyzing social network data. In *Proceedings of the 8th International Conference on Active Media Technology* (pp. 584–595).

Fitts, P. M. (1964). Perceptual-motor skill learning. In A. W. Melton (Ed.), *Categories of human learning* (pp. 243–285). San Diego: Academic Press.

Fitts, P. M., & Posner, M. I. (1967). *Human performance.* Belmont, CA: Brooks/Cole.

Fix, V., Wiedenbeck, S., & Scholtz, J. (1993). Mental representations of programs by novices and experts. In *Proceedings of the INTERACT'93 and CHI'93 Conference on Human Factors in Computing Systems* (pp. 74–79).

Freeman, J. T. (1992). Conference report Empirical Studies of Programmers: Fourth workshop. *SIGCHI Bulletin*, *24*(3), 18–23.

Gallivan, M., Truex, I., D. P., & Kvasny, L. (2004). Changing patterns in IT skill sets 1988–2003: A content analysis of classified advertising. *The DATA BASE for Advances in Information Systems*, *35*(3), 64–87.

Galton, F. (1883). *Inquiries into human faculty and its development* (Second ed.). London: Dent.

Glass, R. L. (1980). The importance of the individual. *ACM SIGSOFT Software Engineering Notes*, *5*(3), 48–50.

Glass, R. L. (2001). Frequently forgotten fundamental facts about software engineering. *IEEE Software*, 110–112.

Glymour, C. (2001). *The mind's arrows: Bayes nets and graphical causal models in psychology*. Cambridge, MA: MIT Press.

Goold, A., & Rimmer, R. (2000). Factors affecting performance in first-year computing. *ACM SIGCSE Bulletin*, *32*(2), 39–43.

Gould, S. J. (1996). *The mismeasure of man* (Rev. and expanded ed.). New York: Norton.

Grant, E. E., & Sackman, H. (1967). An exploratory investigation of programmer performance under on-line and off-line conditions. *IEEE Transactions on Human Factors in Electronics*, *HFE-8*(1), 33–48.

Green, S. B., Lissitz, R. W., & Mulaik, S. A. (1977). Limitations of coefficient alpha as an index of test unidimensionality. *Educational and Psychological Measurement*, *37*(4), 827–838.

Gustafsson, J.-E. (1984). A unifying model for the structure of intellectual abilities. *Intelligence*, *8*(3), 179–203.

Guttman, L. L. (2007). The basis for scalogram analysis. In G. M. Maranell (Ed.), *Scaling: A sourcebook for behavioral scientists* (pp. 142–171). Chicago: Aldine Pub. Co.

Hacking, I. (1990). *The taming of chance*. Cambridge: Cambridge University Press.

Haerem, T., & Rau, D. (2007). The influence of degree of expertise and objective task complexity on perceived task complexity and performance. *Journal of Applied Psychology*, *92*(5), 1320–1331.

Hagan, D., & Markham, S. (2000). Does it help to have some programming experience before beginning a computing degree program? *ACM SIGCSE Bulletin*, *32*(3), 25–28.

Hands, B., Sheridan, B., & Larkin, D. (1999). Creating performance categories from continuous motor skill data using a Rasch measurement model. *Journal of Outcome Measurement*, *3*(3), 216–232.

Hannay, J. E., Arisholm, E., Engvik, H., & Sjøberg, D. I. K. (2010). Personality and pair programming. *IEEE Transactions on Software Engineering*, *36*(1), 61–80.

Hannay, J. E., & Jørgensen, M. (2008). The role of deliberate artificial design elements in software engineering experiments. *IEEE Transactions on Software Engineering*, *34*, 242–259.

Harris, J. (2014). Testing programming aptitude in introductory programming courses. *Journal of Computing Sciences in Colleges*, *30*(2), 149–156.

Hawk, S., Kaiser, K. M., Goles, T., Bullen, C. V., Simon, J. C., Beath, C. M., ... Frampton, K. (2012). The information technology workforce: A comparison of critical skills of clients and service providers. *Information Systems Management*, *29*(1), 2–12.

Hitchens, C. (2007). *God is not great: How religion poisons everything.* New York: Twelve.

Holden, E., & Weeden, E. (2004). The experience factor in early programming education. In *Proceedings of the 5th Conference on Information Technology Education* (pp. 211–218).

Hollingsworth, J. (1960). Automatic graders for programming classes. *Communications of the ACM*, *3*(10), 528–529.

Holt, R. W., Boehm-Davis, D. A., & Schultz, A. C. (1987). Mental representations of programs for student and professional programmers. In G. H. Olson, S. Sheppard, & E. Soloway (Eds.), *Empirical studies of programmers: Second workshop* (pp. 33–46). Ablex Publishing.

Höst, M., Regnell, B., & Wohlin, C. (2000). Using students as subjects—a comparative study of students and professionals in lead-time impact assessment. *Empirical Software Engineering*, *5*(3), 201–214.

Humphrey, W. S. (1996). Using a defined and measured Personal Software Process. *IEEE Computer*, *13*(3), 77-88.

Humphry, S. M., & Andrich, D. (2008). Understanding the unit in the Rasch model. *Journal of Applied Measurement*, *9*(3), 249–264.

Jeffery, D. R., & Lawrence, M. J. (1979). An inter-organisational comparison of programming productivity. In *Proceedings of the 4th International Conference on Software Engineering* (p. 369-377).

Jeffery, D. R., & Lawrence, M. J. (1985). Managing programming productivity. *Journal of Systems and Software*, *5*, 49–58.

Jones, C. (1997). *Applied software measurement: Assuring productivity and quality* (Second ed.). New York: McGraw-Hill.

Jones, T. C. (1978). Measuring programming quality and productivity. *IBM Systems Journal*, *17*(1), 39–63.

Jöreskog, K. G. (1969). A general approach to confirmatory maximum likelihood factor analysis. *Psychometrika*, *34*(2), 183-202.

Jørgensen, M. (1995). An empirical study of software maintenance tasks. *Software Maintenance: Research and Practice*, *7*(1), 27–48.

Jørgensen, M. (2014). Failure factors of small software projects at a global outsourcing marketplace. *Journal of Systems and Software*, *92*(June), 157–169.

Joy, M., Griffiths, N., & Boyatt, R. (2005). The BOSS online submission and assessment system. *Journal of Educational Resources in Computing*, *5*(3), 1–28.

Kampenes, V. B., Dybå, T., Hannay, J. E., & Sjøberg, D. I. K. (2007). A systematic review of effect size in software engineering experiments. *Information and Software Technology*, *49*(11), 1073–1086.

Kampenes, V. B., Dybå, T., Hannay, J. E., & Sjøberg, D. I. K. (2009). A systematic review of quasi-experiments in software engineering. *Information and Software Technology*, *51*(1), 71–82.

Karabatos, G. (2001). The Rasch model, additive conjoint measurement, and new models of probabilistic measurement theory. *Journal of Applied Measurement*, *2*(4), 389–423.

Karahasanović, A., Levine, A. K., & Thomas, R. C. (2007). Comprehension strategies and difficulties in maintaining object-oriented systems: An explorative study. *Journal of Systems and Software*, *80*(9), 1541–1559.

Karahasanović, A., & Thomas, R. C. (2007). Difficulties experienced by students in maintaining object-oriented systems: An empirical study. In *Proceedings of the Australasian Computing Education Conference* (pp. 81–87).

Kearney, J. K., Sedlmeyer, R. L., Thompson, W. B., Gray, M. A., & Adler, M. A. (1986). Software complexity measurement. *Communications of the ACM*, *29*(11), 1044–1050.

Kelly, T. L. (1927). *Interpretation of educational measurements*. New York: World Book Company.

Kitchenham, B., & Pfleeger, S. L. (1996). Software quality: The elusive target. *IEEE Software*, *13*(1), 12-21.

Kjølberg, L.-K., & Hjorth-Johansen, K. E. (2012). *Expertise: What does education give you?—on education and task complexity and their moderating effect on expertise.* Master's thesis, BI Norwegian Business School. Oslo, Norway.

Kleinschmager, S., & Hanenberg, S. (2011). How to rate programming skills in programming experiments? A preliminary, exploratory study based on university marks, pretests, and self-estimation. In *Proceedings of the 3rd ACM SIGPLAN Workshop on Evaluation and Usability of Programming Languages and Tools* (pp. 15–24).

Koenemann, J., & Robertson, S. P. (1991). Expert problem solving strategies for program comprehension. In *Proceedings of the SIGCHI Conference on Human Factors in Computing Systems* (pp. 125–130).

Koubek, R. J., & Salvendy, G. (1991). Cognitive performance of super-experts on computer program modification tasks. *Ergonomics*, *34*(8), 1095–1112.

Krantz, D. H., Luce, R. D., Suppes, P., & Tversky, A. (1971). *Foundations of measurement* (Vol. 1). New York: Academic Press.

Kruger, J., & Dunning, D. (1999). Unskilled and unaware of it: How difficulties in recognizing one's own incompetence lead to inflated self-assessments. *Journal of Personality and Social Psychology*, *77*(6), 1121–1134.

Kværn, K. (2006). *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. Oslo, Norway.

Kyburg, J., H. E. (1984). *Theory and measurement*. Cambridge: Cambridge University Press.

Kyllonen, P. C., & Stephens, D. L. (1990). Cognitive abilities as determinants of success in acquiring logic skill. *Learning and Individual Differences*, *2*(2), 129–160.

Kyllonen, P. C., & Woltz, D. J. (1989). Role of cognitive factors in the acquisition of cognitive skill. In R. Kanfer, P. L. Ackerman, & R. Cudeck (Eds.), *Abilities, motivation, and methodology: The Minneapolis symposium on learning and individual differences* (pp. 239–280). Mahwah, NJ: Erlbaum.

Kyngdon, A. (2008). The Rasch model from the perspective of the representational theory of measurement. *Theory & Psychology*, *18*(1), 89–109.

Kyngdon, A. (2011). Psychological measurement needs units, ratios, and real quantities: A commentary on Humphry. *Measurement*, *9*(1), 55–58.

Lampson, B. W. (1967). A critique of 'an exploratory investigation of programmer performance under on-line and off-line conditions'. *IEEE Transactions on Human Factors in Electronics*, *HFE-8*(1), 48–51.

Land, L. P. W., Wong, B., & Jeffery, R. (2003). An extension of the behavioral theory of group performance in software development technical reviews. In *10th Asia-Pacific Software Engineering Conference* (pp. 520–530).

Lanning, D. L., & Khoshgoftaar, T. M. (1994). Modeling the relationship between source code complexity and maintenance difficulty. *Computer*, *27*(9), 35–40.

Latham, G. P., & Pinder, C. C. (2005). Work motivation theory and research at the dawn of the twenty-first century. *Annual Review of Psychology*, *56*(1), 485–516.

Le Deist, F. D., & Winterton, J. (2005). What is competence? *Human Resource Development International*, *8*(1), 27–46.

Leidlmair, K. (2009). *After cognitivism: A reassessment of cognitive science and philosophy*. Dordrecht: Springer.

Lethbridge, T. C. (2000). What knowledge is important to a software professional? *Computer*, *33*(5), 44–50.

Lethbridge, T. C., LeBlanc, J., R. J., Sobel, A. E. K., Hilburn, T. B., & Diaz-Herrera, J. L. (2006). SE2004: Reccomendations for undergraduate software engineering curricula. *IEEE Software*, *23*(6), 19–25.

Linacre, J. M. (1994). Sample size and item calibration stability. *Rasch Measurement Transactions*, *7*(4), 328.

Linn, M. C., & Dalbey, J. (1989). Cognitive consequences of programming instruction. In E. Soloway & J. C. Spohrer (Eds.), *Studying the novice programmer* (pp. 57–81). Hillsdale, NJ: Lawrence Erlbaum.

Lister, R., Fidge, C., & Teague, D. (2009). Further evidence of a relationship between explaining, tracing and writing skills in introductory programming. *ACM SIGCSE Bulletin*, *41*(3), 161–165.

Locke, E. A., & Lathham, G. P. (1990). *A theory of goal setting and task performance*. Englewood Cliffs, NJ: Prentice Hall.

Loehlin, J. C. (2004). *Latent variable models: An introduction to factor, path, and structural equation analysis* (Fourth ed.). Mahwah, NJ: Lawrence Erlbaum.

Lord, D. (1997). *The acquisition of programming skills: Effects of learning style and teaching method*. Doctoral dissertation, Lancaster University. Lancaster, England.

Lord, F. M., & Novick, M. R. (1968). *Statistical theories of mental test scores*. Reading, MA: Addison-Wesley.

Lucas, J., H. C., & Kaplan, R. B. (1976). A structured programming experiment. *The Computer Journal*, *19*(2), 136–138.

Luce, R. D. (1997). Quantification and symmetry: Commentary on Michell, Quantitative science and the definition of measurement in psychology. *British Journal of Psychology*, *88*(3), 395–398.

Luce, R. D., & Tukey, J. W. (1964). Simultaneous conjoint measurement: A new type of fundamental measurement. *Journal of Mathematical Psychology*, *1*(1), 1–27.

Lui, K. M., & Chan, K. C. C. (2006). Pair programming productivity: Novice-novice vs. expert-expert. *International Journal of Human-Computer Studies*, *64*(9), 915–925.

MacCallum, R. C., Zhang, S., Preacher, K. J., & Rucker, D. D. (2002). On the practice of dichotomization of quantitative variables. *Psychological Methods*, *7*(1), 19–40.

MacKay, D. G. (1982). The problems of flexibility, fluency, and speed-accuracy trade-off in skilled behavior. *Psychological Review*, *89*(5), 483–506.

MacKenzie, S. B., Podsakoff, P. M., & Podsakoff, N. P. (2011). Construct measurement and validation procedures in MIS and behavioral research: integrating new and existing techniques. *MIS Quarterly*, *35*(2), 293–334.

Mair, P., & Hatzinger, R. (2007). Extended Rasch modeling: The eRm package for the application of IRT models in R. *Journal of Statistical Software*, *20*(9), 1–20.

Marais, I., & Andrich, D. (2012). RUMMss (Rasch Unidimensional Measurement Models Simulation Studies software) and user guide [Computer software manual]. Perth: University of Western Australia.

Maris, G., & van der Maas, H. (2012). Speed-accuracy response models: Scoring rules based on response time and accuracy. *Psychometrika*, *77*(4), 615–633.

Markus, K. A., & Borsboom, D. (2012). The cat came back: Evaluating arguments against psychological measurement. *Theory & Psychology*, *22*(4), 452-466.

Maul, A., Wilson, M., & Irribarra, D. T. (2013). On the conceptual foundations of psychological measurement. *Journal of Physics: Conference Series*, *459, 012008*. doi: 10.1088/1742-6596/459/1/012008

Mayer, D. B., & Stalnaker, A. W. (1968). Selection and evaluation of computer personnel—the research history of SIG/CPR. In *Proceedings of the 23rd ACM National Conference* (pp. 657–670). ACM.

Mayer, R. E., Dyck, J. L., & Vilberg, W. (1986). Learning to program and learning to think: What's the connection? *Communications of the ACM*, *29*(7), 605–610.

Maynard, D. C., & Hakel, M. D. (1997). Effects of objective and subjective task complexity on performance. *Human Performance*, *10*(4), 303–330.

Mazlack, L. J. (1980). Identifying potential to acquire programming skill. *Communications of the ACM*, *23*(1), 14–17.

McCall, J. (1994). Quality factors. In J. J. Marciniak (Ed.), *Encyclopedia of software engineering* (Vol. 2, pp. 958–969). Wiley-Interscience.

McClelland, D. C. (1973). Testing for competence rather than for 'intelligence'. *American Psychologist*, *28*(1), 1–14.

McCloy, R. A., Campbell, J. P., & Cudeck, R. (1994). A confirmatory test of a model of performance determinants. *Journal of Applied Psychology*, *79*(4), 493–505.

McConnell, S. (1998). Problem programmers. *IEEE Software*, *15*(2), 126–128.

McCracken, M., Almstrum, V., Diaz, D., Guzdial, M., Hagan, D., Kolikant, Y. B.-D., ... Wilusz, T. (2001). A multi-national, multi-institutional study of assessment of programming skills of first-year CS students. *ACM SIGCSE Bulletin*, *33*(4), 125–140.

McDaniel, M. A., Schmidt, F. K., & Hunter, J. E. (1988). Job experience correlates of job performance. *Journal of Applied Psychology*, *73*, 327–330.

McGill, M. (2008). Critical skills for game developers: An analysis of skills sought by

industry. In *Proceedings of the 2008 Conference on Future Play: Research, Play, Share* (pp. 89–96).

McKeithen, K. B., Reitman, J. S., Rueter, H. H., & Hirtle, S. C. (1981). Knowledge organization and skill differences in computer programmers. *Cognitive Psychology*, *13*(3), 307–325.

McNab, F., & Klingberg, T. (2008). Prefrontal cortex and basal ganglia control access to working memory. *Nature Neuroscience*, *11*(1), 103–107.

McNamara, R. A. (2004). Evaluating assessment with competency mapping. In *Proceedings of the 6th Australasian Conference on Computing Education* (pp. 193–199).

McNamara, W. J., & Hughes, J. L. (1955). The selection of card punch operators. *Personnel Psychology*, *8*(4), 417–427.

Messick, S. (1989a). Meaning and values in test validation: The science and ethics of assessment. *Educational Researcher*, *18*(2), 5–11.

Messick, S. (1989b). Validity. In R. L. Linn (Ed.), *Educational measurement* (Third ed., pp. 12–103). New York: American Council on Education/Macmillan.

Messick, S. (1994). The interplay of evidence and consequences in the validation of performance assessments. *Educational Researcher*, *23*(2), 13–23.

Messick, S. (1995). Standards of validity and the validity of standards in performance asessment. *Educational Measurement: Issues and Practice*, *14*(4), 5–8.

Meyer, B. (2006). The unspoken revolution in software engineering. *Computer*, *39*(1), 121–124.

Michell, J. (1997). Quantitative science and the definition of measurement in psychology. *British Journal of Psychology*, *88*(3), 355–383.

Michell, J. (1999). *Measurement in psychology: A critical history of a methodological concept*. New York: Cambridge University Press.

Michell, J. (2008). Rejoinder. *Measurement*, *6*(1–2), 125–133.

Miller, G. A. (1956). The magical number seven, plus or minus two: Some limits on our capacity for processing information. *Psychological Review*, *63*(2), 81–97.

Miller, G. E. (1990). The assessment of clinical skills/competence/performance. *Academic Medicine*, *65*(9), S63–67.

Millman, J., & Greene, J. (1989). The specification and development of tests of achievement and ability. In R. L. Linn (Ed.), *Educational measurement* (Third ed., pp. 335–366). New York: American Council on Education/Macmillan.

Mills, H. D. (1983). *Software productivity*. Boston: Little, Brown and Company.

Neisser, U., Boodoo, G., Bouchard Jr., T. J., Boykin, A. W., Brody, N., Ceci, S. J., ... Urbina, S. (1996). Intelligence: Knowns and unknowns. *American Psychologist*, *51*(2), 77–101.

Neves, D. M., & Anderson, J. R. (1981). Knowledge compilation: Mechanisms for the automatization of cognitive skills. In J. R. Anderson (Ed.), *Cognitive skills and their acquisition* (pp. 57–84). Mahwah, NJ: Erlbaum.

Newell, A., & Rosenbloom, P. (1981). Mechanisms of skill acquisition and the law of practice. In J. R. Anderson (Ed.), *Cognitive skills and their acquisition* (pp. 1–56). Hillsdale, NJ: Erlbaum.

Nunnally, J. C., & Bernstein, I. H. (1994). *Psychometric theory* (Third ed.). New York: McGraw-Hill.

Oyer, P. D. (1969). An approach to computer personnel evaluation. In *Proceedings of the 7th Annual Conference on SIGCPR* (pp. 100–113).

Pear, T. H. (1927). Skill. *Journal of Personnel Research*, *12*, 478–489.

Pear, T. H. (1928). The nature of skill. *Nature*, *122*(3077), 611-614.

Pearl, J. (2000). *Causality: Models, reasoning, and inference*. Cambridge: Cambridge University Press.

Perry, D., & Cantley, G. (1965). *Computer programming selection and training in System Development Corporation* (Technical memorandum No. TM-2234). Santa Montica, CA: System Developement Corporation.

Pioro, B. T. (2006). Introductory computer programming: gender, major, discrete mathematics, and calculus. *Journal of Computing Sciences in Colleges*, *21*(5), 123–129.

Pirolli, P., & Wilson, M. (1998). A theory of the measurement of knowledge content, access, and learning. *Psychological Review*, *105*(1), 58–82.

Pirolli, P. L., & Anderson, J. R. (1985). The role of learning from examples in the acquisition of recursive programming skills. *Canadian Journal of Psychology*, *39*(2), 240–272.

Plomin, R., Shakeshaft, N. G., McMillan, A., & Trzaskowski, M. (2014). Nature, nurture, and expertise: Response to Ericsson. *Intelligence*, *45*(July-August), 115–117.

Pocius, K. E. (1991). Personality factors in human-computer interaction: A review of the literature. *Computers in Human Behavior*, *7*(3), 103–135.

Popper, K. (1968). *Conjectures and refutations*. New York: Harper & Row.

Porter, A., Siy, H., Mockus, A., & Votta, L. (1998). Understanding the sources of variation in software inspections. *ACM Transactions on Software Engineering Methodology*, *7*(1), 41–79.

Prechelt, L. (1999a). *The 28:1 Grant/Sackman legend is misleading, or: How large is interpersonal variation really?* (Tech. Rep. No. 18). Karlsruhe, Germany: University of Karlsruhe.

Prechelt, L. (1999b). Comparing Java vs. C/C++ efficiency differences to interpersonal differences. *Communications of the ACM*, *42*(10), 109-112.

Prechelt, L. (2000). An empirical comparison of seven programming languages. *IEEE Computer*, *33*(10), 23–29.

Prechelt, L. (2011). Plat_Forms: A web development platform comparison by an exploratory experiment searching for emergent platform properties. *IEEE Transactions on Software Engineering*, *37*(1), 95–108.

Prechelt, L., & Unger, B. (2000). An experiment measuring the effects of Personal Software Process (PSP) training. *IEEE Transactions on Software Engineering*, *27*(5), 465–472.

Proctor, R. W., & Dutta, A. (1995). *Skill acquisition and human performance*. Thousand Oaks, CA: Sage Publications.

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

Rasch, R. H., & Tosi, H. L. (1992). Factors affecting software developers' performance: An integrated approach. *MIS Quarterly*, *16*(3), 395–413.

Rist, R. S. (1989). Schema creation in programming. *Cognitive Science*, *13*(3), 389–414.

Rist, R. S. (1995). Program structure and design. *Cognitive Science*, *19*(4), 507–562.

Roberts, J. A., Hann, I.-H., & Slaughter, S. A. (2006). Understanding the motivations, participation, and performance of open source software developers: A longitudinal study of the Apache projects. *Management Science*, *52*(7), 984–999.

Robillard, P. N. (1999). The role of knowledge in software development. *Communications of the ACM*, *42*(1), 87–92.

Robins, A., Rountree, J., & Rountree, N. (2003). Learning and teaching programming: A review and discussion. *Computer Science Education*, *13*(2), 137–172.

Rolfhus, E. L., & Ackerman, P. L. (1999). Assessing individual differences in knowledge: Knowledge, intelligence, and related traits. *Journal of Educational Psychology*, *91*(3), 511-526.

Roth, P. L., BeVier, C. A., Switzer, F. S., III, & Schippmann, J. S. (1996). Meta-analyzing the relationship between grades and job performance. *Journal of Applied Psychology*, *81*(5), 548–556.

Rowan, T. C. (1957). Psychological tests and selection of computer programmers. *Journal of the Association for Computing Machinery*, *4*, 348-353.

Runeson, P. (2003). Using students as experiment subjects—an analysis on graduate and freshmen student data. In *Proceedings of the 7th International Conference on Empirical Assessment in Software Engineering* (pp. 95–102).

Runkel, P. J., & McGrath, J. E. (1972). *Research on human behavior: A systematic guide to method*. New York: Holt, Rinehart and Winston.

Sackman, H., Erikson, W. J., & Grant, E. E. (1968). Exploratory experimental studies comparing online and offline programming performance. *Communications of the ACM*, *11*(1), 3-11.

Saikkonen, R., Malmi, L., & Korhonen, A. (2001). Fully automatic assessment of programming exercises. *ACM SIGCSE Bulletin*, *33*(33), 133–136.

Salicath, A. A. (2008). *Administration and evaluation of solutions to programming tasks: A prototype implementation of a framework for automatic and manual evaluation and scoring of java programming tasks [my translation].* Master's thesis, University of Oslo. Oslo, Norway.

Salleh, N., Mendes, E., & Grundy, J. C. (2011). Empirical studies of pair programming for CS/SE teaching in higher education: A systematic literature review. *IEEE Transactions on Software Engineering*, *37*(4), 509–525.

Sandelands, L. E. (1990). What is so practical about theory? Lewin revisited. *Journal for the Theory of Social Behaviour*, *20*(3), 235–262.

Schmidt, F. L., Hunter, J. E., McKenzie, R. C., & Muldrow, T. W. (1979). Impact of valid selection procedures on work-force productivity. *Journal of Applied Psychology*, *64*(6), 609–626.

Schmitt, N. (1996). Uses and abuses of coefficient alpha. *Psychological Assessment*, *8*(4), 350–353.

Seppälä, O. (2012). *Advances in assessment of programming skills.* Doctoral dissertation, Aalto University. Espoo, Finland.

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

Shah, H. B., Görg, C., & Harrold, M. J. (2010). Understanding exception handling: Viewpoints of novices and experts. *IEEE Transactions on Software Engineering*, *36*(2), 150–161.

Shanteau, J. (1992). Competence in experts: The role of task characteristics. *Organizational Behavior and Human Decision Processes*, *53*, 252–266.

Shanteau, J., Weiss, D. J., Thomas, R. P., & Pounds, J. C. (2002). Performance-based assessment of expertise: How to decide if someone is an expert or not. *European Journal of Operational Research*, *136*, 253–263.

Sheetz, S. D. (2002). Identifying the difficulties of object-oriented development. *Journal of Systems and Software*, *64*(1), 23–36.

Sheil, B. A. (1981). The psychological study of programming. *ACM Computing Surveys*, *13*(1), 101–120.

Sheppard, S. B., Curtis, B., Milliman, P., & Love, T. (1979). Modern coding practices and programmer performance. *IEEE Computer*, *12*(12), 41–49.

Shiffrin, R. M. (1988). Attention. In R. C. Atkinson, R. J. Herrnstein, G. Lindzey, & R. D. Luce (Eds.), *Stevens' handbook of experimental psychology* (Second ed., Vol. 2, pp. 789–811). New York: John Wiley & Sons.

Shiffrin, R. M., & Schneider, W. (1977). Controlled and automatic human information processing: II. Perceptual learning, automatic attending and a general theory. *Psychological Review*, *84*(2), 127–190.

Shneiderman, B. (1976). Exploratory experiments in programmer behavior. *International Journal of Computer and Information Sciences*, *5*(2), 123–143.

Shneiderman, B. (1980). *Software psychology: Human factors in computer and information systems.* Cambridge, MA: Winthorp Publishers.

Shneiderman, B., & Mayer, R. (1979). Syntactic/semantic interactions in programmer behavior: A model and experimental results. *International Journal of Computer and Information Sciences*, *8*(3), 219-238.

Shute, V. J. (1991). Who is likely to acquire programming skills? *Journal of Educational Computing Research*, *7*(1), 1–24.

Shute, V. J. (1992). *A comparison of learning environments: All that glitters ...* (Tech. Rep. No. AL-TP-1991-0042). Brooks Air Force Base, Texas: Manpower and Personnel Division, Air Force Human Resources Laboratory.

Shute, V. J., & Kyllonen, P. C. (1990). *Modeling individual differences in programming skill acquisition* (Tech. Rep. No. AFHRL-TP-90-76). Brooks Air Force Base, Texas: Manpower and Personnel Division, Air Force Human Resources Laboratory.

Shute, V. J., & Pena, C. M. (1990). *Acquisition of programming skill* (Tech. Rep. No. 89-54). Brooks Air Force Base, Texas: Manpower and Personnel Division.

Sijtsma, K., & Molenaar, I. W. (2002). *Introduction to nonparametric item response theory* (Vol. 5). Thousand Oaks, CA: Sage Publications.

Simon, H. A. (1990). Invariants of human behavior. *Annual Review of Psychology*, *41*(1), 1–20.

Simpson, D. (1973). Psychological testing in computing staff selection—a bibliography. *ACM SIGCPR Computer Personnel*, *4*(1–2), 2–5.

Sjøberg, D. I. K., Anda, B., Arisholm, E., Dybå, T., Jørgensen, M., Karahasanović, A., ... Vokáč, M. (2002). Conducting realistic experiments in software engineering. In *Proceedings of the International Symposium Empirical Software Engineering* (pp. 17–26).

Sjøberg, D. I. K., Anda, B., & Mockus, A. (2012). Questioning software maintenance metrics: A comparative case study. In *Proceedings of the ACM-IEEE International Symposium on Empirical Software Engineering and Measurement* (pp. 107–110).

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

Sjøberg, D. I. K., Hannay, J. E., Hansen, O., Kampenes, V. B., Karahasanovic, A., Liborg, N.-K., & Rekdal, A. C. (2005). A survey of controlled experiments in software engineering. *IEEE Transactions on Software Engineering*, *31*(9), 733–753.

Sjøberg, D. I. K., Yamashita, A., Anda, B., Mockus, A., & Dybå, T. (2013). Quantifying the effect of code smells on maintenance effort. *IEEE Transactions on Software Engineering*, *39*(8), 1144–1156.

Skiena, S. S., & Revilla, M. A. (2003). *Programming challenges: The programming contest training manual*. New York: Springer.

Smith, E. V., Jr. (2002). Detecting and evaluating the impact of multidimensionality using item fit statistics and principal component analysis of residuals. *Journal of Applied Measurement*, *3*(2), 205–231.

Snoddy, G. S. (1926). Learning and stability: A psychophysiological analysis of a case of motor learning with clinical applications. *Journal of Applied Psychology*, *10*(1), 1–36.

Soloway, E. (1986). Learning to program = learning to construct mechanisms and explanations. *Communications of the ACM*, *29*(9), 850–858.

Soloway, E., & Ehrlich, K. (1984). Empirical studies of programming knowledge. *IEEE Transactions on Software Engineering*, *SE-10*(5), 595–609.

Soloway, E., & Spohrer, J. C. (Eds.). (1989). *Studying the novice programmer*. Hillsdale, NJ: Lawrence Erlbaum.

Sonnentag, S. (1998). Expertise in professional software design: a process study. *Journal of Applied Psychology*, *83*(5), 703–715.

Sonnentag, S., & Frese, M. (2002). Performance concepts and performance theory. In S. Sonnentag (Ed.), *Psychological management of individual performance* (pp. 3–25). Chichester: John Wiley & Sons.

Sørlie, L. P. (2007). *Automatic and manual scoring of java tasks in programming tests: A prototype implementation of persistence for use in skill tests [my translation].* Master's thesis, University of Oslo. Oslo, Norway.

Stack Overflow. (2015, May 6). *2015 developer survey.* Retrieved from `http://stackoverflow.com/research/developer-survey-2015`

Stanislaw, H., Hesketh, B., Kanavaros, S., Hesketh, T., & Robinson, K. (1994). A note on the quantification of computer programming skill. *International Journal of Man-Machine Studies*, *41*(3), 351–362.

Sternberg, R. J., & Kaufman, J. C. (1998). Human abilities. *Annual Review of Psychology*, *49*(1), 479–502.

Stevens, S. S. (1946). On the theory of scales of measurement. *Science*, *103*(2684), 677–680.

Stoof, A., Martens, R. L., van Merriënboer, J. J. G., & Bastiaens, T. J. (2002). The boundary approach of competence: A constructivist aid for understanding and using the concept of competence. *Human Resource Development Review*, *1*(3), 345–365.

Surakka, S. (2007). What subjects and skills are important for software developers? *Communications of the ACM*, *50*(1), 73–78.

Svahnberg, M., Aurum, A., & Wohlin, C. (2008). Using students as subjects—an empirical evaluation. In *Proceedings of the 2nd International Symposium on Empirical Software Engineering and Measurement* (pp. 288–290).

Syang, A., & Dale, N. B. (1993). Computerized adaptive testing in computer science: Assessing student programming abilities. *ACM SIGCSE Bulletin*, *25*(1), 53–56.

Taatgen, N. A., Huss, D., Dickison, D., & Anderson, J. R. (2008). The acquisition of robust and flexible cognitive skills. *Journal of Experimental Psychology: General*, *137*(3), 548–565.

Taatgen, N. A., & Lee, F. J. (2003). Production compilation: A simple mechanism to model complex skill acquisition. *Human Factors: The Journal of the Human Factors and Ergonomics Society*, *45*(1), 61–76.

Thomas, R. C., Karahasanovic, A., & Kennedy, G. E. (2005). An investigation into keystroke latency metrics as an indicator of programming performance. In *Proceedings of the 7th Australasian Conference on Computing Education* (pp. 127–134).

Thurstone, L. L. (1926). The scoring of individual performance. *Journal of Educational Psychology*, *17*(7), 446–457.

Thurstone, L. L. (1927). A law of comparative judgment. *Psychological Review*, *34*(4), 273–286.

Tichy, W. F. (1998). Should computer scientists experiment more? *IEEE Computer*, *31*(5), 32–40.

Tichy, W. F. (2000). Hints for reviewing empirical work in software engineering. *Empirical Software Engineering*, *5*(4), 309–312.

Trendowicz, A., & Münch, J. (2009). Factors influencing software development productivity—state-of-the-art and industrial experiences. *Advances in Computers*, *77*, 185–241.

Truong, N., Roe, P., & Bancroft, P. (2004). Static analysis of students' Java programs. In *Proceedings of the 6th Australasian Conference on Computing Education* (pp. 317–325).

Tukiainen, M., & Mönkkönen, E. (2002). Programming aptitude testing as a prediction of learning to program. In *Proceedings of the 14th Workshop of the Psychology of Programming Interest Group* (pp. 45–57).

Turley, R. T., & Bieman, J. M. (1995). Competencies of exceptional and nonexceptional

software engineers. *Journal of Systems and Software*, *28*(1), 19–38.

Unsworth, N., Fukuda, K., Awh, E., & Vogel, E. K. (2015). Working memory delay activity predicts individual differences in cognitive abilities. *Journal of Cognitive Neuroscience*, *27*(5), 853–865.

Unsworth, N., Heitz, R. P., Schrock, J. C., & Engle, R. W. (2005). An automated version of the operation span task. *Behavior Research Methods*, *3*(37), 498–505.

Upshaw, H. S. (1968). Attitude measurement. In H. M. Blalock Jr. & A. B. Blalock (Eds.), *Methodology in social research* (pp. 61–111). New York: McGraw-Hill.

U.S. Department of Labor. (2014, January 30). *Occupational Outlook Handbook, 2014–15 Edition, Software Developers.* Retrieved from `http://www.bls.gov/ooh/computer-and-information-technology /software-developers.htm`

van der Maas, H. L. J., Kan, K.-J., & Borsboom, D. (2014). Comment: Intelligence is what the intelligence test measures. seriously. *Journal of Intelligence*, *2*(1), 12–15.

van der Maas, H. L. J., Molenaar, D., Maris, G., Kievit, R. A., & Borsboom, D. (2011). Cognitive psychology meets psychometric theory: On the relation between process models for decision making and latent variable models for individual differences. *Psychological Review*, *118*(2), 339–356.

van der Maas, H. L. J., & Wagenmakers, E.-J. (2005). A psychometric analysis of chess expertise. *American Journal of Psychology*, *118*(1), 29–60.

VanLehn, K. (1996). Cognitive skill acquisition. *Annual Review of Psychology*, *47*(1), 513–539.

Volmer, J. (2006). *Individual expertise and team performance: Results of three empirical studies.* Doctoral dissertation, Technischen Universität Carolo-Wilhelmina in Braunschweig. Braunschweig, Germany.

von Mayrhauser, A., & Vans, A. M. (1995). Program comprehension during software maintenance and evolution. *IEEE Computer*, *28*(8), 44–55.

von Mayrhauser, A., & Vans, A. M. (1996). Identification of dynamic comprehension processes during large scale maintenance. *IEEE Transactions on Software Engineering*, *22*(6), 424–437.

von Mayrhauser, A., Vans, A. M., & Howe, A. E. (1997). Program understanding behaviour during enhancement of large-scale software. *Journal of Software Maintenance: Research and Practice*, *9*(5), 299–327.

Wagenmakers, E.-J., Wetzels, R., Borsboom, D., & van der Maas, H. L. J. (2011). Why psychologists must change the way they analyze their data: The case of psi: Comment on Bem (2011). *Journal of Personality and Social Psychology*, *100*(3), 426–432.

Waldman, D. A., & Spangler, W. D. (1989). Putting together the pieces: A closer look at the determinants of job performance. *Human Performance*, *2*(1), 29–59.

Walston, C. E., & Felix, C. P. (1977). A method of programming measurement and estimation. *IBM Systems Journal*, *16*(1), 54–73.

Watson, C., Li, F. W. B., & Godwin, J. L. (2013). Predicting performance in an introductory programming course by logging and analyzing student programming behavior. In *IEEE 13th International Conference on Advanced Learning Technologies* (pp.

319–323).

Weinberg, G. M. (1971). *The psychology of computer programming.* New York: Van Nostrand Reinhold.

Weinberg, G. M., & Schulman, E. L. (1974). Goals and performance in computer programming. *Human Factors*, *16*(1), 70–77.

Welford, A. T. (1968). *Fundamentals of skill.* London: Methuen.

Wernimont, R. F., & Campbell, J. P. (1968). Signs, samples, and criteria. *Journal of Applied Psychology*, *52*(5), 372–376.

Whipkey, K. L. (1984). Identifying predictors of programming skill. *ACM SIGCSE Bulletin*, *16*(4), 36–42.

Wiedenbeck, S. (1985). Novice/expert differences in programming skills. *International Journal of Man-Machine Studies*, *23*(4), 383–390.

Wiedenbeck, S., Fix, V., & Scholtz, J. (1993). Characteristics of the mental representations of novice and expert programmers: An empirical study. *International Journal of Man-Machine Studies*, *39*, 793–812.

Wiedenbeck, S., & Ramalingam, V. (1999). Novice comprehension of small programs written in the procedural and object-oriented styles. *International Journal of Human-Computer Studies*, *51*(1), 71–87.

Wiedenbeck, S., Ramalingam, V., Sarasamma, S., & Corritore, C. L. (1999). A comparison of the comprehension of object-oriented and procedural programs by novice programmers. *Interacting with Computers*, *11*(3), 255–282.

Wilking, D., Schilli, D., & Kowalewski, S. (2008). Measuring the human factor with the Rasch model. In B. Meyer, J. R. Nawrocki, & B. Walter (Eds.), *Balancing agility and formalism in software engineering* (Vol. 5082, pp. 157–168). Berlin: Springer.

Williamson, D. M., Bejar, I. I., & Hone, A. S. (1999). 'Mental model' comparison of automated and human scoring. *Journal of Educational Measurement*, *36*(2), 158–184.

Wittman, W. W., & Keith, H. (2004). The relationship between performance in dynamic systems and intelligence. *Systems Research and Behavioral Science*, *21*(4), 393–409.

Wittmann, W. W., & Süß, H.-M. (1999). Investigating the paths between working memory, intelligence, knowledge, and complex problem-solving performances via Brunswik symmetry. In P. L. Ackerman, P. C. Kyllonen, & R. D. Roberts (Eds.), *Learning and individual differences: Process, trait, and content determinants* (pp. 77–108). Washington, DC: American Psychological Association.

Wohlin, C. (2002). Is prior knowledge of a programming language important for software quality? In *Proceedings of the 2002 International Symposium on Empirical Software Engineering* (pp. 27–34).

Wohlin, C. (2004). Are individual differences in software development performance possible to capture using a quantitative survey? *Empirical Software Engineering*, *9*(3), 211–228.

Woit, D., & Mason, D. (2003). Effectiveness of online assessment. *ACM SIGCSE Bulletin*, *35*(1), 137–141.

Woltz, D. J. (1988). An investigation of the role of working memory in procedural skill acquisition. *Journal of Experimental Psychology: General*, *117*(3), 319–331.

Wood, R. E. (1986). Task complexity: Definition of the construct. *Organizational Behavior and Human Decision Processes*, *37*(1), 60–82.

Wright, B. D., & Masters, G. N. (1979). *Rating scale analysis*. Chicago: Mesa Press.

Zhou, M., & Mockus, A. (2010). Developer fluency: Achieving true mastery in software projects. In *Proceedings of the 18th ACM SIGSOFT International Symposium on Foundations of Software Engineering* (pp. 137–146).

Paper I:

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

Gunnar R. Bergersen, Jo E. Hannay, Dag I. K. Sjøberg, Tore Dybå, and Amela Karahasanović

Proceedings of the 5th International Symposium on Empirical Software Engineering and Measurement, 2011, pp. 305–314.

Abstract

The skills of software developers are important to the success of software projects. Also, when studying the general effect of a tool or method, it is important to control for individual differences in skill. However, the way skill is assessed is often ad hoc, or based on unvalidated methods. According to established test theory, validated tests of skill should infer skill levels from well-defined performance measures on multiple, small, representative tasks. In this respect, we show how time and quality, which are often analyzed separately, can be combined as task performance and subsequently be aggregated as an approximation of skill. Our results show significant positive correlations between our proposed measures of skill and other variables, such as seniority, lines of code written, and self-evaluated expertise. The method for combining time and quality is a promising first step to measuring programming skill in both industry and research settings.

Keywords: skill, perforamance, time, quality, productivity.

1 Introduction

The skills of individual software developers have a large impact on the success of software projects. Also, differences in programming performance reported in the late 1960s, indicate that the authors believed levels of performance varied dramatically. Although more recent research (DeMarco & Lister, 1999; Prechelt, 1999) is more conservative in their assertions, companies that succeed in hiring the best people will nevertheless achieve great economic and competitive benefits (Glass, 2003; Schmidt & Hunter, 1998; Spolsky., 2007).

Individual differences in skill also affect the outcome of empirical studies. When evaluating alternative processes, methods, or tools, the effect of using a specific alternative may be moderated by skill levels. For example, in an experiment on the effect of a centralized versus delegated control style, the purportedly most-skilled developers performed better using a delegated control style than with a centralized one, while the less-skilled developers performed better with the centralized style (Arisholm & Sjøberg, 2004). In another experiment, skill had a moderating effect on the benefits of pair programming (Arisholm, Gallis, Dybå, & Sjøberg, 2007).

However, determining the skill level of software developers is far from a trivial task. In the work life, there are common-sense guidelines from experienced practitioners on how to distinguish the good from the bad (Spolsky., 2007). But there seems to be consensus that this crucial human resource management task remains difficult. Often, job recruitment personnel use tests that purport to measure a variety of traits, such as general cognitive abilities (intelligence), values, interests, and measures of personality, to predict job performance (Campbell, 1990). Research has, however, established that work sample tests in combination with General Mental Ability (GMA) testing are among the best predictors of job performance (Schmidt & Hunter, 1998). GMA is a general aspect of intelligence and is best suited for predicting performance on entry-level jobs or job-training situations. By contrast, work sample tests are task-specific and are integrated closely with the concept of job skill (Ericsson, Charness, Feltovich, & Hoffman, 2006). Although the predictive validity of standardized work samples exceed that of GMA alone, these predictors seem to yield the best results when combined (Schmidt & Hunter, 1998).

In the context of empirical studies in software engineering, the notion of programming skill is generally not well founded. This has led to studies that failed in adequately correcting for bias in quasi-experimental studies (Kampenes, Dybå, Hannay, & Sjøberg, 2009). Often the more general concept of *programming expertise* is used, with little validation. For example, in a recent study (Hannay, Arisholm, Engvik, & Sjøberg, 2010), we conceptualized programming expertise as the level of seniority (junior, intermediate, senior) of the individual programmer as set by their manager. While bearing some relevance to the consultancy market, this conceptualization is not sufficient to capture the skill of individual programmers. The concepts of expertise and skill are also operationalized in questionable ways in other domains; see (Hærem, 2002) for a survey of operationalizations in IT management.

The focus of this paper is as follows. Given a small set of programming tasks, how does one infer the candidates' programming skill from both the quality of the task solutions

and the time spent performing the tasks? It is well recognized that the combination of task quality and time is essential to define skill (Ericsson et al., 2006; Fitts & Posner, 1967), but how to combine them in practice is challenging. For example, how does one rank programmers who deliver high quality slowly, relative to those who deliver lesser quality more quickly? This paper addresses such challenges and proposes a method for combining quality and solution time into a single ordinal score of performance (i.e., *low, medium, high*). Multiple performance scores are then aggregated to form an ordinal approximation of programming skill. The method is demonstrated by using data from two existing experiments.

Section 2 gives the theoretical and analytical background for skill as a subdomain of expertise and discusses how quality and time are currently dealt with. Section 3 describes how time and quality were combined as programming performance on tasks. Section 4 reanalyzes two existing data sets according to the arguments given in the previous sections. Sections 5 and 6 discuss the results and conclude the paper.

2 Background

2.1 Expertise

Expertise is one of the classic concepts of social and behavioral science. Expertise is usually related to specific tasks within a given domain and does not in general transfer across domains or tasks (Ericsson et al., 2006; Shanteau, 1992). Expertise has several aspects; we present five of these in Figure 1(a). The aspects are all related. For example, in the usual descriptions of skill acquisition (Anderson, 1982; Dreyfus, Dreyfus, & Athanasiou, 1988; Fitts & Posner, 1967), which is a subdomain of expertise, an individual starts by acquiring declarative knowledge, which for experts is qualitatively different in representation and organization compared to novices (Ericsson et al., 2006; Wiedenbeck, Fix, & Scholtz, 1993). Next, through practice, declarative knowledge is transformed into procedural skill, which at first is slow and error-prone (Fitts & Posner, 1967). However, though extended experience, performance improves and experts tend also to converge on their understanding of the domain in which they are an expert as well (Shanteau, 1992; Shanteau, Weiss, Thomas, & Pounds, 2002) (i.e., consensual agreement). Experts also regard themselves as being experts, for example, through the use of self-assessments. Ultimately, the desired effect of expertise is superior performance on the tasks in which one is an expert. In our context, this is performance on real-world programming tasks. However, predicting future job performance by observing actual job performance is unreliable and inefficient (Campbell, 1990). It is therefore desirable to design quick tests based on how well an individual reliably performs on representative tasks (Ericsson et al., 2006).

2.2 Skill

We generally understand skill as performance on small representative tasks. Note, though, that inferring skill from a reliable level of performance on representative tasks is not the

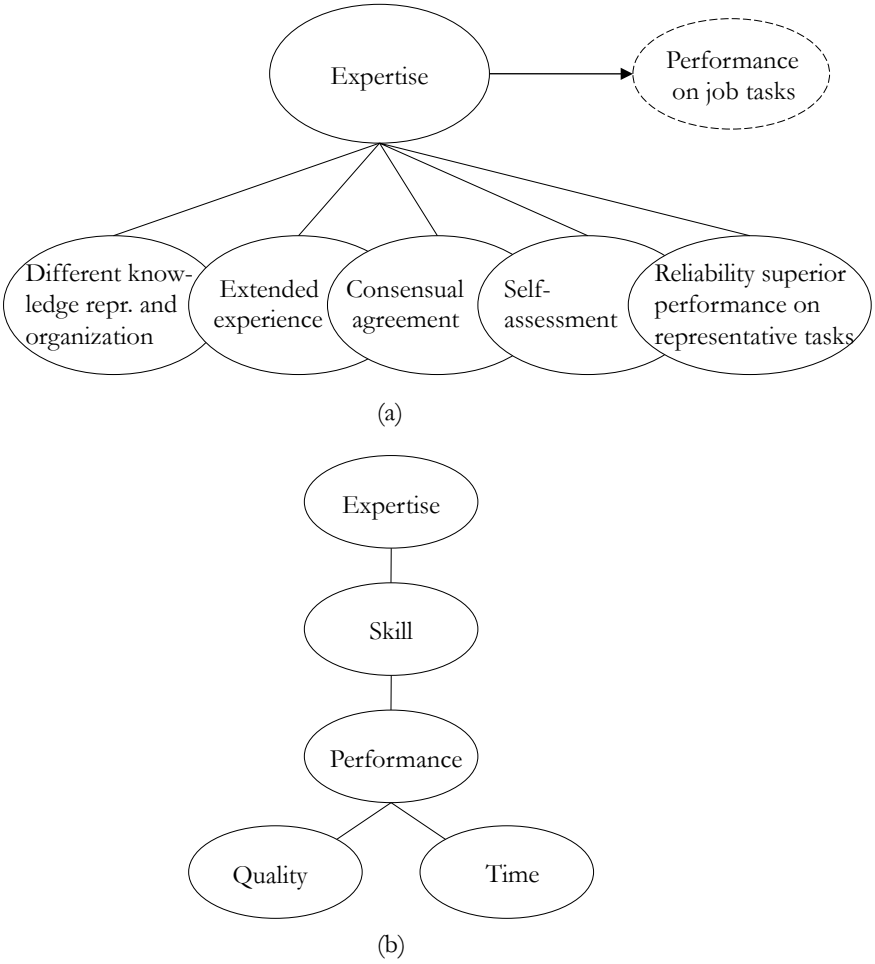


Figure 1: Expertise (a) and skill as one aspect of expertise (b) with relations to time and qualtiy as variables though the concept of performance. The desired effect of expertise is superior performance on job tasks.

same as defining it by performance on the job. Representative tasks in our context are those smaller tasks which merely represent real-world tasks, and for which there are well-defined measures of performance (Ericsson et al., 2006). Additionally, such measures are typically regarded as situations of maximum performance, whereas behavior on the job would constitute typical performance (Campbell, 1990). Motivation plays a central role in predicting typical performance in a job situation (see Beecham, Baddoo, Hall, Robinson, & Sharp, 2008 for an overview), whereas potential positive or negative consequences for a test-taker would affect a situation demanding maximum performance.

Generalizing from performance on small representative tasks to performance on the job requires an understanding of key mechanisms at play shared between tasks in the two settings. This is theory-driven generalization (Shadish, Cook, & Campbell, 2002), based on the economy of artificiality (Hannay & Jørgensen, 2008). In the absence of, or as a complement to, strong theory, it is useful to seek confirmation of how well skill measures coincide with other aspects of expertise. This is relevant for skill in programming.

Anderson and others (Anderson, 1982; Anderson, Conrad, & Corbett, 1989) investigated programming skill from a psychological perspective. They reported that both coding time and the number of programming errors decreased as skill improved. Further, programming in LISP required the learning of approximately 500 if-then rules. The acquisition of these rules followed a power-law learning curve: the improvement in performance was largest at first and then decelerated until an asymptote was reached. Thus, the relationship between amount of practice (extended experience) and performance was non-linear. However, if amount of practice and performance were logarithmically transformed, an approximately linear trend was observed. This phenomenon is widely observed and is often referred to as the *log-log law of practice* (Newell & Rosenbloom, 1981).

Fitts and Posner have extensively studied skill acquisition. Within many different domains of expertise, they found that with increased skill, the number of errors in performance decreases and the speed with which a task is executed increases. Regarding measures of skill, they state: "[t]he measure should take into account the length of time taken to perform a skill as well as the accuracy with which it is performed" (Fitts & Posner, 1967, p. 85). Therefore, time and quality (the latter being a generalization of accuracy) are intimately linked to skill, and the term performance is linked to all three concepts. Because skill affects performance (Campbell, 1990), we can hierarchically structure the five concepts *expertise, skill, performance, time,* and *quality* as shown in Figure 1(b). From the top, expertise, which should affect job performance, is a generalization of skill. Beneath, skill is inferred from performance on multiple tasks where reliably superior performance is a requirement. At the lowest level, time and quality, in combination, dictate whether programming performance overall is, say, low or high.

2.3 Measures of Programming Performance

It is common in empirical software engineering to deal with quality and time separately when analyzing results; that is, one studies performance first in terms of quality and then in terms of time, often under the assumption that a solution meets some particular criterion for correctness (see, for example, Arisholm et al., 2007; Arisholm & Sjøberg, 2004). We acknowledge that for many studies, this is acceptable. However, when the purpose is to characterize individual differences, problems may occur.

Time is a ratio variable with an inverse relation to performance (i.e., less time implies better performance). Quality, on the other hand, may consist of a plethora of variables where each one may have complex relations with each other and where all often cannot be optimized simultaneously (McCall, 1994). Further, depending on how quality is operationalized, these variables may have different scale properties (i.e., nominal, ordinal, interval, or ratio). Therefore, when aiming to characterize individual differences, one may (a) disregard quality and report differences only in time spent or (b) only analyze time for observations surpassing some specific level of quality (often correctness), thereby adhering to the basic principle delineated by Thorndike and others in the 1920s: "the more quickly a person produces the *correct* response, the greater is his [ability]" (Carroll, 1993, p. 440, emphasis added). It is also possible to (c) devise acceptance tests that force everyone to work until an acceptable solution is achieved. Generally, we regard this as perhaps the

most viable approach today, because variability in performance is expressed through time spent in total. However, by using (b) or (c), large portions of the dataset may be excluded from analysis, in particular when the proportion of correct solutions is low.

At the most fundamental level of the time/quality tradeoff problem, it is not clear how to place programmers who deliver high quality slowly relative to those who deliver lesser quality more quickly. In the datasets that are available to us, correctness and time are often negatively correlated. This indicates that the longer it takes to submit a solution, the lower is the likelihood of the solution being correct. Although this may seem contrary to what may be expected—that higher quality requires more time while lower quality requires less—there are two important distinctions to be made: First, there is a difference between quality in general and correctness specifically. Second, there is also a difference between *within-subject* and *between-subject* interpretations (Borsboom, 2005). When a correct solution can be identified, a highly skilled individual can arrive at this solution in less time and with higher quality than a less capable individual (between-subject interpretation). But given more time, a single individual can generally improve an existing solution (within-subject interpretation).

Another challenge is identifying to what degree individual performance in a study is stable at a specific level, or high/low from one time to another. One way to address such concerns is to use multiple indicators of performance (Basilevsky, 1994; Gorsuch, 1983). Based on the same principles for combining time and quality as performance delineated in this paper, we have already advanced the measurement of skill using multiple indicators of performance (Bergersen & Gustafsson, 2011). However, a more detailed discussion of these principles involved is needed. It is to this discussion we will now turn.

2.4 Using the Guttman Structure for Time and Quality

The two-by-two matrix in Figure 2 has two possible values for quality (*low, high*) and two possible values for time (*slow, fast*). It is easy to agree that in this simplified example, "high performance" is represented by the upper right quadrant (fast and high quality) whereas "low performance" is represented by the lower left quadrant (slow and low quality). Further, it should also be straightforward to agree that the two remaining quadrants lie somewhere between these two extremes, say, "medium performance". However, which one of the two alternatives one would rate as superior, or whether they should be deemed equal, is a value judgment: in some instances, "fast and low quality" may be deemed superior to "slow and high quality". To address how performance should relate to different values for time and quality, we propose to use the principles delineated by Louis Guttman.

The Guttman scale was originally developed to determine whether a set of attitude statements is unidimensional (Guttman, 2007). In Guttman's sense, a perfect scale exists if a respondent who agrees with a certain statement also agrees with milder statements of the same attitude. The Guttman-structured scoring rules that we propose utilize the same underlying principle as the Guttman scale, although at a lower level of abstraction (a scale is a formal aggregation of indicators, whereas the structure we employ refers to the indicators themselves). The approach utilizes general principles as delineated

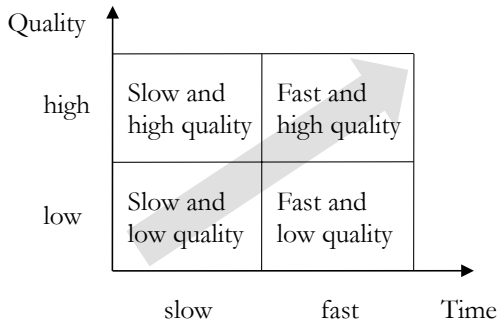


Figure 2: An example of scoring time and quality as performance based on value judgments.

by others (Andrich, 1988), but which we have only addressed somewhat informally so far (Bergersen, 2011).

With a Guttman structure it is possible to rank combinations of quality and time relative to each other as well as being explicit about how different tradeoffs in time and quality are scored. Performance on a programming task is thus determined by a series of well-ordered thresholds. Combined, these thresholds constitute a set of monotonically ordered response categories (i.e., an ordinal variable) in relation to performance. Surpassing a given threshold implies that all thresholds below it have been passed as well. For example, for a score of, say 2 (of 5 possible), the thresholds for obtaining scores of 1 and 2 must have been passed, while the threshold for obtaining score 3 has failed. Quality can be deliberately emphasized over time (or vice versa) by adjusting score categories accordingly. A task that differentiates more on quality aspects may, further, be scored on multiple quality categories and a task that also differentiates more on time aspects may have more time categories.

3 Research Methods

This section describes how time and quality were combined as task performance using multiple indicators. Using two data sets, we show how scoring rules for tasks were operationalized and reanalyzed.

3.1 Data Set 1

The first data set we reanalyzed is from a one-day study (Arisholm & Sjøberg, 2004). In the experiment, 99 consultants from eight software consultancy companies and 59 undergraduate and graduate students were paid to participate. The independent (treatment) variable in the experiment was the control style of the code (whether it is centralized or delegated). Five programming tasks were presented in succession to the subjects during the experiment. The first task i_1 (the pretest) was identical for both experimental groups, and the four next tasks i_2–i_5 involved the independent variable. We analyze only the first four tasks here due to challenges in applying the last task to our purpose (see Arisholm & Sjøberg, 2004 for why).

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

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

The matrix representation of this scoring rule is illustrated in Table 1(a). In using this structure, a solution must be correct before time is taken into consideration. Increasing scores for time are, further, only awarded in order ($T3$ before $T2$ and $T2$ before $T1$). Additionally, the precedence of quality in this type of scoring rule reflects the view that, for this study, we do not consider an incorrect solution to reflect high or medium performance, even when it is developed quickly.

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

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

Table 1: Score according to time and quality thresholds

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

(a) Dataset 1

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

(b) Dataset 2

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

It should be noted that the relationship between the score vector and the overall skill score is a many-to-one (surjective) function. For example, an individual with correct but very slow solutions for all four tasks when using $Q1{\times}T3{\times}T2{\times}T1$ receives the sum score of 4. An individual with a single correct solution with very fast time but the other three tasks incorrect would also receive the same sum score. Obviously, it is incorrect to characterize the latter instance as "reliably (superior) performance" because the individual exhibits superior performance on only a single task. We return to this issue in Section 3.3.

3.2 Data Set 2

The second data set stems from three quasi-experiments which all used the same programming tasks. During a one-day study, the subjects were required to perform three different change tasks in a library application system of 3600 LOC, containing 26 Java classes. Two of the studies used students as subjects; one study used professionals. The study in (Karahasanović & Thomas, 2007) investigated the effects of different comprehension strategies using 38 subjects; the study in (Karahasanović, Levine, & Thomas, 2007) compared feedback collection and think-aloud methods for 34 subjects; and the study in (Kværn, 2006) studied the effects of expertise and strategies on program comprehension for 19 subjects. Additionally, the same pretest task as in Dataset 1 (i_1) was used. However, one of the studies had missing data for the last change task, thereby reducing the number of available tasks for analysis from four to three. Human graders scored the quality of each task on the following scale:

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

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

2: functional anomalies, one or more subtasks are achieved

3: functionally correct, major visual anomalies

4: functionally correct, only minor cosmetic anomalies

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

We defined a Guttman structure $Q1{\times}Q2{\times}Q3$ for the dimension of quality as follows: The original scoring categories 0 and 1 should be collapsed into a single category, because

neither might be preferred over the other. Thus, variable $Q1$ was defined as "one or more subtasks achieved" (category 2 above). Next, the $Q2$ variable was "functionally correct, but with major visual anomalies allowed" (category 3). Finally, we regarded the level of detail used for separating functionally correct with minor visual anomalies (category 4) and a "perfect solution" (category 5) as somewhat arbitrary; these two categories were therefore combined for $Q3$. For the time dimension, we used $T1{\times}T2$ to partition the time for those individuals who passed $Q3$ into three groups. The matrix representation of this scoring rule, denoted $Q1{\times}Q2{\times}Q3{\times}T2{\times}T1$, is provided in Table 1(b).

We also devised alternative scoring rules using one and two dichotomous quality variables: $Q1{\times}Q2{\times}T2{\times}T1$ does not separate between major and minor visual anomalies when the solution is otherwise correct. And finally, $Q1{\times}T2{\times}T1$ only separates between functionally correct solutions and those that are not functionally correct, with no attention given to visual anomalies. Finally, we devised scoring rules for $Q{+}T$, $Q{+}lnT$, Q, and T using the same procedure as in Dataset 1, but using three tasks instead of four.

3.3 Analysis Method and Handling of Missing Data

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

3.3.1 Using Exploratory Factor Analysis

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

3.3.2 Inspecting External and Internal Results

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

Conversely, all the reported *internal results* are influenced by how each scoring rule was constructed. For internal results, we used the proportion of explained variance for the first Principal Component (PC), which is analogous to the sum score, as the signal-to-noise ratio for each scoring rule. Cronbach's α was used as an estimate of the internal consistency of the scores. To ascertain the applicability of each score operationalization, we used confirmatory factor analysis. We report the Root Mean Square Error of Approximation (RMSEA) using Amos™ 18.0. RMSEA is a parsimony-adjusted index, as it favors models with fewer parameters. Further, RMSEA will be influenced negatively

if level of performance is not consistent over multiple tasks (see Section 3.1). We used a tau-equivalent reflective measurement model with multiple indicators (Loehlin, 2004). This implies that all tasks receive the same weight when calculating the sum score. All scoring rules are further regarded as ordinal scale approximations of skill.

3.3.3 Handling of Missing Data

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

4 Results

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

4.1 External Correlations

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

The other experience variables were self-assessed. Years of programming experience (*lnProfExp*) is an aspect of extended experience. In general, the correlations for this variable were low and insignificant for all scoring alternatives, but were slightly improved having been logarithmically transformed (a justifiable transformation given the log-log law of practice). Java programming expertise (*SEJavaExp*) is a Likert scale variable ranging from *novice* = 1 to *expert* = 5. This variable was significantly and positively correlated around 0.3 with all of the scoring alternatives for Dataset 1. However, for Dataset 2, the correlations were lower and less systematic; caution should be shown when

interpreting this result due to the low number of observations (n). Overall, self-assessed Java programming expertise seems to have a non-linear but monotonically increasing relation to the proposed score operationalizations. Lines Of Code ($lnLOCJava$) written in Java is a self-estimated variable with positive skew and kurtosis, but approximates a normal distribution after logarithmic transformation. All scoring operationalizations were significantly and positively correlated with LOC (around rho = 0.3) with two exceptions: Q in Dataset 1 and T in Dataset 2.

4.2 Internal Fit Indices

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

The proportion of explained variance by the first PC ($\%E$) indicates the signal-to-noise ratio for each score operationalization. The additive scoring rules ($Q+T$, $Q+lnT$) have the highest proportion of explained variance: logarithmic transformation before standardization of time produces additional explained variance.

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

Finally, we report how the different scoring alternatives fit according to confirmatory factor analysis. Lower RMSEA values signify better fit: values less than or equal to 0.05 indicate close approximate fit, values between 0.05 and 0.08 indicates reasonable error of approximation, and values above 0.10 suggest poor fit (Kline, 2005). For Dataset 1, $Q1{\times}T2{\times}T1$ and $Q1{\times}T3{\times}T2{\times}T1$ display reasonable model fit (< 0.08) whereas T and Q have poor fit. The additive scoring rules alternatives lie somewhere in between, and the logarithmically-transformed version ($Q+lnT$) has better overall fit than the untransformed version. Q for Dataset 2 has poor fit as well, even though this dataset has better overall confirmatory fit, despite the lower statistical power as can be seen by the wider 90% Confidence Intervals (CI).

Nevertheless, upon inspecting the lower CI of Dataset 1, there is sufficient statistical power to state that T fits poorly. However, as is evident by the upper CI of all alternatives, there is not sufficient statistical power to claim support for a close model fit for any of these alternatives either; $Q1{\times}Q2{\times}T2{\times}T1$ in Dataset 2 is overall the best fitting alternative with upper CI slightly above 0.10.

In summary, an analysis of Q and T separately seems problematic in terms of some correlations, relatively low explained variance, and problematic confirmatory fit in three out of four instances. The additive scoring rules show the highest levels of explained variance and internal consistency, but they display some problems with confirmatory model fit. Overall we found the best-fitting score operationalizations to be $Q1{\times}T3{\times}T2{\times}T1$ for Dataset 1 and $Q1{\times}Q2{\times}T2{\times}T1$ for Dataset 2.

Table 2: Correlations and confirmatory model fit of scoring alternatives

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

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

4.3 Details for Factors in Data Sets 1 and 2

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

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

To verify that the scoring rules predict performance on other programming tasks, we used the $Q1{\times}T2{\times}T1$ scoring rule of Dataset 1 to separate individuals into *low* and *high* skill groups. Using the results for task i_5, which is not a part of the investigated scoring rules, as a dependent variable and above/below mean sum score of tasks i_1–i_4 as the independent variable, we found that the high group performed much better than

Table 3: Correlations for dataset 1 and 2

Scoring rule	(1)	(2)	(3)	(4)	(5)	(6)
Q (1)	—	0.36	0.76	0.82	0.75	0.71
T (2)	0.33	—	0.82	0.64	0.73	0.74
$Q{+}\ln T$ (3)	0.72	0.85	—	0.92	0.95	0.94
$Q1{\times}T1$ (4)	0.80	0.70	0.92	—	0.96	0.96
$Q1{\times}T2{\times}T1$ (5)	0.75	0.78	0.96	0.95	—	0.98
$Q1{\times}T3{\times}T2{\times}T1$ (6)	0.72	0.81	0.97	0.96	0.98	—

(a) Dataset 1

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

(b) Dataset 2

the low group: 67.1% had correct solutions for i_5 while the corresponding results for the low group was 27.8% correct (time could not be analyzed this way for i_5; see Arisholm & Sjøberg, 2004 for an explanation). We could further confirm that the high group had written significantly more LOC in Java and had more programming and Java experience as well. Moreover, by using three groups instead of two (i.e., *low, medium* and *high*) in terms of overall skill, similar results were obtained: the groups are well ordered according to external background variables, as well as on performance for i_5. For Dataset 2, we performed the same analysis, using the quality of task i_4 as the dependent variable and the sum score of i_1–i_3 as the independent variable. All correlations between the Guttman-structured scoring rules and the quality of i_4 were large ($n = 52$, rho = 0.51–0.53) and significant ($p < 0.001$).

We were also able to identify the treatment effect of Dataset 1. The dichotomous treatment variable was significantly and moderately correlated with the second PC of $Q1{\times}T3{\times}T2{\times}T1$ (rho = 0.42) and for the two additive scoring rules as well (rho = 0.36). Further, the treatment effect was not correlated with any of the first PC of the proposed scoring rules, suggesting that the effect of the treatment in this study is less than the individual variability. This implies that unless some degree of experimental control is available for individual variability—for example, through the use of pre-tests (Kampenes et al., 2009; Shadish et al., 2002)—an experimenter would require many more subjects in a study to achieve the same statistical power. Perhaps worse, effects of practical importance might go undetected in the early phases of a research project.

5 Discussion

We begin by discussing the implications for research and practice when combining time and quality in empirical studies on programmers. Next, we discuss limitations and address how this work can be expanded in the future.

5.1 Implications for Research and Practice

In this reanalysis, we have presented a method for combining time and quality as an ordinal variable of performance. Results show that when programming performance on multiple tasks were aggregated, significant and positive correlations with skill could be obtained with several relevant expertise-related background variables. The strongest and most consistent correlations were obtained for LOC (around 0.3), which is highly similar to the value (of 0.29) reported in (Bergersen & Gustafsson, 2011). Seniority and self-evaluated expertise indicated more mixed results. For seniority in Dataset 1, statistical significant positive correlations could only be obtained when students were removed from analysis. For general programming experience, low and insignificant correlations were present. However, (Bergersen & Gustafsson, 2011) reports a correlation of 0.29 between skill and months of Java programming experience; this may indicate that more precision (months instead of years) as well as specificity (*Java* experience as opposed to *general* programming experience) is required to yield higher correlations for this variable.

Nevertheless, using correlations as the only criteria for evaluating tests poses a problem; what the actual correlation is between a test score and a background variable will never be known for certain (Borsboom, 2005). We therefore used confirmatory factor analysis to investigate whether performance on multiple tasks could be considered as "reliable (superior) performance" according to established literature on expertise and skill. We found unacceptable model fit in three of four instances when analyzing quality or time as separate variables. Furthermore, the only well-fitting variable (T in Dataset 2) did not accord with expectations of correlations with expertise background variables. Therefore, some concerns seem to exist when performance as a dependent variable is operationalized as time alone, or quality alone, in programming experiments. Such a problem will, however, remain undetected unless multiple tasks are present to be compared.

This study also demonstrates the importance of considering experiment constraints when analyzing individual variability of performance. For example, when loose time limits, or no limits, are used in a study and most subjects solve a task correctly, it is entirely plausible that, when analyzing correct solutions, between-subject variability is mostly present in the time variable. Conversely, if strict time limits are used and few subjects are able to finish on time, variability will mainly reside in the quality variable. Hence, the scoring of time and quality as a combined variable is dependent on the instrumentation as well as upon empirical results. This also implies that no universal scoring rule exists that applies equally well to all tasks in all situations.

For time as a variable, we generally obtained stronger and more consistent correlations when using non-parametric correlations and untransformed experience variables or parametric correlations with logarithmically-transformed experience. We concede that other transformations may be applicable as well, such as 1/time. Nevertheless, variables analyzed in this manner should almost always be plotted first and verified against theoretical expectations; there is often little theoretical rationale for expecting a priori that variables have a linear relation, even if each variable displays an approximate normal distribution. A similar caution should be observed when analyzing variables of quality. For example, for Dataset 2, it is problematic to assume that an increase in score from 2 to 3 (a difference of 1) amounts to the same increase as from 3 to 4; improving a correct solution from "major" to "minor visual anomalies" may require negligible time, whereas the improvement from one functionally correct subtask to a fully functionally correct solution may require substantial effort. Hence, quality variables should be treated as ordinal when in doubt and they are, further, most likely to have non-linear relations to other variables. Therefore, non-parametric correlations should be the first, not last, resort.

For the scoring rules that combine time and quality, we see two main competing alternatives in the present analysis: the additive scoring rules treated "slow and high quality" and "fast and low quality" (Figure 2) as "medium performance". The Guttman-structured alternatives treated "fast and low quality" differently; quality had to be at an (operationally defined) acceptable level, before additional score points were awarded for time. Both alternatives demonstrated highly similar results in terms or correlations as well as rank order of individuals in terms of skill. Specifically, the two alternatives were highly similar in terms of the rank ordering of individuals in Dataset 1. It is therefore interesting to note that confirmatory model fit was much stronger in favor of the Guttman-structured

scoring rules in Dataset 2 where less agreement between the Guttman-structured and additive scoring rules exist.

Some challenges are also present with using the additive scoring rules; partial scores of performance are awarded for fast solutions even though no improvement may be present. This implies that an individual who chooses not to participate seriously in a study will receive a higher score than an individual who seriously attempts all tasks, but fails.

Finally, given that some merit for the use of Guttman structures is established at this point, a practical issue arises around the number of score categories for time and quality. Because neither quality alone, nor time alone, fits the data particularly well in most instances, a sensible choice is to start at the extremes and work towards a compromise using the indices and recommendations reported here. Based on our experience, each score operationalization will usually display a peak of overall fit at some point; for example, when adding $Q3$ to $Q1{\times}Q2{\times}T2{\times}T1$ in Dataset 2, confirmatory model fit began to decrease while the proportion of explained variance only marginally improved.

5.2 Limitations

The weaknesses in the present analysis have to do with the two sources of data as well as the methods we have used to analyze time and quality as an aggregated variable.

The experimental treatments in both datasets add noise to the data, which probably entails that all investigated scoring rules show worse fit than they would in data sets without such treatment. However, for this issue, it is at least true that the conditions were equal throughout our comparisons. There are also statistical independency problems between tasks. When a new task expands upon the solution of the previous task (i.e., a testlet-structure), this is still held to constitute one observation. Furthermore, performance in both studies is affected by motivational components; some peer pressure was present for most subjects in Dataset 1, whereas we believe motivation was more variable in Dataset 2. Finally, we used factor analysis on discrete ordinal variables as well as on dichotomous variables for quality. Optimally, one should use Bayesian estimation (Lee, 2007) or polychoric correlation matrices if the aim is to conclude more strongly. It is nevertheless somewhat common to see, for example, Likert scale variables analyzed using factor analysis, with a claim that the resulting factor score has interval-scale properties.

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

Other, perhaps, justifiable objections to our approach may be that multiple-factor (e.g., two rotated factors) or simplex-structured models should have been investigated. For Dataset 1, we have already done these analyses, but with inconclusive overall results and poor confirmatory fit. Therefore, we have chosen not to report these here. We could also have used non-linear regression or neural networks to maximize explained variance. However, by doing so, it would have been more difficult to interpret the results and the

implied models for measurement would lack parsimony. Better, and more recent, models for measurement exist than the one we have used here. We now turn to such improvements where patterns in the score vector impose additional constraints on what scoring rules can be considered well-fitting overall.

5.3 Reccomendations for Future Research

In software engineering it is somewhat common to read about differences in programming performance ranging in an order of magnitude or more. However, as illustrated in Figure 1, these differences pertain to performance *per se*, and not necessarily to individuals. For ratio comparisons of individuals, a quantitative scale is required in which units are separated by equal distances and where zero is well defined. This may be attempted though axiomatic measurement, which is measurement in a stricter sense than the putative measures discussed here. But it is unlikely that advances in this direction will happen in the near future (see Borsboom, 2005 for present challenges). Nevertheless, we consider Rasch analysis (Andrich, 1988) as a useful intermediary step for obtaining interval-scale approximations of programming skill. In this model, task difficulty is also an integral part of the measurement process, something the current model for measurement we employ (i.e., classical test theory) is lacking.

We have identified some additional directions for future work. For example, two or more internally consistent scoring perspectives based on the same data could be devised in which more score points are awarded either to quality or to time. By estimating skill for an individual based on these two approaches separately, it may be possible to make inferences about an individual's preferred working style during a test. If an individual is ranked higher on one score operationalization than the other, the relative difference may provide information about the value system an individual has for "fast and low quality" versus "slow and high quality" solutions. Further, different scoring rules on a task-by-task basis should be investigated as well. Although we applied the same scoring principle to all tasks in this study, there should be no *a priori* reason for requiring that a scoring rule must fit all tasks equally well to be valid.

Future work using Guttman-structured scoring rules for time and quality may also be directed at how to avoid the degradation of time as a continuous variable into discrete categories. However, any solution to this problem must somehow account for the likelihood that time will have skewed distributions. Although the additive scoring rule partially solves this problem by logarithmically transforming time, we would like to see more alternatives in the future that differ in how quality and time is combined as well as how measurement is conceptualized.

6 Conclusion

In a reanalysis of two existing studies, we have shown through multiple score operationalizations how time and quality for programming task solutions may be meaningfully combined as an ordinal variable of performance. By aggregation of multiple performance variables into an ordinal skill score approximation, we obtained significant and positive

correlations with relevant experience variables for all score alternatives that combined time and quality. Some degree of internal consistency was present, but only at a level high enough to roughly characterize group differences. We also showed that from correlations alone, it was difficult to choose what the so-called "best" scoring alternative might be. However, by using confirmatory factor analysis, we found some support for the Guttman-structured scoring rules over other alternatives, such as analyzing time or quality independently. Statistical power in our reanalysis was, however, not sufficient to conclude strongly, even though upper confidence intervals for some scoring alternatives almost indicate reasonable model fit for some alternatives.

This study implies that if time or quality is analyzed separately as a dependent variable, and experiment results are contrary to theoretical expectations, a reanalysis using the principles delineated here may be warranted. Threats to validity may further be present if the dependent variable of a study is not a monotonically increasing function of expertise or if multiple operationalizations of the same treatment do not indicate similar interpretations of the results. How individual variability in a study affects time, quality, or the two variables in combination also warrants closer scrutiny; instrumentation issues, such as time limits or task difficulty match with subject population, may also influence results in unexpected directions. Finally, we recommend the use of scoring rules that converge in terms of measurement while being parsimonious and interpretable.

Alternative score operationalizations in which time and quality are combined often were more similar to each other than they were to time or quality analyzed separately. Therefore, more attention is required to analyze how "performance" is operationalized in programming studies. We call for others to reanalyze their existing datasets where time and quality variables have been collected, and to score time and quality as a combined variable of performance. Such new datasets might even contain relevant background variables that can be further used to inform the strength of relations with performance and background variables, thereby making meta-analytic studies feasible in the future.

Acknowledgment

The authors are grateful to Erik Arisholm, Kaja Kværn, Annette Levine, Richard Thomas, and Gøril Tømmerberg for use of their data, and to Jan-Eric Gustafsson for detailed feedback on an earlier analysis of Dataset 1.

References

Anderson, J. R. (1982). Acquisition of cognitive skill. *Psychological Review*, *89*(4), 369–406.

Anderson, J. R., Conrad, F. G., & Corbett, A. T. (1989). Skill acquisition and the LISP Tutor. *Cognitive Science*, *13*(4), 467–505.

Andrich, D. (1988). *Rasch models for measurement* (No. 68). Sage Publications.

Arisholm, E., Gallis, H., Dybå, T., & Sjøberg, D. I. K. (2007). Evaluating pair programming with respect to system complexity and programmer expertise. *IEEE Transactions on Software Engineering*, *33*(2), 65–86.

Arisholm, E., & Sjøberg, D. I. K. (2004). Evaluating the effect of a delegated versus centralized control style on the maintainability of object-oriented software. *IEEE Transactions on Software Engineering*, *30*(8), 521–534.

Basilevsky, A. (1994). *Statistical factor analysis and related methods: Theory and applications*. New York: John Wiley & Sons.

Beecham, S., Baddoo, N., Hall, T., Robinson, H., & Sharp, H. (2008). Motivation in software engineering: A systematic literature review. *Information and Software Technology*, *50*(9–10), 860–878.

Bergersen, G. R. (2011). Combining time and correctness in the scoring of performance on items. In T. Nielsen, S. Kreiner, & J. Brodersen (Eds.), *Probabilistic models for measurement in education, psychology, social science and health* (pp. 43–44). Copenhagen.

Bergersen, G. R., & Gustafsson, J.-E. (2011). Programming skill, knowledge and working memory capacity among professional software developers from an investment theory perspective. *Journal of Individual Differences*, *32*(4), 201–209.

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

Campbell, J. P. (1990). Modeling the performance prediction problem in industrial and organizational psychology. In M. D. Dunnette & L. M. Hough (Eds.), *Handbook of industrial and organizational psychology* (Second ed., Vol. 1, pp. 687–732). Palo Alto, CA: Consulting Psychologists Press.

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

DeMarco, T., & Lister, T. (1999). *Peopleware: Productive projects and teams* (Second ed.). New York: Dorset House Publishing Company.

Dreyfus, H. L., Dreyfus, S. E., & Athanasiou, T. (1988). *Mind over machine: The power of human intuition and expertise in the era of the computer*. New York: The Free Press.

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

Fitts, P. M., & Posner, M. I. (1967). *Human performance*. Belmont, CA: Brooks/Cole.

Glass, R. L. (2003). *Facts and fallacies of software engineering*. Boston: Addison-Wesley Professional.

Gorsuch, R. L. (1983). *Factor analysis* (Second ed.). Hillsdale, NJ: Lawrence Erlbaum Associates.

Guttman, L. L. (2007). The basis for scalogram analysis. In G. M. Maranell (Ed.), *Scaling: A sourcebook for behavioral scientists* (pp. 142–171). Chicago: Aldine Pub. Co.

Hærem, T. (2002). *Task complexity and expertise as determinants of task perceptions and performance: Why technology structure research has been unreliable and inconclusive*. Doctoral dissertation, BI Norwegian Business School. Oslo, Norway.

Hannay, J. E., Arisholm, E., Engvik, H., & Sjøberg, D. I. K. (2010). Personality and pair programming. *IEEE Transactions on Software Engineering*, *36*(1), 61–80.

Hannay, J. E., & Jørgensen, M. (2008). The role of deliberate artificial design elements in software engineering experiments. *IEEE Transactions on Software Engineering*, *34*, 242–259.

Kampenes, V. B., Dybå, T., Hannay, J. E., & Sjøberg, D. I. K. (2009). A systematic review of quasi-experiments in software engineering. *Information and Software Technology*, *51*(1), 71–82.

Karahasanović, A., Levine, A. K., & Thomas, R. C. (2007). Comprehension strategies and difficulties in maintaining object-oriented systems: An explorative study. *Journal of Systems and Software*, *80*(9), 1541–1559.

Karahasanović, A., & Thomas, R. C. (2007). Difficulties experienced by students in maintaining object-oriented systems: An empirical study. In *Proceedings of the Australasian Computing Education Conference* (pp. 81–87).

Kline, R. B. (2005). *Principles and practice of structural equation modeling* (Second ed.). New York: Guilford Press.

Kværn, K. (2006). *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. Oslo, Norway.

Lee, S.-Y. (2007). *Structural equation modeling: A Bayesian approach.* Chichester: John Wiley.

Loehlin, J. C. (2004). *Latent variable models: An introduction to factor, path, and structural equation analysis* (Fourth ed.). Mahwah, NJ: Lawrence Erlbaum.

McCall, J. (1994). Quality factors. In J. J. Marciniak (Ed.), *Encyclopedia of software engineering* (Vol. 2, pp. 958–969). Wiley-Interscience.

Newell, A., & Rosenbloom, P. (1981). Mechanisms of skill acquisition and the law of practice. In J. R. Anderson (Ed.), *Cognitive skills and their acquisition* (pp. 1–56). Hillsdale, NJ: Erlbaum.

Prechelt, L. (1999). *The 28:1 Grant/Sackman legend is misleading, or: How large is interpersonal variation really?* (Tech. Rep. No. 18). Karlsruhe, Germany: University of Karlsruhe.

Schmidt, F. L., & Hunter, J. E. (1998). The validity and utility of selection methods in personnel psychology: Practical and theoretical implications of 85 years of research findings. *Psychological Bulletin*, *124*(2), 262–274.

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

Shanteau, J. (1992). Competence in experts: The role of task characteristics. *Organizational Behavior and Human Decision Processes*, *53*, 252–266.

Shanteau, J., Weiss, D. J., Thomas, R. P., & Pounds, J. C. (2002). Performance-based assessment of expertise: How to decide if someone is an expert or not. *European Journal of Operational Research*, *136*, 253–263.

Spolsky., J. (2007). *Smart and gets things done: Joel Spolsky's concise guide to finding the best technical talent.* Berkeley, CA: Apress.

Wiedenbeck, S., Fix, V., & Scholtz, J. (1993). Characteristics of the mental representations of novice and expert programmers: An empirical study. *International Journal*

of Man-Machine Studies, *39*, 793–812.

Paper II:

Construction and Validation of an Instrument for Measuring Programming Skill

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

Transactions on Software Engineering, Vol. 40, No. 12, pp. 1163–1184, 2014.

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.

Keywords: skill, programming, performance, instrument, measurement.

1 Introduction

Software engineering folklore states that the skill of programmers is crucial to the success of software projects (Brooks, 1987; Curtis, 1984). 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 resumes 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 (Campbell, McCloy, Oppler, & Sager, 1993). 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 (Beecham, Baddoo, Hall, Robinson, & Sharp, 2008). 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 (Soloway & Ehrlich, 1984; Wiedenbeck, 1985). Other factors, such as experience, education, and personality, also *indirectly* affect individual performance through their influence on motivation, knowledge, and skill (Schmidt, Hunter, & Outerbridge, 1986; Waldman & Spangler, 1989). 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 (Fitts & Posner, 1967), 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 general principles of instrument construction (Messick, 1994; Nunnally & Bernstein, 1994; Pedhazur & Schmelkin, 1991). 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 (Ericsson, Charness, Feltovich, & Hoffman, 2006). 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 (Bergersen & Gustafsson, 2011) and to investigate how skill moderates the benefit of software technologies and methods (Bergersen & Sjøberg, 2012). It has also been used to select programmers as research subjects in a multiple-case study (Sjøberg, Yamashita, Anda, Mockus, & Dybå, 2013). 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 paper may also guide the construction and validation of instruments for measuring other

aspects of software engineering.

The remainder of this paper 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. Figure 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* (Cohen, 1989), which in turn is *defined* by a scoring rule that is applied to the time and quality of task solutions (i.e., a "response"

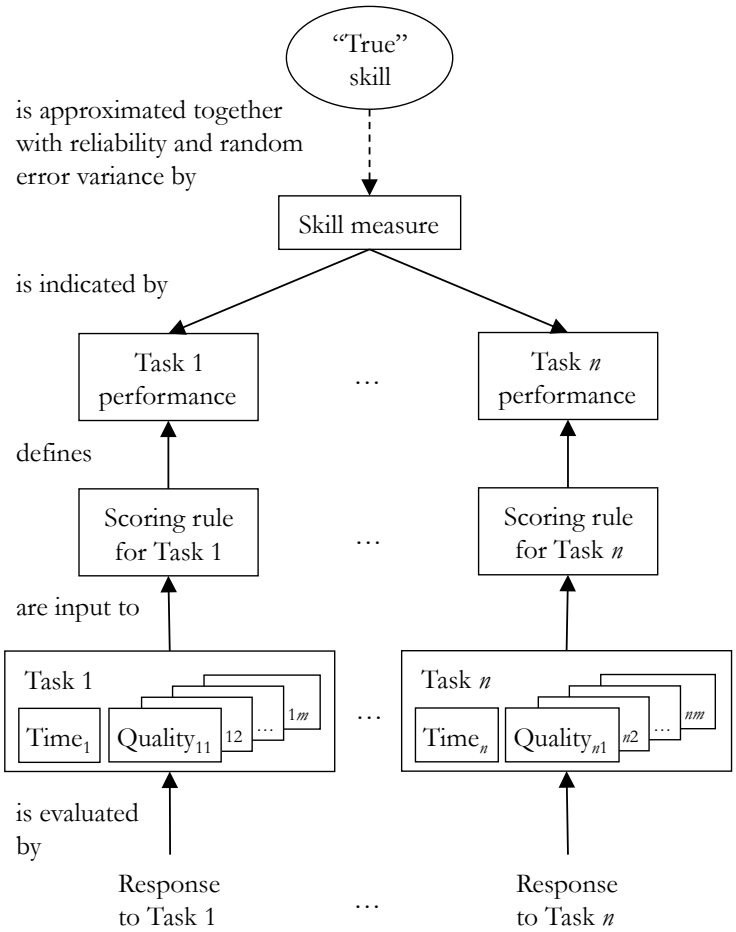


Figure 1: The relations between variables of skill, task performance, and time and quality.

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* (Edwards & Bagozzi, 2000).

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" (Carroll, 1993, 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" (Ferguson, 1956, 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 (Ericsson & Smith, 1991). 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" (Ericsson, 2003, p. 52).

Unlike some abilities, skill is a psychological variable that can be defined theoretically. Over 80 years ago, Pear (1928) recommended using the term for higher levels of performance and then only in conjunction with well-adjusted performance. According to Fitts and Posner (1967), 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" (Robillard, 1999, 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., Anderson, 1982) and *programming skills* in particular (e.g., Anderson, 1987; Anderson, Conrad, & Corbett, 1989; Shute, 1991) using Fitts and Postner's (1967) early work. Anderson (1987) 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 (Fenton, 1994; Messick, 1989). The choice of a measurement model also exposes

the assumptions that underlie one's effort to measure something (Borsboom, Mellenbergh, & van Heerden, 2003).

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" (Stevens, 1946, p. 667); see, for example, the early paper by Curtis (1980). 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*" (Michell, 1997, p. 358). More generally, Stevens' definition is not universally accepted (Pedhazur & Schmelkin, 1991) because even meaningless rules can yield measurements, according to this definition (Borsboom, 2005), also see (Fenton, 1994).

A challenge is that Stevens' definition is commonly used in software engineering, while scholars (Fenton, 1994; Fenton & Kitchenham, 1991) advocate that measurement in software engineering should adhere to the scientific principles of measurement (i.e., Krantz, Luce, Suppes, & Tversky, 1971; Kyburg, 1984). 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" (Briand, El Emam, & Morasca, 1996, 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 (Luce & Tukey, 1964).

In this work, we chose the Rasch measurement model, which resembles additive conjoint measurement, albeit from a probabilistic viewpoint (Borsboom & Scholten, 2008). Although this viewpoint is still being debated (Kyngdon, 2008), the use of probability is central to science in general (Hacking, 1990) and experimentation in particular (Shadish, Cook, & Campbell, 2002). 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 (1960) 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:

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

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 (2008) 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 (Andrich, 1978) 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. A 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 (Andrich, 2010). 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 (Bond & Fox, 2001). The Rasch model has also been used to measure programming ability in C (Wilking, Schilli, & Kowalewski, 2008), Lisp (Pirolli & Wilson, 1998), and Pascal (Syang & Dale, 1993), and to explain software engineering practices that are based on CMM (Dekleva & Drehmer, 1997).

2.4 Operationalization of Performance

When inferring skill, only performance that is under the complete control of the individual is of interest (Campbell et al., 1993). 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 (1994) 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 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 (Bergersen, Hannay, Sjøberg, Dybå, & Karahasanović, 2011):

- *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 (Bergersen et al., 2011), 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 paper 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" (Shadish et al., 2002, 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 [psychological] test is valid for measuring an attribute if (a) the attribute exists and (b) variations in the attribute causally produce variation in the measurement outcomes" (Borsboom, Mellenbergh, & van Heerden, 2004, p. 1061).

We distinguish validity from the *process* of evaluating validity, that is, validation (Borsboom et al., 2004). According to the American Psychological Association, support for or evidence against validity may be addressed according to the aspects shown in Table 1 (1999).

When multiple observations of task performance are used as indicators of skill (Figure 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 (Figure 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

Table 1: Addressing validity aspects recommended by the APA guidelines

Aspect	Description	Addressed in sections
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

Notes. APA also includes "consequences of testing", which addresses social policies of testing. This aspect is beyond the scope of this paper.

in itself invalidate a measure. However, *systematic error variance* indicates systematic patterns in the variance that are not a part of the intended measure.

According to Messick (1989), 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 (Messick, 1994). 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 (Nunnally & Bernstein, 1994), but we

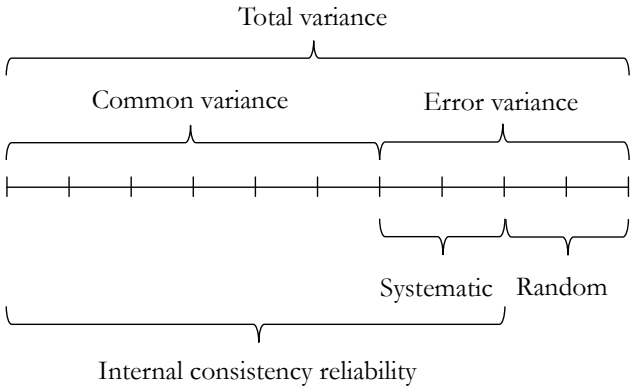


Figure 2: Variance components (adapted from Rummel, 1970).

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 Figure 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 activities of writing code from scratch, and modifying and debugging code. In particular, the last two activities also include 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.

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 the fit of task performance data as the criterion in 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.

In a study of four software projects (Anda, Sjøberg, & Mockus, 2009), 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%, $n = 1{,}490$).[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 (Shadish et al., 2002). However, there is no such thing as a pool of all programming tasks that 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 (Dybå, Sjøberg, & Cruzes, 2012). Therefore, one has to use convenience sampling.

To help achieve generalizability into an industrial programming setting, we *purposively sampled typical instances* (Shadish et al., 2002). Thus, the set of tasks were selected or constructed to capture a range of aspects of industrial programming tasks to increase realism (Sjøberg et al., 2002).

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. (Kane, Crooks, & Cohen, 1999) 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.

[1] www.aboutprogrammers.org

- *Time limit* was varied across a mix of short (∼10 minutes), medium (∼25 minutes), and long tasks (∼45 minutes).
- *Subtasks*, which require multiple submissions (i.e., testlet structure; see Section 2.4), 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 3 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 (Bergersen et al., 2011). Each scoring rule is uniquely associated with a specific task. An example of a scoring rule that we constructed is shown in Table 4. Three subtasks extend the functionality of the Library Application described in Table 3: 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 for e-mail functionality 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.

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

Table 3: 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	Constructed	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	Constructed	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	Constructed[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	Constructed	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	Constructed	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 task 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 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 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	Constructed	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	Constructed	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.

Notes. 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 4: The scoring rule for the Library Application task

	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

For all tasks, the time limits used were either based on empirical data from earlier studies (see Table 3) or results from pilot tests. Some strategies for deciding on time limits in the scoring of performance are provided in our previous work (Bergersen et al., 2011).

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

An experiment support environment (Arisholm, Sjøberg, Carelius, & Lindsjørn, 2002) 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 (Arisholm, Gallis, Dybå, & Sjøberg, 2007; Arisholm & Sjøberg, 2004). 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 (Feynman, 1998). 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 (Chatfield, 1995). Because the ways to combine time and quality variables into a single performance variable are potentially infinite (Bergersen et al., 2011), 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 (Dahl, Grotle, Benth, & Natvig, 2008) or other claims. Using generalizability theory (Shavelson & Webb, 1991), we first investigated the magnitude of different sources of variance for the experiment reported in (Arisholm & Sjøberg, 2004). This analysis confirmed that variability between persons and tasks was large, which was also reported in (DeMarco & Lister, 1999; Prechelt, 1999). 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 Figure 1). This rule is called a scoring rule for the item i:

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

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) (Campbell et al., 1993). However, within most other fields, it is difficult to obtain a suitable external criterion (Nunnally & Bernstein, 1994). For example, it is problematic to use a supervisor's rating of overall performance as *the* external criterion for the job performance of individuals (Campbell, Gasser, & Oswald, 1996). More generally, the idea of using an "ultimate criterion" has also been described as "an idea that has severely retarded personnel research" (Campbell, 1990, p. 713) and as "truly one of the most serious mistakes ever made in the theory of psychological measurement" (Borsboom et al., 2004, 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 (Campbell, 1969, p. 43).

3.8 Constructing and Adjusting Scoring Rules Using Rasch Analysis

Figure 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 3),

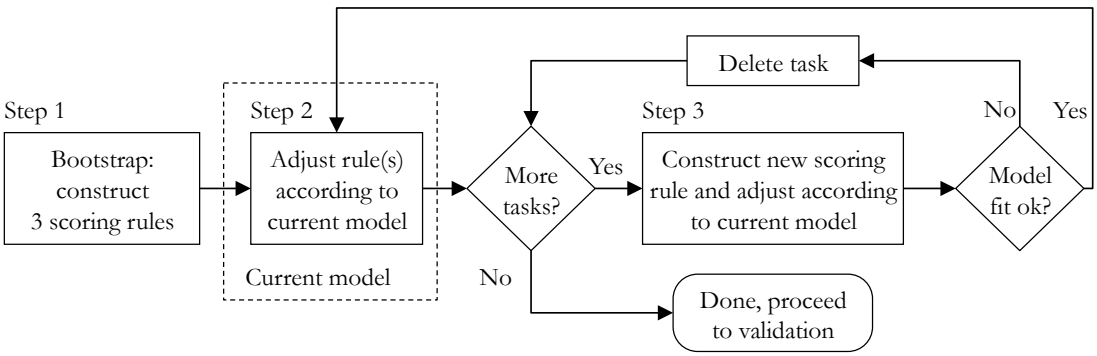


Figure 3: Constructing and adjusting scoring rules prior to instrument validation.

for which we already had extensive programming performance data available (Arisholm et al., 2007; Arisholm & Sjøberg, 2004; Karahasanović, Levine, & Thomas, 2007; Karahasanović & Thomas, 2007; Kværn, 2006). We had determined a set of scoring rules based on this data for these three tasks (Bergersen et al., 2011), 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 (Andrich, Sheridan, & Luo, 2006). (See, for example, Bond & Fox, 2001; Wilson, 2005; Wright & Masters, 1979 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 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 5.

4.1 Test of 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

Table 5: 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.

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 (Nunnally & Bernstein, 1994) was performed to investigate measurement bias. Task 17 (see Table 3) 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 (Loehlin, 2004). 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 Figure 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 make 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 Figure 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 (2002) proposed a test that uses principal component analysis (PCA) on standardized 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 individual's 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 3) contributed negatively to instrument unidimensionality. By removing these three tasks, Smith's test indicated acceptable unidimensionality ($p = 0.046$, low 95% CI = 0.00).

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 6, 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 that relates to tasks and their residuals.

Table 6: Task performance scores for the subjects in the final instrument

Subject ID	Dataset	Skill	Task ID											
			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

Notes. 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.

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 (Smith, 1988). 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. Figure 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 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.

[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.

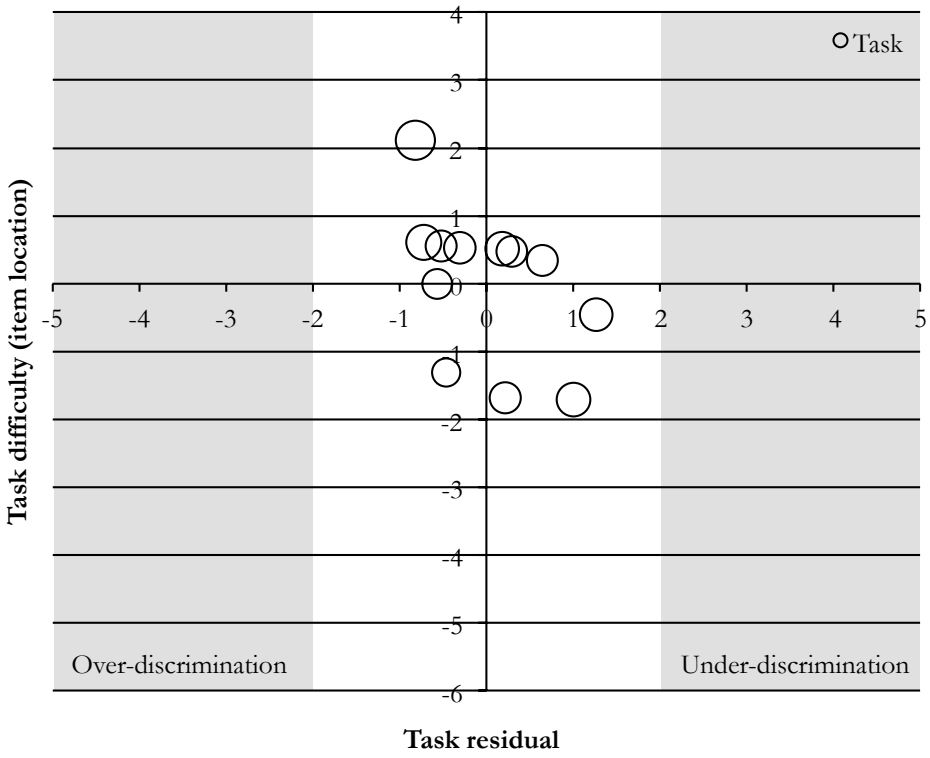


Figure 4: Task fit to the model.

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 (Bond & Fox, 2001).

Figure 5 shows that the individual's response pattern in general fits the model; five 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 (Embretson, 1996). Figure 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 MacKay, 1982, generally).

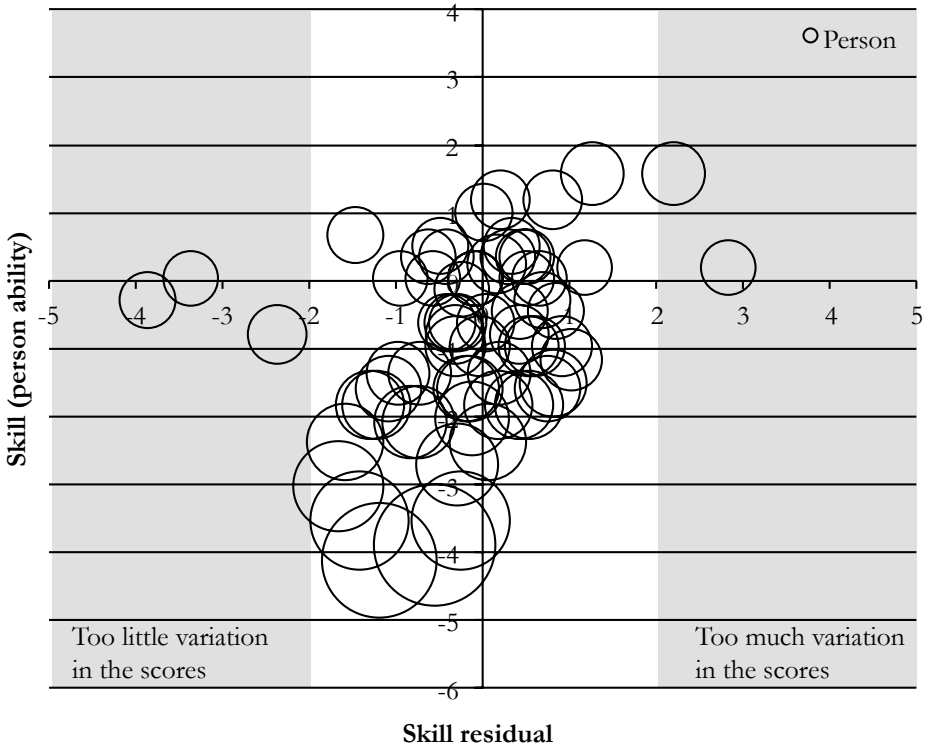


Figure 5: Person fit to the model.

Another concern is whether the subjects increased their performance throughout the two days they solved the tasks. Figure 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 (Shadish et al., 2002) or other improvements in performance (Ericsson, 2003) were present, negative skill-task residuals 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 (Chinn, 2000) for converting logits into a standardized effect size combined with software engineering

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

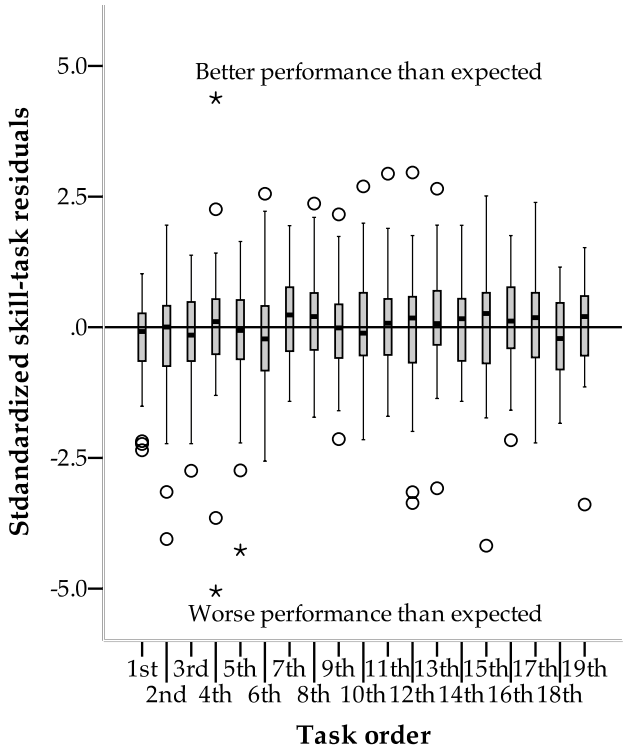


Figure 6: Skill-task residuals depending on task order.

conventions for "small", "medium", and "large" effect sizes (Kampenes, Dybå, Hannay, & Sjøberg, 2007), 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) (Streiner, 1995). 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 (Schmitt, 1996).

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; Syang & Dale, 1993). 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 (Campbell & Fiske, 1959). Conversely, divergent validity is that variables that, in theory, are not meant to assess the same construct, should not correlate in practice.

Table 7 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 described throughout this section. The four experience variables and lines of code use ratio scale. *JavaSkill* uses interval scale. The remaining variables of the table are all ordinal scale.

Table 8 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 $7,000 and administered to 60 of the 65 developers one to four months after they solved the programming tasks. From this test, we sampled 30 multiple-choice questions that covered the same domain as the tasks used in the skill instrument. Table 8 shows that *javaKnowledge* was the variable with the highest correlation with *javaSkill*. Because knowledge and skill should be close to unity for developers currently learning how to program, but should diverge as skill evolves through the second and third phase of skill acquisition (Section 2.1), we split the individuals into two groups. For those with *javaSkill* below the mean (see Table 7), *javaKnowledge* and *javaSkill* were highly correlated ($\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 Figure 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

[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. (Basili, Zelkowitz, Sjøberg, Johnson, & Cowling, 2007) will be required.

Table 7: 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	–3.3	3.6	Sum of three standardized tests[a]
Test based (subjective)	Technical skills	65	11.5	1.9	5	15	Sum of 3 Likert scale questions[b]
	Managerial skills	65	26.9	4.2	17	34	Sum of 8 Likert scale questions[b]
	People skills	65	2.2	2.9	16	29	Sum of 6 Likert scale 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]	136k	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] One observation was removed as an extreme outlier according to Grubbs' test. [f] Maximum = 10 (minimum = 1 is implied). [g] Strongly disagree = 1; disagree = 2; neither = 3; agree = 4; strongly agree = 5.

information. In general, working memory capacity is central to theories of skill acquisition (Anderson, 1982; Chase & Ericsson, 1982). In particular, working memory has been found to predict technical skill acquisition (Kyllonen & Stephens, 1990) and programming skill acquisition to a large extent (Shute, 1991). In our study, three tests of working memory were acquired from Unsworth et al. (Unsworth, Heitz, Schrock, & Engle, 2005). 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 (Unsworth et al., 2005). 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 8 shows 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. (Unsworth, Redick, Heitz, Broadway, & Engle, 2009).

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

Table 8: Cross correlations between Java skill and external variables

Variable	(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 experience (4)	**0.37****	0.27*	0.54**	—	65	65	27	65	64	65	65	65	65	65	65	65
Programming 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
Developer 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
Programming 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	—

Notes. 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.

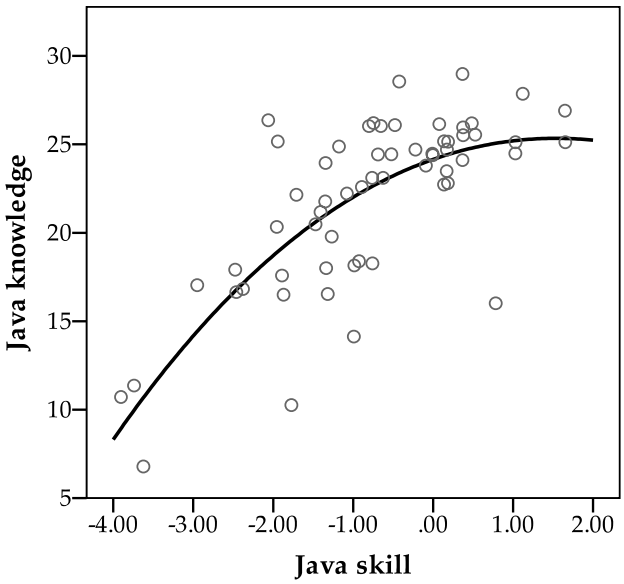


Figure 7: The relation between Java skill and knowledge. Note: The data are jittered to show overlapping observations.

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 8 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 (Bergersen et al., 2011). 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 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 (Bergersen et al., 2011).

Among the self-reported variables in Table 8, Java expertise (*javaExpertise*), which uses a 5-point Likert scale from "novice" = 1 to "expert" = 5 (see Table 7), had the highest, significant correlation of 0.46 with *javaSkill*. This is similar to the correlation reported in (Ackerman & Wolman, 2007) 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 (Nunnally & Bernstein, 1994).

We also administered a questionnaire, published in (Chilton & Hardgrave, 2004), 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 8 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 8 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 (Kanfer & Ackerman, 1989) 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.

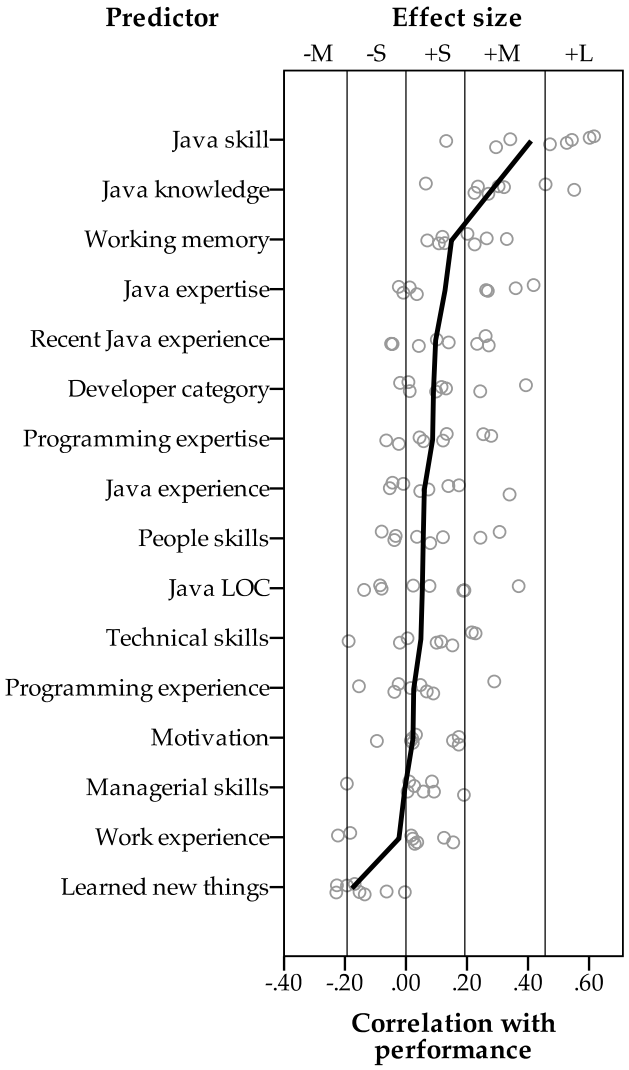
Finally, the subjects were asked about the extent to which they learned new things while solving the 19 tasks (*learnedNewThings*). Table 8 shows a statistically significant negative correlation between *learnedNewThings* and *javaSkill*. This demonstrates divergent validity, because people with high 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 (Nunnally & Bernstein, 1994). 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 3). 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 external 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.

Figure 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 negated and analyzed using Spearman's ρ. The vertical lines divide between small (S), medium (M), and large (L) correlations according



Note: The data are jittered to show overlapping observations.

Figure 8: Java skill and alternative predictors of task performance. Note: The data are jittered to show overlapping observations.

to the effect size guidelines stated in (Kampenes, Dybå, Hannay, & Sjøberg, 2009). The trend line shows the mean of the correlations for each predictor and confirms that the instrument (i.e., *javaSkill*) was the best predictor, ahead of *javaKnowledge*.

Figure 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 (Jeffery & Lawrence, 1979). 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 of experience" (Jeffery & Lawrence, 1979, 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 a monotonically increasing but deaccelerating relationship is expected from the log-log law of practice (Newell & Rosenbloom, 1981), as well as research on expertise (Ericsson, Krampe, & Tesch-Römer, 1993).

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 (Sjøberg et al., 2005). Compared with (Syang & Dale, 1993) and (Wilking et al., 2008), 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 (Syang & Dale, 1993) and (Wilking et al., 2008) 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 (Dybå et al., 2012).

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 (Syang & Dale, 1993). The open response format (e.g., used in Wilking et al., 2008) 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 (Freedman, 2013). This indicates that programming skill accounts for the major proportion of the large differences observed in programming performance (i.e., the "common variance" in Figure 2) reported elsewhere (Curtis, 1980; DeMarco & Lister, 1999; Grant & Sackman, 1967; Prechelt, 1999). 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 (Syang & Dale, 1993), 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 largely 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 (Bergersen & Gustafsson, 2011), we found that five of the variables in Table 7 display a pattern in the correlations that is consistent with Cattell's investment theory, see (Cattell, 1971/1987). This psychological theory describes how the effect of intelligence (in our context, working memory) and experience on skill is mediated by knowledge. Previous work by Anderson (1987) 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 (1999) 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* (Locke, 1986), in which the concept of transfer (Ferguson, 1956) 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 reported that "[t]here is transfer from other programming experience to the extent that this programming experience involves the same productions" (Anderson et al., 1989, 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 (Popper, 1968).

6.2 Contributions to Research

Theory-driven investigations are rare in empirical software engineering (Hannay, Sjøberg, & Dybå, 2007), even though theory is often required to interpret and test results (Popper, 1968; Shadish et al., 2002). In (Sjøberg, Dybå, Anda, & Hannay, 2008), 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 (Hands, Sheridan, & Larkin, 1999) to account for the time-quality problems when scoring performance on programming tasks (Bergersen et al., 2011).

Scoring rules are rarely justified or evaluated. In (Dybå, 2000), 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 paper 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 (Sheil, 1981) by analyzing residual variance. Moreover, we demonstrated that by requiring that residual variance be uncorrelated, the testability of the proposed model is enhanced. In (Dybå, 2000), 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 (Dybå, Moe, & Arisholm, 2005), 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 (Campbell & Fiske, 1959), 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 (Bergersen & Sjøberg, 2012). 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 (Sjøberg et al., 2002). 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 (Sjøberg et al., 2013).

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 (Sjøberg et al., 2005) 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 (Bergersen & Sjøberg, 2012). 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 (Dybå, Kampenes, & Sjøberg, 2006; Shadish et al., 2002).

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" (Kampenes et al., 2009, 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 (Arisholm et al., 2007), 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 (Campbell et al., 1993), 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" (Campbell et al., 1993, p. 46). In a meta-analysis with over 32,000 employees (Schmidt & Hunter, 1998), 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 (Braun, Bennett, Frye, & Soloway, 1990). However, they are more costly to develop and score and more time-consuming to administer (Kane et al., 1999). 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 (Schmidt & Hunter, 1998). 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 (Feigenspan, Kästner, Liebig, Apel, & Hanenberg, 2012).

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 3 to be used as a one-hour version of the instrument. Although the measurement precision of the instrument is greatly reduced by having only two tasks to measure skill instead of 12, the validity of the instrument should be unaffected because all the tasks still measure "the same", that is, programming skill. 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. Figure 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 (Linacre, 1994), even though authoritative work on Rasch modeling has previously been conducted on a sample size similar to ours (see Wright & Masters, 1979). 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 Figures 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" (Schooler, Ohlsson, & Brooks, 1993); 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 present (12 tasks) 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 (APA, 1999) from industry programming tasks with our instrument tasks. However, we have previously found that such methods are intrusive (Karahasanović, Hinkel, Sjøberg, & Thomas, 2009) 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 (Curtis, 1980; DeMarco & Lister, 1999; Grant & Sackman, 1967; Prechelt, 1999). The standard deviation of skill in our sample of developers was 1.3 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 four out of five programming tasks on average.[5] The instrument is used at present in an

[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.

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 only 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 (Ericsson et al., 1993).

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 based on an individual's performance on a set of programming 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.

Acknowledgment

This work was supported by Simula Research Laboratory and the Research Council of Norway through grants no. 182567, 193236/I40, and 217594/O30. The authors thank the anonymous referees for valuable comments, Erik Arisholm and Jo Hannay for useful discussions and feedback on an earlier version of this paper, 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

Ackerman, P. L., & Wolman, S. D. (2007). Determinants and validity of self-estimates of ability and self-concept measures. *Journal of Experimental Psychology: Applied*, *13*(2), 57–78.

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. (1999). *Standards for educational and psychological testing*. Washington, DC: American Educational Research Association.

Anda, B. C. D., Sjøberg, D. I. K., & Mockus, A. (2009). Variability and reproducibility in software engineering: A study of four companies that developed the same system. *IEEE Transactions on Software Engineering*, *35*(3), 407–429.

Anderson, J. R. (1982). Acquisition of cognitive skill. *Psychological Review*, *89*(4), 369–406.

Anderson, J. R. (1987). Skill acquisition: Compilation of weak-method problem solutions. *Psychological Review*, *94*(2), 192–210.

Anderson, J. R., Conrad, F. G., & Corbett, A. T. (1989). Skill acquisition and the LISP Tutor. *Cognitive Science*, *13*(4), 467–505.

Andrich, D. (1978). A rating formulation for ordered response categories. *Psychometrika*, *43*(4), 561–573.

Andrich, D. (2010). Understanding the response structure and process in the polytomous Rasch model. In M. L. Nering & R. Ostini (Eds.), *Handbook of polytomous item response theory models: Developments and applications* (p. 123-152). New York: Routledge.

Andrich, D., Sheridan, B., & Luo, G. (2006). Rumm2020 [Computer software manual]. Perth.

Arisholm, E., Gallis, H., Dybå, T., & Sjøberg, D. I. K. (2007). Evaluating pair programming with respect to system complexity and programmer expertise. *IEEE Transactions on Software Engineering*, *33*(2), 65–86.

Arisholm, E., & Sjøberg, D. I. K. (2004). Evaluating the effect of a delegated versus centralized control style on the maintainability of object-oriented software. *IEEE Transactions on Software Engineering*, *30*(8), 521–534.

Arisholm, E., Sjøberg, D. I. K., Carelius, G. J., & Lindsjørn, Y. (2002). A web-based support environment for software engineering experiments. *Nordic Journal of Computing*, *9*(3), 231–247.

Basili, V. R., Zelkowitz, M. V., Sjøberg, D. I. K., Johnson, P., & Cowling, A. J. (2007). Protocols in the use of empirical software engineering artifacts. *Empirical Software Engineering*, *12*(1), 107–119.

Beecham, S., Baddoo, N., Hall, T., Robinson, H., & Sharp, H. (2008). Motivation in software engineering: A systematic literature review. *Information and Software Technology*, *50*(9–10), 860–878.

Benander, A. C., Benander, B. A., & Sang, J. (2000). An empirical analysis of debugging performance—differences between iterative and recursive constructs. *Journal of Systems and Software*, *54*(1), 17–28.

Bergersen, G. R., & Gustafsson, J.-E. (2011). Programming skill, knowledge and working memory capacity among professional software developers from an investment theory perspective. *Journal of Individual Differences*, *32*(4), 201–209.

Bergersen, G. R., Hannay, J. E., Sjøberg, D. I. K., Dybå, T., & Karahasanović, A. (2011). Inferring skill from tests of programming performance: Combining time and quality. In *Proceedings of the 5th International Symposium on Empirical Software Engineering and Measurement* (pp. 305–314).

Bergersen, G. R., & Sjøberg, D. I. K. (2012). Evaluating methods and technologies in software engineering with respect to developer's skill level. In *Proceedings of the 16th International Conference on Evaluation & Assessment in Software Engineering* (pp. 101–110). IET.

Bloch, J. (2001). *Effective Java programming language guide*. Mountain View, CA: Sun Microsystems.

Bond, T. G., & Fox, C. M. (2001). *Applying the Rasch model: Fundamental measurement in the human sciences*. Mahwah, NJ: Erlbaum.

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

Borsboom, D., Mellenbergh, G. J., & van Heerden, J. (2003). The theoretical status of latent variables. *Psychological Review*, *110*(2), 203–219.

Borsboom, D., Mellenbergh, G. J., & van Heerden, J. (2004). The concept of validity. *Psychological Review*, *111*(4), 1061–1071.

Borsboom, D., & Scholten, A. Z. (2008). The Rasch model and conjoint measurement theory from the perspective of psychometrics. *Theory & Psychology*, *18*(1), 111-117.

Braun, H. I., Bennett, R. E., Frye, D., & Soloway, E. (1990). Scoring constructed responses using expert systems. *Journal of Educational Measurement*, *27*(2), 93–108.

Briand, L., El Emam, K., & Morasca, S. (1996). On the application of measurement theory in software engineering. *Empirical Software Engineering*, *1*(1), 61–88.

Brooks, F. P., Jr. (1987). No silver bullet: Essence and accidents of software engineering. *IEEE Computer*, *20*(4), 10–19.

Campbell, D. T. (1969). A phenomenology of the other one: Corrigible, hypothetical, and critial. In T. Mischel (Ed.), *Human action: Conceptual and empirical issues* (pp. 41–69). New York: Academic Press.

Campbell, D. T., & Fiske, D. W. (1959). Convergent and discriminant validity by the multitrait-multimethod matrix. *Psychological Bulletin*, *56*(2), 81–105.

Campbell, J. P. (1990). Modeling the performance prediction problem in industrial and organizational psychology. In M. D. Dunnette & L. M. Hough (Eds.), *Handbook of industrial and organizational psychology* (Second ed., Vol. 1, pp. 687–732). Palo Alto, CA: Consulting Psychologists Press.

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

Campbell, J. P., McCloy, R. A., Oppler, S. H., & Sager, C. E. (1993). A theory of performance. In N. Schmitt & W. C. Borman (Eds.), *Personnel selection in organizations* (p. 35-70). San Francisco, CA: Jossey-Bass.

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

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

Chase, W. G., & Ericsson, K. A. (1982). Skill and working memory. *The Psychology of Learning and Motivation*, *16*, 1–58.

Chatfield, C. (1995). Model uncertainty, data mining and statistical inference. *Journal of the Royal Statistical Society, Series A*, *158*(3), 419–466.

Chilton, M. A., & Hardgrave, B. C. (2004). Assessing information technology personnel: Towards a behavioral rating scale. *DATA BASE for Advances in Information Systems*, *35*(3), 88–104.

Chinn, S. (2000). A simple method for converting an odds ratio to effect size for use in meta-analysis. *Statistics in Medicine*, *19*(22), 3127–3131.

Cockburn, A. (1998, May/June). The Coffee Machine design problem: Part 1 & 2. *C/C++ Users Journal*.

Cohen, B. P. (1989). *Developing sociological knowledge: Theory and method* (Second ed.). Chicago: Nelson-Hall.

Curtis, B. (1980). Measurement and experimentation in software engineering. *Proceedings of the IEEE*, *68*(9), 1144–1157.

Curtis, B. (1984). Fifteen years of psychology in software engineering: Individual differences and cognitive science. In *Proceedings of the 7th International Conference on Software Engineering* (pp. 97–106).

Dahl, F. A., Grotle, M., Benth, J. Š., & Natvig, B. (2008). Data splitting as a countermeasure against hypothesis fishing: With a case study of predictors for low back pain. *European Journal of Epidemiology*, *23*(4), 237–242.

Dekleva, S., & Drehmer, D. (1997). Measuring software engineering evolution: A Rasch calibration. *Information Systems Research*, *8*(1), 95–104.

DeMarco, T., & Lister, T. (1999). *Peopleware: Productive projects and teams* (Second ed.). New York: Dorset House Publishing Company.

Dybå, T. (2000). An instrument for measuring the key factors of success in software process improvement. *Empirical Software Engineering*, *5*(4), 357–390.

Dybå, T., Kampenes, V. B., & Sjøberg, D. I. K. (2006). A systematic review of statistical power in software engineering experiments. *Information and Software Technology*, *48*(8), 745–755.

Dybå, T., Moe, N. B., & Arisholm, E. (2005). Measuring software methodology usage: Challenges of conceptualization and operationalization. In *Proceedings of the International Symposium on Empirical Software Engineering* (pp. 447–457).

Dybå, T., Sjøberg, D. I. K., & Cruzes, D. S. (2012). What works for whom, where,

when, and why? On the role of context in empirical software engineering. In *Proceedings of the 6th International Symposium on Empirical Software Engineering and Measurement* (pp. 19–28).

Edwards, J. R., & Bagozzi, R. P. (2000). On the nature and direction of relationships between constructs and measures. *Psychological Methods*, *5*(2), 155-174.

Embretson, S. E. (1996). The new rules of measurement. *Psychological Assessment*, *8*(4), 341-349.

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

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

Ericsson, K. A., Krampe, R. T., & Tesch-Römer, C. (1993). The role of deliberate practice in the acquisition of expert performance. *Psychological Review*, *100*(3), 363–406.

Ericsson, K. A., & Smith, J. (1991). Prospects and limits of the empirical study of expertise: An introduction. In K. A. Ericsson & J. Smith (Eds.), *Towards and general theory of expertise* (pp. 1–38). New York: Cambridge University Press.

Eriksson, H.-E., & Penker, M. (1997). *UML toolkit*. New York: John Wiley & Sons.

Feigenspan, J., Kästner, C., Liebig, J., Apel, S., & Hanenberg, S. (2012). Measuring programming experience. In *IEEE 20th International Conference on Program Comprehension* (pp. 73–82).

Fenton, N. (1994). Software measurement: A necessary scientific basis. *IEEE Transactions on Software Engineering*, *20*(3), 199–206.

Fenton, N., & Kitchenham, B. (1991). Validating software measures. *Journal of Software Testing, Verification, and Reliability*, *1*(2), 27–42.

Ferguson, G. A. (1956). On transfer and the abilities of man. *Canadian Journal of Psychology*, *10*(3), 121–131.

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

Fitts, P. M., & Posner, M. I. (1967). *Human performance*. Belmont, CA: Brooks/Cole.

Fleury, A. E. (2000). Programming in Java: Student-constructed rules. *ACM SIGCSE Bulletin*, *32*(1), 197–201.

Floyd, R. W. (1979). The paradigms of programming. *Communications of the ACM*, *22*(8), 455-460.

Freedman, R. (2013). Relationships between categories of test items in a C++ CS1 course. *Journal of Computing Sciences in Colleges*, *29*(2), 26–32.

Grant, E. E., & Sackman, H. (1967). An exploratory investigation of programmer performance under on-line and off-line conditions. *IEEE Transactions on Human Factors in Electronics*, *HFE-8*(1), 33–48.

Hacking, I. (1990). *The taming of chance*. Cambridge: Cambridge University Press.

Hands, B., Sheridan, B., & Larkin, D. (1999). Creating performance categories from continuous motor skill data using a Rasch measurement model. *Journal of Outcome*

Measurement, *3*(3), 216–232.

Hannay, J. E., Sjøberg, D. I. K., & Dybå, T. (2007). A systematic review of theory use in software engineering experiments. *IEEE Transactions on Software Engineering*, *33*(2), 87–107.

Humphry, S. M., & Andrich, D. (2008). Understanding the unit in the Rasch model. *Journal of Applied Measurement*, *9*(3), 249–264.

Jeffery, D. R., & Lawrence, M. J. (1979). An inter-organisational comparison of programming productivity. In *Proceedings of the 4th International Conference on Software Engineering* (p. 369-377).

Kampenes, V. B., Dybå, T., Hannay, J. E., & Sjøberg, D. I. K. (2007). A systematic review of effect size in software engineering experiments. *Information and Software Technology*, *49*(11), 1073–1086.

Kampenes, V. B., Dybå, T., Hannay, J. E., & Sjøberg, D. I. K. (2009). A systematic review of quasi-experiments in software engineering. *Information and Software Technology*, *51*(1), 71–82.

Kane, M., Crooks, T., & Cohen, A. (1999). Validating measures of performance. *Educational Measurement: Issues and Practice*, *18*(2), 5–17.

Kanfer, R., & Ackerman, P. L. (1989). Motivation and cognitive abilities: An integrative/aptitude-treatment interaction approach to skill acquisition. *Journal of Applied Psychology*, *74*(4), 657–690.

Karahasanović, A., Hinkel, U. N., Sjøberg, D. I. K., & Thomas, R. (2009). Comparing of feedback-collection and think-aloud methods in program comprehension studies. *Behaviour & Information Technology*, *28*(2), 139–164.

Karahasanović, A., Levine, A. K., & Thomas, R. C. (2007). Comprehension strategies and difficulties in maintaining object-oriented systems: An explorative study. *Journal of Systems and Software*, *80*(9), 1541–1559.

Karahasanović, A., & Thomas, R. C. (2007). Difficulties experienced by students in maintaining object-oriented systems: An empirical study. In *Proceedings of the Australasian Computing Education Conference* (pp. 81–87).

Krantz, D. H., Luce, R. D., Suppes, P., & Tversky, A. (1971). *Foundations of measurement* (Vol. 1). New York: Academic Press.

Kværn, K. (2006). *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. Oslo, Norway.

Kyburg, J., H. E. (1984). *Theory and measurement.* Cambridge: Cambridge University Press.

Kyllonen, P. C., & Stephens, D. L. (1990). Cognitive abilities as determinants of success in acquiring logic skill. *Learning and Individual Differences*, *2*(2), 129–160.

Kyngdon, A. (2008). The Rasch model from the perspective of the representational theory of measurement. *Theory & Psychology*, *18*(1), 89–109.

Linacre, J. M. (1994). Sample size and item calibration stability. *Rasch Measurement Transactions*, *7*(4), 328.

Locke, E. A. (1986). Generalizing from laboratory to field: Ecological validity or ab-

straction of essential elements? In E. A. Locke (Ed.), *Generalizing from laboratory to field setting: Research findings from industrial-organizational psychology, organizational behavior, and human resource management* (pp. 3–42). Lexington, MA: Lexington Books.

Loehlin, J. C. (2004). *Latent variable models: An introduction to factor, path, and structural equation analysis* (Fourth ed.). Mahwah, NJ: Lawrence Erlbaum.

Luce, R. D., & Tukey, J. W. (1964). Simultaneous conjoint measurement: A new type of fundamental measurement. *Journal of Mathematical Psychology*, *1*(1), 1–27.

MacKay, D. G. (1982). The problems of flexibility, fluency, and speed-accuracy trade-off in skilled behavior. *Psychological Review*, *89*(5), 483–506.

McCall, J. (1994). Quality factors. In J. J. Marciniak (Ed.), *Encyclopedia of software engineering* (Vol. 2, pp. 958–969). Wiley-Interscience.

McCracken, M., Almstrum, V., Diaz, D., Guzdial, M., Hagan, D., Kolikant, Y. B.-D., ... Wilusz, T. (2001). A multi-national, multi-institutional study of assessment of programming skills of first-year CS students. *ACM SIGCSE Bulletin*, *33*(4), 125–140.

Messick, S. (1989). Validity. In R. L. Linn (Ed.), *Educational measurement* (Third ed., pp. 12–103). New York: American Council on Education/Macmillan.

Messick, S. (1994). The interplay of evidence and consequences in the validation of performance assessments. *Educational Researcher*, *23*(2), 13–23.

Michell, J. (1997). Quantitative science and the definition of measurement in psychology. *British Journal of Psychology*, *88*(3), 355–383.

Newell, A., & Rosenbloom, P. (1981). Mechanisms of skill acquisition and the law of practice. In J. R. Anderson (Ed.), *Cognitive skills and their acquisition* (pp. 1–56). Hillsdale, NJ: Erlbaum.

Nunnally, J. C., & Bernstein, I. H. (1994). *Psychometric theory* (Third ed.). New York: McGraw-Hill.

Pear, T. H. (1928). The nature of skill. *Nature*, *122*(3077), 611-614.

Pedhazur, E. J., & Schmelkin, L. P. (1991). *Measurement, design, and analysis: An integrated approach.* Hillsdale, NJ: Lawrence Erlbaum.

Pirolli, P., & Wilson, M. (1998). A theory of the measurement of knowledge content, access, and learning. *Psychological Review*, *105*(1), 58–82.

Popper, K. (1968). *Conjectures and refutations.* New York: Harper & Row.

Prechelt, L. (1999). *The 28:1 Grant/Sackman legend is misleading, or: How large is interpersonal variation really?* (Tech. Rep. No. 18). Karlsruhe, Germany: University of Karlsruhe.

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

Robillard, P. N. (1999). The role of knowledge in software development. *Communications of the ACM*, *42*(1), 87–92.

Rummel, R. J. (1970). *Applied factor analysis.* Evanston, IL: Northwestern University Press.

Schmidt, F. L., & Hunter, J. E. (1998). The validity and utility of selection methods in personnel psychology: Practical and theoretical implications of 85 years of research

findings. *Psychological Bulletin*, *124*(2), 262–274.

Schmidt, F. L., Hunter, J. E., & Outerbridge, A. N. (1986). Impact of job experience and ability on job knowledge, work sample performance, and supervisory ratings of job performance. *Journal of Applied Psychology*, *71*(3), 432–439.

Schmitt, N. (1996). Uses and abuses of coefficient alpha. *Psychological Assessment*, *8*(4), 350–353.

Schooler, J. W., Ohlsson, S., & Brooks, K. (1993). Thoughts beyond words: When language overshadows insight. *Journal of Experimental Psychology: General*, *122*(2), 166–183.

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

Shavelson, R. J., & Webb, N. M. (1991). *Generalizability theory: A primer.* Thousand Oaks, CA: Sage Publications.

Sheil, B. A. (1981). The psychological study of programming. *ACM Computing Surveys*, *13*(1), 101–120.

Shute, V. J. (1991). Who is likely to acquire programming skills? *Journal of Educational Computing Research*, *7*(1), 1–24.

Sjøberg, D. I. K., Anda, B., Arisholm, E., Dybå, T., Jørgensen, M., Karahasanović, A., ... Vokáč, M. (2002). Conducting realistic experiments in software engineering. In *Proceedings of the International Symposium Empirical Software Engineering* (pp. 17–26).

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

Sjøberg, D. I. K., Hannay, J. E., Hansen, O., Kampenes, V. B., Karahasanovic, A., Liborg, N.-K., & Rekdal, A. C. (2005). A survey of controlled experiments in software engineering. *IEEE Transactions on Software Engineering*, *31*(9), 733–753.

Sjøberg, D. I. K., Yamashita, A., Anda, B., Mockus, A., & Dybå, T. (2013). Quantifying the effect of code smells on maintenance effort. *IEEE Transactions on Software Engineering*, *39*(8), 1144–1156.

Skiena, S. S., & Revilla, M. A. (2003). *Programming challenges: The programming contest training manual.* New York: Springer.

Smith, E. V., Jr. (2002). Detecting and evaluating the impact of multidimensionality using item fit statistics and principal component analysis of residuals. *Journal of Applied Measurement*, *3*(2), 205–231.

Smith, R. M. (1988). The distributional properties of Rasch standardized residuals. *Educational and Psychological Measurement*, *48*(3), 657–667.

Soloway, E., & Ehrlich, K. (1984). Empirical studies of programming knowledge. *IEEE Transactions on Software Engineering*, *SE-10*(5), 595–609.

Stevens, S. S. (1946). On the theory of scales of measurement. *Science*, *103*(2684), 677–680.

Streiner, D. L. (1995). *Health measurement scales.* Oxford: Oxford University Press.

Syang, A., & Dale, N. B. (1993). Computerized adaptive testing in computer science: Assessing student programming abilities. *ACM SIGCSE Bulletin*, *25*(1), 53–56.

Unsworth, N., Heitz, R. P., Schrock, J. C., & Engle, R. W. (2005). An automated version of the operation span task. *Behavior Research Methods*, *3*(37), 498–505.

Unsworth, N., Redick, T., Heitz, R. P., Broadway, J. M., & Engle, R. W. (2009). Complex working memory span tasks and higher-order cognition: A latent-variable analysis of the relationship between processing and storage. *Memory*, *17*(6), 635–654.

Vokáč, M., Tichy, W., Sjøberg, D. I. K., Arisholm, E., & Aldrin, M. (2004). A controlled experiment comparing the maintainability of programs designed with and without design patterns—a replication in a real programming environment. *Empirical Software Engineering*, *9*(3), 149–195.

Waldman, D. A., & Spangler, W. D. (1989). Putting together the pieces: A closer look at the determinants of job performance. *Human Performance*, *2*(1), 29–59.

Wiedenbeck, S. (1985). Novice/expert differences in programming skills. *International Journal of Man-Machine Studies*, *23*(4), 383–390.

Wilking, D., Schilli, D., & Kowalewski, S. (2008). Measuring the human factor with the Rasch model. In B. Meyer, J. R. Nawrocki, & B. Walter (Eds.), *Balancing agility and formalism in software engineering* (Vol. 5082, pp. 157–168). Berlin: Springer.

Wilson, M. (2005). *Constructing measures: An item response modeling approach*. Mahwah, NJ: Lawrence Erlbaum Associates.

Wright, B. D., & Masters, G. N. (1979). *Rating scale analysis*. Chicago: Mesa Press.

Paper III:

Programming Skill, Knowledge, and Working Memory among Professional Developers from an Investment Theory Perspective

Gunnar R. Bergersen and Jan-Eric Gustafsson

Journal of Individual Differences, Vol. 32, No. 4, pp. 201–209, 2011.

Abstract

This study investigates the role of working memory and experience in the development of programming knowledge and programming skill. An instrument for assessing programming skill—where skill is inferred from programming performance—was administered along with tests of working memory and programming knowledge. We recruited 65 professional software developers from nine companies in eight European countries to participate in a two-day study. Results indicate that the effect of working memory and experience on programming skill is mediated through programming knowledge. Programming knowledge was further found to explain individual differences in programming skill to a large extent. The overall findings support Cattell's investment theory. Further, we discuss how this study, which currently serves a pilot function, can be extended in future studies. Although low statistical power is a concern for some of the results reported, this work contributes to research on individual differences in high-realism work settings with professionals as subjects.

Keywords: programming, investment theory, knowledge, working memory.

1 Introduction

Software production is a highly competitive and globalized industry focused on delivering high-quality software at low cost. One important way to stay competitive is to recruit and retain highly productive software developers. Although companies often use different methods for assessing competence when recruiting developers, tests of cognitive abilities are frequently utilized.

Generally, cognitive abilities can be organized into a hierarchical structure (Carroll, 1993; Gustafsson, 2002). At the apex, general mental ability (g) exerts influence over all lower factors. At the next stratum there are a handful of broad abilities such as crystallized g (Gc, acquired knowledge) and fluid g (Gf, novel and abstract problem solving), while the lowest stratum specifies a large number of narrow abilities.

According to Cattell's investment theory (1971/1987), Gf is involved in all new learning. This, in turn, implies that Gf is to some extent related to individual differences in all domains of knowledge and skills, because they have all been new to the learners. This ubiquitous presence of Gf variance is what makes it closely related to g. Gustafsson (1984) demonstrated that Gf and g are perfectly correlated, and Valentin Kvist and Gustafsson (2008) showed that this perfect relation holds true only for homogeneous populations in which the individuals have had reasonably similar opportunities to acquire the knowledge and skills tested. Thus, Gf (and g) has a wider breadth of influence than other factors of intelligence, but it does not necessarily exert a stronger influence on performance on any single task (Coan, 1964; Gustafsson, 2002; Humphreys, 1962; Valentin Kvist & Gustafsson, 2008).

Based on the investment theory, as well as on later extensions to this theory (see Ackerman, 2000), we expect the influence of Gf and experience on skill, as well as job performance, to be mediated through knowledge. That a mediating relationship exists is supported by theories of skilled behavior and skill acquisition (Anderson, Conrad, & Corbett, 1989; Neves & Anderson, 1981), in which knowledge is a key component. For example, Anderson states that knowledge initially "comes in declarative form and is used by weak methods to generate solutions [which] ... form new productions ... [and a] key step is the knowledge compilation process, which produces the domain specific skill" (Anderson, 1987, p. 197). Further, knowledge is also a central component of adult intelligence (Ackerman, 2000) and can therefore be an important factor in the acquisition of new knowledge and skills (Ackerman, 2007).

Constructs close to the apex of the hierarchical model can be described as having high referent generality, and constructs that are highly specific to a limited situation have low referent generality (see, e.g., Coan, 1964). For example, programming skill in a single programming language can be regarded as a narrow construct with low referent generality. It is well-suited for predicting the outcomes of an individual for a specific programming language, but there may be limited transfer of knowledge and skills to, for example, other kinds of programming languages. Therefore, when assessing constructs with low referent generality, it is important also to assess constructs with high referent generality (Gustafsson, 2002).

It is also possible to conceptualize criteria using a hierarchical model. Using Brunswik

symmetry, Wittmann and Süß (1999) demonstrated how an aggregated performance measure was best predicted by an aggregated knowledge measure. Furthermore, they found independent paths leading from intelligence and personality to knowledge, but not to performance. Moreover, working memory was central in their investigation; this construct was placed at the apex of their model with significant direct paths to intelligence, knowledge, and performance.

Working memory is a construct that has a close relationship to *Gf* and *g*, and can further be regarded as a construct with high referent generality. Although the relationship is complex (Ackerman, Beier, & Boyle, 2002), several researchers have reported a large degree of overlap between working memory and *g* (Ackerman et al., 2002; Colom, Rebollo, Palacios, Juan-Espinosa, & Kyllonen, 2004; Unsworth, Heitz, Schrock, & Engle, 2005). Furthermore, limitations of working memory, which are revealed in theories of skilled behavior, are an important source of errors in skilled performance (Anderson, 1987).

Computer programming is sometimes described as one of several archetypes of complex cognitive behavior; overall, programming requires the programmer to have a high level of declarative knowledge, as well as much practice to perform well. Working memory has previously been found to be a good predictor of programming skill acquisition (Shute, 1991), as has experience, at least to some extent (Arisholm & Sjøberg, 2004).

It has, however, been noted that performance on tasks that operate under skill constraints can be good predictors of other tasks. For example, for programming in the LISP programming language, Anderson states that "the best predictor of individual subject differences in errors on problems that involved one LISP concept was number of errors on other problems that involved different concepts" (Anderson, 1987, p. 203). Further, the amount of errors was correlated with the amount of programming experience (see Anderson et al., 1989 for further details).

Skill as a latent construct is inferred from observed performance that varies in both time and accuracy (or quality) (Anderson, 1987; Fitts & Posner, 1967). Together with knowledge and motivation, skill is one of three direct antecedents of performance (Campbell, McCloy, Oppler, & Sager, 1993). Skill is sometimes also referred to as procedural knowledge, and its acquisition consists of three overlapping phases. During each phase, different abilities (as antecedents) are hypothesized to exert different levels of influence on the acquisition of skill (Ackerman, 1988): General mental ability is predominant during the first cognitive phase, perceptual speed during the second associative phase, and psychomotor ability during the final and autonomous phase. Kyllonen and Woltz (1989) refer to the first phase as the "knowledge acquisition phase", while phases two and three are the skill acquisition and skill refinement phases, respectively.

Related to the view of how different abilities may exert different levels of influence during the skill acquisition phases is the concept of transfer (Ferguson, 1956). As practice on specific tasks increases, relations with previously strong determinants may diminish in favor of more task-specific factors due to limited opportunities for transfer. However, relations may also remain stable (after some decrease in correlation with initial performance) or even increase (Ackerman, 1987). This would be somewhat dependent, however, on how task performance is sampled together with what subject population is investigated. Nevertheless, research on the stability of predictions for high referent generality constructs are

important, as they apply to a wide range of novel situations that may occur in a specific job.

Much of the research on the prediction of job performance has relied on a construct of general mental ability indexed by test batteries, which measure a mixture of *Gf* and *Gc* (Valentin Kvist & Gustafsson, 2008). To distinguish this conception of general mental ability from the apex factor *g*, we will refer to it as *G*. A comprehensive meta-analysis by Schmidt and Hunter (1998) showed *G* to be a strong predictor of job performance, but they also showed that the predictive validity of work sample tests exceeds that of *G*. Also, in another meta-analysis (Schmidt, Hunter, & Outerbridge, 1986), the strongest determinant of job sample performance was job knowledge (0.74), much higher than the direct path from experience (0.08) or general mental ability (0.04). These results bear a close resemblance to what would be expected from the investment theory, despite the obvious conceptual and theoretical differences between *G* and working memory as well as job performance and skill.

Although research has been conducted on group differences in the acquisition of programming skill, individual differences in already acquired skill have not been adequately quantified. One reason for the lack of research on the quantification of skills is that the construct validity of programming performance is currently unresolved (Hannay, Arisholm, Engvik, & Sjøberg, 2010); indeed, the construct validity of work samples in general seems unresolved (Campbell, 1990). Programming usually operates under a time-quality tradeoff, making any single score assigned to a solution an aggregate tradeoff, where the relative weight of the quality of the solutions becomes important with respect to the time used to obtain that solution. In addition, to what degree observed programming performance can be generalized to other tasks, systems, people or countries is uncertain.

This study investigates the relationship between programming skill and its main antecedents, using Cattell's investment theory (1971/1987) as a conceptual framework. We predict that, in accordance with skilled behavior theory, investment theory, and previous research on work samples, programming knowledge is the main causal antecedent of programming skill. Further, programming knowledge is expected to mediate the relationship between programming skill and the causal variables of working memory and experience.

2 Method

2.1 Participants

Sixty-five professional software developers were hired from nine companies located in eight Central/Eastern European countries: Belarus, Czech Republic, Italy, Lithuania, Moldova, Norway, Poland, and Russia. Each participant received their base salary for participation, but they were not compensated beyond and above this. The overall cost of hiring the developers was approximately EUR 50,000 including travel (EUR 10,000). All companies and subjects were guaranteed anonymity, and subjects were free to terminate participation at any time without loss of compensation. Although we requested voluntary participation, this was difficult to guarantee, as negotiations were conducted with company project leaders, not the subjects themselves. All developers were required to be proficient

in English and to have at least 6 months of consecutive programming experience in the Java programming language prior to participating in the study.

The mean age of participants was 28 years (SD = 5.66, range = 21–53) with a mean professional working experience of 5.3 years (SD = 4.94, range = 0.3–30). Of this group, 63.1% had a master's degree, 33.8% a bachelor's degree, and 3.1% a high-school degree; 10.8% were female.

2.2 Tests Administered

All materials and instructions were in English, which is today the de facto business language of software developers working in the global IT industry. Some minor language problems occurred during the study for a few of the subjects.

2.2.1 Java Programming Skill

As a proxy for a job sample test of programming performance, we used an instrument for assessing programming skill (Bergersen, in preparation). The instrument contains 12 programming tasks (items) in the Java programming language, which has become one of the most common programming languages during the last decade. All tasks required either implementation or modification of programming code; the duration of each task was between 10 and 50 minutes. Between 5 and 10 minutes was also allowed for reading the task instructions before code was downloaded.

Performance on each task was scored as an aggregate of both the time required to implement a correct solution and the quality of the solution, following principles delineated in (Bergersen, Hannay, Sjøberg, Dybå, & Karahasanović, 2011). Starting from incorrect tasks (or tasks submitted too late), increasingly higher (ordinal) scores were assigned to more correct solutions. Further, even higher scores were given to tasks of acceptable quality (or correctness), but with less time used. This scoring procedure follows the maxim recognized in the 1920s by Thorndike and others: "[S]peed measurements should properly be taken only for correct responses if they ... [are] to be studied in relation to [ability]" (Carroll, 1993, p. 442).

The time versus quality tradeoff function of tasks cannot be calculated because of the many different and noncomparable operationalizations of programming tasks in the skill instrument. However, in general, a negative correlation is frequently observed in empirical studies of programmers (Bergersen et al., 2011). The instrument consists of both automatically and semi-automatically scored tasks, as some programming quality aspects can only be evaluated by humans. The scoring scheme for these kinds of tasks used objective criteria (such as, "is functionality x present for attribute y?" 0 = *no*, 1 = *partially*, 2 = *yes*). Some tasks were testlets, implying that multiple requirements were to be solved in consecutive steps until time ran out.

For subject scaling, task scores were fitted to the polytomous Rasch model (Andrich, 1978). This is a generalization of the Rasch model (1960), where interval scale estimates of person abilities can be obtained, provided certain conditions are met. In particular, the ability investigated must be a quantifiable variable (Michell, 1997)—an assumption that is tested only indirectly by the Rasch model.

2.2.2 Programming Knowledge

A 30-item multiple choice knowledge test of Java programming was purchased from a large international (anonymous) test vendor. Items were selected from a large pool of existing items currently used for assessing software professionals globally. All items were specifically selected to cover the same content present in the programming skill instrument so that both skill and knowledge operate on the same level of generality. The knowledge test imposes a 3-minute time limit per item, but is not speeded. The test takes approximately 1 hour to complete and is scored according to the number of correct items within each item's time limit.

2.2.3 Working Memory

Three 20-minute tests of working memory (Operation Span, Symmetry Span, and Reading Span) were acquired from Unsworth et al. (2005). Each test requires the memorization of letters or locations while simultaneously having to solve simple math problems, determining whether a figure is symmetrical, or determining whether an English sentence is correctly formulated. For the distracting task, subjects were informed that accuracy must be kept over 85% to limit memorization of letters or locations. However, for Reading Span, we informed each subject that accuracy below 85% would be acceptable as none of the subjects were native English speakers. Each test was scored according to the number of perfectly recalled sets.

2.3 Procedure

The programming skill test was administered at each company's office location during two full work days (16 hours). The first author was present during the study to answer questions, but the test was administered online through a specially built experiment support environment (Arisholm, Sjøberg, Carelius, & Lindsjørn, 2002). Subjects were free to use whatever software development tools they normally used in their job. All subjects were given instructions and a practice task before the instrument was administered. All programming tasks were administered in random order, and the subjects were only allowed to take breaks between tasks. The working memory tests were, furthermore, administered during these individual breaks. The working memory tests were, however, not available for the first four companies visited, implying fewer subjects ($n = 29$) for these tests. The knowledge test was administered online, 1 to 4 months after the programming skill test, on a subsample of the subjects (92.3%, $n = 60$). Missing developers in this subsample had either changed employer, did not want to participate, or were currently involved in other projects.

2.4 Statistical Analysis and the Model Tested

The hypothesized causal relationship between the variables investigated is as follows: Working memory and experience are hypothesized to affect programming knowledge di-

rectly. Furthermore, direct paths from working memory and experience to skill should not be present as the causal effect of these two constructs should be mediated through knowledge. Finally, programming knowledge affects programming skill.

In the model, programming skill was represented as a latent variable with a single indicator, following the procedure described by Jöreskog and Sörbom (as cited in Mathieu, Tannenbaum, & Salas, 1992, p. 837):

> [T]he path from a latent variable to its corresponding observed variable (lambda) is equal to the square root of the reliability of the observed score. In addition, the associated amount of random error variance (theta) is equal to one minus the reliability of the observed score times the variance of the observed score.

We used person separation index (PSI) as a reliability estimate (see e.g., Streiner, 1995), because PSI can be calculated with missing data. However, results are virtually identical to those obtained when using Cronbach's α.

Programming knowledge was represented as a latent variable by two indicators with equal loadings and variances. The 30 items were divided randomly into two parcels, each with the same average difficulty (items were delivered by the test provider in increasing order of difficulty based on data from their existing item bank). Working memory was represented by the three working memory tests, and neither factor loadings nor error variances were constrained.

Programming experience consists of two indicators. The first indicator, *mJava*, is the total number of months the individual has programmed in Java, both during training (if applicable) and professionally. The second indicator is the subject's estimate of how many lines of code (LOC) that person has written in the Java programming language in total (during training and professional career). This variable is right skewed (skew = 2.70), something which poses a statistical problem in the analysis. However, *lnLOC* approximates a normal distribution after log transformation ($skew = -0.06$).

The model reported was estimated with Amos 16.0, using maximum likelihood estimation for missing data which is implemented in this program. The raw and transformed correlation matrix for the variables investigated is shown in Table 1.

3 Results

3.1 Descriptive Statistics and Reliability Estimates

Descriptive results are shown in Table 2. The Java skill instrument was sufficiently reliable to assess individual differences (PSI = 0.86 and $\alpha = 0.85$). As mentioned, results on the working memory test were available only for approximately half the subjects, while for the other variables few data are missing.

The programming skill instrument is unidimensional according to the dependent sample t-test for unidimensionality (Smith, 2002), a procedure also implemented in the software used to carry out the Rasch analysis, RUMM2020 (Andrich, Sheridan, & Luo, 2006).

Table 1: Original and transformed correlation matrix

	(1)		(2)		(3)		(4)		(5)		(6)		(7)		(8)	
	N	*r*	N	*r*	N	*r*	N	*r*	N	*r*	N	*r*	N	*r*	N	*r*
raschSkill (1)	65	—	60	0.640	60	0.688	28	0.385	28	0.165	29	0.374	64	0.285	65	0.285
knowP1 (2)		0.667	60	—	60	0.708	25	0.255	25	0.111	26	0.385	59	0.319	60	0.324
knowP2 (3)		0.667		0.713	60	—	25	0.344	25	0.407	26	0.438	59	0.295	60	0.208
wmS (4)		0.237		0.248		0.248	28	—	27	0.290	28	0.190	28	0.328	28	0.036
wmR (5)		0.214		0.221		0.221		0.189	28	—	28	0.418	28	0.034	28	–0.037
wmO (6)		0.339		0.353		0.353		0.297		0.265	29	—	29	0.337	29	–0.011
lnLOC (7)		0.333		0.347		0.347		0.184		0.167		0.264	64	—	64	0.476
mJava (8)		0.261		0.272		0.272		0.144		0.130		0.206		0.484	65	—

Notes. The part below the diagonal is the transformed correlation matrix. Correlations of 1.00 are denoted —. N is the number of observations, and *r* is the correlation. Variable names are: raschSkill = Java programming skill; knowP1 and knowP2 = knowledge for Parcel 1 and 2 respectively; wmS = working memory Symmetry span; wmR = working memory Reading span; wmO = working memory Operation span; lnLOC = logarithmic transformation of lines of code written in the Java programming language; mJava = months of experience programming in Java.

Table 2: Descriptives of all variables used

Variable	Score	N	Min	Max	Mean	SD	1KS
Programming skill	in logits*	65	–4.12	1.58	–0.83	1.30	0.489
Programming knowledge	NC	60	7	29	21.72	4.80	0.013
Working memory							
Symmetry span	NR	28	10	42	25.11	8.36	0.979
Operation span	NR	29	21	75	52.14	15.33	0.945
Reading span	NR	28	25	75	54.25	14.48	0.750
Experience							
Total lines of code written	ln(LOC)	64	6.21	13.82	10.46	1.76	0.806
Total programming experience	months	65	2	130	39.98	25.21	0.627

Notes. * 0 logits is defined in the Rasch model as the mean difficulty of items. NC = number of correct responses, NR = number of perfectly recalled sets.

The programming knowledge test had acceptable reliability ($\alpha = 0.81$), but was not unidimensional. We did not calculate reliability of the working memory tests; these have previously been reported to have acceptable levels of reliability (Unsworth et al., 2005). Also, although ceiling effects were present for all the working memory tests, the mean and SD of our population was comparable to results reported elsewhere (Unsworth, Redick, Heitz, Broadway, & Engle, 2009). All variables used in the analysis except programming knowledge can be regarded as normally distributed by the one-sample Kolmogorov-Smirnov Test (1KS).

3.2 The Investigated Model According to the Investment Theory

The model investigated and shown in Figure 1 had a close model fit ($\chi^2[19] = 15.2$, *ns*, RMSEA = 0.000, LO90 = 0.000, HI90 = 0.084). Because of the wide confidence interval for RMSEA, the statistical power to reject a poor-fitting model is relatively low. However, the confidence interval was almost entirely within the range that indicates a well-fitting model (RMSEA < 0.08). We also tested adding direct paths from working memory and experience to skill, something that should not be present according to investment theory. The loadings were -0.02, *ns* and -0.06, *ns*, respectively, with a slightly reduced model fit ($\chi^2[17] = 15.0$, *ns*, RMSEA = 0.000, LO90 = 0.000, HI90 = 0.100).

The high latent correlation between skill and knowledge merits further considerations as to whether in the current study these two constructs differ only in method variance. In a posthoc analysis, we first tried to combine knowledge and skill data as a unidimensional measure. However, this operationalization displayed clear signs of misfit as well as departure from unidimensionality. We therefore attempted to model knowledge separately by using Rasch analysis again. By dropping three items that contributed negatively to model fit as well as unidimensionality, we obtained a reasonably well-fitting, unidimensional, and internally consistent ($\alpha = 0.81$) representation of programming knowledge that was normally distributed. Nevertheless, the overall results were highly similar to those already reported.

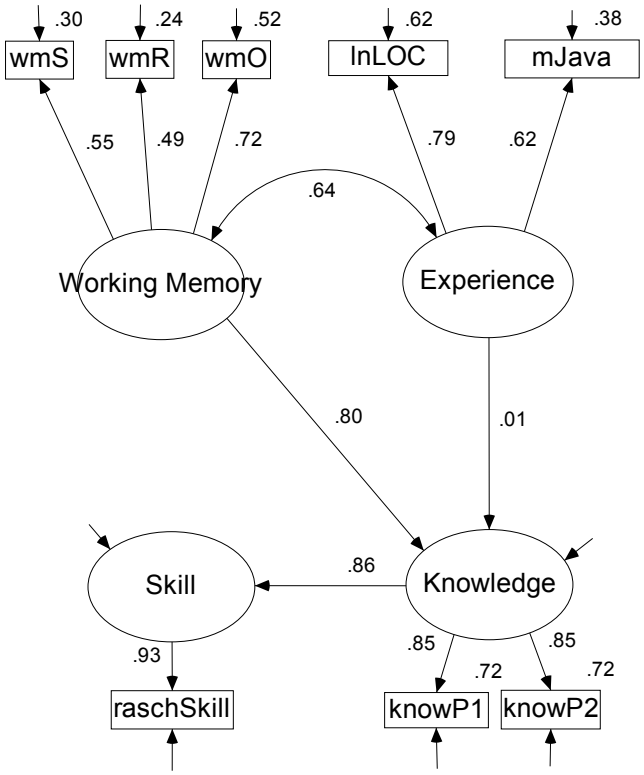


Figure 1: Results for the investigated model.

One explanation for the high latent correlation between these two constructs therefore probably lies elsewhere. For theoretical reasons alone, we would expect knowledge and skill to be close to unity for individuals who currently are in the first phase of skill acquisition (i.e., currently acquiring knowledge). However, this relationship should begin to diverge in the second and third phase of skill acquisition. In the programming skill instrument, the programming tasks can be regarded as consistent, something that makes automatic processing (see Ackerman, 1988) possible. However, developers who do not have sufficient programming experience would be forced to use controlled processing, which would lower their estimated level of programming skill, as the controlled processing would be much slower and prone to errors. Conversely, the score on the knowledge test would not benefit from automatic processing as items were presented with leisure time limit.

There are also other important differences between these two tests. For the knowledge test, guessing is possible, and the test requires only the identification of the correct response among a set of alternatives. In contrast, the skill test has severe time limits and requires maximal performance over extended periods of time. This performance is further constrained by limitations in perceptual and psychomotor speed. Moreover, subjects use their normal tools and programming environment and benefit from knowing these well in the skill instrument. Finally, many of the tasks in the skill instrument are too comprehensive to comprehend in whole; instead subjects must use their experience to quickly identify areas where modifications should be implemented, using a suitable strategy for this.

4 Discussion

The good fit of our data to the investigated model supports Cattell's investment theory: The effect of working memory capacity and experience on programming skill is mediated by programming knowledge, which in turn accounts for a large degree of variance in programming skill. Furthermore, as elaborated in theories of skill acquisition (Ackerman, 2000; Anderson, 1987; Fitts & Posner, 1967), there are good theoretical reasons why skill is mediated by knowledge; i.e., knowing what to do is prior to improved performance. We now turn to how the findings can be interpreted in more specific terms.

Our results show several similarities to the study by Wittmann and Süß (1999), who also addressed the investment theory. In both studies, working memory was placed at the apex, whereas knowledge was best placed as a mediator variable between higher-order constructs and performance. There are, however, several noticeable differences in results. First, their knowledge construct, which operated at a higher level of generality, had a weaker path (0.54) to performance compared to our results. Further, they reported weaker paths from working memory to knowledge (0.24), and working memory furthermore had a direct path to performance as well (0.27). We believe the latter might be attributed to their studying students on novel tasks where sufficient time may not have been available to develop a high level of skill. Alternatively, results may be different when studying skills that take years to evolve compared to systems where the last stage of skill acquisition is reached within a short duration (their study lasted 3 days).

Our research also reveals at least three important similarities with the meta-analysis by Schmidt et al. (1986), bearing in mind that Schmidt et al. reported on different types of jobs as well as different job-related constructs. First, knowledge was found in both studies to be the most important antecedent of skill/job sample performance. Second, the direct effect on skill/job sample performance from working memory/G and experience was small in Schmidt et al. and insignificant and close to zero in our study. Finally, Schmidt et al. reported a path between G and job knowledge of 0.65 for civilian data when excluding experience, while the path from working memory to knowledge in our study is 0.77 when experience is excluded.

The most obvious difference between our results and those of Schmidt et al. (1986), is the high correlation we observed between working memory and experience; G and experience were not correlated in their study. However, when we inspect Table 1, we see that lnLOC is clearly the variable responsible for this positive latent correlation. One reason for this may be that programmers with a low working memory capacity may not continue to program over many years, due to the continuous learning required to stay updated on the programming language they use for development.

Further, in the Schmidt et al. (1986) meta-analysis, the impact on knowledge from experience was 0.57, which is somewhat larger than for G (0.46). In our study, working memory explained most of the variance, although a lot of this variance was shared with experience. However, the path from experience to knowledge increased to 0.54 when we omitted working memory, indicating a similar result.

One possible reason for these differences is that Schmidt et al. (1986) used the number of months on the present job for the experience variable, while we used both the number

of months programming in Java (irrespective of job changes) and number of lines of code previously written in Java. As programming should be seen as a high-complexity job, it would be expected that the effect of experience on knowledge would be less than for low-complexity jobs (McDaniel, Schmidt, & Hunter, 1988). The reason for this is that initial experience with programming is often obtained through the educational system rather than on the job (in our subject sample, 96.9% had a bachelor's degree or higher).

Other questions remain. We do not know how well the instrument predicts programming performance "on the job". Also, we do not know if skill inferred from programming performance and attested by a unidimensional instrument can partially resolve the poorly understood construct validity of work samples (see Campbell, 1990). However, this study does meet Campbell, Gasser, and Oswald's criticism (1996) that using ratings of overall job performance as the dependent variable in SEM models is inadequate; we used instead a unidimensional measure of programming skill. Further, we acknowledge that working memory as a construct with high referent generality is used as a proxy for g/Gf measures, which, ideally, should both have been included in the study as well. Moreover, the use of SEM to investigate purportedly causal relations (such as those expressed in Cattell's investment theory) does not provide a test of causality per se. Instead, the path coefficients express the strength of between-construct relations under the assumption that the causal model is true. More advanced research designs that are longitudinal or experimental are required in the future to directly test whether these causal relationships hold true or not.

The main limitation of this study is low statistical power, in particular the number of observations for working memory. Because of the low power, the confidence intervals for RMSEA were wide. Additionally, the standard errors associated with correlations and loadings for working memory as well as the path from working memory to knowledge were large. Calculating Bayesian posterior distribution estimates in AMOS, our model seldom converged according to conventional criteria (Gelman, Carlin, Stern, & Rubin, 2004). Although the path coefficients from knowledge to skill were acceptable, the standard errors for loadings and path coefficient for working memory were not. Therefore, the loadings for working memory as well as influence on knowledge should be regarded as preliminary and interpreted with caution. Another concern was the wide age range of subjects with respect to working memory. However, there were only insubstantial differences when older developers were removed from the sample.

Another possible concern was the parceling procedure used for the knowledge test. We also investigated other approaches to representing knowledge, but found only insubstantial differences for regression weights. However, RMSEA estimates were somewhat more affected—both for better and for worse—by different combinations of items in each parcel, but well within the reported 90% confidence limits.

Clearly, there are several loose ends requiring further work. In addition to including more subjects, we want to include a purer g measure, such as Ravens. Further, to better test the investment theory, we should also require Gc measures that are applicable for use in heterogeneous samples of nonnative English-speaking software professionals from multiple countries. We initially attempted to represent Gc by self-reported grades for middle school, high school, and postgraduate studies. However, these results were in-

conclusive as correlations with programming skill were near zero and insignificant, and further detrimental to model fit.

We should additionally require measures of motivation and personality. However, for the latter, we have so far met with limited success when using personality to predict programming performance in studies carried out at Simula Research Laboratory (see, e.g., Hannay et al., 2010). Nevertheless, if personality is viewed from the perspective of Brunswik symmetry as a construct of high referent generality (Wittmann & Süß, 1999), we would most likely benefit from obtaining higher referent generality measures of "job performance" as well. One such representation could be peer or supervisor ratings of performance, which may yield interesting results when inspected together with the tests already employed in this study.

Our overall results support Cattell's investment theory. The results are also similar to those reported by others, albeit with some differences in operationalizations. Although the statistical power of this study is low, the results show reasonably close fit to an investment theory model framed within a high-realism setting of software developers from multiple countries.

Acknowledgments

This research was supported by the FORNY program at the Research Council of Norway. It is based on the first author's ongoing PhD thesis with Dag Sjøberg and Tore Dybå as supervisors. We thank Steinar Haugen for his programming assistance, and Erik Arisholm and Jo Hannay for their useful insights. We also thank the project managers, developers, and companies involved in this study, as well as the test vendor company for allowing us to use their tests. Finally, we thank the helpful anonymous reviewers for comments and suggested improvements in earlier versions this manuscript.

References

Ackerman, P. L. (1987). Within-task intercorrelations of skilled performance: Implications for predicting individual differences (a comment on Henry and Hulin, 1987). *Journal of Applied Psychology*, *74*, 360–364.

Ackerman, P. L. (1988). Determinants of individual differences during skill acquisition: Cognitive abilities and information processing. *Journal of Experimental Psychology: General*, *117*(3), 288–318.

Ackerman, P. L. (2000). Domain-specific knowledge as the "dark matter" of adult intelligence: Gf/Gc, personality and interest correlates. *Journal of Gerontology*, *55B*, P69–P84.

Ackerman, P. L. (2007). New developments in understanding skilled performance. *Current Directions in Psychological Science*, *16*, 235–239.

Ackerman, P. L., Beier, M. E., & Boyle, M. O. (2002). Individual differences in working memory within a nomological network of cognitive and perceptual speed abilities. *Journal of Experimental Psychology: General*, *131*, 567–589.

Anderson, J. R. (1987). Skill acquisition: Compilation of weak-method problem solutions. *Psychological Review*, *94*(2), 192–210.

Anderson, J. R., Conrad, F. G., & Corbett, A. T. (1989). Skill acquisition and the LISP Tutor. *Cognitive Science*, *13*(4), 467–505.

Andrich, D. (1978). A rating formulation for ordered response categories. *Psychometrika*, *43*(4), 561–573.

Andrich, D., Sheridan, B., & Luo, G. (2006). Rumm2020 [Computer software manual]. Perth.

Arisholm, E., & Sjøberg, D. I. K. (2004). Evaluating the effect of a delegated versus centralized control style on the maintainability of object-oriented software. *IEEE Transactions on Software Engineering*, *30*(8), 521–534.

Arisholm, E., Sjøberg, D. I. K., Carelius, G. J., & Lindsjørn, Y. (2002). A web-based support environment for software engineering experiments. *Nordic Journal of Computing*, *9*(3), 231–247.

Bergersen, G. R. (in preparation). *Measuring programming skill.* Unpublished doctoral dissertation, University of Oslo, Oslo, Norway.

Bergersen, G. R., Hannay, J. E., Sjøberg, D. I. K., Dybå, T., & Karahasanović, A. (2011). Inferring skill from tests of programming performance: Combining time and quality. In *Proceedings of the 5th International Symposium on Empirical Software Engineering and Measurement* (pp. 305–314).

Campbell, J. P. (1990). Modeling the performance prediction problem in industrial and organizational psychology. In M. D. Dunnette & L. M. Hough (Eds.), *Handbook of industrial and organizational psychology* (Second ed., Vol. 1, pp. 687–732). Palo Alto, CA: Consulting Psychologists Press.

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

Campbell, J. P., McCloy, R. A., Oppler, S. H., & Sager, C. E. (1993). A theory of performance. In N. Schmitt & W. C. Borman (Eds.), *Personnel selection in organizations* (p. 35-70). San Francisco, CA: Jossey-Bass.

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

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

Coan, R. B. (1964). Facts, factors and artifacts: The quest for psychological meaning. *Psychological Review*, *71*, 123–140.

Colom, R., Rebollo, I., Palacios, A., Juan-Espinosa, M., & Kyllonen, P. (2004). Working memory is (almost) perfectly predicted by g. *Intelligence*, *32*, 277–296.

Ferguson, G. A. (1956). On transfer and the abilities of man. *Canadian Journal of Psychology*, *10*(3), 121–131.

Fitts, P. M., & Posner, M. I. (1967). *Human performance.* Belmont, CA: Brooks/Cole.

Gelman, A., Carlin, J. B., Stern, H. S., & Rubin, D. B. (2004). *Bayesian data analysis* (Second ed.). Boca Raton, FL: Chapman and Hall/CRC.

Gustafsson, J.-E. (1984). A unifying model for the structure of intellectual abilities. *Intelligence*, *8*(3), 179–203.

Gustafsson, J.-E. (2002). Measurement from a hierarchical point of view. In H. Braun, D. N. Jackson, & D. E. Wiley (Eds.), *The role of constructs in psychological and educational measurement* (pp. 73–95). New York: Erlbaum.

Hannay, J. E., Arisholm, E., Engvik, H., & Sjøberg, D. I. K. (2010). Personality and pair programming. *IEEE Transactions on Software Engineering*, *36*(1), 61–80.

Humphreys, L. G. (1962). The organization of human abilities. *American Psychologist*, *17*(475–483).

Kyllonen, P. C., & Woltz, D. J. (1989). Role of cognitive factors in the acquisition of cognitive skill. In R. Kanfer, P. L. Ackerman, & R. Cudeck (Eds.), *Abilities, motivation, and methodology: The Minneapolis symposium on learning and individual differences* (pp. 239–280). Mahwah, NJ: Erlbaum.

Mathieu, J. E., Tannenbaum, S. I., & Salas, E. (1992). Influences of individual and situational characteristics on measures of training effectiveness. *The Academy of Management Journal*, *35*, 828–847.

McDaniel, M. A., Schmidt, F. K., & Hunter, J. E. (1988). Job experience correlates of job performance. *Journal of Applied Psychology*, *73*, 327–330.

Michell, J. (1997). Quantitative science and the definition of measurement in psychology. *British Journal of Psychology*, *88*(3), 355–383.

Neves, D. M., & Anderson, J. R. (1981). Knowledge compilation: Mechanisms for the automatization of cognitive skills. In J. R. Anderson (Ed.), *Cognitive skills and their acquisition* (pp. 57–84). Mahwah, NJ: Erlbaum.

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

Schmidt, F. L., & Hunter, J. E. (1998). The validity and utility of selection methods in personnel psychology: Practical and theoretical implications of 85 years of research findings. *Psychological Bulletin*, *124*(2), 262–274.

Schmidt, F. L., Hunter, J. E., & Outerbridge, A. N. (1986). Impact of job experience and ability on job knowledge, work sample performance, and supervisory ratings of job performance. *Journal of Applied Psychology*, *71*(3), 432–439.

Shute, V. J. (1991). Who is likely to acquire programming skills? *Journal of Educational Computing Research*, *7*(1), 1–24.

Smith, E. V., Jr. (2002). Detecting and evaluating the impact of multidimensionality using item fit statistics and principal component analysis of residuals. *Journal of Applied Measurement*, *3*(2), 205–231.

Streiner, D. L. (1995). *Health measurement scales.* Oxford: Oxford University Press.

Unsworth, N., Heitz, R. P., Schrock, J. C., & Engle, R. W. (2005). An automated version of the operation span task. *Behavior Research Methods*, *3*(37), 498–505.

Unsworth, N., Redick, T., Heitz, R. P., Broadway, J. M., & Engle, R. W. (2009). Complex working memory span tasks and higher-order cognition: A latent-variable analysis of the relationship between processing and storage. *Memory*, *17*(6), 635–654.

Valentin Kvist, A., & Gustafsson, J.-E. (2008). The relation between fluid intelligence and the general factor as a function of cultural background: A test of Cattell's

investment theory. *Intelligence*, *36*(5), 422–436.

Wittmann, W. W., & Süß, H.-M. (1999). Investigating the paths between working memory, intelligence, knowledge, and complex problem-solving performances via Brunswik symmetry. In P. L. Ackerman, P. C. Kyllonen, & R. D. Roberts (Eds.), *Learning and individual differences: Process, trait, and content determinants* (pp. 77–108). Washington, DC: American Psychological Association.

Paper IV:

Evaluating Methods and Technologies in Software Engineering with Respect to Developers' Skill Level

Gunnar R. Bergersen and Dag I. K. Sjøberg

Proceedings of the 5th International Symposium on on Evaluation and Assessment in Software Engineering, 2012, pp. 101–110.

Abstract

Background: It is trivial that the usefulness of a technology depends on the skill of the user. Several studies have reported an interaction between skill levels and different technologies, but the effect of skill is, for the most part, ignored in empirical, human-centric studies in software engineering. *Aim*: This paper investigates the usefulness of a technology as a function of skill. *Method*: An experiment that used students as subjects found recursive implementations to be easier to debug correctly than iterative implementations. We replicated the experiment by hiring 65 professional developers from nine companies in eight countries. In addition to the debugging tasks, performance on 17 other programming tasks was collected and analyzed using a measurement model that expressed the effect of treatment as a function of skill. *Results*: The hypotheses of the original study were confirmed only for the low-skilled subjects in our replication. Conversely, the high-skilled subjects correctly debugged the iterative implementations faster than the recursive ones, while the difference between correct and incorrect solutions for both treatments was negligible. We also found that the effect of skill (odds ratio = 9.4) was much larger than the effect of the treatment (odds ratio = 1.5). *Conclusions*: Claiming that a technology is better than another is problematic without taking skill levels into account. Better ways to assess skills as an integral part of technology evaluation are required.

Keywords: programming skill, pretest, experimental control, debugging, performance, replication.

1 Introduction

When studying the effects of software processes, products, or resources, a researcher is often forced to keep constant or control for factors that may influence the outcome of the experiment. Because previous studies have shown large variability in programming performance, it is important to control for this. However, it is not a simple task to control for programming skill (Basili, Shull, & Lanubile, 1999; Brooks, 1980; Kampenes, Dybå, Hannay, & Sjøberg, 2009; Sjøberg et al., 2002), which is one of several factors that affect programming performance (Bergersen & Gustafsson, 2011; Bergersen, Hannay, Sjøberg, Dybå, & Karahasanović, 2011).

Individual differences may also moderate the claimed benefit of different technologies. In a one-day experiment on professionals maintaining two different implementations of the same system, seniority had an effect on which system was better: The system that used a "poor" object-oriented design was better for juniors, whereas the system that used a "good" design was better for seniors (Arisholm & Sjøberg, 2004). The effect of pair programming on the same system was also investigated in (Arisholm, Gallis, Dybå, & Sjøberg, 2007); overall, the juniors benefitted from working in pairs whereas the seniors did not. Such results are clearly problematic if one aims to generalize from the study population to a target population specified only as "software developers".

When an independent variable, such as skill or seniority, is correlated with the dependent (outcome) variable of a study, it is relevant to address this variable in relation to the experimental results (Shadish, Cook, & Campbell, 2002). Improved control can be achieved during experiment design (e.g., through blocking or matching) or in analysis (e.g., as a covariate). In both instances, the statistical power increases (Maxwell, 1993).

Another way to increase statistical power in studies is to reduce subject variability (Shadish et al., 2002). However, the individual differences of developers are, perhaps, some of the largest factors that contribute to the success or failure of software development in general (Brooks, 1995; Glass, 2001). Several studies report an "individual-differences" factor (e.g., due to differences in skill) that is highly variable across individuals (Brooks, Daly, Miller, Roper, & Wood, 1996), teams (Prechelt, 2011), companies (Anda, Sjøberg, & Mockus, 2009), and universities (Krein, Knutson, Prechelt, & Juristo, 2012), thereby complicating analysis and adding uncertainty to the results. Meta-analysis has also confirmed that individual variability in programming is large, even though it may appear less than the 1:28 differences reported in the early days of software engineering (Prechelt, 1999). Nevertheless, large variability in skill levels implies that one should be meticulous when defining the sample population as well as the target population in empirical studies in software engineering.

An indicator of programming skill that is easy to collect is months of experience or lines of code written by the subjects. Large meta-analyses have indicated that biographical measures, such as experience, generally have low predictive validity in studies on job performance (Schmidt & Hunter, 1998). At the same time, work sample tests that involve actual (job) tasks have the highest degree of validity.

Using one set of tasks to predict performance on another set of tasks is not new: Anderson studied the acquisition of skills in LISP programming and found that "the best

predictor of individual differences in errors on problems that involved one LISP concept was number of errors on other problems that involved different concepts" (1987, p. 203). Therefore, pretests appear to be better measures of skill than biographical variables, even though they require a lot more instrumentation.

Calls for better pretests for programmers can be traced back to at least 1980 (Brooks, 1980). Yet, in a 2009 literature review on quasi experiments in software engineering, only 42% of the reported 113 studies applied controls to account for potential selection bias (Kampenes et al., 2009). Among the studies that applied controls, only three experiments involved actual pretest tasks. (The remaining studies used covariates such as lines of code, exam scores, or years of experience.) The authors therefore restated previous calls (see Basili et al., 1999; Brooks, 1980) for initiatives where the interaction between different types of technologies and software developer capabilities could be investigated.

This paper reports a replication of a debugging study where a measure of programming skill is available using a pretest. Unlike (Arisholm et al., 2007; Arisholm & Sjøberg, 2004), where a *small* pretest and a comprehensive experiment were used, we conducted a *comprehensive* pretest and a small replication. Our overall research question is: what are the moderating effects of skill levels on the purported benefit of different technologies or methods? In this study, we investigated whether the usefulness of recursive implementations in debugging is invariant of skill level. Further, we also aimed to assess the individual variability in skill, which is potentially a confounding factor, with respect to different implementations of two small debugging tasks.

To do so, we analyzed the effect of using different debugging implementations in a specific measurement model (the Rasch model) where the effect of treatment is expressed as a function of skill. Moreover, we use professional software developers, thereby addressing the common criticism that researchers habitually use students in experiments (see, e.g., Basili et al., 1999; Brooks, 1980; Sjøberg et al., 2002). Although debugging studies and replications are interesting in their own right, the focus here is on methodical issues: Specifically, we investigate the effect of skill levels on the generalizability of the main conclusions of two earlier studies.

Section 2 describes method and materials. Section 3 reports results and Section 4 analyzes these results using programming skill as a covariate. Section 5 discusses implications, limitations, and suggestions for further work. Section 6 concludes the study.

2 Methods and Materials

Section 2.1 describes the material, experimental procedure and results of the original study. Section 2.2 describes the material and experimental procedure of our replication. Section 2.3 introduces the Rasch measurement model, which is used in the analysis in Section 4.

2.1 The Original Study

The original study involved 266 students who took a course on data structures (Benander, Benander, & Sang, 2000). The students were presented with two C implementation tasks:

(1) a small (< 25 lines of code) search program for a linked list ("Find" task) and (2) a linked list that was to be copied ("Copy" task). Each task had either a recursive or iterative implementation (the treatment of the study). Both tasks contained a bug that had to be correctly *identified* and *corrected*. The study found that significantly more subjects identified the bug for the recursive versions than they did for the iterative version. A similar result was also found for one of the tasks in an earlier comprehension study using PASCAL (Benander, Benander, & Pu, 1996). Regarding correcting the bug, recursion also gave significantly better results with respect to the proportion of correct solutions for the Copy task ($p = 0.019$). However, the results for the Find task were in weak (non-significant) favor of iteration. When the results of the two tasks were combined, the recursive versions had 4.1% more correct solutions overall, a result that was not significant ($p = 0.311$). For the time required to correctly debug the tasks, the original study was not significantly in favor of any of the treatments.

The original study used a randomized within-subject (repeated measures) crossover design. Both treatments were presented to all subjects. Either one of them used the iterative treatment first and the recursive treatment second or vice versa. However, the Find task was always presented before the Copy task. Therefore, it is unknown to what extent an *ordering effect* is present (see generally Shadish et al., 2002), for example, whether iterative Find and then recursive Copy is an easier order to solve the tasks than recursive Find and then iterative Copy. The tasks were debugged manually using "hand tracing".

2.2 This Replication

Our replication is part of an ongoing work for constructing an instrument for assessing programming skill. We conducted a study with sixty-five professional software developers who were hired from nine companies for approximately €40,000. The companies were located in eight different Central or Eastern-European countries. All the subjects were required to have at least six months of recent programming experience in Java. The subjects used the same development tools that they normally used in their jobs. The programming tasks, which included code and descriptions, were downloaded from an experiment support environment (Arisholm, Sjøberg, Carelius, & Lindsjørn, 2002) that was responsible for tracking the time spent on implementing each solution. Neither the developers nor their respective companies were given individual results.

The study lasted two days and consisted of 17 Java programming tasks in total. A subset of 12 of these tasks had previously been found to adequately represent a programming skill as a single measure that is normally distributed and sufficiently reliable to characterize individual differences (Bergersen & Gustafsson, 2011). Further, this measure was also significantly positively correlated with programming experience, a commercially available test of programming knowledge, and several tests of working memory, which is an important psychological variable in relation to skill (see Shute, 1991; Woltz, 1988). The overall results accorded with a previous meta-analysis on job performance (see Schmidt & Hunter, 1998) and Cattell's investment theory, which describes how the development of skills in general is mediated by the acquisition of knowledge (Cattell, 1971/1987).

Table 1: The design of the replicated study

Find		Copy		
Recursive	Iterative	Recursive	Iterative	*n*
X			X	22
	X	X		21
X		X		9
	X		X	12

Note. Which of the two tasks were presented first was randomized

In this replication, the subjects received the two debugging tasks described above in addition to the 17 Java programming tasks. Allocation to treatment version (recursive or iterative) for both tasks was random. One subject was removed because the subject was an extremely low outlier regarding skill.

This resulted in 64 pairs of Find and Copy tasks in a crossover design, as shown in Table 1. To reduce the risk of an ordering effect (see, e.g., Shadish et al., 2002), we improved the design of the original study by randomizing on task order (Find versus Copy first) and including the recursive-recursive and iterative-iterative designs. Further, all the 19 tasks were allocated to the subjects in random order on a subject-by-subject basis.

The subjects were given 10 minutes to solve each debugging task. They were also allowed three additional minutes to upload the solution. Up to five minutes were allowed for reading the task description prior to downloading the code task. It was explicitly explained to the subjects that the time they spent reading task descriptions was not included in the time recorded for solving the tasks. Tasks that were submitted too late (i.e., more than 13 minutes in total) were scored as incorrect. This procedure was explained to the subjects prior to the start of the study. Time was only analyzed and reported for correct solutions.

In our replication, we focus only on differences for whether a bug was *corrected* or not. Our study design and available resources did not enable us to identify whether a bug was correctly identified (see Section 2.1) and then incorrectly corrected. The time to correctly debug a task in our study was not comparable to the original study because of differences in how the tasks were presented to the subjects.

We used R (R Development Core Team, 2008) for statistical analysis. Unless otherwise noted, Fisher's exact test was used to test differences in correctness, Welch's t-test for differences in time, and Spearman's rho to report (non-parametric) correlations. A common feature of all these statistics is that they do not make strong assumptions about the distribution of the data. We use Fisher's test, which can report exact probabilities, in the presence of few observations rather than the Chi-squared differences test that calculates approximate p-values. Welch's t-test is similar to the Student's t-test, but it does not assume that the compared variables have equal variance.

We use two-tailed tests for differences when reporting p-values. For standardized effect sizes, we use Cohen's d and follow the behavioral science conventions in (Cohen, 1992) when reporting the magnitude of an effect (see Kampenes, Dybå, Hannay, & Sjøberg, 2007 for software engineering conventions). We use (Chinn, 2000) for effect size conversions

from odds ratio (OR) to d and report arithmetic means.

2.3 The Rasch Measurement Model

A measurement model explicates how measurement is conceptualized. Within psychological testing, the choice of measurement model establishes how abilities, such as intelligence or skills, are related to a person's responses on items (Nunnally & Bernstein, 1994). An *item* is a generic term for any question, essay, task, or other formulated problem presented to an individual to elicit a response. The choice of measurement model dictates how patterns in responses to items should and should not appear. Failure to detect expected patterns and the presence of unwanted patterns may invalidate a researcher's claims to what is measured by a psychological test (Andrich, 1988).

The original Rasch model (1960) was published in 1960 and conceptualizes measurement of abilities according to a probabilistic framework. The model has similarities to conditional logistic regression and is sometimes referred to as a one-parameter Item Response Theory (IRT) model (see e.g., Nunnally & Bernstein, 1994; Ostini & Nering, 2006). The use of IRT models has increased in last half century. Nowadays, IRT models are central to large, multi-national testing frameworks, such as the PISA test (Bond & Fox, 2001), which is used to measure educational achievement of students across approximately 40 OECD countries.

The Rasch model belongs to a class of models that assumes unidimensionality, that is, the investigated ability can be represented by a single numerical value (Andrich, 1988; Nunnally & Bernstein, 1994). Central to the Rasch model is the invariant estimation of abilities and item difficulties (Andrich, 1988). This is consistent with the general test-theory requirements set forth by pioneers in psychology nearly a century ago (see Thurstone, 1928).

The original Rasch model only permits two score categories when a person solves a task: *incorrect* = 0 or *correct* = 1. Therefore, it is called the dichotomous Rasch model. In this model, the probability of a person with skill β to correctly answer a task with difficulty δ can be expressed as

$$Pr = \frac{e^{\beta-\delta}}{1+e^{\beta-\delta}}. \quad (1)$$

The parameters β and δ are represented in log odds (i.e., *logits*). When β equals δ, the probability for a correct response is 0.50. The relative distance between skill and task difficulty follows a logistic (sigmoid) function that is S-shaped.

A generalization of the dichotomous Rasch model, derived by Andrich (1978), allows the use of *more than two* score categories. It is therefore called the *polytomous* Rasch model. Although this model is more complex to express mathematically than the dichotomous model shown above (1), the general principles are the same for both models.

Even though the Rasch model uses discrete score categories, continuous variables such as time have previously been adapted to the scoring structure of the polytomous Rasch model. In (Bergersen, 2011; Bergersen et al., 2011), we explicated the requirements for including programming tasks that vary in both time and quality dimensions simultane-

ously in the polytomous Rasch model. We now use the same model to express the effect of recursive versus iterative treatments for the two debugging tasks conditional on skill.

The Rasch analysis was conducted using the Rumm2020 software package (Andrich, Sheridan, & Luo, 2006). A difference of x logits has uniform implications over the whole scale and is equal to an OR of e^x.

3 Results

Table 2 shows the proportion of correct solutions in the original study and the replicated one. There are three clear differences. First, the professionals in the replicated study clearly have a larger proportion of correct solutions than the students of the original study when comparing the mean of both tasks combined for the two studies (OR = 7.6, $d = 1.12$, $p < 0.001$).

Second, the probability of a correct solution for the Find task was higher in both studies (i.e., it is an *easier* task). For our replication, the difference in mean correctness between Find and Copy is large as well (OR = 4.5, $d = 0.83$, $p = 0.001$).

Third, our prediction that recursion would have a larger proportion of correct responses was disproved on a task-by-task basis compared with the original study. Our study supports the conclusion of the original study only for the recursive Find task: we found a larger proportion of correct solutions (OR = 6.5, $d = 1.03$, $p = 0.106$) with a 95% confidence interval (95CI) for the OR that ranges from 0.72 to 316. However, for the Copy task, the result was in favor of iteration because the odds ratio is less than 1 (OR = 0.94, 95CI = [0.30, 3.0], $d = -0.03$, $p = 1.0$).

In the original study, the mean time required to fix the debugging problem correctly yielded mixed and not significant results that were slightly in favor of recursion. (Mean time was in favor of recursion for the Find task and in favor of iteration for the Copy task.) In this replication, the mean time, which were measured in minutes, were not in favor of any of the treatments (recursive Find = 5.48, SD = 2.48; iterative Find = 5.65, SD = 2.82; recursive Copy = 6.30, SD = 2.58; iterative Copy = 5.95, SD = 2.68). However, the mean does not adequately represent the central tendency of the data in the presence of outliers or when the distribution is skewed.

Figure 1 shows boxplots of the time for correct solutions. As indicated by the whiskers,

Table 2: Proportion of correct solutions for both studies

Task	Treatment	Original study (*n*)		Original study (*n*)	
Find	Recursive	34.1%	(132)	96.1%	(31)
Find	Iterative	38.1%	(134)	81.8%	(33)
Copy	Recursive	29.9%	(134)	63.3%	(30)
Copy	Iterative	17.6%	(131)	64.7%	(34)
Mean of Find	Both	36.1%	(266)	89.1%	(64)
Mean of Copy	Both	23.8%	(265)	64.0%	(64)
Mean of both tasks	Both	30.0%	(531)	76.6%	(128)

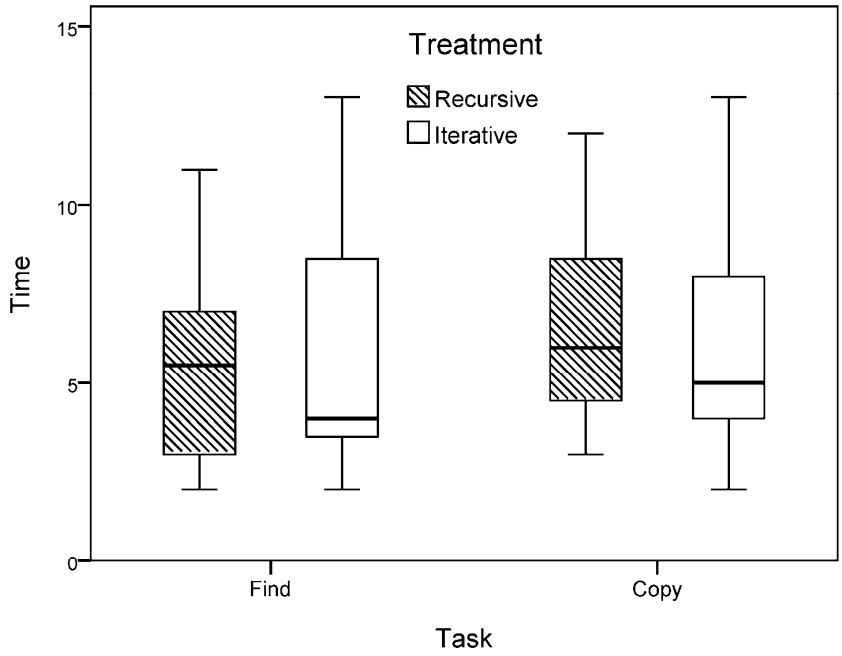


Figure 1: Distribution of time for correct solutions in the replication.

the spread of the data is somewhat wider for both iterative treatments. Further, the median is much closer to the first quartile than the third quartile, which indicates a positively skewed distribution. The original study only reported mean differences for time without referring to the distribution, standard deviation or median values; it is therefore unclear whether the mean is a good representation of the central tendency of their data or not.

Within-subject designs permit pair-wise comparisons that may limit the confounding effect of individual variability (Shadish et al., 2002). In our replication, the recursive-recursive and iterative-iterative designs (Table 1) cannot be used in such an analysis, because the same treatment is used for both tasks. Nevertheless, two-thirds of our subjects ($n = 43$) were given both treatments using a randomized crossover design. There are two relevant outcomes to analyze: those who had a correct recursive and incorrect iterative solution (i.e., *in favor of recursion*) and vice versa (i.e., *in favor of iteration*). Eleven subjects displayed results in favor of recursion and six subjects in favor of iteration. (A total of 26 subjects performed identically across treatments and are thus excluded from this analysis.) A null hypothesis of "equal probability in favor of either treatment" (i.e., in favor of recursion = in favor of iteration = 0.5) can then be tested against the alternative hypothesis of a higher probability in favor of recursion, using a binomial test. However, 11 successes in favor of recursion over 17 trials (11+6) give a p-value of only 0.167. Hence, the null hypothesis could not be falsified. Only weak support in favor of recursion was therefore present.

We have previously shown that the proportion of correct responses is higher for Find than for Copy (Table 2). It is therefore interesting that 15 of the same 17 individuals who

were in favor of one of the treatments were also in favor of the same treatment that they received for the (easiest) Find task. A null hypothesis of "no differences between tasks" using a binominal distribution could be falsified ($p = 0.001$). Hence, there is support for that the effect of differences in the difficulty between tasks appears larger than the effect of treatment across both tasks in an analysis using pair-wise comparisons.

In summary, only for the Find task, our replication supported the overall finding of the original study that recursion is associated with a larger proportion of correct answers. This result is contrary to the findings of the original authors who only found a significant difference in favor of recursion for the Copy task.

4 Results Using Rasch Model Analysis

This section expands upon the previous section by including results for skill differences (Section 2.2). Section 4.1 gives justification as to why the measures of skill should be included in the analysis. Section 4.2 addresses to what extent random assignment to treatment was successful in our replication. Finally, while Sections 4.1 and 4.2 treat skill estimates as a "black box", Section 4.3 shows how the preference for iterative or recursive debugging tasks changes when the results (Section 3) are reanalyzed as a function of skill.

4.1 Justification for Using Skill in the Analysis

In order to include a covariate such as skill in an analysis, the covariate must be correlated with the outcome of the experiment (Shadish et al., 2002). Table 3 shows the correlations between skill and the dual experiment outcomes of correctness and time to correctly debug the two tasks irrespective of treatment. All four correlations were large and significant with p-values below 0.003.

Correlations alone do not illustrate to what extent differences in skill accounts for practical differences in debugging performance on the two tasks. We therefore split the study population into two groups: The more skilled (variable *MoreSkill*) and the less skilled (variable *LessSkill*) groups consist of individuals with skill above and below the mean respectively. For the proportion of correct solutions for the Find task irrespective of treatment, *MoreSkill* had all tasks correct (100%), whereas *LessSkill* had 75.8% correct (OR cannot be computed, 95CI = $[2.0, \infty]$, $p = 0.003$). For the Copy task, *MoreSkill* had a higher mean proportion of correct answers as well (85.7% correct, *LessSkill*: 37.9% correct, OR = 9.4, 95CI = [2.6, 41], $d = 1.24$, $p < 0.001$).

For the time to correctly debug the tasks irrespective of treatment, *LessSkill* spent 73.1% more time than *MoreSkill* on correctly solving the Find task, $t(40) = 5.05$, $p < 0.001$,

Table 3: Dependent variable correlations with skill in the replication

Task	Correctness[a] (N)	Time (N)
Find	0.51 (64)	–0.56 (57)
Copy	0.55 (64)	–0.44 (41)

Note. [a] Point-biserial correlation

$d = 1.38$. Further, this group also spent 36.8% more time on correctly solving the Copy task, $t(21) = 2.32$, $p = 0.030$, $d = 0.80$, than *MoreSkill*. For Find, the mean time for *LessSkill* was 7.21 minutes (SD = 2.59, $n = 25$) and 4.16 minutes (SD = 1.75, $n = 32$) for *MoreSkill*. For Copy, the mean time for *LessSkill* was 7.49 minutes (SD = 2.69, $n = 13$) whereas *MoreSkill* used 5.47 minutes (SD = 2.36, $n = 28$). Moreover, when the results for Find and Copy were combined, *MoreSkill* was faster in correctly debugging as well ($p < 0.001$).

Hence, we can regard measures of skill as a relevant predictor of the debugging performance in the replication that may be used in the further analysis.

4.2 Random Assignment in the Replication

An implicit assumption in randomized designs is that randomization limits systematic effects of unequal groups with respect to factors that can significantly influence the experiment outcome (Shadish et al., 2002; Kampenes et al., 2009). Although this holds true for large samples, when sample size is small it may pose a serious threat to the validity of inferences.

The subjects who were assigned to the recursive Find task had slightly higher mean skills than those who were assigned to the iterative version. The difference in mean skill, as estimated by the Rasch model, was 0.35 logits (OR = 1.42, $d = 0.19$). For the Copy tasks, the effect of randomized assignment to treatment was reversed; the iterative group had slightly higher skills than the recursive group on average ($\Delta = 0.13$ logits, OR = 1.14, $d = 0.07$). We now turn to the more detailed analysis where the effect of recursive and iterative treatments for the two tasks can be analyzed conditional on skill.

4.3 Results

In this section, we use Rasch analysis to expand the previously reported results. Information about each individual's skill is now used to estimate the difficulty of four *treatment pairs*: recursive Find, iterative Find, recursive Copy and iterative Copy. The procedure uses the principles of differential item functioning (DIF) that is commonly is used to investigate whether questions in a psychological test are biased towards subgroups such as non-native speakers of a language or ethnic minorities (see Borsboom, 2006). However, we perform some adaptations to traditional DIF analysis along the lines discussed in (Chang & Chan, 1995).

We will use the term item to denote one of the four treatment-task pairs above. The estimation process uses a conditional maximum likelihood function where residual (unexplained) variance is minimized. Further, unlike the differences in mean skill we reported in Section 4.2, the Rasch model accounts for skill differences on a *person-by-person* level. This implies that even if a group of individuals is more skilled on average, individual response patterns that yield more relevant information in determining the difficulty of tasks are used. Nevertheless, the inequality of treatment groups with respect to skill (as reported in Section 4.2) is now taken into consideration in determining the difficulty estimates for the four items.

All the four items were scored identically using three score categories: *Incorrect* = 0, *correct and slow* = 1, and *correct and fast* = 2. This score structure is a monotonic function of what "high performance" implies and uses an ordinal scale (Bergersen et al., 2011). We (operationally) defined six minutes as the difference between "slow" and "fast" solutions, thereby roughly splitting the observations of Figure 1 in half. Although this procedure degrades time into a dichotomous variable, it allows time and correctness to be co-located on the same scale for more detailed analysis. This, in turn, makes it possible to express the difficulty of all four items as a function of skill.

Each individual's aggregated debugging score over both tasks (i.e., a sum score from 0–4) appeared normally distributed according to the Kolmogorov-Smirnov statistic ($p = 0.058$). Further, this sum score was also well predicted by skill (rho = 0.673, 95CI[1] = [0.484, 0.806], $p < 0.001$).

The Rasch model places item difficulty and a person's ability on the same interval scale (Andrich, 1978). An interval scale implies that additive transformations are permitted but not multiplicative transformations. Further, ratio interpretations are not meaningful because the number zero is not defined (Stevens, 1946).

Figure 2 shows the expected probabilities for the recursive Find item using the three score categories (0–2) above. The mean population skill is transformed to be located at 5 logits and has a standard deviation of 1.3 logits (not shown) on the x-axis. Starting from the left, the figure shows that the incorrect (score = 0) response category has the highest probability for individuals with less skill than about 1.2 logits. Further, the probability of an incorrect solution decreases as skill increases, and the probability of an incorrect response becomes negligible at about 5 logits and above. Between 1.2 and 5.2 logits, the most probable response category is 1 (correct and slow). Above 5.2 logits, the most probable response category is 2 (correct and fast). The sum of the probability for the three score categories always equals 1.0 for any level of skill.

The dotted line that departs upwards from the "score = 1" category shows the cumulative distribution function for this score category. Because a score of 2 also implies that the first threshold has been passed (see, e.g., Andrich, 1988; Bergersen et al., 2011), the probability of *not* achieving at least a correct and slow solution when skill is above about 5 logits is also negligible.

A *threshold* is the location on the logit scale where one response category replaces another one as being the most likely response. This is indicated in Figure 2 by two vertically dotted lines. There, the number of thresholds for each item equals the number of score categories minus 1: The *first threshold* is where score 0 and 1 intersect, and the *second threshold* is where score 1 and 2 intersect.

Figure 3 shows the most likely response category for all four items as a function skill. Although the exact category probability curves for each item is not shown (as in Figure 2), the differences in the location of thresholds provide the needed information about differences in difficulty between the four items. (The two thresholds in Figure 2 can be found for "recursive Find" row in Figure 3.) The standard error of measurement for each threshold is represented by a horizontal bar. Overall, the lower difficulty of the first threshold for both recursive tasks compared with their iterative alternatives supports

[1]Calculated using PASW™18.0 for 2500 bootstrapped samples.

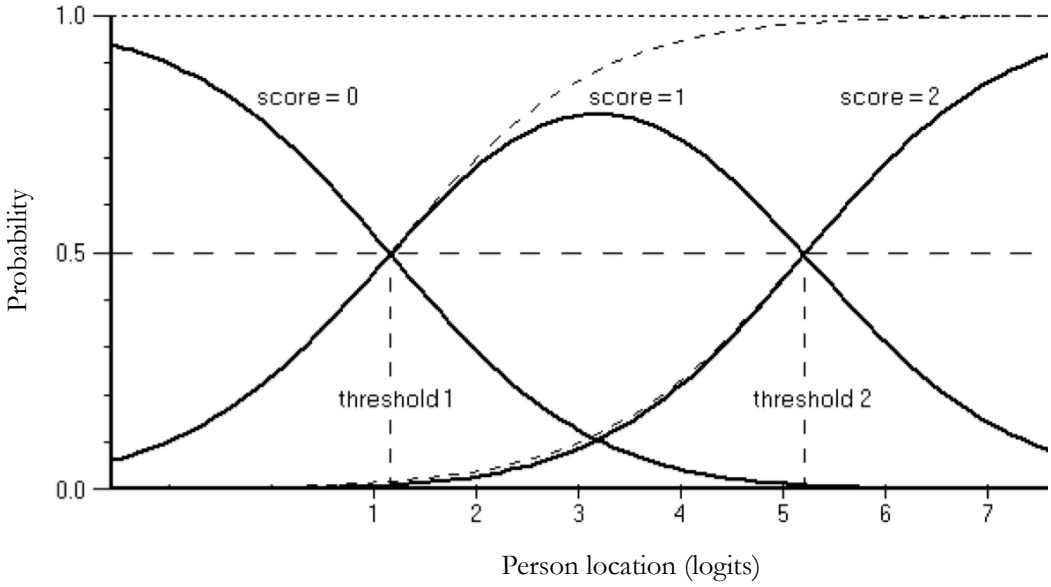


Figure 2: Category probability curves for different scores as a function of skill for the recursive Find task.

the findings of original study: the recursive versions of the two tasks are easier to debug correctly. The effect is much larger for the Find task (Δ 2.1 logits) than for the Copy task (Δ 0.6 logits).

The second threshold represents the difficulty of debugging an item correctly in a "slow" versus "fast" manner. As expected, it is more difficult to achieve a correct and fast solution than a correct and slow solution. The results for the second thresholds were reversed with respect to what the better treatment was for both tasks; the iterative versions were easier to debug correctly and fast than the recursive versions of the tasks. However, inspecting the width of the standard errors in Figure 3 shows that none of the differences in threshold locations are significant; a 95% confidence interval for item thresholds can be obtained by roughly doubling the width of each standard error bar.

Based on the information contained in the threshold map in Figure 3, it is now possible to turn to a concrete example of how "what is the better treatment" varies as a function of skill. We first define three (hypothetical) groups: the *low-skilled* group has a skill of 3

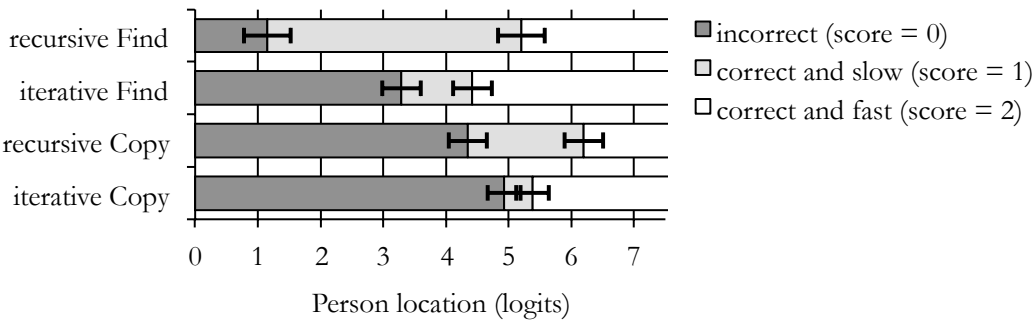


Figure 3: Estimated task difficulty thresholds by the Rasch model (threshold map).

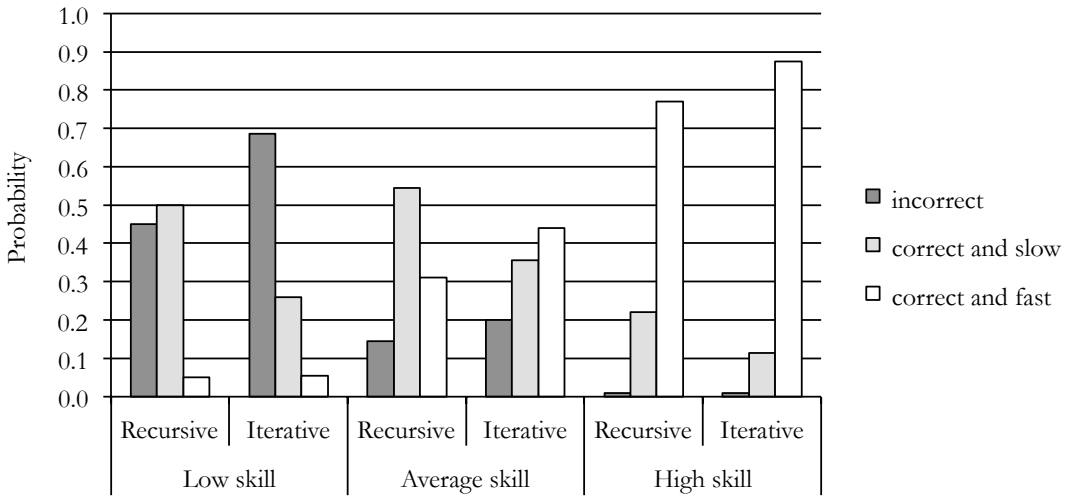


Figure 4: Expected score category probabilities for Find and Copy combined depending on skill and treatment.

logits (at 7th percentile), the *average-skilled* group has the mean skill of the investigated population and the *high-skilled* group has a skill of 7 logits (at 93rd percentile). (The low- and high-skilled groups are about ±1.5 standard deviations below or above the mean skill.)

In Figure 4 we have combined the probabilities for the Find and Copy tasks. For the low-skilled group, the recursive implementations appear best because the probability of incorrectly debugging these tasks (0.45) is lower than for the iterative versions (0.69). At the same time, the difference in probability between the treatments for a correct and fast implementation also appears negligible (0.05 for recursive versus 0.06 for iterative). For the high-skilled group, the iterative versions appear best. The expected probability of a correct and fast implementation for the iterative versions of the tasks combined is 0.88, whereas the corresponding probability for the recursive versions is 0.77. At the same time, the probability for an incorrect solution for both recursion and iteration is only 0.01.

For the average-skilled group, the results are inconclusive, because a choice must be made with respect to preference in a time-quality tradeoff (see, e.g., Fitts & Posner, 1967). For example, the iterative versions have a slightly higher probability of being incorrect (0.20 versus 0.15), whereas the probability of a correct and fast solution is also slightly higher (0.44 versus 0.33). Whether recursion or iteration is the better treatment for this group cannot be decided because there are no negligible differences for any of the score categories.

5 Discussion

This section discusses and contrasts the results from the original and replicated study, first with respect to implication for research and then with respect to implications for practice. We then address limitations of our replicated study and discuss issues for future work.

5.1 Implications for Research

In this study, the largest effect on debugging performance was neither the treatment nor the task complexity; it was the skill of the subjects. Overall, the two recursive implementations were slightly easier to debug correctly (OR = 1.50) than the iterative implementations. The original study had a similar result, although the effect size was smaller (OR = 1.21). By pooling the data from both studies (Table 2), the effect of recursion being easier to debug correctly is marginal (OR = 1.18, 95CI = [0.85, 1.63], $d = 0.09$, $p = 0.34$).

Both studies show that the difference in difficulty between the tasks is larger than the effect of treatment. Combining the studies, the standardized effect size of task difficulty irrespective of treatment is between small and medium (OR = 1.87, 95CI = [1.34, 2.6], $d = 0.35$, $p < 0.001$).

However, in our replication, these effect sizes are dominated by the effect of individual differences in skill: When correctness for Find and Copy was merged and analyzed irrespective of treatment, the more skilled group had 92.9% correct solutions and the less skilled group had 56.9% correct solutions. This difference represents a large standardized effect size (OR = 9.4, 95CI = [2.6, 41], $d = 1.25$, $p < 0.001$) that ranks in the top 25th percentile of 284 software engineering experiments (see Kampenes et al., 2007). Differences in skill must therefore be controlled for in empirical studies of programmers.

Generalization over tasks usually requires that the results be consistent across several tasks. The original study had only two tasks. Consider Segal's law: "a man with a watch knows what time it is. A man with two watches is never sure." Only by having multiple operationalizations is it possible to make more qualified inferences on the extent to which a result can be generalized. When expressing the effect of treatment as a function of skill by using the Rasch model, we obtained results that were consistent across two tasks despite the challenge that a large task difficulty factor presented.

The combined results of three studies now support the conclusion of the authors of the original study. Table 4 shows the overall results for the comprehension, original, and replicated studies, where "+" denotes positive support of the original authors' conclusion with respect to debugging correctness. Yet, their large study still failed to yield results in support of the recursive version of the Find task as being easier to debug. Although the difference of the effect size of recursion versus iteration for the two tasks is relatively small and difficult to detect, the sample size of the original study requires us to conjecture why they did not find that the recursive Find task was easier than the iterative one.

A *practice effect* is when performance increases for each new task in a study when the performance is supposed to be stable in order not to bias the study (Sheil, 1981). Practice effects are common threats to validity in within-subject designs (Shadish et al., 2002) and are known to increase individual variability and thereby decrease the statistical power to detect differences. In the skill instrument, we have previously reported the presence of a small "warm-up" effect for the first three tasks (1–2 hours), but it is not present afterwards (Bergersen & Hannay, 2011). It is therefore tenable that the original study, which only involved two small tasks, is influenced by a similar practice effect (any potential practice effect in our replication is averaged over 19 tasks using randomization).

Table 4: Support for recursion being more easy to debug correctly

Study	# Responses (both tasks)	Debugging phase	Task	
			Find	Copy
Comprehension[a]	275	Identification	+	(–)
Original[b]	531		+	+
		Correction	(–)	+
This replication	128		(+)	(+)

Notes. [a] see (Benander, Benander, & Pu, 1996); [b] see (Benander, Benander, & Sang, 2000); + denote positive support and – denote negative support for the original author's conclusion; ( ) represents non-significant results.

The mean degree of correctness for Find and Copy for the students in the original study and the professionals in the replicated study (Table 2) deserves to be addressed in order to address a potential practice effect. The student's probability of correct solutions for their first (Find) task was 0.53 lower than that for the professionals, but only a difference of 0.40 separated the two populations for the second (Copy) task. This indicates that the students improved their performance on their second task more than did the professionals. Typically, professionals are more skilled than students and therefore learn less from practice (see Fitts & Posner, 1967). Nevertheless, a systematic improvement in performance during a study is problematic, because it implies that the subjects are not well versed in using the technology; hence the results cannot be generalized (Sheil, 1981). An improvement of 0.13 (i.e., $0.53 - 0.40$) in the proportion of correct responses is almost equal in size to the difference in mean difficulty of the two tasks (0.16). Hence, it may appear that that a practice effect, in addition to the effect of skill and task difficulty, may also be larger than the effect of treatment in the original study.

5.2 Implications for Practice

We found weak results in favor of iteration being easier to debug fast and correctly than recursion. Although this result was not significant, it was consistent over both tasks.

It is self-evidently true that a technology is better when it is easier *and* faster to use than when it is not. However, what if a "faster" technology comes at the price of added complexity, which makes the technology harder to use properly? Then the faster technology would require more training to be used successfully. Without training, the faster and more complex technology would be associated with a higher proportion of incorrect uses, thereby making the faster technology appear worse than the existing alternative that is slow but easy to use correctly already.

There are several examples of occasions when a faster technology is more difficult to use. For example, a bicycle is a faster means of transportation than walking, but one must know how to ride it. Similarly, a typewriter is easier to use than a computer, but the computer is faster for text editing when used correctly. More complex technologies frequently require more training than less complex technologies; at the same time, more complex technologies are adopted because they add to productivity.

Fundamental to our reported results that the potential advantage of different debug-

ging implementations may depend on skill levels are two basic assumptions. First, a correct solution is better than an incorrect solution. Second, a fast solution is better than a slow solution *if* both solutions are correct (Carroll, 1993). As previously shown in Figure 4, when the probabilities for an incorrect solution is high, it is normal to take steps to improve the degree of correctness before less time becomes an important factor. This seems to be the case for our (hypothetically defined) low-skilled subjects. An opposite situation was present for the high-skilled subjects: because the proportion of incorrect and correct answers was negligible whereas the difficulty of a fast and correct solution was lower for the iterative versions, the preference for what treatment was better was reversed. The tradeoff between quality and time is certainly present when practitioners evaluate the benefits of new software engineering technologies.

5.3 Limitations

A limitation of this replication is low statistical power. Although we had sufficient power to detect large and systematic differences in the skills of the subjects and the differences in the difficulty of the tasks, our results with respect to the treatments were not significant. Further, even though the study magnitude of our replication is large according to the conventions of (Sjøberg et al., 2005) (the professionals spent more than 1000 hours combined), the Rasch model can be data intensive when the purpose is to characterize individual differences (see Linacre, 1994). Because our research question considers group differences (rather than individual differences), fewer than the recommended number of subjects for using the Rasch model is therefore acceptable. We also regard the limitation of only having two debugging tasks to generalize from as a greater concern than the statistical power at present.

In this replication, there was only one response for the "incorrect" score category for the recursive Find task. This implies that the standard error associated with the first threshold for this task is not adequately represented. Ideally, all response categories should be well populated to obtain accurate item thresholds in the Rasch model. For the two tasks investigated here, a new sample of less skilled subjects is needed to obtain lower standard errors of measurement in the item difficulty thresholds.

5.4 Future Work

To measure the skill of the subjects in this study we used a specifically tailored research prototype. We are now working on making the skill instrument industry strength. New or replicated experiments may then be administered as part of ongoing assessments of professionals and students, something that facilitates the use of more statistically powerful experimental designs (e.g., matching or pairing, see Shadish et al., 2002). Such designs are particularly relevant for studies where few subjects are available, within-subject designs are not feasible, or where random assignment to treatment is not possible. We will also conduct more studies where skill is taken into account when investigating the effect of a technology. We welcome future collaborations.

6 Conclusion

An implicit assumption in many research studies in software engineering is that the benefit of a new technology or method is invariant of skill levels. The study reported in this paper illustrates why such an assumption is problematic. Using a measurement model where the effect of recursive versus iterative implementations of two small debugging tasks was expressed as a function of skill, we provided additional evidence that "what is the better" of two competing technologies requires the additional qualifier "for whom?"

We found that for the low-skilled subjects, the results were in favor of recursive implementations, which supports the original study. An opposite result was found for the high-skilled subjects; the iterative versions were debugged faster while the difference in the proportion of correct answers compared to the recursive version was negligible. Hence, the benefit of debugging the iterative versions (less difficult to debug fast but more difficult to debug correctly), is based on an important principle: The probability of incorrectly using both debugging alternatives must be low and negligible before the faster technology can be assumed to be better.

This study does not stand in isolation; previous large-scale experiments have reported similar interaction effects between skill levels and the technology or method being investigated. Still, there is often a gap between researcher expectations and empirical results because one fails to acknowledge that potentially more powerful technologies may be more complex to use, or may require new skills in order to use correctly. The community must raise its awareness of how skill levels, which in this study was much larger than the difference between treatments, affect the claimed benefits of alternatives being evaluated. Consequently, we need better ways to measure relevant skills with respect to the product or process being investigated.

Acknowledgment

This research was supported by the Research Council of Norway through the FORNY program, and by Simula Research Laboratory. We thank Steinar Haugen and Gunnar Carelius for technical assistance, and Magne Jørgensen for reading through an earlier version of this manuscript. We also thank Christian Brinch for pointing out that our pair-wise comparison can be analyzed using a binominal test.

References

Anda, B. C. D., Sjøberg, D. I. K., & Mockus, A. (2009). Variability and reproducibility in software engineering: A study of four companies that developed the same system. *IEEE Transactions on Software Engineering*, *35*(3), 407–429.

Anderson, J. R. (1987). Skill acquisition: Compilation of weak-method problem solutions. *Psychological Review*, *94*(2), 192–210.

Andrich, D. (1978). A rating formulation for ordered response categories. *Psychometrika*, *43*(4), 561–573.

Andrich, D. (1988). *Rasch models for measurement* (No. 68). Sage Publications.

Andrich, D., Sheridan, B., & Luo, G. (2006). Rumm2020 [Computer software manual]. Perth.

Arisholm, E., Gallis, H., Dybå, T., & Sjøberg, D. I. K. (2007). Evaluating pair programming with respect to system complexity and programmer expertise. *IEEE Transactions on Software Engineering*, *33*(2), 65–86.

Arisholm, E., & Sjøberg, D. I. K. (2004). Evaluating the effect of a delegated versus centralized control style on the maintainability of object-oriented software. *IEEE Transactions on Software Engineering*, *30*(8), 521–534.

Arisholm, E., Sjøberg, D. I. K., Carelius, G. J., & Lindsjørn, Y. (2002). A web-based support environment for software engineering experiments. *Nordic Journal of Computing*, *9*(3), 231–247.

Basili, V. R., Shull, F., & Lanubile, F. (1999). Building knowledge through families of experiments. *IEEE Transactions on Software Engineering*, *25*(4), 456–473.

Benander, A. C., Benander, B. A., & Pu, H. (1996). Recursion vs. iteration: an empirical study of comprehension. *Journal of Systems and Software*, *32*(1), 73–82.

Benander, A. C., Benander, B. A., & Sang, J. (2000). An empirical analysis of debugging performance—differences between iterative and recursive constructs. *Journal of Systems and Software*, *54*(1), 17–28.

Bergersen, G. R. (2011). Combining time and correctness in the scoring of performance on items. In T. Nielsen, S. Kreiner, & J. Brodersen (Eds.), *Probabilistic models for measurement in education, psychology, social science and health* (pp. 43–44). Copenhagen.

Bergersen, G. R., & Gustafsson, J.-E. (2011). Programming skill, knowledge and working memory capacity among professional software developers from an investment theory perspective. *Journal of Individual Differences*, *32*(4), 201–209.

Bergersen, G. R., & Hannay, J. E. (2011). Detecting learning and fatigue effects by inspection of person-item residuals. In T. Nielsen, S. Kreiner, & J. Brodersen (Eds.), *Probabilistic models for measurement in education, psychology, social science and health* (pp. 56–57).

Bergersen, G. R., Hannay, J. E., Sjøberg, D. I. K., Dybå, T., & Karahasanović, A. (2011). Inferring skill from tests of programming performance: Combining time and quality. In *Proceedings of the 5th International Symposium on Empirical Software Engineering and Measurement* (pp. 305–314).

Bond, T. G., & Fox, C. M. (2001). *Applying the Rasch model: Fundamental measurement in the human sciences*. Mahwah, NJ: Erlbaum.

Borsboom, D. (2006). When does measurement invariance matter? *Medical Care*, *44*, S176–S181.

Brooks, A., Daly, J., Miller, J., Roper, M., & Wood, M. (1996). *Replication of experimental results in software engineering* (Tech. Rep. No. EFoCS-17-95). Glasgow: University of Strathclyde.

Brooks, F. P. (1995). *The mythical man-month: Essays on software engineering* (Anniversary ed.). Reading, MA: Addison-Wesley.

Brooks, R. E. (1980). Studying programmer behavior experimentally: the problems of proper methodology. *Communications of the ACM*, *23*(4), 207–213.

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

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

Chang, W.-C., & Chan, C. (1995). Rasch analysis for outcome measures: Some methodological considerations. *Archives of Physical Medicine and Rehabilitation*, *76*, 934–939.

Chinn, S. (2000). A simple method for converting an odds ratio to effect size for use in meta-analysis. *Statistics in Medicine*, *19*(22), 3127–3131.

Cohen, J. (1992). A power primer. *Psychological Bulletin*, *112*(1), 155–159.

Fitts, P. M., & Posner, M. I. (1967). *Human performance.* Belmont, CA: Brooks/Cole.

Glass, R. L. (2001). Frequently forgotten fundamental facts about software engineering. *IEEE Software*, 110–112.

Kampenes, V. B., Dybå, T., Hannay, J. E., & Sjøberg, D. I. K. (2007). A systematic review of effect size in software engineering experiments. *Information and Software Technology*, *49*(11), 1073–1086.

Kampenes, V. B., Dybå, T., Hannay, J. E., & Sjøberg, D. I. K. (2009). A systematic review of quasi-experiments in software engineering. *Information and Software Technology*, *51*(1), 71–82.

Krein, J. J., Knutson, C. D., Prechelt, L., & Juristo, N. (2012). Report from the 2nd international workshop on replication in empirical software engineering research (RESER 2011). *SIGSOFT Software Engineering Notes*, *37*, 27–30.

Linacre, J. M. (1994). Sample size and item calibration stability. *Rasch Measurement Transactions*, *7*(4), 328.

Maxwell, S. E. (1993). Covariate imbalance and conditional size: Dependence on model-based adjustments. *Statistics in Medicine*, *12*, 101–109.

Nunnally, J. C., & Bernstein, I. H. (1994). *Psychometric theory* (Third ed.). New York: McGraw-Hill.

Ostini, R., & Nering, M. L. (2006). *Polytomous item response theory models.* CA: Sage Publications.

Prechelt, L. (1999). *The 28:1 Grant/Sackman legend is misleading, or: How large is interpersonal variation really?* (Tech. Rep. No. 18). Karlsruhe, Germany: University of Karlsruhe.

Prechelt, L. (2011). Plat_Forms: A web development platform comparison by an exploratory experiment searching for emergent platform properties. *IEEE Transactions on Software Engineering*, *37*(1), 95–108.

R Development Core Team. (2008). *R: A language and environment for statistical computing [computer software v. 2.13.2].* http://www.R-project.org.

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

Schmidt, F. L., & Hunter, J. E. (1998). The validity and utility of selection methods in personnel psychology: Practical and theoretical implications of 85 years of research findings. *Psychological Bulletin*, *124*(2), 262–274.

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

Sheil, B. A. (1981). The psychological study of programming. *ACM Computing Surveys*, *13*(1), 101–120.

Shute, V. J. (1991). Who is likely to acquire programming skills? *Journal of Educational Computing Research*, *7*(1), 1–24.

Sjøberg, D. I. K., Anda, B., Arisholm, E., Dybå, T., Jørgensen, M., Karahasanović, A., ... Vokáč, M. (2002). Conducting realistic experiments in software engineering. In *Proceedings of the International Symposium Empirical Software Engineering* (pp. 17–26).

Sjøberg, D. I. K., Hannay, J. E., Hansen, O., Kampenes, V. B., Karahasanovic, A., Liborg, N.-K., & Rekdal, A. C. (2005). A survey of controlled experiments in software engineering. *IEEE Transactions on Software Engineering*, *31*(9), 733–753.

Stevens, S. S. (1946). On the theory of scales of measurement. *Science*, *103*(2684), 677–680.

Thurstone, L. L. (1928). Attitudes can be measured. *American Journal of Sociology*, *4*, 529–554.

Woltz, D. J. (1988). An investigation of the role of working memory in procedural skill acquisition. *Journal of Experimental Psychology: General*, *117*(3), 319–331.