Empirical evaluation of the effects of experience on code quality and programmer productivity: an exploratory study

Oscar Dieste[1] · Alejandrina M. Aranda[1] · Fernando Uyaguari[1] · Burak Turhan[2] · Ayse Tosun[3] · Davide Fucci[2] · Markku Oivo[2] · Natalia Juristo[1,2]

Abstract
Context There is a widespread belief in both SE and other branches of science that experience helps professionals to improve their performance. However, cases have been reported where experience not only does not have a positive influence but sometimes even degrades the performance of professionals.
Aim Determine whether years of experience influence programmer performance.
Method We have analysed 10 quasi-experiments executed both in academia with graduate and postgraduate students and in industry with professionals. The experimental task was to apply ITLD on two experimental problems and then measure external code quality and programmer productivity.
Results Programming experience gained in industry does not appear to have any effect whatsoever on quality and productivity. Overall programming experience gained in academia does tend to have a positive influence on programmer performance. These two findings may be related to the fact that, as opposed to deliberate practice, routine practice does not appear to lead to improved performance. Experience in the use of productivity tools, such as testing frameworks and IDE also has positive effects.
Conclusion Years of experience are a poor predictor of programmer performance. Academic background and specialized knowledge of task-related aspects appear to be rather good predictors.

✉ Oscar Dieste
odieste@fi.upm.es

Alejandrina M. Aranda
alearanda@gmail.com

Fernando Uyaguari
fuyaguari01@gmail.com

Burak Turhan
Burak.Turhan@oulu.fi

Ayse Tosun
tosunmisirli@itu.edu.tr

Davide Fucci
Davide.Fucci@oulu.fi

Markku Oivo
Markku.Oivo@oulu.fi

Natalia Juristo
Natalia.Juristo@oulu.fi

[1] Escuela Técnica Superior de Ingenieros en Informática, Universidad Politécnica de Madrid, Campus de Montegancedo, 28660 Boadilla del Monte, Spain

[2] Department of Information Processing Science, University of Oulu, P. O. Box 3000, 90014 Oulu, Finland

[3] Faculty of Computer & Informatics, Istanbul Technical University, 34469 MaslakIstanbul, Turkey

1 Introduction

The older you are, the wiser you get; An old ox makes a straight furrow; They who live longest will see most: the passage of time is, proverbially, one and perhaps the major factor facilitating learning. This factor is none other than *experience.*

Things are not very different in software engineering (SE) either. Some people within an organization know more or better, and their participation in the project can be vital to its success (e.g., Curtis et al. 1988). These truisms are backed up by a large number of papers in a range of SE areas, e.g., requirements (Marakas and Elam 1998), design (Sonnentag 1998), usability (MacDorman et al. 2011) or testing (Chmiel and Loui 2004), where it is generally agreed that experience makes the difference with respect to practitioner performance.

There are two different definitions of experience (Merriam-Webster 2015): (1) skill or knowledge that you get by doing something, and (2) the length of time that you have spent doing something (such as a particular job). The two definitions mirror the fact that experience is a theoretical construct: the substance of experience (skills, knowledge) cannot be directly observed, and its existence has to be estimated, where the length of time that a subject has been performing a particular task is the most obvious and easiest-to-measure operationalization. Accordingly, it is common practice to divide subjects into two groups: (1) *experts*, whose characteristic is that they have been working in an area for quite a long time, typically years, and (2) *novices* who not been working in the field for very long.

Focusing on SE, programming, which is the area addressed in this paper, is the field where most evidence for the beneficial effects of experience has been found. To cite just a few examples, expert programmers are quicker at identifying valid sentences in a programming language (Wiedenbeck 1985), more accurately remember meaningful code snippets (McKeithen et al. 1981) or have more sophisticated reasoning strategies than novices (Jeffries et al. 1981). These results match the findings for other areas of SE (e.g., cited above), and other fields outside SE, e.g., physics (Larkin et al. 1980). Until quite recently at least, it looked as if achieving expert performance was the inevitable result of a length of service from around 10 years in an area (Ericsson 2006a).

Later research into experience has tinged the above picture. The key difference between the previous and present conception of experience is the *intensity of practice.* Activity execution does not in itself appear to lead to improvements in a subject's performance; improvements come when a *deliberate effort* is made in order to improve performance (Ericsson 2006b). In fact, performance has even been found to drop as experience increases (Ericsson 2006a). This should not come as a surprise. Surely everyone can think of someone that they know who has a lot of experience but is a poor performer. There are some (not very many) SE studies that conclude that there no differences of performance between experts and novices, e.g., (Agarwal and Tanniru 1991). Some of these studies also focus on programming (Adelson 1984).

There is therefore a lot of uncertainty surrounding whether experience is associated with better performance. As regards programming, this uncertainty is especially worrying because: (1) programming, together with testing, are quantitatively the most important activities in the software development process, and (2) experience is one of the key variables used by employers to hire programmers. The aim of this paper is to determine whether expert programmers exhibit better performance than novice programmers. To do this, we have conducted a series of quasi-experiments analysing the quality of the generated code by programmers and programmer productivity depending on their years of experience. We collected data at four companies and three universities from a total of 115 programmers with a range of experiences, averaging from 0 to 10 years. A key issue is the inclusion of professional programmers currently working in industry, as many earlier studies were conducted without access to real programmers (Votta 1994).

Our results suggest that: (1) experience gained in industry is not related to better quality or higher productivity, (2) secondary issues, like familiarity with the unit testing framework or integrated development environments (IDE), appear to have quite a positive effect on quality and productivity, and (3) academic learning, which could be considered as an instance of deliberate practice, does influence quality and productivity as opposed to on-the-job learning.

The conclusion from our findings is that years of experience are a poor predictor of programmer performance. In turn, academic background (probably also formal training courses in industry) and knowledge of specialized task-related aspects (e.g., the IDE in our case) are good predictors.

This paper is structured as follows. Section 2 briefly describes research into the effects of experience, focusing especially on programming. Section 3 describes the family of experiments. Section 4 describes the working hypotheses and working methodology. Section 4 describes the quasi-experiments, characteristics of the collected data, and the choice of the best-suited statistical analysis method. Section 5 reports the results of the linear analysis, whereas Section 6 reports the nonlinear analysis; both are discussed in Section 7. The paper ends with a discussion of the validity threats and conclusions in Section 8 and 9, respectively.

2 Background

The study of experience goes way back. The original aim was to determine which factors caused expert subjects to perform better than novices. Studies by (De Groot 1978) revealed that experts had two key characteristics in common: an in-depth knowledge of their field of expertise and a long length of service in the area. Chase and Simon (1973) formalized experience as a process by means of which, over time, experts acquired knowledge that they stored as complex mental patterns and that they used to quickly and effectively solve problems in their area of expertise. Experience had nothing to do with natural talent, such as intelligence, and was very specialized, that is, it was not transferable from one area to another (Colvin 2008). Related literature reports that it takes around 10 years or 10,000 working hours to acquire a substantial amount of patterns, although this is by no means a fixed number and depends on the area and type of instruction received (Ericsson et al. 1993). For quite some time, therefore, experience was assumed to be a natural consequence of the passage of time (Ericsson 2006a). From this viewpoint, experience could be likened to a measure of time, e.g., the above 10 years of service.

SE has also studied experience ever since the early days of the discipline. The focus in the 1980s was on programming and low-level design (Curtis 1984). Since then, however, experience has been studied in almost all areas of SE: requirements (Marakas and Elam 1998), design (Sonnentag 1998), usability (MacDorman et al. 2011), testing (Chmiel and Loui 2004), etc.

A weakness of the study of experience in SE is that there are hardly any synthesis papers. Curtis (1984) conducted a broad literature review, which is, however, completely out of date today. Mayer (1997) reviewed 33 studies on the effect of experience on programing published prior to 1997 but, since then, a number of similar studies have been published e.g., (Lui and Chan 2006). Siegmund et al. (2014) conducted a review on how to measure programming experience, but without any reference to expert-novice behaviour.

As opposed to an exhaustive state of the art, which is beyond the scope of this paper, Table 1 shows a summary of the existing studies that address the effects of experience in programming. Programming is a rather complex area, in which a diversity of notations, languages, design approaches, programming techniques, etc. have been investigated. In Table 1, we have included only studies that explore programming abilities. For instance, (Burkhardt et al. 1997) examine the mental representations of objects (in a program). This study seems to be exploring a design but not a programming aspect, and therefore has not been included in Table 1. In the same vein (Ricca et al. 2007), investigates the impact of annotations in UML diagrams, depending on the subjects' experience. Again, this study has been excluded because it addresses a modelling but not a programming feature.

The studies in Table 1 have the following characteristics:

- *Research methodology:* The vast majority of studies are experiments. In most cases, they compare two groups (novices vs. experts). Studies with only one group of subjects, characterized by their experience years, are also common, e.g., (Sheppard et al. 1979). These later studies, with the exception of (Askar and Davenport 2009), are quasi-experiments.
- *Characterization of novices and experts:* All studies use novices with very little, e.g., (Wiedenbeck 1985) or no experience, e.g., (McKeithen et al. 1981). On what regards expert characterization, the situation is far from uniform. Some studies use experts with limited experience, e.g., graduate students (Weiser and Shertz 1984; Ye and Salvendy 1994). In other cases, experts may have quite a lot of experience, e.g., (Burkhardt et al. 2002) use professional programmers with experience in object-oriented design with C++.
- *Response variables and measurement procedures:* Programmer performance is typically measured indirectly, e.g., ability to identify valid programming language sentences (Wiedenbeck 1985), ability to remember meaningful code snippets (McKeithen et al. 1981), etc. Subjective measures, e.g., self-efficacy scores (Askar and Davenport 2009) have also been used. Direct measurements e.g., effort (Arisholm et al. 2007) are uncommon.

Table 1 Related empirical studies

Ref	Study	#Subjects	Independent Variable	Dependent Variable	Metric	Experimental task	Results
(Wiedenbeck 1985)	Experiment	10 Experts 10 Novices	**Experts**: 11,000 h. programming experience. **Novices**: 200 h. programming experience.	Speed and accuracy for generating syntactically programming sentences.		Recognition of syntactic errors and understanding of the structure and function of simple stereotyped code segments	The experts were approximately 25 % faster and had 40 % lower error rates than the novices.
(Ye and Salvendy 1994))	Quasi-Experiment	10 Experts 10 Novices	**Experts**: graduate students in computer science. **Novices**: undergraduates with one course in C programming.	Ratings	0 to 6 Likert scale	Rate the relatedness of 23 concepts in C computer programming.	The experts beat novices on defining relationships between terms and categories creation.
(McKeithen et al. 1981)	Experiment	6 Experts 23 Intermediates 4 Novices	**Experts**: 400 h. in ALGOL W and 2000 h. of general programming. **Novices**: 0 h.	Recall	Number of lines written verbatim in their proper relative order.	Participants viewed a 31-line ALGOL program presented in either normal or scrambled order for 2 min, and then were given 3 min to recall the program.	For the normal programs, experts recalled approximately three times as many lines as the novices. For the scrambled programs, the expert-novice differences were largely erased.
(Arisholm et al. 2007)	Quasi-Experiment	Junior Intermediate Senior	**Junior**: Ind.: 0-23 years., Pair: 0-25 yr. **Intermediate**: Ind.: 0-26 years., Pair: 0-24 yr. **Senior professional Java consultants**: Ind.: 0-27 yr., Pair: 1-30 yr.	Duration	Elapsed time taken to perform a set of change tasks.	Perform maintenance tasks on Java code.	Differences between pairs and individuals do not depend on programmer expertise.
				Effort	Total number of programmer hours taken to develop a correct program.		
				Correctness	Whether or not the final, maintained program possessed the required functionality.		
(Askar and Davenport 2009)	Survey	200 first year students from the computer, electronics, and	Experience years	Self-efficacy scores.	1 to 7 Likert scale	Answer a questionnaire regarding self-efficacy in JAVA programming.	The number of years of experience a student had with computers had a significant linear

Table 1 (continued)

Ref	Study	#Subjects	Independent Variable	Dependent Variable	Metric	Experimental task	Results
		industrial engineering departments. 20 first year science students, all of whom were enrolled in an introductory Java programming course. 106 second, third and fourth year computer engineering students.					contribution to their self-efficacy scores. As the computer experience increases, there is a tendency to gain self-efficacy in programming.
(Adelson 1981)	Experiment	5 Experts 5 Novices	**Novices:** undergraduates who had just completed an introductory course in computer programming in PPL. **Experts:** teaching fellows for the same course.	Recall	The number of items recalled.	Recalling 16 lines of PPL code.	The experts recalled more than the novices, and had larger recall chunks.
				Chunk size	Number of items in a burst of recall.		
(Sheppard et al. 1979)	Quasi-experiment	Professional programmers	Programming years	Comprehension	Percent of statements correctly recalled.	The participants were allowed 25 min to study several programs, during which time they could make notes or draw flowcharts. At the end of study period, the program and all scrap paper were collected, and each participant was given 20 min to reconstructs functionally equivalent code from memory on blank sheet of paper.	The number of years of programming experience did not correlate with performance.
			Programming years	Modification	Percent correct time (to complete the modification).		The number of years of programming experience did not

Table 1 (continued)

Ref	Study	#Subjects	Independent Variable	Dependent Variable	Metric	Experimental task	Results
							correlate with performance.
(Soloway et al. 1983)	Experiment	Students	**Novices** had been taught about and had experience with the while loop and the other two looping constructs; this occurred three-quarters of the way through the semester. **Intermediates** were students currently two-thirds through a second course in programming (e.g., either a data structures course using Pascal or an assembly language course). The **advanced** group were juniors and seniors in systems programming and programming methodologies.	Accuracy	*Not defined in the manuscript*	Students were given a two-part test. In the first part, they were asked to write a plan that would solve the stated problem. In the the second part, half students were asked to write a Pascal program that solved the problem, while the other half were asked to solve the problem using Pascal L.	Accuracy improves from 19 % for the novice group to 49 % for the intermediate group to 83 % for the advanced group.
(Crosby et al. 2002)	Experiment	45 Programmers	**Novice:** CS students. **Intermediate**: CS Students at the junior or senior undergraduate level. **Advanced:** graduate CS students and CS faculty.	The programmers were then asked to rank (on a 6 point scale) how likely it was that a particular line of code came from a binary search program.	0 to 6 Likert scale	Rate lines of code as to how indicative they were of a program function.	Novices discriminate very little between program areas. More experienced programmers tend to concentrate on the important areas of a program.
	Experiment	15 intermediate programmers 15 advanced programmers	**Intermediate**: CS Students at the junior or senior undergraduate level.	Correctness	The response was classified as correct if the programmer was able to identify the origin	Determine which program a line of code was most likely to represent.	The advanced group completed the task in less time.

Table 1 (continued)

Ref	Study	#Subjects	Independent Variable	Dependent Variable	Metric	Experimental task	Results
			Advanced: graduate CS students and CS faculty.	Response time	(depth first search, shell sort or binary search) of a line of code.		The intermediate and more advanced programmers displayed the same ability in terms of percentage of correct responses. The more advanced programmers were able to accomplish this in significantly less time, regardless of the line of code.
	Experiment	9 novice programmers 10 advanced programmers	**Novice:** students with one semester of programming experience. **Advanced:** CS faculty members and students from advanced undergraduate and graduate classes.	Fixations (attention that a programmer pays to a piece of code detected by an eye tracking device).	Number of fixations	This experiment used eye tracking to precisely determine not only what programmers viewed but also how long they fixated on each type of program statement.	The average fixation duration was statistically significant for experience.
				Fixations times	Fixation times		
(Soloway and Ehrlich 1984)	Experiment	94 novice programmers 45 advanced programmers	**Novice:** programmers were students at the end of a first course in Pascal programming. **Advanced:** programmers who had completed at least 3 programming courses, and most were either computer science majors or first year graduate students in computer science; all had extensive experience with Pascal.	Accuracy of response	A correct response was one that completed the intended plan.	One line of code was taken out from the program and replaced with a blank line. The subjects filled the blank line with the piece of code that, in their opinion, best complete the program.	The experts performed better than novices.
				Time			

Table 1 (continued)

Ref	Study	#Subjects	Independent Variable	Dependent Variable	Metric	Experimental task	Results
(Adelson 1984)	Experiment	18 Novices 18 Experts	**Novice:** undergraduates who had completed an introductory computer programming course. **Expert:** fellows of the same course.	Type of program representation.	Abstract or concrete representation.	Each participant was given a concrete or abstract flowchart followed by the program and by concrete or abstract comprehension questions. Abstract questions focused on the underlying plan of the program. Concrete questions focused on surface details of how the program earned out the plan.	Experts performed better on abstract questions, whereas novices performed better on concrete questions.
(Burkhardt et al. 2002)	Experiment	21 Novices 30 Experts	**Novice:** advanced students. **Expert:** professional programmers experienced in OO design with C++.	Comprehension	Correctness of responses	During phase 1, participants were allowed to study the program, run it (an executable was provided), and take notes. After this initial study phase, participants answered a questionnaire.	Experts are better than novices at constructing correct mental representations of the program.
(Weiser and Shertz 1984)	Experiment	6 Novices 9 Experts	**Novices**: students of second and third semester **Experts**: graduate students	Programming problem representation. Sorting time	A set of categories proposed by experimental subjects after sorting task.	Each participant was given specifications for 27 different programming problems, and was asked to group together problems whose method of solution would be similar.	Novices consider the surface features of the problem but represent problems inconsistently. Experts consider the deep features and tend to agree with one another.
(Müller and Höfer 2007)	Experiment	7 TDD Experts 9 TDD Novices	**Experts.** Professional programmers with industrial TDD experience	Conformance to TDD. Changes to lines of code.	TDD changes plus refactorings, over all changes The number of changed lines of code (CLOC)	Performing a coding task in TDD.	The experts achieve a higher conformance to the rules of the test-driven development process than the nov-ices.

Table 1 (continued)

Ref	Study	#Subjects	Independent Variable	Dependent Variable	Metric	Experimental task	Results
			Novices. Students who finished an XP lab course		during the whole implementation process		The number of changed lines of code during the whole implementation process is smaller for the experts than for the novices.
				Speed for application code and test code	Changed lines of code per hour		The experts are faster in changing application code and test code.
				Duration of Implementation	Duration of the implementation until the first acceptance test		The experts are significantly faster than the novices.
				Quality of Test	The quality of the developed tests in terms of statement and block coverage		The tests developed by the experts achieve a higher coverage on the application code than the tests of the novices
(Muller and Padberg 2004)	Quasi-experiment	38 computer science students with JAVA programming experience	Programming years, both at the individual and pair levels	Implementation time		Performing a coding task in TDD.	Individual experience levels and the pair performance are uncorrelated. There is no correlation between the experience level and the implementation time. Java programming experience (measured either in years or lines of code) and implementation time are uncorrelated. There is no correlation between these individual experience levels and the implementation time.

- *Experimental tasks:* Most of the studies have many points in common with classical experiments on expert behaviour. There are plenty of recognition, matching and recall tasks. There are relatively few studies where subjects are called upon to generate code, e.g., (Muller and Padberg 2004). Experimental objects are ad-hoc and especially prepared for each study. With regard to complexity, the objects are generally simple. Studies where actual coding is performed, e.g., (Müller and Höfer 2007) use general problem domains where specialized (domain-specific) knowledge is not required.

Save very rare exceptions, Chase and Simon's theory of experience (Chase and Simon 1973) has generally been repeatedly confirmed. Experts identify or remember more programming language sentences than novices (McKeithen et al. 1981), consider deeper program features (Weiser and Shertz 1984) or are faster than novices (Müller and Höfer 2007). Experts do not always outperform novices, e.g., (McKeithen et al. 1981) but this can usually be explained by the non-transferability of experience, i.e., in such cases, experts are working outside their area of expertise, where their strategies are not applicable and, therefore, they perform similarly to novices.

There have been reports in the literature of cases where experience does not always lead to better performance. McDaniel et al. (1988) reported low correlations between experience and performance. Camerer and Johnson (1997) conclude that subjects with experience make decisions or predictions that are no better or even worse than those made by inexperienced subjects. There are studies with similar outcomes in the area of programming (Muller and Padberg 2004; Sheppard et al. 1979), as well as in other areas of SE, e.g., (Marakas and Elam 1998; Sonnentag 1995). This apparent contradiction can be explained if a distinction is made between experience and *expertise*. In order to achieve the performance of an expert, subjects need to complete a period of intensive practice, with the deliberate intention of improving performance (i.e., achieving expertise). The mere practice of an activity (i.e., the years of experience) may improve performance but not to the point of it being equal to that of people who are generally recognized as experts in an area (Ericsson and Charness 1994).

Finally, it is noteworthy that experience has mainly been studied indirectly. The typical study presents some task(s) to expert and novices subjects, and some facet of the problem solving process (e.g., the top-down or bottom-up programming strategy) is observed. Later, on the basis of Chase and Simon's theory of experience (Chase and Simon 1973) a given strategy (e.g., top-down) is associated to expert behaviour. It is assumed that such strategy will lead to better programs and subjects are categorized according to it. However, expert behaviour does not equate to expert performance. Existing studies are missing direct measures of programmer performance, e.g., whether expert programmers are more productive or generate programs of better quality than novices. In fact, one of the two existing studies reporting negative results (Muller and Padberg 2004) uses direct measures. Recent research, e.g., (Ericsson 2006b) emphasizes the need of explicitly measuring expert performance, instead of relying on (apparent) expert behaviour. On this ground, this paper addresses the following research question:

RQ: *Is the performance (measured directly) of expert programmers (i.e., with longer periods of service) superior than that of novice programmers?*

3 Family of Experiments

3.1 Conducted Quasi-Experiments

We conducted 10 quasi-experiments, six of which were run in industry and four in academia. All the quasi-experiments were conducted as part of the *Empirical Software Engineering Industry Lab (ESEIL)* project, led by N. Juristo and funded by TEKES.[1] The research has been conducted according to the regulations laid out by the Universidad Politécnica de Madrid and University of Oulu's Ethical Boards. Both the funding agency and the participating researchers state that they have no conflicts of interest with respect to the research results. In all cases, the experimental procedure was as follows:

- Before conducting the quasi-experiments, the experimental tasks (shown in Appendix 2) were selected and the code templates were prepared. H. Erdogmus, B. Turhan, D. Fucci, A. Tosum and T. Raty performed this task.
- Again before the quasi-experiments were performed, the forms described in Appendices 3 and 4 were used to acquire data about the experimental subjects. A. Santos processed the demographic data.
- Each quasi-experiment used a particular programming language, testing framework and IDE depending on the preferences of the host organization. The most commonly used technology was *Java + jUnit + Eclipse*.
- The quasi-experiment had a total duration of 8 h:

– The first 4 h were spent on training the subjects to use the selected testing frameworks and practical exercises. B. Turhan delivered the training for quasi-experiments 1–5 (with the help of T. Raty in one case[2]). O. Dieste delivered the training for experiments 6–10.
– The experimental task (MR, BSK, with or without slicing) was completed after training. It had a duration of 2 h without breaks. The task assignment to experimental subjects differed slightly in each quasi-experiment for the purpose of alignment with the needs of the research on programming strategies and TDD of which this study is part. Tasks were assigned rigorously without introducing validity threats.
– D. Fucci, A. Tosun and S. Vegas (depending on the case) supervised the experimental task. At the end of the experimental task, subjects handed in their code and the quasi-experiment was concluded with a short debriefing.

Table 2 shows the particular conditions under which each quasi-experiment was conducted. As such contextual variables can have a bearing on code quality and programmer productivity (e.g., *C++* and *Boost Test* are more complicated to use than *Java* and *jUnit*), they have to be specifically considered and, where appropriate, added as blocking variables to the analysis. One exception to this rule is the IDE, as it is the same in almost all cases (*Eclipse*) and has no predictive ability. Although the same might be said of programming language (*C++* and *Java*) and testing framework (*jUnit*, *Google Test* and *Boost Test*), each group has a sizeable number of subjects in these two cases (e.g., 29 subjects used C++). It is therefore preferable not to jump to

[1] TEKES: Finnish Funding Agency for Technology and Innovation
[2] Not specified so as not to disclose T. Raty's organization.

Table 2 Contextual variables characterizing each of the conducted quasi-experiments

		Trainer	Site	Programming language	Testing Framework	IDE
Experiment code	1	B. Turhan	Industry	Java	jUnit	Eclipse
	2	B. Turhan	Industry	Java	jUnit	Eclipse
	3	B. Turhan	Industry	Java	jUnit	Eclipse
	4	B. Turhan	Industry	C++	Google Test	Eclipse
	5	B. Turhan	Academia	Java	jUnit	Eclipse
	6	O. Dieste	Industry	Java	jUnit	Eclipse
	7	O. Dieste	Academia	Java	jUnit	Eclipse
	8	O. Dieste	Academia	Java	jUnit	Eclipse
	9	O. Dieste	Academia	Java	jUnit	Eclipse
	10	O. Dieste	Industry	C++	Boost test	Eclipse (17 cases) Vim (2 cases)

conclusions and have the actual analysis procedure determine (e.g., by collinearity) whether or not these contextual variables should be omitted.

3.2 Dependent Variables

The effect construct is programmer performance. As the research question states, we passed over the response variables typically used to study the effect of programmer experience, e.g., ability to identify valid sentences in a programming language (Wiedenbeck 1985), and we used operationalizations focused on code properties that could be directly measured.

Code can be examined from different viewpoints. In this research, we used the *External quality* of the code and the *Productivity* of programmers as response variables. *External quality* is equivalent to the functional concept of quality defined in ISO/IEC 25010 as the extent to which a software product satisfies certain needs (ISO 2011). In this respect, quality is related to what functionality code users get rather than the internal structure of the code, which is why we use the adjective external. *Productivity* is generally defined as the amount of work done. Section 3.9 details the metrics and measurement procedures for both variables, which are basically percentages representing the ratio of *External Quality* or *Productivity* to their respective maximum values.

The use of the above variables has two practical advantages. On one hand, this research into the effect of programmer experience is part of a wider research project into programming strategies and test-driven development (TDD). The *External Quality* and *Productivity* variables are often used in TDD studies, e.g., (Erdogmus et al. 2005; Munir et al. 2014). Therefore, their use will keep both research projects aligned and create synergies. On the other hand, *External quality* and *Productivity* can be defined separately from the task, programming language, etc. This provides for the comparison and joint analysis of data from a range of experiments. This is a very important point, as sample sizes of over a hundred subjects are required to achieve adequate statistics power (see Section 5.1.2). The sample size of a single experiment is not usually this big, and several experimental replications have to be conducted and jointly analysed.

3.3 Subject Selection

The experimental subjects were convenience sampled (i.e. selected by availability). They are members of two separate groups:

- Programmers with different levels of experience from four European companies located in Finland and Estonia.
- Senior undergraduate and postgraduate students from three universities located in Spain and Ecuador. Most of the students do not have professional experience, although some have already worked or are working in industry.

3.4 Experimental Task

The quasi-experiment has only one experimental task, which is to apply an *Incremental test-last development* (ITLD) strategy (Madeyski 2005). This strategy involves writing production and testing code in parallel, without prioritizing testing code as in TDD. The ITLD strategy is in widespread use in industry, where there is a recognized need for automated testing to increase production code quality (Williams et al. 2009). ITLD is not unusual, albeit less common, in academia. No further conditions were imposed on ITLD, i.e., each programmer was allowed to select whichever slice granularity and tests he or she wanted to use. In other words, the programmers completed the task more or less as per usual practice. All programmers were informed verbally, at the beginning of the experimental session, that the goal was to complete the experimental problem in the allocated time frame.

3.5 Experimental Problems

The subjects applied ITLD on two experimental problems, MarsRover API (MR) and Bowling Scorekeeper (BSK). BSK and MR are generic programming assignments, and thus they do not specifically belong to the domain of experience of any of the experimental subjects. They enable a clear separation between potential domain knowledge effects (i.e., the performance improvements achieved because the programming assignment is familiar) and the effects due only to the length of programming experience, which are the ones relevant for this research.

Appendix 2 gives a full description of the programming assignments that the subjects were set. There are two versions: with and without slices. Slices conform the original definition by (Weiser 1981), although we are using them from a different perspective (Lee et al. 2001). MR and BSK problems are not at all challenging for professional programmers and should be doable for undergraduate and postgraduate students.

3.5.1 MarsRover API

MR is a programming exercise that requires the development of a public interface for controlling the movement of a fictitious vehicle on a grid with obstacles. MR is a popular exercise used by the agile community to teach and practice unit testing.

MR is an algorithm-oriented task and does not involve the creation of a user interface. The implementer needs to handle several boundary cases in order to produce the expected results. The implementation of MR leverages a NxN matrix data structure representing an imaginary planet on which the rover moves. Each matrix cell may store an obstacle on the planet's surface. Obstacles are without behaviour and can be modelled using simple data types (e.g., a Boolean for representing presence/absence). Subjects have to implement six main operations necessary to move the rover on the planet's surface. The task can easily be solved using just one class. The possible operations are:

- Matrix initialization and assignment of obstacles to cells
- Command parsing
- Forward and backward moves
- Left and right turns.

The forward and backward moves are the most complex operations. Command parsing and left/right turns are straightforward operations. The assignment of obstacles to cells upon initialization requires some parsing and type casting.

Subjects were given the MR specification document and a project template in order to get them started and provide a common package structure that would make data collection easier to automate.

3.5.2 Bowling Scorekeeper

BSK is a modified version of Robert Martin's Bowling Scorekeeper (Bob 2005). This task is popular in the agile community. The goal of the task is to calculate the score of a single bowling game. The task is algorithm-oriented and it does not involve the creation of a user interface. The task does not require prior knowledge of bowling scoring rules: this knowledge is embedded in the specification. BSK also has six main operations:

- Add a frame or bonus throws
- Detect when a frame is a spare or strike
- Calculate a frame score
- Calculate the game score.

The most complex operation is the calculation of the frame score. It depends on the type of the frame (regular, spare or strike), the position of the frame in the game, and whether or not the next frame is a strike.

We gave subjects the BSK specification document and a code template.

3.6 Treatment Assignment to Subjects

We have used a quasi-experimental design to study the effect of experience. Quasi-experiments are used when the subjects cannot be randomly assigned to an experimental condition, or, alternatively, a treatment cannot be assigned to a group. This applies in our case, as the experimental subjects' characteristics are intrinsic and cannot be randomized or blocked. Consequently, all the subjects have performed the same task (ITLD) to the same experimental object (MR or BSK, either sliced or not). Note that each subject participated only once. The quasi-experimental design of this study means that the relationship between the independent and dependent variables cannot be said to be causal.

3.7 Instrumentation

The subjects implemented the experimental tasks in *Java* or *C++*. The language was selected depending on preferences at the site where each quasi-experiment was conducted. They used the *jUnit*, *Google Test* and *Boost Test* testing frameworks. In all cases, we gave subjects stubs so that they did not have to write the testing framework initialization code (not necessarily evident in the case of *Boost Test*) and could focus exclusively on writing the tests that they considered necessary. Most subjects used the *Eclipse* integrated development environment (IDE), although some subjects preferred to use text-mode editors like *Vim*.

3.8 Measurement Procedure

3.8.1 Independent Variables

We gathered the values of the independent variables using a questionnaire implemented in *Google Forms*. Appendices 3 and 4 show the questionnaires for professionals and students, respectively.

3.8.2 Dependent Variables

We used acceptance tests as the main instrument for extracting the *Productivity* (PROD) and *External Quality* (QLTY) response variable data. We wrote a set of acceptance tests for all tasks. MR can be decomposed into 11 subtasks (or slices) that represent all the functionality required to complete this task. A total of 13 subtasks can be defined for BSK. Appendix 2 lists the MR and BSK subtasks. One of the researchers (D. Fucci) wrote tests for MR, whereas the BSK tests were adapted from a previous experiment (Erdogmus et al. 2005). MR's acceptance test suite has 11 test classes, 52 test methods and 89 assertions. BSK has 13 test classes, 51 test methods and 56 assertions. Each test class implements the test of a particular subtask.

Productivity can be defined as (Kitchenham and Mendes 2004):

$$\text{Productivity} = \frac{\text{Process output}}{\text{Process inputs}} \tag{1}$$

Process output is some measure of size, such as the number of lines of code (LOC) produced by a developer, number/percentage of user stories implemented, or the number/ percentage of passing test cases. LOC has known weaknesses as metric (Armour 2004). Conformance-based metrics are widely used (Darcy and Ma 2005). Therefore, we opted to use the percentage of passing test assertions over all assertions as the basis for output calculation.

The most common *input* is some measure of effort (Fenton and Bieman 2014), such as man-months or monetary cost. In our case, subjects have a maximum time to complete the tasks and tend to use in its entirety. Therefore, a time-based metric is useless. We also ruled out monetary cost, due to the quasi-experimental character of our research. Therefore, the *process input* is constant across subjects and experimental runs. Being constant, it can be discarded for productivity calculation.

Thus, PROD represents the amount of functionality delivered by programmers (i.e., the *amount of work done*), and it is defined as shown in Eq. 2:

$$PROD = \frac{\#Assert(Pass)}{\#Assert(All)} \times 100 \tag{2}$$

The concept of quality that we are using is the extent to which a software product satisfies certain needs (ISO 2011). Defined as such, quality can be interpreted as the *amount of functionality* delivered by programmers, i.e., productivity. However, this equality is only valid when coding is complete, that is, when programmers are able to finish the task before delivery. When it does not happen, the *amount of functionality* underestimates quality. For instance, let's assume that a programmer completes only a fraction (e.g., 80 %) of a given task, with no errors. His or her productivity is clearly 80 % (the amount of delivered functionality), but the quality of the code cannot be 80 % because it is completely correct; quality should be 100 %.

In this research, most experimental subjects have been unable to complete the programming tasks. Therefore, we need to find out the degree of termination of each task to fine-tune quality accordingly. We have accomplished this goal examining MR and BSK subtasks. We have considered that an experimental subject has worked on a given subtask when at least one assert statement in the acceptance test suite associated with that subtask passes. This criterion is used to objectively separate subtasks to which a subject made a reasonable amount of effort to complete, from other subtasks in which a subject invested little or no effort.

In order to formalize this criterion, we have defined whether a subtask *i* has been "tackled" (TST_i) as indicated in Eq. 3:

$$TST_i = \begin{cases} 1 & \#ASSERT_i(Pass) > 0 \\ 0 & otherwise \end{cases} \tag{3}$$

The number of tackled subtasks (TST) is calculated using Eq. 4, where *n* is the total number of subtasks of the experimental problem.

$$TST = \sum_{i=0}^{n} TST_i \tag{4}$$

We use TST to calculate QLTY in Eq. 5:

$$QLTY = \sum_{TST_i=1} \frac{QLTY_i}{TST} \tag{5}$$

where $QLTY_i$ is the quality of the i-th tackled subtask, and is defined as:

$$QLTY_i = \frac{\#Assert_i(Pass)}{\#Assert_i(All)} \times 100 \tag{6}$$

$\#Assert_i(Pass)$ represents the number of passing assertions in the acceptance test suite associated with the i-th subtask. In other words, QLTY represents how correct (in percentage) the code corresponding to the tackled tasks is.

The dependent variables PROD and QLTY are related in such a way that QLTY > = PROD. This restriction implies that QLTY can take any value when PROD is low but, as PROD increases, QLTY increases accordingly. When PROD nears 100 %, QLTY also approaches 100 %. The strong relationship between PROD and QLTY makes that both constructs cannot be differentiated when subjects are highly productive, i.e., in such a case we observe *Productivity* or *External Quality* (likely the later), but not *both*. Nevertheless, in the set of quasi-experiments that we are using for this research, the time is constrained and only a fraction of subjects achieve high PROD values. Therefore, we are rather confident that the constructs *Productivity* and *External Quality* have been reasonably operationalized.

3.8.3 Data Collection

The measurement procedure involved executing the test suites on the code written by subjects. Subjects were told not to modify the API for the MR and BSK problems which was well defined in the code templates that they were given. Even so, they did. This caused compilation errors in the test suite. These errors were corrected by adapting the production code to the test code and vice versa depending on each case. We tried to modify the code written by subjects as little as possible so as not to introduce validity threats. However, the alternative in many cases was to assign QLTY = PROD = 0 values for the subjects that had altered the API, which was clearly going too far.

Measurement was based on a set of test cases, but some type of adaptation of the subjects' source code (e.g.: aligning return data types, fixing problems with leading/training spaces, etc.) should be made in almost all cases. The measurer can have an influence here, e.g., one measurer might make more changes to the production code than another. This possible threat to validity is addressed in Section 7. D. Fucci measured quasi-experiments 1–3 (a total of 24 subjects), O. Dieste measured quasi-experiments 8–9 (a total of 22 subjects), and F. Uyaguari measured the other five quasi-experiments (a total of 80 subjects). We collected data from a total of 126 subjects.

3.9 Experimental Repository

Experimental data is confidential nowadays. A sanitized version is available at http://www.grise.upm.es/sites/extras/11. This website also stores the test cases used for measurement.

4 Methodology

The study of the effects of experience on programmers' performance was conducted by means of a series of quasi-experiments. The programmers completed a programming assignment, and their experience and performance were then compared. The quasi-experiment design is detailed in the following.

4.1 Hypothesis

The main hypothesis of this paper, stated as null/alternative hypothesis, is as follows:

H_0 programmer experience *does not influence* their performance

H_1 programmer experience *does influence* their performance.

It is a generally accepted fact in SE that experience improves programmer performance. Therefore, one might be tempted to test the hypotheses using one-tailed (i.e., programmer experience improves their performance) rather than two-tailed tests. However, the reviewed literature shows that there are contradictory opinions with respect to experienced programmers performing better. As this is an exploratory study, we decided provide for possible effects in both directions, i.e., experience having both positive and negative effects, to be on the safe side.

In this research, performance has been operationalized as the quality (QLTY) and productivity (PROD) response variables. QLTY represents the degree of correctness in the experimental task that the programmers were able to achieve. PROD represents the amount of functionality delivered. If experience influence performance positively, then we should find a direct relationship between any experience-related variable and quality/productivity. It is unlikely that experience influence positively quality or productivity alone. Common sense suggests that an expert programmer does not only do more work than novices in the same time, but the work outcome is also better, i.e., of higher quality. Nevertheless, we will test H_0 independently for quality and productivity to evaluate all possible alternatives.

4.2 Independent Variables

4.2.1 Experience-Related Independent Variables

The cause construct refers to *Programmer experience*. Experience is not a directly observable construct (Siegmund et al. 2014) that can be operationalized using multiple independent variables (e.g., programming experience, unit testing experience), where each independent variable can be measured in different ways (e.g., years, Likert scales).

In this research, we decided to use as many independent variables as possible to prevent mistaken conclusions being reached due to the operationalization. For example, *Unit testing experience* could be considered a poor operationalization of *Programmer experience*, as a good tester is not necessarily a good programmer. However, it is reasonable to assume that a programmer with some *Unit testing experience* might produce better quality code. Therefore, it is not wrong to use the *Unit testing experience* as an independent variable. Table 3 details the studied independent variables.

The categorical variables have two possible values (No/Yes) with a numerical equivalence for ease of interpretation if their effect is as expected (e.g., 1 = NO CS degree, 2 = YES CS degree, assuming that CS degree holdership improves both programmer quality and productivity). The ordinal variables are measured by means of four-point Likert scales, coded as follows:

- 1 = No experience (<2 years)
- 2 = Novice (2–5 years)
- 3 = Intermediate (5–10 years)
- 4 = Expert (>10 years)

The Likert scale is based on year ranges that are equivalent to the time spans commonly specified in the literature that it takes to acquire the respective expertise. Campbell and Bello (1996) point out that programmers need (at least) 2 years to become Smalltalk experts. Sim et al. (2006) consider that 5 experience years are a reasonable period (not necessarily sufficient) for an engineer to achieve expertise. Additionally, these ranges counteract the optimism with which the subjects interpret the text labels (i.e., novice, expert), which biases measurements (Aranda et al. 2014). Positive biases have been reported in several SE activities, e.g., (Jørgensen et al. 2007).

Programming experience is probably the most interesting aspect in this research. The ordinal variables *Experience in the programming language used in the experiment* and *Overall programming experience* are very useful for studying the effect of experience on programming, as they are handy means for subjects to rate and report their experience. On the other hand, however, their accuracy is limited on two grounds:

- The results of the multiple linear regression analyses (i.e., the analysis method used in this research, see Section 4.3) may be biased by the use of ordinal values (Winship and Mare 1984).
- The experimental subjects (see Section 3.4) are both professionals working in industry and students taking different programmes in academia. The extent and rate of exposure to the

Table 3 Independent variables used

Categorical (dummies)	Ordinal	Scalar
• Holds a CS degree • Currently uses a unit testing framework • Has specialized training in unit testing • Has specialized training in TDD • Current uses the IDE used in the experiment • Currently uses TDD	• Experience in the unit testing framework used during the experiment • Experience in the programming language used in the experiment • Overall programming experience • Unit testing experience • TDD experience in TDD (if currently uses TDD = YES)	• Experience in the programming language used in the experiment acquired in academia • Experience in the programming language used in the experiment acquired in industry • Overall programming experience acquired in academia • Overall programming experience acquired in industry

programming activity in both groups should be considerably different and it is seldom clear that they can be measured using the same variables.

In order to set off the ordinal variables, we have also captured separate scalar variables for industry and academia measured in years and referred to both experience in the programming language used in the experiment and overall experience.

4.2.2 Other Independent Variables

This paper is, in essence, a secondary analysis that relies on data collected in diverse contexts using different experimental designs. Such diversity gives rise to the appearance of several variables, such as TRAINER, SLICING or TASK_ITLD, not directly related to the experience construct. For the reader convenience, the independent variables used are listed in Appendix 1.

4.3 Dataset

Table 4 summarizes the key demographic sample data. As we can see, both professional programmers and students state that they have from 2 to 10 years of overall programming experience and slightly less (from 0 to 5 years) experience in the specific programming language used in each experiment. Appendix 5 shows the breakdown of experience measured in years. Generally, the experience measured in years is quite well aligned with the Likert-scale data. The experience on the programming language used in each quasi-experiment is slightly greater among students than in industry (2.1 vs. 1.8 years). This is probably a reflection of the widespread use of Java in academia as opposed to the wider range of programming languages that are used at companies. Overall programming experience is, predictably, greater in industry (4 vs. 5.1 years).

As regards experience broken down by subject types, we found that students have slightly more academic experience than practitioners regarding the programming language used in each quasi-experiment (2.2 vs. 2.0 years on average, respectively). However, practitioners, predictably, have more experience than students in industry (0.7 vs. 2.7 years). The pattern is similar for overall programming experience (5.9 vs. 2.7 and 2.4 vs. 7.1 years). Experience measured in years clearly appears to better account for population characteristics than the ordinal variables, and will be given preference.

The biggest difference between both groups (professionals and students) is with respect to years of unit testing experience, IDE use and academic training received. Most students are pursuing a degree in computer science, whereas professionals have different educational backgrounds. On the other hand, professional programmers have more experience in unit testing, whereas students are more acquainted with IDE use.

The number of subjects in each independent variable category is reasonably well balanced, on which ground we expect the research results will not be biased by group size, except perhaps with regard to educational background and TDD use. This could be considered as a possible threat to validity as described in Section 7. Additionally, the experience ranges at our disposal match the specifications in the related literature, where theoretically expertise is acquired after from 5 to 10 years of deliberate practice (see Section 2). The experimental subjects should therefore be suitable for identifying the effects of experience on code quality and programmer productivity.

Table 4 Characterization of subjects. Totals do not match due to missing responses

Characteristics	Levels	Environment		
		Academia	Industry	Total
CS degree holdership	No computer science	1	29	30
	Computer science	54	36	90
	Total	*55*	*65*	*120*
Overall programming experience	No experience (<2 years)	7	5	12
	Novice (2-5 years)	25	20	45
	Intermediate (5–10 years)	23	29	52
	Expert (>10 years)	1	15	16
	Total	*56*	*69*	*125*
Experience in programming language used in the experiment	No experience (<2 years)	16	24	40
	Novice (2–5 years)	28	26	54
	Intermediate (5–10 years)	12	12	24
	Expert (>10 years)	0	8	8
	Total	*56*	*70*	*126*
Experience in unit testing	No experience (<2 years)	50	31	81
	Novice (2–5 years)	6	25	31
	Intermediate (5–10 years)	0	11	11
	Expert (>10 years)	0	3	3
	Total	*56*	*70*	*126*
Current usage of the IDE used in the experiment	No	10	36	46
	Yes	45	34	79
	Total	*55*	*70*	*125*
Current usage of TDD	No	50	51	101
	Yes	6	19	25
	Total	*56*	*70*	*126*

4.4 Analysis Strategy

Each quasi-experiment separately is insufficient for detecting effects in either of the response variables. For example, a correlation analysis requires 67 subjects to identify medium effects ($r = 0.3$) with a power of 80 %. The simultaneous analysis of several variables would be less statistically powerful. Consequently, the data collected from the quasi-experiments must be analysed jointly.

We have to rule out meta-analysis on two grounds: (1) there are not many well-developed meta-analysis models for multiple independent variables, and (2) we have subject-level data, meaning that the most common analysis methods (e.g., ANOVA, multiple regression) are applicable (provided the right blocking variables are introduced (Hedges and Olkin 1985). The analysis of subject data just might, although the literature on this point is unclear, output more solid findings (e.g., with a higher statistical power) than experiment-level analyses (Riley et al. 2010).

The independent variables that we have used in this research are ordinal (dummy-coded binary) or scalar. Apart from these independent variables, the particular characteristics of each quasi-experiment have generated categorical contextual variables (e.g., testing framework, programming language, etc., see Appendix 1) which have to be accounted for in the analysis as blocking variables. The mix of variables is problematic, as there is no method that can analyse all of these variables together. Possible scenarios follow:

1. The usual experiment analysis methods, such as ANOVA or mixed models, cannot use scalar independent variables.
2. If we were to omit the scalar variables and use only ordinal independent variables, we could use ANOVA and mixed models but the different ordinal variable values would be considered as different categories. This means that we would lose all the information associated with the order relationship between the ordinal variables. We do not think that this is a good strategy as: (1) it is equivalent to a dichotomization that may lead to incorrect results (MacCallum et al. 2002), and (2) the very phenomenon under study (the effect of *Programmer experience*) requires the magnitudes to be specifically considered, e.g., 3 years of experience < 4 years of experience, irrespective of the fact that 3 and 4 years of experience can be considered as *intermediate experience.*
3. The linear regression model can deal with categorical variables. When categorical variables have just two values (/levels), they can be used directly (provided that they are recoded as dummy variables). Categorical variables with more than 2 levels require a more complicated apparatus (Weisberg 2005). Ordinal and scalar variables can be used without restrictions.

We believe that the best analysis option is to use the multiple linear regression model (MLR), because, as discussed in Section 3.1, there are only two categorical variables with more than two values (*Testing framework used in the quasi-experiment* and *Experiment code*) in our dataset. In the first case, we would not be running too much of a risk if we recoded the variable, as specified in Section 4.4. This way we would be able to take advantage of the fact that MLR is better able to deal with ordinal and scalar variables. The second case is not as straightforward. There is a definite possibility of subject performance being better at one company or university than another. Therefore, the analysis should take into account *Experiment code*. However, as we will see later, *Experiment code* is highly collinear (i.e., the values of *Experiment code* are confounded with other variables). This rules out its use in MLR. A possible trade-off is to ignore the *Experiment code* during the first stage of the analysis using MLR, and then study whether the model residuals are systematically related to *Experiment code*. This is the approach that we take.

Four basic conditions have to hold for MLR to be reliable. They are: collinearity, sample size, normality and homoscedasticity (Field et al. 2012).

1. Collinearity. For the model to be reliable, we have to assure that the model predictor variables are not collinear. Collinearity occurs in a regression model when one or more of the predictor variables (dummies, ordinal or scalar) are linearly correlated with other model variables. We used the variance inflation factor (VIF), tolerance (T) and condition index (CI) to test for the collinearity between variables.
2. Sample size. The study will be more statistically powerful the larger the sample size is, that is, the statistical power of a study with a small sample size will be low. Consequently, the estimates will be less accurate, and we will be less likely to detect significant effects. This highlights the importance of a large enough sample size.

3. Normality. The distribution of residuals must be normal with zero mean and random but constant variance. We used the Lilliefors-corrected Kolmogorov-Smirnov test, the Shapiro-Wilks test, and Q-Q plots to test for the normal distribution of residuals.
4. Homoscedasticity. We tested for homogeneity of variance using scatter plots of model residuals against predicted values.

In addition to the MLR, we will use decision trees to explore nonlinear effects (Brandmaier et al. 2013). Several algorithms for building decision trees could be used: CHAID, exhaustive CHAID, CART and QUEST. Each one has strengths and weaknesses. We will use CART (Classification and Regression Trees), because it has intimate connections with MLR analysis (they both use mean squared errors for scale dependent variables). Therefore, the outcomes of the CART trees and the MLR support each other. Furthermore, CART does not impose restrictions on independent and dependent variables, and it is not affected by the variable type (categorical, ordinal o scale), outliers, heteroskedasticity, collinearity or distributional error structures (Nisbet et al. 2009). CART can be used with smaller datasets than e.g., CHAID as well (Chulis 2012).

4.5 Data Transformations

As illustrated in Section 4.2, Table 3, the categorical variable *Testing framework* has three levels: gTest, jUnit and Boost Test. In order to use MLR, we had to recode one of the variable levels to output a dummy variable. Specifically, we have recoded the gTest levels and jUnit levels as a single xUnit value. We believe that this is feasible as the syntax of gTest and jUnit is very similar and the gTest code templates given to students mean that it is used in more or less the same way as jUnit in practice. We did not equate Boost Test to jUnit and gTest, because Boost Test's syntax is much more complex and it has a number of concepts that are quite far removed from jUnit and gTest. It could therefore be considered more complex than jUnit and gTest, for which reason we decided to consider it separately in the analysis.

After this procedure, the *Testing framework* was transformed to a dummy value with the following levels: xUnit and Boost Test.

5 Linear Model Analysis

5.1 Data Exploration

This section reports some descriptive statistics about the dataset that we will use to answer the research questions. First, we show the overall distribution of the QLTY and PROD variables, separated by programming assignment (MR, BSK). Later, we give an account of the average quality and productivity scores obtained by the subjects, depending on their experience level and site (industry or academy).

5.1.1 Overall Distribution

Figure 1 contains histograms describing the distribution of the quality and productivity scores. We provide separated histograms for MR and BSK because, although both experimental objects have comparable complexity, other aspects (e.g., domain knowledge, ability with algorithmic programming, etc.) may influence programmers' performance. The plots suggest that MR and BSK are not exactly alike. Subjects fail quite more often (see the tall column in the 0–10 class) when they work on MR. The lesser complexity of BSK can also be seen in the skewness (to the right) of the distribution: more subjects achieve high quality/productivity when solving BSK. Leaving this apart, the shapes of the histograms do not reveal dramatically different patterns.

5.1.2 *Quality*

Tables 5 and 6 report the number of subjects and the corresponding quality averages for different experience levels. The grand means (both in Tables 5 and 6) are similar, although the scores are slightly higher (3 % difference) for industry than academia. The most striking difference is the relationship between experience levels and scores. In academia, students improve in quality as their experience increase. In industry, the scores keep essentially constant (with some exceptions, as the zigzag pattern shown in Table 6).

The patters are more evident when we run Pearson correlations, as we show in Table 7. Experiences in academy have low/medium effects (Cohen 1988), statistically

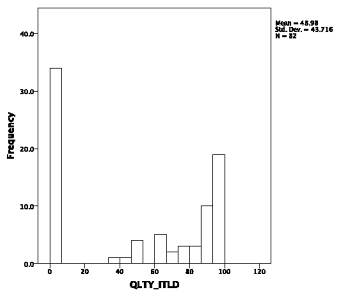
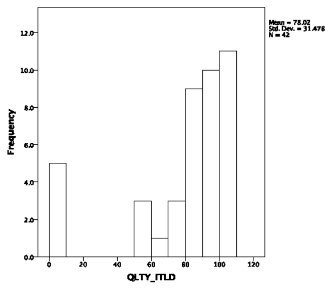
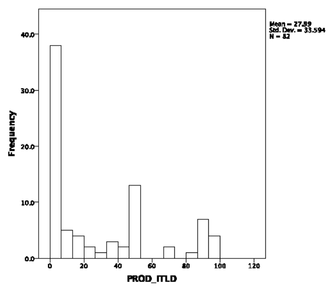
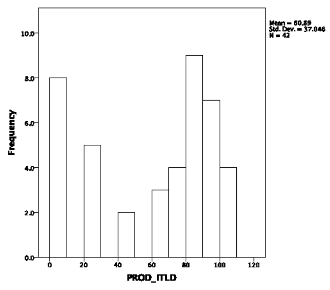
	MR	**BSK**
QLTY		
PROD		

Fig. 1 Data distribution (per programming assignment)

Table 5 Mean quality of subjects depending on programming language experience

QLTY	Academy		Industry	
	#Subjects	Mean	#Subjects	Mean
No experience (<2 years)	14	40.33 %	24	68.89 %
Novice (2–5 years)	28	56.91 %	26	45.65 %
Intermediate (5–10 years)	12	77.04 %	12	65.55 %
Expert (>10 years)			8	73.01 %
Total subjects	54	57.08 %	70	60.15 %

significant or close to significance. In turn, experiences in industry are low in both cases and non-significant.

5.1.3 Productivity

Tables 8 and 9 show the productivity scores using the same conventions than previous section. There are several differences as compared to quality. First regards the grand mean for the *Academia* and *Industry* categories: students achieve higher productivity than practitioners. Second, students increase productivity with experience, whereas practitioners exhibit a decreasing trend.

Pearson correlations, shown in Table 10, confirm the visual exploration of Tables 7 and 9. The overall programming experience in academia has a very strong correlation with productivity. Industry-related experience exhibit very low correlation coefficients, negative (confirming the decreasing trend), and non-significant.

The descriptive statistics suggest that industry experience does not seem to be related to superior performance. Academic experience *could*. However, the previous tables and correlation coefficients summarize the dataset in a very coarse-grained manner. There are many other independent variables that may have an influence on the quality and productivity scores. A more in-depth analysis will be conducted in the following sections.

Table 6 Mean quality of subjects depending on overall programming experience

QLTY	Academia		Industry	
	#Subjects	Mean	#Subjects	Mean
No experience (<2 years)	6	36.08 %	5	58.97 %
Novice (2–5 years)	25	60.09 %	20	61.24 %
Intermediate (5–10 years)	22	61.99 %	29	63.36 %
Expert (>10 years)	1	.00 %	15	53.57 %
Total subjects	54	57.08 %	69	60.30 %

Table 7 Pearson correlations (for QLTY)

Response variable	SITE	Independent variable	correlation		
			r	p-value	N
QLTY	Academia	Experience Programming Language	.155	.086	124
		Overall Programming Experience	.240*	.007	124
	Industry	Experience Programming Language	.131	.146	124
		Overall Programming Experience	.108	.235	122

5.2 Choosing the Best Regression Model

The aim of this section is to determine which regression model best fits the data. The original model contained all the demographic variables and contextual variables (see Appendix 6). We then checked that the independent variables were not collinear. If they were, we eliminated any variables that were strongly correlated to the others, thereby simplifying the regression model.

5.2.1 Checking for Collinearity

One way of determining whether the independent variables are collinear is to use the variance inflation factor with the condition index.

- *The variance inflation factor* (VIF): a measure of the impact of collinearity between the regression model variables. High VIF values are a sign that a variable can be largely explained by the other variables, that is, that the model variables are collinear. A VIF-related parameter is tolerance (T), which is defined as $T = 1/VIF$. A guideline often used by researchers is to use a high VIF, that is, $VIF > 10$, which is output when $R^2 > 0.9$ and $T < 0.1$. A second, more rigorous, option is to lower the bounds to $VIF > 5$ with $R^2 > 0.8$ and $T < 0.2$ (Heiberger and Holland 2013) as evidence of collinearity.
- *Condition index (IC): a* measure of ill-conditioning in a matrix. Belsley (1991) suggest three levels of collinearity depending on the CI: slight ($CI < 10$), moderate ($10 < CI < 30$) and severe ($CI \geq 30$). When a model has a severe CI, the variance of one or more of its variables is substantially collinear with the other variables. A high proportion of

Table 8 Mean productivity of subjects depending on programming language experience

PROD	Academy		Industry	
	#Subjects	Mean	#Subjects	Mean
No experience (<2 years)	14	37.74 %	24	40.08 %
Novice (2–5 years)	28	40.89 %	26	26.64 %
Intermediate (5–10 years)	12	66.74 %	12	39.34 %
Expert (>10 years)			8	31.46 %
Total subjects	54	45.82 %	70	33.97 %

Table 9 Mean productivity of subjects depending on overall programming experience

PROD	Academia		Industry	
	#Subjects	Mean	#Subjects	Mean
No experience (<2 years)	6	25.98 %	5	47.16 %
Novice (2–5 years)	25	46.06 %	20	36.44 %
Intermediate (5–10 years)	22	53.04 %	29	36.14 %
Expert (>10 years)	1	0.00 %	15	24.26 %
Total subjects	54	45.82 %	69	34.44 %

variance explained (greater than 0.5) is usually considered to be a sign that the respective variable is involved in the collinear relationship.

As shown in Appendix 6 we have 15 independent variables that might be included in the regression model. The collinearity statistics shown in Table 11 suggest that none of the variables has a VIF greater than 10 (a T less than 0.1). Looking at the more rigorous option ($VIF > 5$ or $T < 0.2$), we find that the pattern for the *Testing framework* variable (UNIT_TESTING_FRAMEWORK_ ADAPTED) could pose problems of collinearity, as its values are close to the bounds established for the VIF ($VIF = 4.943$) and tolerance is ($T = 0.202$). On the other hand, the collinearity statistics for the other variables are within the expected bounds ($FIV < 5$ and $T > 0.2$), which suggests that they are not collinear.

The collinearity diagnostics of the model specified in Table 11, as shown in Appendix 6, report that the UNIT_TESTING_FRAMEWORK_ADAPTED and EXPERIMENT_ PROGRAMMING_LANGUAGE variables have an collinearity problem. One way of solving the collinearity problem is to remove the most collinear variable, which, in this case, is UNIT_TESTING_FRAMEWORK_ADAPTED.

The removal of collinear variables has two implications: one positive and one negative. The positive consequence is the elimination of the UNIT_TESTING_FRAMEWORK_ADAPTED variable, which represents the recoding of the three testing frameworks (gTest, jUnit and Boost Test) into two (xUnit and Boost Test). The removal of UNIT_TESTING_FRAMEWORK_ ADAPTED variable eliminates the potential threats to validity posed by recoding. In either case, we checked that UNIT_TESTING_FRAMEWORK would have been collinear even if the UNIT_TESTING_FRAMEWORK had not been recoded.

On the negative side, (O'Brien 2007) discourages the removal of variables as a means to solve collinearity problems. One exception to this advice is that the elimination is *theoretically*

Table 10 Pearson correlations (for PROD)

Response variable	SITE	Independent variable	correlation		
			r	*p*-value	N
PROD	Academia	Experience Programming Language	.064	.481	124
		Overall Programming Experience	$.378^{**}$	.000	124
	Industry	Experience Programming Language	−.010	.913	124
		Overall Programming Experience	−.096	.292	122

Table 11 VIF and T for original MLR model

Model		Unstandardized Coefficients		Standardized Coefficients	t	*P*-val.	Collinearity Statistics	
		B	Std. Error	Beta			Tolerance	VIF
1	(Constant)	−64.527	65.070		−.992	.324		
	SITE	36.151	10.468	.425	3.454	.001	.429	2.330
	TRAINER	2.476	11.380	.028	.218	.828	.398	2.512
	CS_TITLE	17.018	9.767	.177	1.742	.085	.628	1.592
	UNIT_TESTING_FRAMEWORK_ADAPTED	−14.927	21.838	−.123	−.684	.496	.202	4.943
	EXPERIENCE_UNIT_TESTING_FRAMEWORK_LIKERT_SCALE	8.903	8.841	.119	1.007	.316	.464	2.157
	EXPERIMENT_PROGRAMMING_LANGUAGE	23.861	17.329	.230	1.377	.172	.233	4.292
	EXPERIENCE_EXPERIMENT_PROGRAMMING_LANGUAGE_ACADEMY_YEARS	.337	1.995	.022	.169	.866	.382	2.621
	EXPERIENCE_EXPERIMENT_PROGRAMMING_LANGUAGE_INDUSTRY_YEARS	1.198	1.978	.086	.606	.546	.321	3.119
	OVERALL_EXPERIENCE_PROGRAMMING_ACADEMY_YEARS	3.326	1.289	.285	2.581	.011	.534	1.873
	OVERALL_EXPERIENCE_PROGRAMMING_INDUSTRY_YEARS	.959	1.039	.135	.923	.358	.304	3.292
	EXPERIENCE_UNIT_TESTING_LIKERT_SCALE	−9.577	7.411	−.162	−1.292	.199	.412	2.426
	EXPERIMENT_IDE_USED_DUMMY	16.605	9.187	.190	1.807	.074	.590	1.694
	TDD_USED_DUMMY	−1.873	10.723	−.017	−.175	.862	.650	1.540
	TASK_ITLD	8.511	13.514	.094	.630	.530	.290	3.449
	SLICED_ITLD_DUMMY	29.735	13.477	.330	2.206	.030	.292	3.430

Dependent Variable: QLTY

motivated. In our case, the collinearity between UNIT_TESTING_FRAMEWORK_ADAPTED, EXPERIMENT_PROGRAMMING_LANGUAGE and the other variables is probably due to most of the experiments were run using Java and jUnit. In other words, the data that we have are not diverse enough to identify the moderator effects of UNIT_TESTING_FRAMEWORK_ADAPTED and EXPERIMENTAL_PROGRAMMING_LANGUAGE. Furthermore, those variables are not related to the construct of interest, i.e., programmer experience. Thus, the elimination of those variables looks justified and, in turn, we obtain a reduction in the variance of the model residuals and thus more power to identify significant effects.

Note that this is not a single-phase process; it is repeated as often as necessary to output the best model whose variables do not have serious collinearity problems. In our case, the final regression model was output after three rounds, as shown in Appendix 6. The regression model that meets the collinearity conditions is composed of 12 predictor variables, as shown below:

DEPENDENT VARIABLE =

β_1*SITE +

β_2*CS_DEGREE +

β_3*EXPERIENCE_UNIT_TESTING_FRAMEWORK_LIKERT_SCALE +

β_4* EXPERIENCE_EXPERIMENT_PROGRAMMING_LANGUAGE_ACADEMIA_YEARS +

β_5*EXPERIENCE_EXPERIMENT_PROGRAMMING_LANGUAGE_INDUSTRY_YEARS +

β_6*OVERALL_EXPERIENCE_PROGRAMMING_ACADEMIA_YEARS +

β_7*OVERALL_EXPERIENCE_PROGRAMMING_INDUSTRY_YEARS +

B_8*EXPERIENCE_UNIT_TESTING_LIKERT_SCALE +

β_9*EXPERIMENT_IDE_USED_DUMMY +

β_{10}*TDD_USED_DUMMY +

β_{11}*TASK_ITLD +

β_{12}*SLICED_ITLD_DUMMY + Error

5.2.2 Determining the Sample Size Necessary in Order to Achieve a Statistical Power of 80 %

There are many ways of determining the minimum sample size for a regression model. The most often used are based on: 1) number of model predictors or 2) the effect size and expected statistical power.

- Determining the sample size depending on the number of predictors

  Green (1991) suggests two heuristic rules for determining an acceptable sample size. The first refers to the overall goodness of fit of the model and the second to the goodness of fit of each of the independent variables in the model.

  1. *Overall goodness of fit of the regression model.* A rule of thumb often used to determine overall goodness of fit is that the required sample size for k variables is $n = 50 + 8*k$.

2. *Goodness of fit for each independent variable in the model.* The suggested minimum samples size is $n = 104 + k$.

   As we have 12 independent variables, we would need approximately $50 + 8*12 = 146$ subjects for a good overall model fit, whereas we would need $104 + 12 = 116$ experimental subjects in order to detect a significant effect for each predictor variable. The two heuristic rules do not appear to be consistent (it does not make sense that the overall goodness of fit of a model should be more demanding than for the 12 individual predictors). On this ground, we also use other methods to estimate the sample size later. In any case, the required sample size is consistent with the number of subjects[3] in our dataset.

- Determining sample size depending on the effect size

Apart from using the number of predictors, it is possible to determine the sample size depending on the effect size and required statistical power. There are several ways of conducting this analysis. The most common one is to use specialized tools like G*Power (Faul et al. 2007). In this case, for 12 predictor variables, with a moderate effect size ($f^2 = 0.15$) and a statistical power of 80 % (which is usually required to consider the results of an empirical study to be reliable), we would need 127 subjects for a good overall regression model fit.

On the other hand, Miles and Shevlin (2001) propose some very useful plots that illustrate the sample sizes required to achieve a power of 80 % for different effect sizes and predictor numbers. In order to detect a moderate effect size with 12 variables, we would need approximately 150 experimental subjects. A large effect requires only 60. In sum, we believe that the available 126 (in actual fact 115) subjects are enough to detect moderate effects with a statistical power very close to 80 %. Additionally, as the sample size is large enough, we avoid the risk of overfitting. Overfitting occurs when the model is a very good fit for the data because there are a large number of independent variables with respect to number of cases/observations. This does not appear to apply in our case.

5.3 Results of Model Application

Tables 12 and 13 show the results of the model regression for both QLTY and PROD, respectively. Note that the observed patterns and effects are quite similar with respect to both quality and productivity. In both cases, the models were significant, with $R^2 = 0.339$ and $R^2 = 0.422$, respectively. It is thus possible to interpret the results for each independent variable reported below.

5.3.1 Quality

As Table 12 shows, none of the programming experiences, except OVERALL_ EXPERIENCE_ PROGRAMMING_ ACADEMIA_YEARS, have a significant effect:

- Experience in the specific programming language used in the experiment in industry (EXPERIENCE_EXPERIMENT_PROGRAMMING_LANGUAGE_INDUSTRY_

[3] Although we had 126 experimental subjects, 11 observations were lost during the analysis as two subjects failed to complete the experimental task, six failed to report their academic qualifications and four failed to report any experience. Consequently, we were only able to effectively process 115 cases.

Table 12 MLR results for QLTY

Model		Unstandardized Coefficients		Standardized CoefficientsBeta	t	*p*-val.	95.0 % Confidence Interval for B		Effect Size (d)	Collinearity Statistics	
		B	Std. Error				Lower Bound	Upper Bound		Tolerance	VIF
1	(Constant)	−58.384	29.399		−1.986	.050	−116.697	−.070			
	SITE	32.447	9.901	.382	3.277	.001	12.809	52.085	.646	.477	2.095
	CS_TITLE	18.813	9.412	.196	1.999	.048	.145	37.482	.394	.673	1.485
	EXPERIENCE_UNIT_TESTING_FRAMEWORK_LIKERT_SCALE	12.411	8.489	.166	1.462	.147	−4.428	29.249	.288	.501	1.998
	EXPERIENCE_EXPERIMENT_PROGRAMMING_LANGUAGE_ ACADEMY_YEARS	−.761	1.786	−.050	−.426	.671	−4.303	2.780	−.084	.474	2.109
	EXPERIENCE_EXPERIMENT_PROGRAMMING_LANGUAGE_ INDUSTRY_YEARS	.793	1.945	.057	.408	.684	−3.064	4.650	.080	.330	3.029
	OVERALL_EXPERIENCE_PROGRAMMING_ACADEMY_YEARS	3.599	1.209	.308	2.976	.004	1.200	5.997	.586	.604	1.657
	OVERALL_EXPERIENCE_PROGRAMMING_INDUSTRY_YEARS	1.085	1.033	.153	1.051	.296	−.963	3.134	.207	.306	3.267
	EXPERIENCE_UNIT_TESTING_LIKERT_SCALE	−11.256	7.260	−.191	−1.550	.124	−25.656	3.143	−.305	.428	2.339
	EXPERIMENT_IDE_USED_DUMMY	18.514	8.810	.212	2.102	.038	1.040	35.989	.414	.639	1.565
	TDD_USED_DUMMY	−.463	9.917	−.004	−.047	.963	−20.133	19.206	−.009	.756	1.323
	TASK_ITLD	8.332	13.385	.092	.622	.535	−18.218	34.881	.123	.294	3.400
	SLICED_ITLD_DUMMY	31.962	13.330	.354	2.398	.018	5.521	58.403	.473	.297	3.372

Dependent Variable: QLTY

Table 13 MLR results for PROD

Model		Unstandardized Coefficients		Standardized Coefficients	t	*p*-val.	95.0 % Confidence Interval for B		Effect Size (d)	Collinearity Statistics	
		B	Std. Error	Beta			Lower Bound	Upper Bound		Tolerance	VIF
1	(Constant)	−61.699	24.927		−2.475	.015	−111.142	−12.257			
	SITE	20.252	8.394	.263	2.412	.018	3.601	36.902	.475	.477	2.095
	CS_TITLE	14.333	7.980	.165	1.796	.075	−1.496	30.162	.354	.673	1.485
	EXPERIENCE_UNIT_TESTING_FRAMEWORK_LIKERT_SCALE	13.245	7.198	.196	1.840	.069	−1.031	27.522	.363	.501	1.998
	EXPERIENCE_EXPERIMENT_PROGRAMMING_LANGUAGE_ ACADEMY_YEARS	−1.938	1.514	−.140	−1.280	.203	−4.941	1.065	−.252	.474	2.109
	EXPERIENCE_EXPERIMENT_PROGRAMMING_LANGUAGE_ INDUSTRY_YEARS	−.574	1.649	−.046	−.348	.728	−3.844	2.696	−.068	.330	3.029
	OVERALL_EXPERIENCE_PROGRAMMING_ACADEMY_YEARS	4.345	1.025	.410	4.238	.000	2.311	6.379	.835	.604	1.657
	OVERALL_EXPERIENCE_PROGRAMMING_INDUSTRY_YEARS	.519	.876	.081	.593	.554	−1.217	2.256	.117	.306	3.267
	EXPERIENCE_UNIT_TESTING_LIKERT_SCALE	−5.160	6.155	−.096	−.838	.404	−17.370	7.049	−.165	.428	2.339
	EXPERIMENT_IDE_USED_DUMMY	17.573	7.470	.221	2.353	.021	2.757	32.389	.464	.639	1.565
	TDD_USED_DUMMY	−9.295	8.408	−.096	−1.106	.272	−25.972	7.382	−.218	.756	1.323
	TASK_ITLD	12.029	11.349	.147	1.060	.292	−10.482	34.540	.209	.294	3.400
	SLICED_ITLD_DUMMY	28.777	11.303	.352	2.546	.012	6.358	51.196	.502	.297	3.372

Dependent Variable: PROD

YEARS) and in academia (EXPERIENCE_EXPERIMENT_PROGRAMMING_ LANGUAGE_ACADEMIA_YEARS) are nowhere near statistical significance (p-value = 0.671 and 0.684, respectively) and have a very small and practically negligible effect ($\beta_4 = -0.76$ and $\beta_5 = 0.79$ respectively, which is equivalent in the independent variable metric to increases or decreases of −0.76 % and 0.79 % per year, respectively). The same could be said about overall programming experience gained by subjects in industry (OVERALL_EXPERIENCE_ PROGRAMMING_ INDUSTRY_YEARS).

- On the other hand, overall programming experience gained in academia (OVERALL_EXPERIENCE_ PROGRAMMING_ ACADEMIA_YEARS) has a clearly significant (p-value = 0.004) moderate effect (3.6 % per year).
- Experience in the unit testing framework (EXPERIENCE_UNIT_TESTING_ FRAMEWORK_ LIKERT_SCALE) has a relatively large effect on quality compared to the other experience variables. The quality of the product output by subjects improved according to experience level (i.e., "novice" as opposed to "no experience" or "intermediate" as opposed to "novice"). Additionally, although not significant, the p-value is relatively small (0.147). This variable is not significant because of its high standard error (8.5), which is probably due to this variable being measured on a Likert scale. This suggests that this variable actually does have an impact on the quality of the code produced by programmers.
- Unit testing (EXPERIENCE_UNIT_TESTING_LIKERT_SCALE) has a similar pattern to EXPERIENCE_UNIT_TESTING_FRAMEWORK_LIKERT_SCALE, albeit in the opposite direction. The variable has a sizeable, but negative, effect (−11.25 %). The p-value is also quite low, although not significant (0.124) because of the high standard error associated with the variable (7.3), which was again measured on a Likert scale.

Apart from the variables directly related to programmer experience, the analysis also yielded results related to other influential variables, all of which, except for subject academic background (CS_DEGREE), are moderator variables:

- Subjects' academic background (CS_DEGREE) has a statistically significant (p-value = 0.048) and big positive effect ($\beta_2 = 18.8$). Subjects with specialized training in computer science tend to produce products whose quality is 18.8 % better than non-computer scientists.
- Subject typology (students vs. professionals) or, rather, the SITE where the quasi-experiments were run (academia vs. industry) has a statistically significant (p-value = 0.001) and marked positive influence ($\beta_1 = 32.0$). The industry subjects tend to output better quality code than students.
- When subjects are familiar with the use of the IDE used in the experiments (EXPERIMENT_IDE_USED_DUMMY), it has a statistically significant (p-value = 0.038) and positive effect ($\beta_9 = 18.5$). In other words, code quality improves if subjects have used the IDE before.
- The use of sliced specifications (SLICED_ITLD_DUMMY) has a big positive influence ($\beta_{12} = 31.96$) on quality irrespective of the task completed (TASK_ITLD), which is not significant. The extent to which TDD skills might improve the quality of programmer output (remember that the treatment was an Iterative test-last strategy) also turned out not to be significant.

The results of the MLR cannot be graphically displayed, due to the existence of multiple independent variables (the corresponding scatter plot would be 13-dimensional). However, we can create scatter plots for the most interesting variables (overall programming experience, both for industry and academy), provided that we plot them independently, using the model residuals (which is probably debatable from the statistical viewpoint, but revealing). The strategy is the following:

1. We have created a predictive model including all the influential variables (e.g.: SITE, CS_DEGREE, etc.) for quality, with the exception of the OVERALL_ PROGRAMMING_EXPERIENCE_ACADEMY _YEARS.
2. We have obtained the residuals of the model. The residuals represent the original data, once the influence of the statistically significant variables (all the model variables, actually) has been removed.
3. We have plotted the model residuals against the variable OVERALL PROGRAMMING EXPERIENCE ACADEMY YEARS.

A similar procedure has been applied to the variable OVERALL PROGRAMMING EXPERIENCE INDUSTRY YEARS. The corresponding scatter plots are shown in Fig. 2. It can be easily perceived that the point cloud has an appreciable ascending direction. The regression lines confirm the visual impression. The variable OVERALL_EXPERIENCE_PROGRAMMING_ACADEMY_YEARS is strongly correlated with quality ($r = 0.26$). Correlation is statistically significant. In turn, OVERALL_EXPERIENCE_PROGRAMMING_INDUSTRY_YEARS is weakly correlated with quality ($r = 0.13$), and this correlation is non-significant (p-value $= 0.15$).

5.3.2 Productivity

The results with respect to Productivity reported in Table 13 are more or less that same as the above, although they differ as to the specific values. There are only two new noteworthy points:

- The significance associated with testing framework experience (EXPERIENCE_ UNIT_TESTING_ FRAMEWORK_LIKERT_SCALE) is p-value $= 0.069$, that is, very

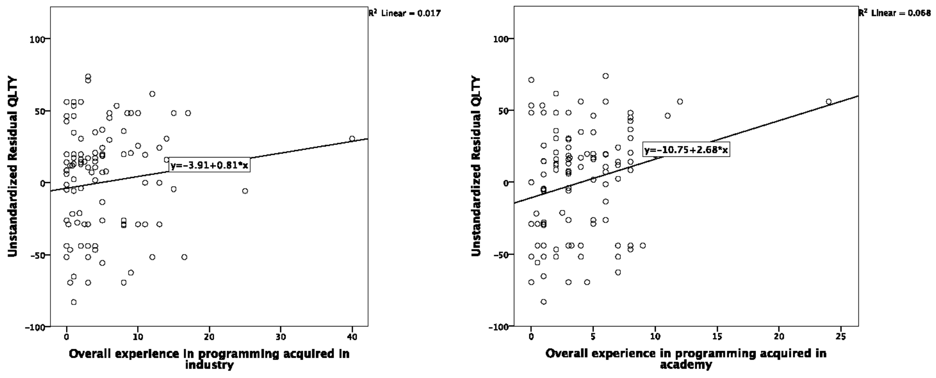


Fig. 2 Correlation between industry/academy experience and the residuals of the linear model containing the variables SITE, CS_DEGREE, EXPERIMENT_IDE_USED_DUMMY and SLICED_ITLD_DUMMY

nearly significant. This strengthens our belief that this variable does have an influence on both code quality and productivity (effect = 13.25 %).

- The statistical significance of unit testing experience (EXPERIENCE_UNIT_TESTING_LIKERT_SCALE) is much greater (p-value = 0.404). The simplest, albeit not altogether convincing, explanation is that unit testing experience does not affect productivity, despite it downgrading code quality.

Figure 3 shows the scatter plot for the overall programming experience, both for industry and academy, using the same strategy than in previous section. The variable OVERALL_EXPERIENCE_PROGRAMMING_ACADEMY _YEARS is strongly and significantly correlated with productivity ($r = 0.349$). The correlation between OVERALL_EXPERIENCE_PROGRAMMING_INDUSTRY_YEARS and productivity is virtually zero.

5.4 Normality and Homoscedasticity Examination

The MLR has two requirements: (1) the model residuals should be normally distributed and (2) the variance should be the same across all independent variable levels. These conditions are studied below.

5.4.1 Normality of Model Residuals

We used the Lilliefors-corrected Kolmogorov-Smirnov and the Shapiro-Wilk tests in order to test for the normality of model residuals, and Q-Q plots to illustrate the tests results.

The Kolmogorov-Smirnov test shows that the residuals are normal (p-value = .200 > 0.05) for both Quality (QLTY) and Productivity (PROD), as shown in Table 14. The Shapiro –Wilk test (which is better suited for small sample sizes) returns a similar result. Skewness and kurtosis statistics are within the range ± 1, as expected for normal distributions.

Q-Q plots simultaneously plot for each data point the observed residual value against the standardized residual value. If the residuals are normally distributed, the points are arranged on a straight line (bisecting the coordinate axes). Fig. 4a and b show that the residuals for both QLTY and PROD line in a more or less a straight line. Q-Q plots confirm that the residuals follow a normal distribution.

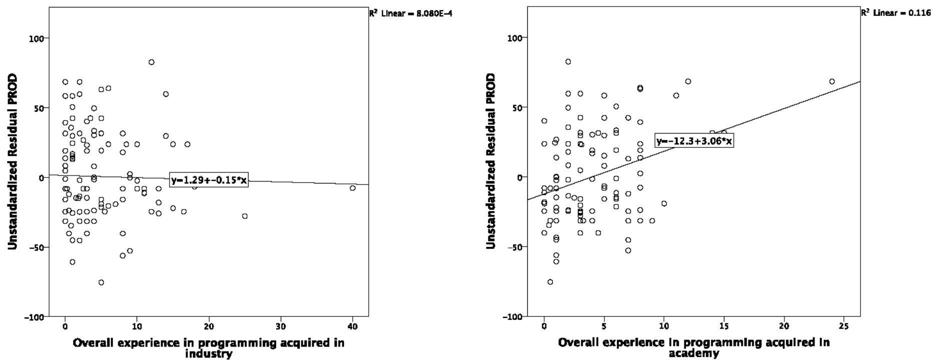


Fig. 3 Correlation between industry/academy experience and the residuals of the linear model containing the variables SITE, EXPERIMENT_IDE_USED_DUMMY and SLICED_ITLD_DUMMY

Table 14 Normality tests

	Statistic		Kolmogorov-Smirnov[a]			Shapiro-Wilk		
	Skewness	Kurtosis	Statistic	df	*P*-val.	Statistic	df	*P*-val.
QLTY	.023	−.470	.076	115	.096	.986	115	.261
PROD	.310	.021	.072	115	.200[b]	.989	115	.464

[a] Lilliefors-corrected significance

[b] This is a lower bound of the true significance

5.4.2 Testing for Homoscedasticity

This condition can be tested visually using a scatter plot of the predicted and expected values of the standardized residuals. As the plots in Fig. 5a and b show, the variance is quite uniform across the range of standardized predicted values in both cases. Thus, the data meet the homoscedasticity or equality of variances condition for both Quality (QLTY) and Productivity (PROD). Note that this effect is clearer for PROD than for QLTY. For QLTY, there is a region to the left of the plot with missing data points. This could pose a validity threat, as discussed in Section 7.

6 Nonlinear Analysis

Multiple linear regression (MLR) is often used for the analysis of large datasets. In this research, the directional character of the research question (/hypotheses) and the existence of multiple independent variables make MRL the best-suited analysis method. However, MLR has two limitations:

- The existence of potential nonlinear effects: It is possible that the relationship between experience and performance has e.g., a bell-like shape, i.e., growing up to certain point (e.g., 40 years old), and decreasing both to left and right. A linear regression model would report a null effect in this case. The decision tree could split the dataset in three groups: left, middle and right side, along with their respective averages.

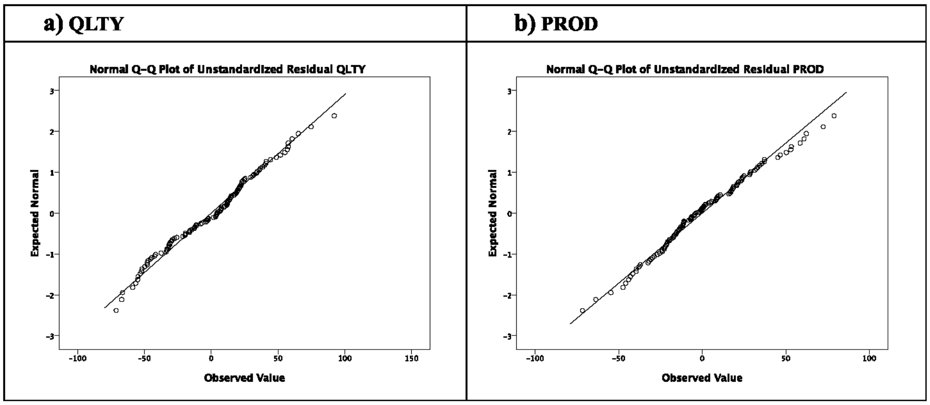


Fig. 4 Q-Q plot of residuals (**a**. QLTY, **b**. PROD)

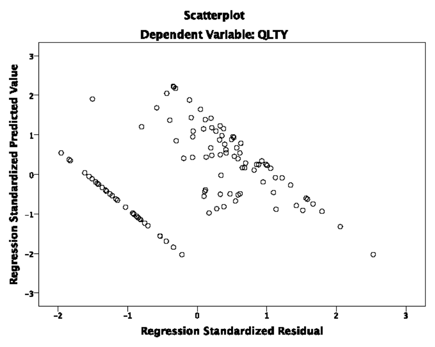
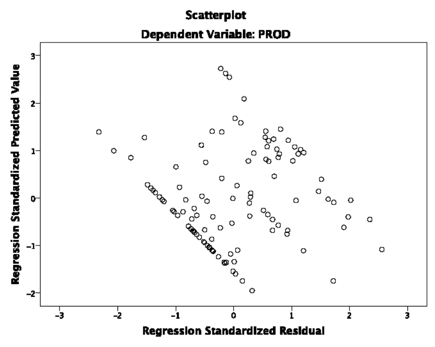
a) QLTY	**b) PROD**
	

Fig. 5 Scatter plot (**a**. QLTY, **b**. PROD)

- Interactions when dummies or ordinal variables are involved: It is easy to define the interactions between scale variables (i.e., multiplying them into a new variable which represents the interaction). However, dummies and ordinal variables have arbitrary numerical codes. In this case, the multiplication makes little sense. That is the reason why we have not included interactions in the MLR (in addition to a propensity to analyse main effects only with limited size datasets).

In the following sections, we include the CART decision trees for the response variables QLTY and PROD, respectively. This statistical procedure has been previously outlined in Section 4.4.

6.1 Quality

In order to create a decision tree, the researcher has some freedom to define the analysis parameters, such as the max tree depth and the minimal number of cases per node. Choosing one of another value yields different (although related) results.

We have set the max tree depth to 5. This is the default value in SPSS. We have tested different values for the number of cases. The respective decision trees are shown in Appendix 10. Trees with few levels are uninformative. Very complex trees (many nodes and levels) are difficult to interpret. We examined the different trees and chose those with average complexity (3–5 levels and 2 nodes per level). The most informative trees were obtained setting the number of cases to 12 for parent nodes, and 6 for child nodes). It is noticeable that, according to Glenwick (2016), for small datasets the optimum number of cases is 10 % and 5 % of the sample size for parent and child nodes, respectively. The values that we have chosen match exactly these percentages (our sample size is $N = 124$).

Figure 6 shows the decision tree for the QLTY response variable. The tree has 5 levels (including the root node). This root node defines the average quality for the entire population (59 %). As we move down from the root node, we find subpopulations defined by values of the independent variables exhibiting different quality averages.

The second level is defined by the SLICED_ITLD_DUMMY variable. This variable represents whether the subjects have used a sliced specification during the experimental sessions. The subjects that have used a sliced specification (SLICED_ITLD_DUMMY > *no*, that is, SLICED_ITLD_DUMMY = *yes*) obtain 80 % quality in average. The average quality for non-sliced specifications is considerably lower (50 %).

Two variables define the third level: OVERALL_EXPERIENCE_PROGRAMMING_INDUSTRY_YEARS and OVERALL_EXPERIENCE_PROGRAMMING_ACADEMY_YEARS for non-sliced and sliced specifications, respectively. This result is equivalent to the existence of two interactions SLICED_ITLD_DUMMY *x*

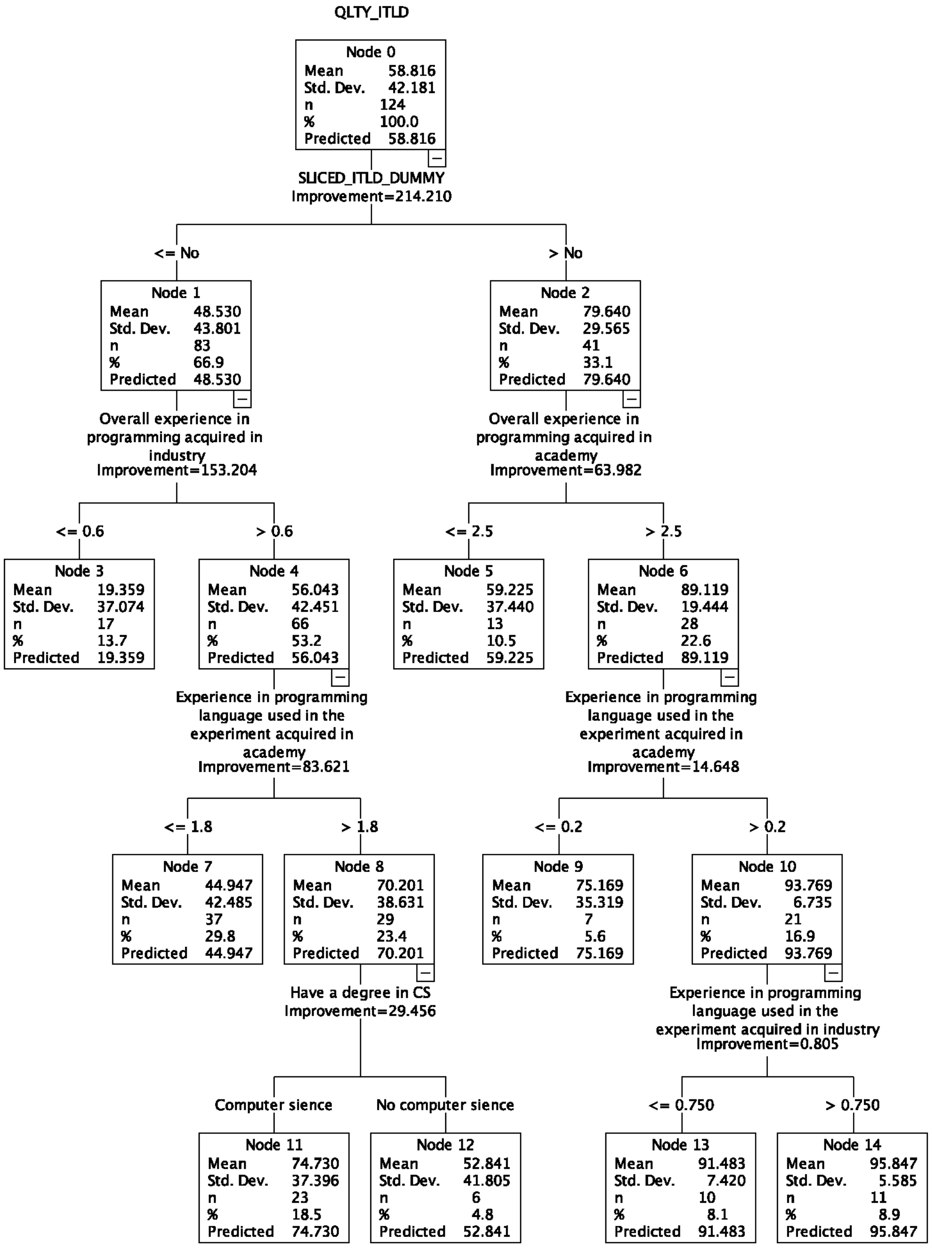

QLTY_ITLD
Node 0 Mean 58.816 Std. Dev. 42.181 n 124 % 100.0 Predicted 58.816
SLICED_ITLD_DUMMY Improvement=214.210
<= No
> No
Node 1 Mean 48.530 Std. Dev. 43.801 n 83 % 66.9 Predicted 48.530
Node 2 Mean 79.640 Std. Dev. 29.565 n 41 % 33.1 Predicted 79.640
Overall experience in programming acquired in industry Improvement=153.204
Overall experience in programming acquired in academy Improvement=63.982
<= 0.6
> 0.6
<= 2.5
> 2.5
Node 3 Mean 19.359 Std. Dev. 37.074 n 17 % 13.7 Predicted 19.359
Node 4 Mean 56.043 Std. Dev. 42.451 n 66 % 53.2 Predicted 56.043
Node 5 Mean 59.225 Std. Dev. 37.440 n 13 % 10.5 Predicted 59.225
Node 6 Mean 89.119 Std. Dev. 19.444 n 28 % 22.6 Predicted 89.119
Experience in programming language used in the experiment acquired in academy Improvement=83.621
Experience in programming language used in the experiment acquired in academy Improvement=14.648
<= 1.8
> 1.8
<= 0.2
> 0.2
Node 7 Mean 44.947 Std. Dev. 42.485 n 37 % 29.8 Predicted 44.947
Node 8 Mean 70.201 Std. Dev. 38.631 n 29 % 23.4 Predicted 70.201
Node 9 Mean 75.169 Std. Dev. 35.319 n 7 % 5.6 Predicted 75.169
Node 10 Mean 93.769 Std. Dev. 6.735 n 21 % 16.9 Predicted 93.769
Have a degree in CS Improvement=29.456
Experience in programming language used in the experiment acquired in industry Improvement=0.805
Computer sience
No computer sience
<= 0.750
> 0.750
Node 11 Mean 74.730 Std. Dev. 37.396 n 23 % 18.5 Predicted 74.730
Node 12 Mean 52.841 Std. Dev. 41.805 n 6 % 4.8 Predicted 52.841
Node 13 Mean 91.483 Std. Dev. 7.420 n 10 % 8.1 Predicted 91.483
Node 14 Mean 95.847 Std. Dev. 5.585 n 11 % 8.9 Predicted 95.847

Fig. 6 CART decision tree for QLTY

OVERALL_EXPERIENCE_PROGRAMMING_INDUSTRY_YEARS and SLICED_ITLD_DUMMY *x* OVERALL_EXPERIENCE_PROGRAMMING_ ACADEMY_YEARS. These interactions have not been considered in the MLR. In the case of non-sliced specifications:

- Subjects with very little industry programming experience (less than 0.6 years) perform poorly (quality = 19 %). Subjects above 0.6 years obtain average quality values (56 %).
- Among the 83 subjects that used a non-sliced specification, there are both students and professionals. However, academy-related experience does not play a role in this level/branch. This suggests that subjects with industry experience can use regular, real-life (non-sliced) specifications effectively, whereas subjects with longer academic experience (probably, the students themselves) perform better with more detailed (sliced) specifications (see below).
- It is also noticeable that the variable OVERALL_EXPERIENCE_PROGRAMMING_ INDUSTRY _YEARS does not show a significant main effect (that is, by itself, without considering the type of specification) in the MLR.

In the case of sliced specifications:

- Subjects with more than 2.5 years programming experience in academia obtain rather high quality scores (89 %). The scores situate in the average for lower experiences (59 %).
- As above, among the 41 subjects in this group, there are both professionals and students. However, the subjects that take more advantage of the sliced specifications are the ones with longer programming experience obtained during their academic training.

The fourth level is defined by the variable EXPERIENCE_EXPERIMENT_ PROGRAMMING _LANGUAGE_ACADEMY_YEARS, regardless of the tree branch. The direction of the effect is as expected: longer experiences increase quality scores by 19–25 %. However, this increment applies only to experienced (either in industry or academy) subjects. Notice that this variable does not show a significant effect in the MLR.

The firth level is defined by the CS_DEGREE and EXPERIENCE_EXPERIMENT_ PROGRAMMING_ LANGUAGE_INDUSTRY_YEARS. Holding a CS degree makes a big difference in the average scores (22 % difference). The impact of the industry experience in the programming language used in the experiment is negligible (4 % difference).

In general, the results of the CART decision tree are aligned to the MLR. The most influential variable is the sliced character of the specification. This variable has the 2nd larger effect size in the MLR, and it appears at the top level in the decision tree. The overall programming experience obtained in academy and holding a CS degree also show beneficial effects both in the MLR and the decision tree.

There are some differences between the MLR and the CART decision tree. The site where the experiment was conducted and the actual usage of the IDE used during the experiment do not appear as explanatory variables in the CART tree. In turn, the overall programming experience in academy appears to be influential, although limited to non-sliced specifications. The experience in the programming language used in the experiment obtained in academy has an influential effect also, but only for experienced (either industry or academy) subjects.

6.2 Productivity

We have used the same values for the max tree depth, and number of cases per node, than in the previous section. The corresponding tree is displayed in Fig. 7.

The tree has only 4 levels, and a much simple splitting pattern than Fig. 6. The grand mean is 39 %. The second level is defined by SLICED_ITLD_DUMMY variable. The subjects that used a sliced specification perform better (36 % difference in average) than those who used a regular

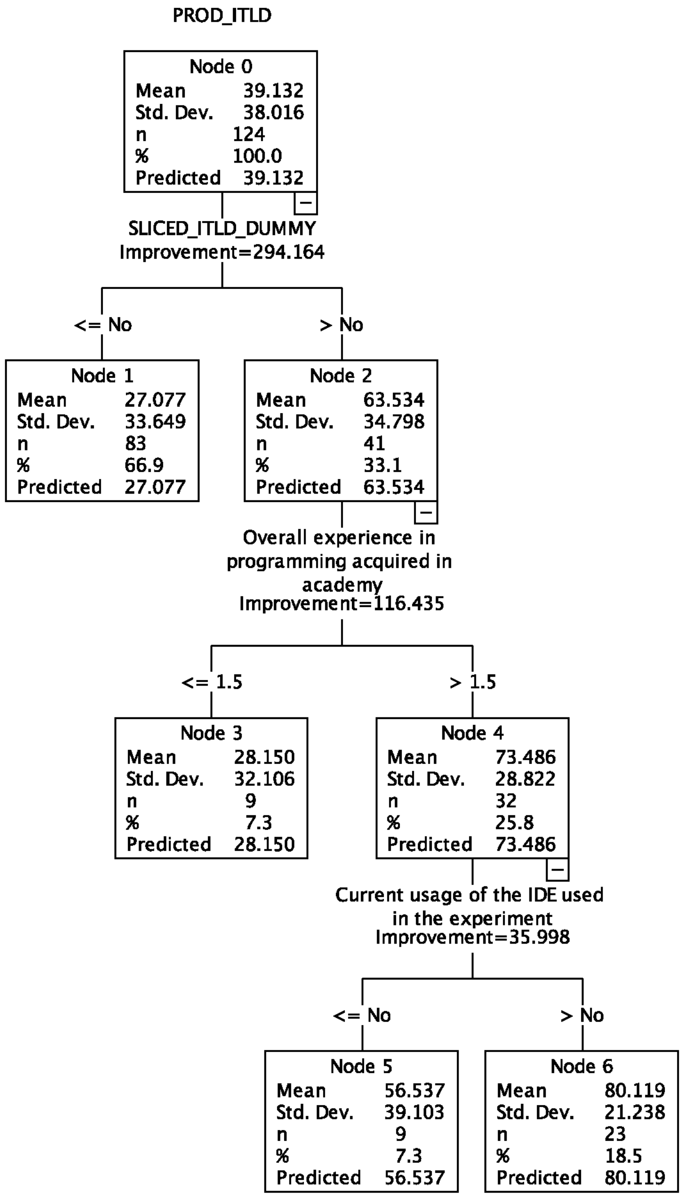


Fig. 7 CART decision tree for PROD

specification. The subjects that used a sliced specification can be further divided at the third level depending on their overall programming experience acquired in academia. Again, those subjects with longer experiences (1.5 or more years) perform dramatically better (45 % difference) than inexperienced ones. Finally, the fourth level is defined by the actual usage of the experimental IDE, exhibiting smaller but also considerable improvement (24 % difference).

The coincidences with the MLR are almost perfect. All variables, with the exception of SITE, that yielded significant results in the MLR also appear as influential in the CART decision tree. It is also noticeable that the second and third levels in Fig. 7 replicate the right branch in Fig. 6. This suggest that the most influential variables are independent of the measurement procedure (i.e., the concrete response variable used).

7 Discussion

7.1 Preliminary Considerations

Before trying to interpret the results, it is worth considering whether: (1) the measurement of experience (in years) yield different results than the measurement of experience using Likert scales, and (2) whether there are any systematic differences (note, for example, that the analysis omitted the EXPERIMENT_CODE variable) between experiments that rule out joint analysis and pose a threat to the validity of the results.

With regard to the first question, Appendix 7 reports the MLR analysis in which experience measured in years was replaced by variables measured on a Likert scale. The observed trends in terms of both β-values and statistical significance are exactly the same. Indeed, the standard error for the EXPERIENCE_EXPERIMENT_PROGRAMMING_LANGUAGE_ LIKERT_SCALE and OVERALL_EXPERIENCE_PROGRAMMING_LIKERT_SCALE variables is inflated with respect to their equivalent values measured in years. This worsens the detection of significant effects.

With regard to the second question, Appendix 8 analyses the model residuals against the EXPERIMENT_CODE variable. We noted in Section 5 that the model residuals were normal and, consequently, had zero mean and random but constant variance. The boxplots charting the residuals by experiment appear to follow the same pattern: each box is centred around zero, and the Q1-Q3 ranges are almost equal (note that there are not many subjects in each experiment, so exact matches are unlikely). The results of the tests of the equality of means (a univariate ANOVA) are not significant. This implies that the quality or productivity does not depend on the concrete company or university were the quasi-experiments were conducted. The Levene test is significant for Quality, but non-significant for Productivity. Nevertheless, it is not a surprise that that the quasi-experiments have different variances due to sample size and diversity of the underlying populations. It appears, therefore, that the data can be jointly analysed and interpreted.

7.2 Effect of Experience

The results of the Multiple Linear Regression (MLR) suggests that programming experience (except for OVERALL_EXPERIENCE_PROGRAMMING_ACADEMIA_YEARS) is not related to better programmer performance (in terms of quality or productivity). In turn, the impact of the programming experience gained in academia is considerable. In terms of percentages, each training year adds around 4 % increment in both quality and productivity, i.e., 3 years of programming experience gained by subjects during their degree (a reasonable assumption) implies that the code contains 12 % less errors (in average). In terms of Cohen's effect size, these values represent a medium effect size for quality ($d = 0.59$) and a large effect size for productivity ($d = 0.84$).

These results appear to be consistent with more modern theories of experience (Ericsson 2006a) that make a distinction between *length of service* (which does not lead to expertise) and *deliberate and intensive practice* (which does lead to expertise):

- The experience gained in industry could (generally) be considered as a *routine*. Professionals are expected to "do their job" within some standard limits of quality and productivity, e.g.: the average company defect rate. Although at the individual level programmers can attend to training courses and/or self-educate to beat those limits, such improvement is not likely intensive enough (e.g., not performed daily for several hours), because the daily work is priority, and the remaining (/spare) time is usually filled up with personal or family activities.
- In academia, students perform programming tasks within training courses. In turn, these courses are typically designed in such a way that: (1) new topics are introduced progressively; (2) the difficulty of the tasks, e.g., programming assignments, increase with time and (3) students make every effort, every day during the academic period, to get high grades. Thus, the salient feature of academia is deliberate and intensive training, and according to Ericsson's theory it should make an effect on performance, which is exactly what we have observed in this research.

Of course, the problem is how to reconcile our results with the findings of previous programming studies. We cannot, of course, rule out error on our part. However, we can venture a hypothesis. Tables 5 to 9 show the average quality and productivity achieved by subjects depending on their programming experience and site (academia, industry), *without considering the other variables in the analysis*. As said before, these tables should be used merely to *identify trends* and not as an independent instance for analysis. However, the data reported are informative:

- Looking at the average values for academia, we find that there is a clear trend towards better performance as experience increases. This being true, studies that use students with different experience levels (e.g.: freshmen vs. seniors) could find significant differences between them. There is some evidence that the positive effects of experience become visible in this context, e.g., (Daun et al. 2015; Runeson 2003).
- In industry, the data plot has zigzag profile with no clear trends. However, the MLR yields a positive, significant effect for SITE, i.e., professionals perform better than students *in average* (see Section 5.3.1). Considering that several studies have been conducted comparing students with professionals, it is not surprising that they found that experience did have an effect.

The decision trees give a somewhat different picture. The most noticeable difference is the absence of the variable SITE in both trees, whereas SITE has a strong, statistically significant effect ($\beta > 20$ %, Cohen's $d > 0.4$) in the MLR for both quality and productivity. The reason for difference lies, most likely, in an interaction among variables.

The decision tree algorithm splits the root note using the variable that more clearly separates the original dataset into subsets. This variable is SLICED_ITLD_DUMMY (i.e., the sliced character of the specification), both for quality and productivity. This decision could be anticipated just by looking at the MLR tables, because SLICED_ITLD_DUMMY has the largest effect size, both for quality and productivity.

Further splitting is dependent upon the decisions taken in the higher level nodes i.e., they represent interactions. Here, the splitting pattern draws a distinction between quality and productivity.

- For quality, the 2nd level nodes are defined by two variables: OVERALL_EXPERIENCE_PROGRAMMING_INDUSTRY_YEARS (for non-sliced specifications) and OVERALL_EXPERIENCE_PROGRAMMING_ACADEMY_YEARS (for sliced specifications).
- For productivity, only the node corresponding to sliced specifications breaks down into two child nodes, defined again by the OVERALL_EXPERIENCE_PROGRAMMING_ACADEMY_YEARS variable.

We venture out that the positive, statistically significant effect for SITE in the MLR is related to the interaction SLICED_ITLD_DUMMY x OVERALL_EXPERIENCE_PROGRAMMING_INDUSTRY_YEARS and SLICED_ITLD_DUMMY x OVERALL_EXPERIENCE_PROGRAMMING_ACADEMY_YEARS. The MLR does not contain this interaction, so that SITE is assigned the variability associated to SLICED_ITLD_DUMMY x OVERALL_EXPERIENCE_PROGRAMMING_INDUSTRY_YEARS. The *p*-values in the MLR tables (see Sections 5.3.1 and 5.3.2) also back up this explanation: SITE has lower *p*-values (or higher effect size) for quality; OVERALL_EXPERIENCE_PROGRAMMING_INDUSTRY_YEARS emerges precisely in the quality decision tree).

If we accept that programming experience in industry has an effect when subjects use non-sliced specifications, the previous argumentation regarding the routine character of the experience in industry would be wrong; however, the impact of the experience in industry is rather low. The child nodes are split at 0.6 experience years, and only a fraction of subjects (17 vs. 66) are located the in low performing node. In other words, after 7 experience months, there are not substantial differences in practitioners' performance (in average).

We have not discussed further above the impact of the sliced character of the specification because it is secondary for this paper' research goal, but a quick look at Appendix 2 clearly shows that sliced specifications are more detailed and provide guidance to the programmers during the coding task. We expected that sliced specification exhibit higher quality and productivity scores. However, it is *somewhat* surprising that sliced specifications interact with programming experience. In our opinion, we are envisioning a domain knowledge effect here:

- Non-sliced specifications (not the ones we provide in Appendix 2 but comparable to some extent) are typically used in industry. After some time (our decision tree suggests 0.6 years), programmers get used to this type of specification and solve the corresponding task professionally.
- Students are not usually exposed to problem assignments where a lot of domain knowledge is required to enable resolution. Problem sheets are typically detailed (again, comparable to the specifications in Appendix 2, including hints and examples to ease understanding. Students get used to this type of documents after some time (2.5 years) and become proficient.

The influence of the type of specification represents, most likely, another confirmation of the *specificity* of the experience (Ericsson and Lehmann 1996). Subjects exhibit expertise in some domains only. Notice the restricted character of *domain*, which is linked in our case to specification types. Domain influence could extend to the types of tasks, development environment, etc as well. We discuss these issues below.

7.3 Effect of Other Variables

Three other variables (besides the sliced character of the specification) have shown a clear influence on programmer performance, although only the first was statistically significant: use of IDE, testing framework experience and unit testing experience.

The results for IDE usage and testing framework experience are not at all surprising. It is reasonable to assume that the use of proper tools should improve programmer performance. It is remarkable, however, that these variables (whose associated β-values are from 10 to 20 %) should have such a noticeable effect.

With regard to unit testing experience, we did not expect to find that it had negative effects. There are possible interpretations for this result. Subjects who are experienced in unit testing might pay more attention to quality and thus be less productive. However, the result of the MLR suggests just the opposite. Unit testing experience has a negative impact on quality (with a β-value of around −10 %, not far from statistical significance).

A possible alternative argument is that the testing activity and the programming activity are performed by different subject profiles, i.e., testers do not make code and programmers do not test. A logical implication of this assumption would be that testers (i.e., people with unit testing experience) achieve lower quality and productivity scores than programmers. This could be true: Quality decreases as unit testing experience increases, and although productivity has a large *p*-value, the associated β-value is negative, around −5 %. However, the correlations between unit testing experience and programming experience are substantial, positive ($r \approx 0.3$) and statistically significant. Also, the correlation between unit testing experience and testing framework experience is very high, positive ($r = 0.568$) and statistically significant. In other words: it seems that testers do know (at least in our sample) how to make code.

The reason why unit testing experience leads to decreasing quality and productivity is unclear for us; it requires further research.

8 Validity Threats

8.1 Threats to Statistical Conclusion Validity

- Homoscedasticity-related problems. Although the regression model for external quality satisfactorily meets the normality condition, we found, when testing for homoscedasticity, that the data were not uniformly distributed. Heteroscedasticity does not affect the estimation of the regression model coefficients (β-values), although it does influence statistical significance. We believe that this threat is not at work, as our results with respect to programming experience show that the associated effect sizes are very small. On this ground, although the statistical significances could be affected, we can likewise conclude that experience does not have a sizeable effect on code quality and subject productivity.
- Unbalancing in some independent variables. The parameter estimation could be subject to unbalanced groups (for example, academic background or use of IDE). However, although we cannot rule out this having a negative effect among variables, we believe that this threat does not challenge our main findings on two grounds: (1) the size and power of the regression models are

large enough, and they are significant, normal and reasonably homoscedastic, and (2) unbalancing does not affect the main variables concerning programming experience.

- Recoding of the *Experience in testing framework used in the experiment* variable. The process of recoding applied to the testing framework experience levels could cause some sort of bias. However, this should not happen on two grounds: (1) this variable has been removed from the model on collinearity grounds, and (2) we have found that, if introduced into the MLR without applying recoding (i.e., considering all three levels —gTest, jUnit and Boost Test—), this variable is still collinear and would therefore also have been removed from the model.
- Measurement bias. Each quasi-experiment was measured by a single measurer. More than one person should conduct the measurement process in order to improve measurement accuracy. In order to counteract this threat, we defined and gave experimental subjects API code templates for the experimental tasks. These code templates contain methods and parameters definitions that can be used to solve the experimental tasks. Those methods and parameters are also used by the test suites. Code templates reduce the manipulations that measurers need to make in the subjects' code, improving between-measurers accuracy.

8.2 Threats to Internal Validity

- Ambiguity surrounding the causality of the effects. Since this is quasi-experimental research, the conclusions cannot be interpreted in causal sense. In our research, we have studied several independent variables ($k = 12$) regarding experience or specialized knowledge for performing an experimental task. However, there could be moderator variables that we have not taken into account and that explain the results, e.g., variables referring to soft skills or programmer personality. The strategy that we used to counteract this was to measure all the moderator variables that looked as if they might realistically have an effect on code quality and programmer productivity. But, of course, we cannot be sure that we have considered all the relevant variables.
- Population heterogeneity. The results of this research may be threatened by combining several experimental populations with different characteristics, which could interact with experience and counteract the effects when analysed jointly. We believe that this threat is not at work.

  Individual experiments (in particular, industry experiments) have insufficient sample size by themselves for the regression model achieving a reasonable power. The approach that we have followed to assess whether the population coming from a given experiment (which, in turn, corresponds to a concrete company/university and moment in time) departs from the global behavior is the examination of the global regression model residuals at the experiment level. We have not detected substantial differences between the model residuals when they are studied separately by experiment (see appendix 8).

  Population subgroups can be defined on different grounds. Probably, the two most relevant (and meaningful) sub-populations are students vs. professionals. When they are analysed independently, the results are not exactly alike, but much the same; in particular, the lack of effect of industry experience, and the positive effect of academic training, does not change.

  On the other side, the mix of populations can be seen as a strength of our study, as the diversity of the populations increases external validity.
- Perturbations caused by the use of ITLD. Although ITLD is a very well-known and popular strategy among programmers, we cannot be sure that all the subjects were familiar with its use. This might lead to a change in the subjects' work method, which would affect their productivity and performance. We have applied two strategies to counteract this threat. First, we provided

specific training on ITLD before applying the treatment. Second, we did not oblige programmers to apply a particular ITLD variant; it was left up to them to apply whichever ITLD strategy they saw fit without this having any impact whatsoever on the response variable measurement.

- Perturbations caused by the use of specific IDEs or unit testing frameworks. A large proportion of subjects do not have experience with the IDE used during the experiment and/or unit testing. Although we have controlled these variables explicitly (notice that they have been included both in the multiple linear regression and decision tree analyses), we cannot rule out that experienced subjects perform particularly bad when they have to code in unfamiliar contexts (e.g., an IDE they are not very familiar with). This makes identifying experience effects more difficult.

8.3 Threats to Construct Validity

- *Nature of the experimental tasks.* We used the MarsRover API (MR) and Bowling Scorekeeper (BSK) experimental tasks. Both are basically algorithmic tasks. BSK uses some terms (e.g., strike, spare) with which the experimental subjects may not be familiar. These tasks were specified in two ways: *sliced* and *non-sliced*. We cannot rule out that these decisions may have biased our results. In order to counteract this threat, we included variables that represent the task and the specification type in the MLR analysis, which we trust will separate their effects from the effects of experience.

8.4 Threats to External Validity

- Effects of programming experience vs. domain knowledge. The area of expertise under study is *programming*. Programming is generally defined here as consisting of knowledge of programming languages, algorithms and strategies (e.g., dynamic programming), good practices (e.g., design patterns), some libraries (e.g., regex), etc. Programming could also be construed as meaning knowledge of how to perform a task in a specific domain, e.g., code a specific network controller. Our aim was to study the effect of programming experience and not the effect of domain knowledge (which is ultimately another facet of expertise). BSK and MR are outside of the domain of the experimental subjects, particularly professional programmers. By using tasks that are outside the programmer's domain, we have separated the effects of domain knowledge from the effects of programming experience. Therefore, our results: (1) should be interpreted *exclusively* in terms of the effect of programming experience (the results might differ if we used other, more familiar experimental problems), and (2) have greater external validity, as they are domain independent.
- Limitation of the number of experimental problems. We have only used two experimental problems (MR and BSK) so that the groups derived from the combination of treatments, tasks and blocking variables have the largest possible number of subjects. This improves the statistical analysis. On the other hand, our study has been conducted in a limited setting. Therefore, our results should be extrapolated to other contexts with due caution.

9 Conclusions

This paper studied the effects of different types of experience (academic background, programming experience, unit testing experience, and IDE and TDD use) on the performance of a set of 126 programmers from four companies and three universities across 10 quasi-experiments. The experimental design used separates the effects of domain knowledge from the effects of programming experience, which is the focus of this study.

The most important result is that years of experience are not able to predict programmer performance at all. The only exception is years of programming experience in academia (in other words, years of training), which does appear to have a positive influence on programmer performance. Other influential variables are testing framework experience and routine use of the IDE, which we believe reflects the positive influence of modern programming tools on programmer performance.

From another viewpoint, companies should give serious consideration to their programmer lifelong training, as the mere repetition of routine tasks does not improve their performance beyond mere competency. However, training courses may, or may not, contribute to increased performance. For instance, industry training courses tend to skip strict performance assessment, on social, psychological or labor law grounds. In turn, academic training is characterized by setting goals and thresholds, and reasonably strict assessment procedures. To what extent transferring academic strategies to industry could be successful? Which strategies have higher yields? Answering those questions require interdisciplinary research, from the perspectives of applied psychology, education, and software engineering disciplines.

From the viewpoint of the representativeness of our sample, as well as the statistical power and rigour of the analysis, the above results are reasonably reliable. There are, however, many open questions. The model's coefficient of determination ($R^2 \approx 0.4$) clearly indicates that there is a lot of unexplained variance. This variance is very likely to due programmers' personal characteristics (soft skills or personal traits). For example, the negative effects of unit testing experience appear to be related to the programmer's profile. We intend to explore this research line in the future.

Acknowledgments We would like to acknowledge Dr.Hakan Erdogmus who contributed to the design of one of the tasks used in the study (BSK) and the corresponding test cases. We also wish to acknowledge Mr. Timo Raty for his participation in the creation of the code templates for C++, and the training given in one of the quasi-experiments. We wish also acknowledge Mr. Adrian Santos for his support in the collection of the subjects' data.

Appendix 1: Description of the Independent Variables

Table 15 shows the 15 independent variables used in this research. The main aim of this appendix is to list each variable giving a brief description of the variable, its type (nominal, ordinal or dummy) and its respective levels. Section 3 details the types and measurement of variables.

Table 15 Independent variables

Independent variable	Description	Variable type	Levels (metric)
CS_DEGREE	Education of experimental subjects	Dummy	0. Non-computer science 1. Computer science
EXPERIENCE_EXPERIMENT_PROGRAMMING_LANGUAGE_ACADEMIA_YEARS	Years of programming language experience gained in academia	Scalar	Number of years
EXPERIENCE_EXPERIMENT_PROGRAMMING_LANGUAGE_INDUSTRY_YEARS	Years of programming language experience gained in industry	Scalar	Number of years
EXPERIENCE_UNIT_TESTING_LIKERT_SCALE	Unit testing experience measured on a Likert scale (1–4)	Ordinal	1. No experience (<2 years) 2. Novice (2–5 years) 3. Intermediate (5–10 years) 4. Expert (>10 years)
EXPERIMENT_IDE_USED_DUMMY	Knowledge of IDE use	Dummy	0. No 1. Yes
EXPERIMENT_PROGRAMMING_LANGUAGE	Programming language used in the experiment	Categorical	1. C++ 2. JAVA
OVERALL_EXPERIENCE_PROGRAMMING_ACADEMIA_YEARS	Years of overall programming experience gained in academia	Scalar	Number of years
OVERALL_EXPERIENCE_PROGRAMMING_INDUSTRY_YEARS	Years of overall programming experience gained in industry	Scalar	Number of years
SITE	Site at which the experiment was run	Dummy	0. Academia 1. Industry
SLICED_ITLD_DUMMY	Whether or not slicing was used in the task	Dummy	0. No 1. Yes
TASK_ITLD	Tasks that subjects had to solve	Categorical	1. MR 2. BSK
TDD_USED_DUMMY	Knowledge of TDD use	Dummy	0. No 1. Yes
TRAINER	Trainer for TDD	Dummy	1. Burak Turhan 2. Oscar Dieste
UNIT_TESTING_FRAMEWORK_LIKERT_SCALE	Framework used to write unit tests	Categorical	1. GTEST 2. BOOST 3. JUNIT
UNIT_TESTING_FRAMEWORK_LIKERT_SCALE_ADAPTED	Testing framework recoded in order to transform the categorical variable into a dummy variable	Dummy	2. BOOST 3. xUNIT

Appendix 2: Details of the Experiment

Specification for Mars Rover API with Slicing

Develop an API that moves a rover around a planet. The planet is represented as a grid with *x* and *y* coordinates. The rover is also facing in a direction. The direction can be north (*N*), south (*S*), west (*W*) or east (*E*). The input received by the rover is a string representing the commands it needs to execute.

The Planet

The planet on which the rover moves is represented as a square grid, with size (x, y).
Requirement: Define a planet of size (x, y).
Example: (100,100) creates a planet of size 100 × 100.

Landing

When the rover lands on the planet, it begins its journey at the start of the grid facing north.
Requirement: When the rover lands on the planet its position shall be (0,0) facing north.
Example: An empty command (i.e., "") to the rover returns its landing status (0,0,N).

Turning

The rover turns right or left. It remains in the same cell of the grid. Its direction changes accordingly.
Requirement: Compute the position of the rover after turning left (command "l") or right (command "r").
Example: A rover at position (0,0,N) is at position (0,0,E) after executing command "r". A rover at position (0,0,N) is at position (0,0,W) after executing command "l".

Moving

The rover moves forward or backward one grid cell in the direction that it is facing. The rover's direction does not change.
Requirement: Compute the position of the rover after moving forward (command "f") or backward (command "b") one grid cell.
Example: A rover at position (7,6,N) moves to (7,7,N) after executing a "f" command. A rover at position (5,8,E) moves to (4,8,E) after executing a "b" command.

Moving and Turning Combined

The rover shall be able to execute arbitrary sequences of “f”, “b”, “l” and “r” commands.

Requirement: Compute the position of the rover after executing a series of commands.

Example: A rover at position (0,0,N) moves to position (2,2,E) after executing “ffrff”.

Wrapping

Since the planet is a sphere the rover wraps at the opposite edge once it moves over it.

Requirement: Compute the position of the rover moving over the edges. The rover shall spawn on the opposite side.

Example: A rover on a planet of size 100 × 100, which moves backward (command “b”) after landing (remember that landing always takes place at position (0,0,N)) moves to position (0,99,N).

Positioning of Obstacles

Obstacles can be positioned on specific cells of the grid.

Requirement: Define the obstacles as a string (x1,y1) (*x*2,y2)… Place the obstacles on the grid.

Example: “(1,1) (4,5)” defines two obstacles, one at position (1,1) and another at position (4,5). Notice that the planet grid should be greater than or equal to 6 × 6.

Identifying a Single Obstacle

The rover might encounter (i.e., tries to move into) an obstacle. When it does it should report the obstacle and continue executing the remaining commands.

Requirement: Compute the position of a rover encountering an obstacle and report the obstacle. The same obstacle should be reported only once.

Example: A rover just landed (position (0,0,N)). There is one obstacle at planet coordinates (2,2). The rover executes “ffrfff” and reports (1,2,E) (2,2). Notice that the same obstacle is encountered twice but reported only once.

Identifying Multiple Obstacles

The rover might encounter multiple obstacles. When it does, it should report all of them once and in the order they were encountered.

Requirement: Compute the position of the rover encountering obstacles, and report the obstacles encountered in the order they are encountered. The same obstacle shall be reported only once.

Example: A rover just landed (position(0,0,N)). There are two obstacles at planet coordinates (2,2) and (2,1). The rover executes "ffrfffrflf" and reports (1,1,E) (2,2) (2,1). Notice that the first obstacle is encountered twice but reported only once.

A Tour Around the Planet

The rover goes on a tour around the planet encountering several obstacles, and wrapping in both axes.

Requirement: Compute the position of a rover that executes a series of commands that result in moving along both axes in both directions, encountering several obstacles and wrapping from both edges of the planet.

Example: The rover lands on a 6 × 6 planet with obstacles at (2,2), (0,5) and (5,0). It executes the command "ffrfffrbbblllfrfrbbl" and returns (0,0,N) (2,2) (0,5) (5,0).

Congratulations, you are done!

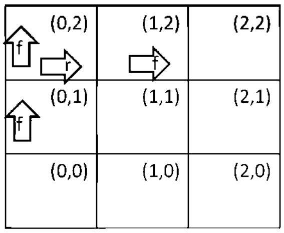
Specification for Mars Rover API without Slicing

The API manages a rover that moves on a planet (/squared grid) of arbitrary size (x,y). The rover starts the movement at position (0,0). The direction of the movement can be N (north), S (south), E (east) and W (west). The rover is north facing at the start.

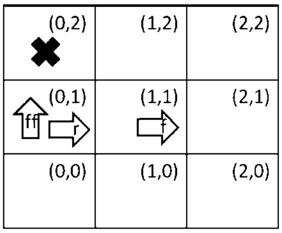
The rover receives a string of commands: l (left), r (right), f (forward) and b (backward). l and r change the rover's direction counter- and clockwise, respectively, but do not alter its position. f and b move the rover 1 position on the grid in or away from the direction that it is facing, respectively. The direction in which the rover is facing does not change. When the rover moves over the edges of the planet, it spawns on the opposite side.

The planet (/grid) may contain obstacles. Obstacles are defined as a list of coordinates "(obs1X, obs1Y) (obs2X, obs2Y)…" . When the rover finds an obstacle during a tour, it skips the current command (i.e., it does not move to the cell in which the obstacle is located) and continues to execute the remaining commands.

Upon processing the string of commands, the rover returns its position and direction in the format "(posX, posY, facing)". If obstacles are found, the output will be "(posX, posY, facing) (obs1X, obs1Y) (obs2X, obs2Y)…" The same obstacle shall be reported only once. Obstacles are reported in the order in which they are found.



Example of a rover's tour on a 3x3 planet in response to the command "ffrf". The starting position is (0,0) facing north. After the 1st f (forward) command, the rover moves to position (0,1) facing north. Subsequent commands keep the rover moving. The expected output is (1,2,E). With two more fs, the rover would spawn over the right edge to the final position (0,2,E).



Example of a rover's tour on a 3x3 planet in response to the command "ffrf", with one obstacle in position (0,2). After the 1st f (forward) command, the rover moves to position (0,1) facing north. The 2nd f command does not change the rover's position, because there is an obstacle in (0,2). This second f command is thus skipped. The expected output is (1,1,E)(0,2).

Specification for Bowling Score Keeper with Slicing

The objective is to develop an application that can calculate the score of a single bowling game using TDD. There is no graphical user interface. All that you will use in this assignment is the objects and JUnit testing. You will not need a main method.

The application requirements are divided into a set of user stories, which is as your to-do list. You should be able to incrementally develop a complete solution without an upfront comprehension of all the game's rules. For this exercise, don't read ahead, and handle the requirements one at a time in the stated order. Solve the problem using TDD, starting with the requirement for the first story. Remember to always lead with a test case, taking hints from the examples provided. Do not move to the next story until you have done with the last one. A story is done when you are confident that your program correctly implements the functionality stipulated by the requirement for the story. This means that *all* of your test cases for that story and *all* of the test cases for the previous stories pass. You may need to tweak your solution as you progress towards more advanced stories.

Frame

Each turn of a bowling game is called a frame. 10 pins are arranged in each frame. The goal of the player is to knock down as many pins as possible in each frame. The player has two chances, or throws, to do so. The value of a throw is given by the number of pins knocked down in that throw.

Story: As the scorekeeper, I want to be able to record a frame as composed of two throws. The first and second throws should be distinguishable.

Example: [2, 4] is a frame with two throws, in which two pins were knocked down in the first throw and four pins were knocked down in the second.

Frame Score

An ordinary frame's score is the sum of its throws.

Story: As the scorekeeper, I want to be able to compute the score of an ordinary frame after a player has rolled both throws.

Examples: The score of the frame [2, 6] is 8. The score of the frame [0, 9] is 9.

Game

A single game consists of 10 frames.

Story: As the scorekeeper, I want to define a game as a sequence of 10 frames.

Example: The sequence of frames [1, 5] [3, 6] [7, 2] [3, 6] [4, 4] [5, 3] [3, 3] [4, 5] [8, 1] [2, 6] represents a game. You may reuse this game from now on to represent and test different scenarios, modifying only a few frames each time.

Partial Game

When the player rolls a throw, the throw is automatically recorded in the correct frame.

Story: As the scorekeeper, when a player rolls throws, I want the game to keep track of the frames and figure out in which frame to place the next throw depending on the past throws. You think this is easy. Maybe for now. We'll see.

Example: If the game currently consists of the frames [1, 5] [3, 6] [7, 2] [3, ?] and the player rolls a throw with a value of 4, the game becomes [1, 5] [3, 6] [7, 2] [3, 4]. Another roll with a value of 5 transforms the game to [1, 5] [3, 6] [7, 2] [3, 4] [5, ?].

Game Score

The score of a bowling game is the sum of the individual scores of its frames.

Story: As the scorekeeper, I want to know a player's current game score at all times.

Example: The score of the game [1, 5] [3, 6] [7, 2] [3, 6] [4, 4] [5, 3] [3, 3] [4, 5] [8, 1] [2, 6] is 81. Partial scores are possible for an incomplete game if the frame scores are known up to the last complete frame. The score of the game [1, 5] [3, 6] [7, ?] is 15. The frame [7, ?] is not yet complete.

Strike

A frame is called a strike if all 10 pins are knocked down in the first throw. In this case, there is no second throw. A strike frame can be written as [10, 0]. The score of a strike equals 10 plus the sum of the next two throws of the subsequent frame.

Story: As the scorekeeper, I want to be able to recognize a strike frame, compute its score after the next frame has been completed, and compute the game score.

Examples: Suppose [10, 0] and [3, 6] are consecutive frames. Then the first frame is a strike and its score equals $10+3+6=19$. The game [10, 0] [3, 6] [7, 2] [3, 6] [4, 4] [5, 3] [3, 3] [4, 5] [8, 1] [2, 6] has a score of 94. The partial game [10, 0] [3, 6] has a score of 28.

Spare

A frame is called a spare when all 10 pins are knocked down in two throws. The score of a spare frame is 10 plus the value of the first throw from the subsequent frame.

Story: As the scorekeeper, I want to be able to recognize a spare frame, compute the score of a game containing a spare frame after the first throw of the next frame has been rolled, and compute the game's score.

Examples: [1, 9], [4, 6], [7, 3] are all spares. If you have two frames [1, 9] and [3, 6] in a row, the spare frame's score is $10+3=13$. The game [1, 9] [3, 6] [7, 2] [3, 6] [4, 4] [5, 3] [3, 3] [4, 5] [8, 1] [2, 6] has a score of 88. The partial game [1, 9] [3, 6] has a score of 22.

Strike and Spare

A strike can be followed by a spare. The strike's score is not affected when this happens.

Story: As the scorekeeper, I want to make sure that the score of a strike is computed right when it's followed by a spare.

Examples: In the sequence [10, 0] [4, 6] [7, 2], a strike is followed by a spare. In this case, the score of the strike is 10 + 4 + 6 = 20, and the score of the spare is 4 + 6 + 7 = 17. The game [10, 0] [4, 6] [7, 2] [3, 6] [4, 4] [5, 3] [3, 3] [4, 5] [8, 1] [2, 6] has a score of 103.

Multiple Strikes

Two strikes in a row are possible. You must take care when this happens as you need the values of throws from the next two frames to compute the score of the first strike..

Story: As the scorekeeper, I want to make sure that I can record two consecutive strikes correctly in the game, and correctly compute the score of the first strike after the next two throws have been rolled.

Examples: In the sequence [10, 0] [10, 0] [7, 2], the score of the first strike is 10 + 10 + 7 = 27. The score of the second strike is 10 + 7 + 2 = 19. The game [10, 0] [10, 0] [7, 2] [3, 6] [4, 4] [5, 3] [3, 3] [4, 5] [8, 1] [2, 6] has a score of 112. The score of the partial game [10, 0] [10, 0] [7, ?] is 27 (we cannot compute the scores of the last two frames yet).

Multiple Spares

Two spares in a row are possible. The score of the first spare is not affected when this happens.

Story: As the scorekeeper, I want to be able to compute the score of a game with two spares in a row, and the scores of the first spare after the next spare has been completed.

Example: The game [8, 2] [5, 5] [7, 2] [3, 6] [4, 4] [5, 3] [3, 3] [4, 5] [8, 1] [2, 6] has a score of 98.

Spare as the Last Frame

When the last frame in a game is a spare, the player will be given a bonus throw. However, this bonus throw does not belong to a regular frame. It is only used to calculate the score of the last spare.

Story: As the scorekeeper, I hate it when the last frame is a spare: let the game please figure out that the next roll is a bonus throw and compute the score of the last frame and the whole game based on the value of that bonus throw.

Example: The last frame in the game [1, 5] [3, 6] [7, 2] [3, 6] [4, 4] [5, 3] [3, 3] [4, 5] [8, 1] [2, 8] is a spare. If the bonus throw is [7], the last frame has a score of 2 + 8 + 7 = 17. The game has a score of 90.

Strike as the Last Frame

When the last frame of the game is a strike, the player will be given two bonus throws. However, these two bonus throws do not belong to a regular frame. They are only used to calculate score of the last strike frame.

Story: As the scorekeeper, I hate it even more when the last frame of a game is a strike: let the game please figure out that the next rolls are bonus throws and compute the score of the last frame and the whole game based on the value of those bonus throws.

Example: The last frame in the game [1, 5] [3, 6] [7, 2] [3, 6] [4, 4] [5, 3] [3, 3] [4, 5] [8, 1] [10, 0] is a strike. If the bonus throws are [7, 2], the last frame's score is 10 + 7 + 2 = 19. The game score is 92.

Bonus is a Strike

No more bonus throws are granted when the last frame in the game is a spare and the bonus throw is a strike.

Story: As the scorekeeper, I hate it most when the last frame is spare and the bonus throw is a strike: please God, let the game figure this scenario out correctly.

Example: In the game [1, 5] [3, 6] [7, 2] [3, 6] [4, 4] [5, 3] [3, 3] [4, 5] [8, 1] [2, 8], the last frame is a spare. If the bonus throw is [10], the game score is 93.

Best Score

A perfect game consists of all strikes (a total of 12, including the bonus throws), and has a score of 300.

Story: As the scorekeeper, I love it when the game is just a sequence of strikes, including the bonus throws, because I know that the player then deserves a perfect score of 300.

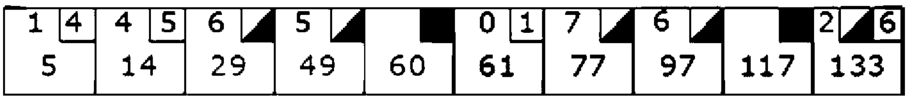
Example: A perfect game looks like [10, 0] [10, 0] [10, 0] [10, 0] [10, 0] [10, 0] [10, 0] [10, 0] [10, 0] [10, 0] with bonus throws [10, 10]. Its score is 300.

Random Game

Story: As the scorekeeper, I want to make sure that the game [6, 3] [7, 1] [8, 2] [7, 2] [10, 0] [6, 2] [7, 3] [10, 0] [8, 0] [7, 3] [10] has a score of 135.

Congratulations, you are done!

Specification for Bowling Score Keeper without Slicing



1 4	4 5	6 ◢	5 ◢	■	0 1	7 ◢	6 ◢	■	2 ◢ 6
5	14	29	49	60	61	77	97	117	133

The game consists of 10 frames as shown above. The player has two opportunities in each frame to knock down 10 pins. The score for the frame is the total number of pins knocked down, plus bonuses for strikes and spares.

A spare is when the player knocks down all 10 pins in two tries. The bonus for that frame is the number of pins knocked down by the next ball rolled. So, the score in frame 3 above is 10 (the total number knocked down), plus a bonus of 5 (the number of pins knocked down on the next roll.).

A strike is when the player knocks down all 10 pins on his or her first try. The bonus for that frame is the value of the next two balls rolled.

A player who rolls a spare or strike in the tenth frame is allowed to roll the extra balls to complete the frame. However, no more than three balls can be rolled in tenth frame.

Appendix 3: Industry Questionnaire

Demographics
Respondent ID* (*) Required
Company*
Location*
Function*

Education
1. Please state your academic degree title(s) (e.g., BS in computer science, MS in management).*
2. Please state any certification(s) that you have received during your professional career (e.g. SEI certification as Personal Software Process (PSP) developer, CMMI certification, or ITIL certification as application engineer).

Professional experience
3. Please state the roles that you have performed during your professional career (e.g. developer XX months/years, tester YY months/years). *
4. Please describe the type of code you currently build (e.g. web interfaces using html+css+javascript; business logic using beans).
5. Please state the programming languages that you have used (during your education as well), and the number of years of experience in each one.

Programming Language 1
5.1.1 Programming language*
5.1.2 Years (education)*
5.1.3 Years (professional career)*

Programming Language 2
5.2.1 Programming language
5.2.2 Years (education)
5.2.3 Years (professional career)

Programming Language 3
5.3.1 Programming language
5.3.2 Years (education)
5.3.3 Years (professional career)

6. How would you rate your programming experience?*
- No experience (only casual usage)
- Little experience (<2 years)
- Novice (2-<=5 years)
- Intermediate (5-<=10 years)
- Expert (>10 years)

7. How would you rate your Java experience?*
- No experience (only casual usage)
- Little experience (<2 years)
- Novice (2-<=5 years)
- Intermediate (5-<=10 years)
- Expert (>10 years)

8. Which development methodologies have you used so far?
(e.g., waterfall, iterative, spiral, agile. If you choose agile, please indicate the type (scrum, tdd, xp, etc.). Include the methodologies you used in academia as well. State the number of years of experience in each one.)

Methodology 1
8.1.1 Methodology*
8.1.2 Years (education)*
8.1.3 Years (professional career)*

Methodology 2
8.2.1 Methodology
8.2.2 Years (education)
8.2.3 Years (professional career)

Methodology 3
8.3.1 Methodology
8.3.2 Years (education)
8.3.3 Years (professional career)

Testing experience
9. How would you rate your unit testing experience?*
- No experience (only casual usage)
- Little experience (<2 years)
- Novice (2-<=5 years)
- Intermediate (5-<=10 years)
- Expert (>10 years)

10. Do you write automated tests?*
- Yes
- No

10.1. If you answered “yes” above, please give a brief explanation.

11. Do you currently use a tool for unit testing (for executing, monitoring)?*

11.1. If you answered “yes” above, please write the names of the tools.

12. What IDE (Integrated Development Environment) do you currently use?*

13. Do you have substantial experience of other IDEs? If so, please specify which ones.

14. How would you rate your experience with the JUnit testing framework?*
- No experience (only casual usage)
- Little experience (<2 years)
- Novice (2-<=5 years)
- Intermediate (5-<=10 years)
- Expert (>10 years)

15. Have you ever used TDD as a development methodology?*
- Yes
- No

15.1. If you answered “yes” above, how would you rate your TDD experience?
- No experience (only casual usage)
- Little experience (<2 years)
- Novice (2-<=5 years)
- Intermediate (5-<=10 years)
- Expert (>10 years)

16. Have you ever attended any training on testing or more specifically unit testing?*
- Yes
- No

16.1. If you answered “yes” above, please give a brief description of its content.

17. Have you ever attended any training on TDD?*
- Yes
- No

If you answered “yes” above, please briefly answer the following questions:
17.1. What was taught during the training?
17.2. How long did the training take (in days or hours if possible)?
17.3. When did you take the training?
17.4. Did you take the training at your current job?
17.5. Are you still practising TDD? Why?
18. Have you ever been involved in TDD studies in industry?*
- Yes
- No

18.1. If you answered “yes” above, please share the results you got from the pilot study, and your opinion on its effectiveness.
19. Have you ever attended any coding kata? Required kata = programming exercise
- Yes
- No

19.1. If you answered “yes” above, please state when and which katas (name of the programming exercises) you completed.

Appendix 4: Academic Questionnaire

Demographics
Respondent ID* (*) Required
University*
Location*

Education
1. Please state your academic degree title(s), if any (e.g., BS in computer science, MS in management).*
2. Please state the roles that you have performed during your professional career, if any (e.g., developer XX months/years, tester YY months/years, etc.).*
3. Please state the programming languages that you have used (during your education as well), and the number of years of experience in each one.

Programming Language 1
3.1.1 Programming language*:
3.1.2 Years (education) Required
3.1.3 Years (professional career), if any

Programming Language 2
3.2.1 Programming language
3.2.2 Years (education)
3.2.3 Years (professional career), if any

Programming Language 3
3.3.1 Programming language
3.3.2 Years (education)
3.3.3 Years (professional career), if any

4. How would you rate your programming experience?*
- No experience (<2 years)
- Novice (2-<=5 years)
- Intermediate (5-<=10 years)
- Expert (>10 years)

5. Which development methodologies have you used so far? (e.g., waterfall, iterative, spiral, agile. If you choose agile, please state the type (scrum, tdd, xp, etc.). State the number of years of experience in each one (e.g., waterfall, 1 year of education, 5 years of professional practice).

Methodology 1
5.1.1 Methodology*
5.1.2 Years (education)*Required
5.1.3 Years (professional career), if any

Methodology 2
5.2.1 Methodology
5.2.2 Years (education)
5.2.3 Years (professional career), if any

Methodology 3
5.3.1 Methodology
5.3.2 Years (education)
5.3.3 Years (professional career), if any

6. How would you rate your unit testing experience?*
- No experience (<2 years)
- Novice (2-<=5 years)
- Intermediate (5-<=10 years)
- Expert (>10 years)

7. Have you used a unit testing tool? If you answered yes above, please write the names of the tools.

8. What IDE (Integrated Development Environment) have you used?*

9. How would you rate your Java experience?*
- No experience (<2 years)
- Novice (2-<=5 years)
- Intermediate (5-<=10 years)
- Expert (>10 years)

10. How would you rate your JUnit testing framework experience?*
- No experience (<2 years)
- Novice (2-<=5 years)
- Intermediate (5-<=10 years)
- Expert (>10 years)

11. Have you ever used TDD as a development methodology?*
- Yes
- No

12. If you answered “yes” above, how would you rate your TDD experience?
- No experience (<2 years)
- Novice (2-<=5 years)
- Intermediate (5-<=10 years)
- Expert (>10 years)

13. Please state the certification(s) you have received during your professional career, if any (e.g., SEI certification as Personal Software Process (PSP) developer, CMMI certification, or ITIL certification as application engineer.)

14. Have you ever attended any training on testing, or more specifically unit testing? If yes, please give a brief explanation of its content.

15. Have you ever attended any training on TDD?*Required
- Yes
- No

16. If you answered “yes” above, please briefly answer the following questions:
a) What was taught during the training?
b) How long did the training take (in days or hours if possible)?
c) When did you take the training?
d) Did you take the training at a company?

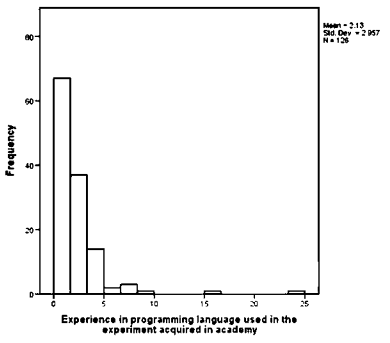
17. Have you ever attended any coding kata?*
- Yes
- No

17a. If you answered “yes” above, please state when and which katas (name of the coding task) you completed.

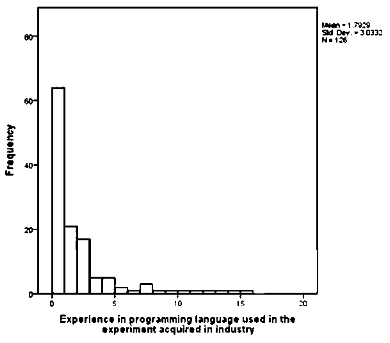
Appendix 5: Breakdown of Experience

Programming Language Experience

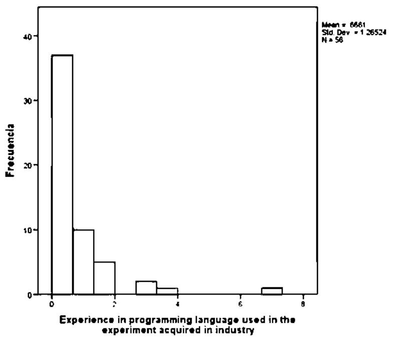
A. ACADEMY - TOTAL



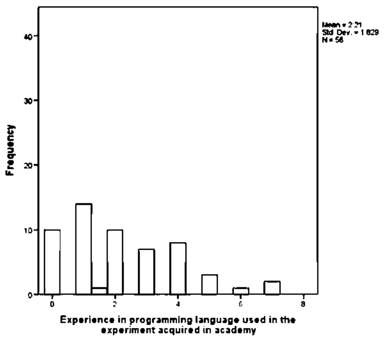
B. INDUSTRY - TOTAL



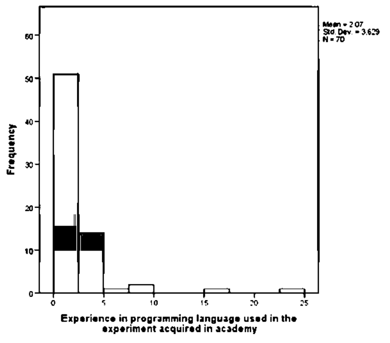
C. ACADEMY – STUDENTS



D. INDUSTRY – STUDENTS



E. ACADEMY - PRACTITIONERS



F. INDUSTRY - PRACTITIONERS

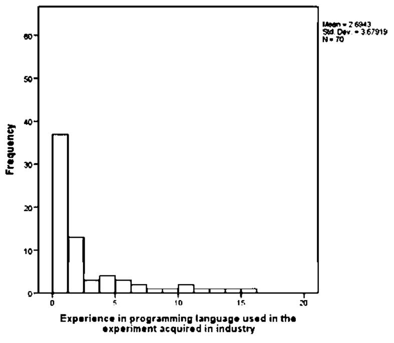
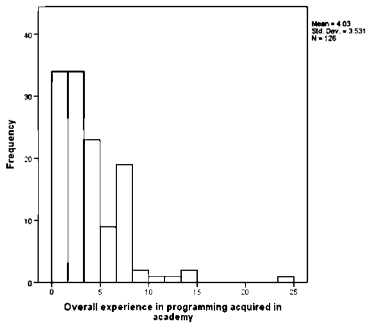


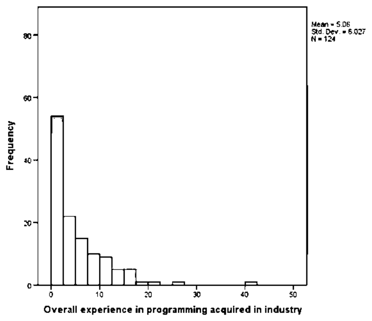
Fig. 8 Breakdown of programming language experience

Overall Programming Language Experience

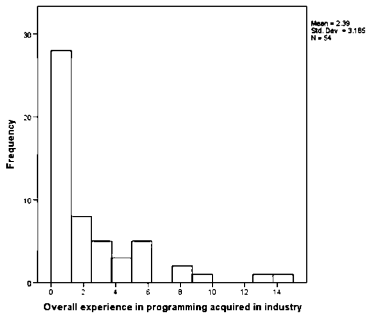
A. Academy - Total



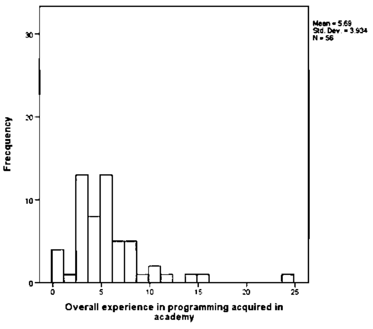
B. Industry - Total



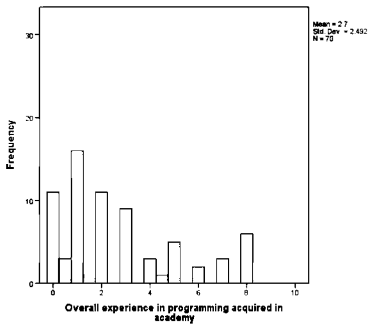
C. Academy– Students



D. Industry – Students



E. Academy - Practitioners



F. Industry - Practitioners

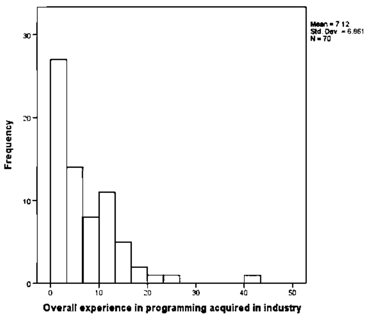


Fig. 9 Breakdown of Overall programming language experience

Appendix 6: Collinearity Conditions

Table 16 reports the results of the collinearity analysis for the model with 15 independent variables. The pattern shown in Table 16 suggests that the testing framework (UNIT_TESTING_FRAMEWORK2_ADAPTED) might be collinear, as it has values close to the bounds established for the variance inflation factor ($VIF = 4.943$) and a low tolerance ($T = 0.202$). On the other hand, the collinearity statistics for the other variables are within the expected values ($VIF < 5$ and $T > 0.2$), which is a sign that they are not collinear.

Table 17 shows the collinearity diagnostics of the model specified in Table 16. Note that component 16 has a very high condition index ($CI = 86.918 > 30$), which suggests that the level of collinearity is high. Comparing the proportion of variance explained for each of the model explanatory variables, we find that the UNIT_ TESTING_FRAMEWORK_ADAPTED and EXPERIMENT_PROGRAMMING_ LANGUAGE variables have an extremely high proportion of variance explained with values of 0.90 and 0.46, respectively. One way of solving the collinearity problem is to remove the most collinear variable, which, in this case, is UNIT_TESTING_ FRAMEWORK_ADAPTED.

Model 2

Table 18 reports the collinearity diagnostics of model 2 with 14 variables, which is composed of all the variables of the original model, except the UNIT_TESTING_ FRAMEWORK_ADAPTED variable that was eliminated on the grounds of collinearity.

Note that dimension 15 still has a very high condition index ($CI = 43 > 30$), which implies that there is a problem of collinearity. There are three closely correlated variables: EXPERIMENT_PROGRAMMING_LANGUAGE, SITE and TRAINER. In order to deal with the collinearity problem, we have opted to eliminate the variable with the highest proportion of variance explained, which in this case is EXPERIMENT_PROGRAMMING_LANGUAGE with a proportion of variance explained of 0.40.

Model 3

Table 19 reports the collinearity diagnostics of model 3 with 13 variables, which is composed of all the variables of model 2 except the EXPERIMENT_PROGRAMMING_ LANGUAGE variable.

Note that dimension 14 still has a condition index greater than 30 ($CI = 33.67 > 30$), which suggests that there is a problem of collinearity. There are three closely correlated variables: SITE, and TRAINER and CS_DEGREE. According to the non-collinearity condition, we should eliminate the variable with the highest proportion of variance explained. Bearing in mind the experimental data type, we know that SITE (which refers to whether the experiment was conducted in academia or industry) is closely related to TRAINER. Therefore, we will eliminate the TRAINER variable, as one of the trainers mostly trained subjects in industry and the other trained subjects in academia, and kept SITE, which is a more interesting variable for this research.

Table 16 Coefficients of the linear regression model with 15 independent variables

Model		Unstandardized Coefficients		Standardized Coefficients	t	Sig.	Collinearity Statistics	
		B	Std. Error	Beta			Tolerance	VIF
1	(Constant)	−64.527	65.070		−.992	.324		
	SITE	36.151	10.468	.425	3.454	.001	.429	2.330
	TRAINER	2.476	11.380	.028	.218	.828	.398	2.512
	CS_TITLE	17.018	9.767	.177	1.742	.085	.628	1.592
	UNIT_TESTING_FRAMEWORK_ADAPTED	−14.927	21.838	−.123	−.684	.496	.202	4.943
	EXPERIENCE_UNIT_TESTING_FRAMEWORK_LIKERT_SCALE	8.903	8.841	.119	1.007	.316	.464	2.157
	EXPERIMENT_PROGRAMMING_LANGUAGE	23.861	17.329	.230	1.377	.172	.233	4.292
	EXPERIENCE_EXPERIMENT_PROGRAMMING_LANGUAGE_ACADEMY_YEARS	.337	1.995	.022	.169	.866	.382	2.621
	EXPERIENCE_EXPERIMENT_PROGRAMMING_LANGUAGE_INDUSTRY_YEARS	1.198	1.978	.086	.606	.546	.321	3.119
	OVERALL_EXPERIENCE_PROGRAMMING_ACADEMY_YEARS	3.326	1.289	.285	2.581	.011	.534	1.873
	OVERALL_EXPERIENCE_PROGRAMMING_INDUSTRY_YEARS	.959	1.039	.135	.923	.358	.304	3.292
	EXPERIENCE_UNIT_TESTING_LIKERT_SCALE	−9.577	7.411	−.162	−1.292	.199	.412	2.426
	EXPERIMENT_IDE_USED_DUMMY	16.605	9.187	.190	1.807	.074	.590	1.694
	TDD_USED_DUMMY	−1.873	10.723	−.017	−.175	.862	.650	1.540
	TASK_ITLD	8.511	13.514	.094	.630	.530	.290	3.449
	SLICED_ITLD_DUMMY	29.735	13.477	.330	2.206	.030	.292	3.430

Dependent Variable: QLTY

Table 17 Collinearity diagnostics (1)

Collinearity Diagnostics

Model	Dimension	Eigenvalue	Condition Index	Variance Proportions							
				(Constant)	SITE	TRAINER	CS_TITLE	UNIT_TESTING_ FRAMEWORK_ ADAPTED	EXPERIENCE_ UNIT_ TESTING_ FRAMEWORK_ LIKERT_SCALE	EXPERIMENT_ PROGRAMMING_ LANGUAGE	EXPERIENCE_ EXPERIMENT_ PROGRAMMING_ LANGUAGE_ ACADEMY_ YEARS
1	1	11.113	1.000	.00	.00	.00	.00	.00	.00	.00	.00
	2	1.349	2.871	.00	.00	.00	.00	.00	.00	.00	.00
	3	.905	3.504	.00	.00	.00	.00	.00	.00	.00	.00
	4	.789	3.754	.00	.00	.00	.00	.00	.00	.00	.17
	5	.534	4.563	.00	.00	.00	.02	.00	.00	.00	.03
	6	.360	5.555	.00	.01	.00	.05	.00	.00	.00	.01
	7	.293	6.161	.00	.00	.00	.06	.00	.00	.00	.02
	8	.213	7.224	.00	.00	.00	.33	.00	.02	.00	.00
	9	.164	8.233	.00	.00	.01	.05	.00	.08	.00	.26
	10	.084	11.497	.00	.00	.01	.09	.00	.08	.00	.06
	11	.075	12.211	.00	.15	.00	.02	.00	.42	.01	.08
	12	.051	14.691	.00	.00	.18	.22	.01	.09	.08	.00
	13	.037	17.346	.00	.32	.24	.11	.00	.20	.01	.18
	14	.025	21.071	.00	.14	.10	.00	.00	.04	.01	.09
	15	.007	38.679	.14	.29	.07	.04	.09	.01	.42	.03
	16	.001	86.918	.86	.09	.38	.00	.90	.04	.46	.05

Model	Dimension	Variance Proportions							
		EXPERIENCE_ EXPERIMENT_ PROGRAMMING_ LANGUAGE_ INDUSTRY_YEARS	OVERALL_ EXPERIENCE_ PROGRAMMING_ ACADEMY_ YEARS	OVERALL_ EXPERIENCE_ PROGRAMMING_ INDUSTRY_ YEARS	EXPERIENCE_ UNIT_ TESTING_ LIKERT_ SCALE	EXPERIMENT_ IDE_USED_ DUMMY	TDD_USED_ DUMMY	TASK_ITLD	SLICED_ITLD_ DUMMY
1	1	.00	.00	.00	.00	.00	.00	.00	.00
	2	.05	.01	.03	.00	.01	.04	.00	.02
	3	.04	.02	.02	.00	.00	.30	.00	.03

Table 17 (continued)

4	.00	.03	.02	.00	.00	.04	.00	.01
5	.00	.00	.01	.00	.03	.17	.00	.19
6	.22	.06	.04	.00	.05	.05	.00	.02
7	.00	.20	.00	.00	.43	.04	.00	.00
8	.01	.25	.04	.04	.00	.06	.00	.00
9	.00	.15	.23	.06	.05	.10	.00	.00
10	.27	.01	.13	.40	.10	.00	.05	.05
11	.18	.00	.26	.00	.06	.03	.01	.01
12	.04	.04	.04	.00	.02	.01	.00	.01
13	.07	.20	.04	.22	.20	.06	.00	.01
14	.06	.01	.11	.18	.05	.01	.80	.54
15	.01	.00	.02	.07	.01	.00	.11	.10
16	.04	.01	.02	.01	.00	.07	.01	.00

Table 18 Collinearity diagnostics (2) with 14 variables

Collinearity Diagnostics

Model	Dimension	Eigenvalue	Condition Index	Variance Proportions							
				(Constant)	SITE	TRAINER	CS_TITLE	EXPERIENCE_ UNIT_TESTING_ FRAMEWORK_ LIKERT_SCALE	EXPERIMENT_ PROGRAMMING_ LANGUAGE	EXPERIENCE_ EXPERIMENT_ PROGRAMMING_ LANGUAGE_ ACADEMY_YEARS	EXPERIENCE_ EXPERIMENT_ PROGRAMMING_ LANGUAGE_ INDUSTRY_ YEARS
1	1	10.153	1.000	.00	.00	.00	.00	.00	.00	.00	.00
	2	1.346	2.747	.00	.00	.00	.00	.00	.00	.00	.05
	3	.905	3.350	.00	.00	.00	.00	.00	.00	.01	.04
	4	.786	3.594	.00	.00	.00	.00	.00	.00	.18	.00
	5	.523	4.404	.00	.00	.00	.03	.00	.00	.04	.00
	6	.353	5.366	.00	.02	.00	.05	.00	.00	.01	.23
	7	.293	5.889	.00	.00	.00	.06	.00	.00	.02	.00
	8	.209	6.964	.00	.00	.00	.36	.02	.01	.00	.02
	9	.164	7.872	.00	.00	.02	.05	.08	.00	.27	.00
	10	.084	11.011	.00	.00	.02	.08	.09	.00	.07	.26
	11	.075	11.672	.00	.15	.00	.01	.45	.02	.08	.18
	12	.043	15.367	.00	.02	.37	.12	.04	.36	.01	.01
	13	.036	16.728	.00	.29	.18	.18	.27	.16	.20	.09
	14	.025	20.150	.00	.14	.12	.00	.04	.05	.10	.06
	15	.005	43.552	1.00	.38	.27	.07	.00	.40	.01	.04

Model	Dimension	Variance Proportions						
		OVERALL_ EXPERIENCE_ PROGRAMMING_ ACADEMY_YEARS	OVERALL_ EXPERIENCE_ PROGRAMMING_ INDUSTRY_YEARS	EXPERIENCE_ UNIT_ TESTING_ LIKERT_SCALE	EXPERIMENT_ IDE_USED_ DUMMY	TDD_USED_ DUMMY	TASK_ITLD	SLICED_ ITLD_ DUMMY
	1	.00	.00	.00	.00	.00	.00	.00
	2	.01	.03	.00	.01	.05	.00	.02
	3	.02	.02	.00	.00	.32	.00	.03
	4	.03	.02	.00	.00	.05	.00	.01

Table 18 (continued)

5	.00	.01	.00	.03	.17	.00	.18
6	.06	.03	.00	.04	.07	.00	.03
7	.20	.00	.00	.43	.05	.00	.00
8	.27	.05	.04	.00	.06	.00	.00
9	.15	.24	.07	.05	.11	.00	.00
10	.02	.12	.40	.09	.00	.06	.06
11	.00	.26	.00	.06	.03	.01	.01
12	.08	.02	.01	.00	.04	.00	.01
13	.16	.06	.22	.21	.04	.00	.01
14	.01	.11	.18	.05	.01	.80	.54
15	.00	.04	.08	.02	.00	.13	.09

Table 19 Collinearity diagnostics (3) with 13 variables

Model	Dimension	Eigenvalue	Condition Index	Variance Proportions						
				(Constant)	SITE2	TRAINER	CS_TITLE	EXPERIENCE_UNIT_TESTING_FRAMEWORK_LIKERT_SCALE	EXPERIENCE_EXPERIMENT_PROGRAMMING_LANGUAGE_ACADEMY_YEARS	EXPERIENCE_EXPERIMENT_PROGRAMMING_LANGUAGE_INDUSTRY_YEARS
1	1	9.237	1.000	.00	.00	.00	.00	.00	.00	.00
	2	1.337	2.629	.00	.00	.00	.00	.00	.00	.05
	3	.903	3.199	.00	.00	.00	.00	.00	.01	.04
	4	.778	3.445	.00	.00	.00	.00	.00	.18	.00
	5	.510	4.254	.00	.00	.00	.03	.00	.05	.01
	6	.350	5.140	.00	.02	.00	.05	.00	.01	.23
	7	.292	5.620	.00	.00	.00	.06	.00	.03	.00
	8	.202	6.755	.00	.00	.00	.45	.01	.00	.03
	9	.163	7.520	.00	.00	.02	.03	.09	.29	.00
	10	.083	10.534	.00	.00	.03	.05	.06	.08	.29
	11	.072	11.321	.00	.16	.00	.00	.59	.09	.18
	12	.038	15.558	.00	.26	.54	.02	.11	.19	.04
	13	.026	18.895	.00	.23	.16	.04	.10	.06	.03
	14	.008	33.679	.99	.32	.24	.26	.02	.01	.11

Table 19 (continued)

Model	Dimension	Variance Proportions OVERALL_ EXPERIENCE_ PROGRAMMING_ ACADEMY_ YEARS	OVERALL_ EXPERIENCE_ PROGRAMMING_ INDUSTRY_ YEARS	EXPERIENCE_ UNIT_TESTING_ LIKERT_SCALE	EXPERIMENT_ IDE_USED_ DUMMY	TDD_USED_ DUMMY	TASK_ITLD	SLICED_ ITLD_ DUMMY
		.00	.00	.00	.00	.00	.00	.00
	2	.01	.02	.00	.01	.05	.00	.02
	3	.02	.02	.00	.00	.32	.00	.03
	4	.03	.02	.00	.00	.07	.00	.01
	5	.00	.02	.00	.04	.15	.00	.18
	6	.05	.03	.00	.03	.08	.00	.03
	7	.20	.00	.00	.44	.05	.00	.00
	8	.28	.04	.03	.00	.05	.00	.00
	9	.13	.25	.07	.05	.11	.00	.00
	10	.02	.14	.40	.10	.00	.07	.07
	11	.00	.27	.02	.06	.03	.00	.00
	12	.24	.02	.19	.15	.09	.00	.01
	13	.01	.06	.21	.01	.01	.70	.49
	14	.01	.09	.06	.09	.00	.22	.14

Table 20 Collinearity diagnostics (4) with 12 variables

Model	Dimension	Eigenvalue	Condition Index	Variance Proportions					
				(Constant)	SITE	CS_TITLE	EXPERIENCE_ UNIT_TESTING_ FRAMEWORK_ LIKERT_SCALE	EXPERIENCE_ EXPERIMENT_ PROGRAMMING_ LANGUAGE_ ACADEMY_ YEARS	EXPERIENCE_ EXPERIMENT_ PROGRAMMING_ LANGUAGE_ INDUSTRY_ YEARS
	1	8.373	1.000	.00	.00	.00	.00	.00	.00
	2	1.288	2.550	.00	.00	.00	.00	.00	.05
	3	.902	3.047	.00	.00	.00	.00	.01	.04
	4	.769	3.299	.00	.00	.00	.00	.20	.00
	5	.498	4.099	.00	.00	.04	.00	.06	.01
	6	.337	4.988	.00	.03	.04	.00	.01	.25
	7	.292	5.356	.00	.00	.06	.00	.03	.01
	8	.202	6.433	.00	.00	.45	.02	.00	.03
	9	.150	7.480	.00	.01	.03	.09	.45	.02
	10	.079	10.272	.00	.02	.05	.03	.03	.27
	11	.072	10.799	.00	.13	.00	.64	.11	.15
	12	.028	17.306	.01	.48	.09	.19	.00	.00
	13	.010	29.003	.98	.33	.23	.03	.09	.18
Model	Dimension	Variance Proportions							
		OVERALL_ EXPERIENCE_ PROGRAMMING_ ACADEMY_ YEARS		OVERALL_ EXPERIENCE_ PROGRAMMING_ INDUSTRY_ YEARS	EXPERIENCE_ UNIT_ TESTING_ LIKERT_ SCALE	EXPERIMENT_ IDE_USED_ DUMMY	TDD_ USED_ DUMMY	TASK_ ITLD	SLICED_ ITLD_ DUMMY
	1	.00		.00	.00	.00	.00	.00	.00
	2	.02		.02	.00	.01	.05	.00	.03
	3	.02		.02	.00	.00	.35	.00	.03
	4	.03		.02	.00	.00	.07	.00	.02
	5	.00		.03	.00	.06	.16	.00	.15

Table 20 (continued)

6	.07	.02	.01	.01	.12	.00	.04
7	.21	.00	.00	.46	.07	.00	.00
8	.31	.05	.04	.00	.06	.00	.00
9	.23	.30	.05	.09	.07	.00	.00
10	.00	.12	.46	.06	.01	.09	.08
11	.00	.25	.06	.06	.04	.00	.00
12	.03	.02	.33	.01	.00	.49	.45
13	.08	.14	.05	.22	.01	.41	.19

Model 4

Table 20 shows the collinearity diagnostics of model 4 with 12 variables, which is composed of all the variables of model 3 except the TRAINER variable. Model 4 is the model that we finally used in this research. Note that this model meets the collinearity conditions: a) the condition index of dimension 13 (CI = 29) is less than 30 and b) the proportions of variance explained are within the established bounds (less than 0.5).

Appendix 7: Multiple Linear Regression – Alternative Model

Quality

Table 21 shows the results of the multiple regression model with respect to the influence of External Quality. Note that experience is measured on a Likert scale in this case.

Table 21 Results of the MRL - Quality

Coefficients[a]

Model		Unstandardized Coefficients		Standardized Coefficients	t	Sig.	Collinearity Statistics	
		B	Std. Error	Beta			Tolerance	VIF
1	(Constant)	−52.740	27.988		−1.884	.062		
	SITE	32.095	9.704	.377	3.308	.001	.520	1.922
	CS_TITLE	24.603	9.323	.254	2.639	.010	.727	1.375
	EXPERIENCE_UNIT_TESTING_FRAMEWORK_LIKERT_SCALE	11.087	8.998	.147	1.232	.221	.473	2.115
	EXPERIENCE_EXPERIMENT_PROGRAMMING_LANGUAGE_LIKERT_SCALE	3.434	5.791	.069	.593	.554	.505	1.980
	OVERALL_EXPERIENCE_PROGRAMMING_LIKERT_SCALE	2.344	5.380	.046	.436	.664	.618	1.618
	EXPERIENCE_UNIT_TESTING_LIKERT_SCALE	−11.366	7.270	−.191	−1.563	.121	.451	2.216
	EXPERIMENT_IDE_USED_DUMMY	20.240	8.448	.231	2.396	.018	.728	1.374
	TDD_USED_DUMMY	2.620	10.077	.024	.260	.795	.776	1.288
	TASK_ITLD	5.744	13.467	.063	.427	.671	.308	3.252
	SLICED_ITLD_DUMMY	36.935	13.617	.406	2.712	.008	.301	3.324

[a] Dependent Variable: QLTY

Productivity

Table 22 shows the results of the multiple regression model with respect to the influence of Productivity. Note that experience is measured on a Likert scale in this case.

Table 22 MRL results – Productivity

Coefficients[a]

Model		Unstandardized Coefficients		Standardized Coefficients	t	Sig.	Collinearity Statistics	
		B	Std. Error	Beta			Tolerance	VIF
1	(Constant)	−43.041	24.194		−1.779	.078		
	SITE	14.541	8.388	.190	1.734	.086	.520	1.922
	CS_TITLE	19.392	8.059	.223	2.406	.018	.727	1.375
	EXPERIENCE_UNIT_TESTING_FRAMEWORK_LIKERT_SCALE	11.869	7.778	.175	1.526	.130	.473	2.115
	EXPERIENCE_EXPERIMENT_PROGRAMMING_LANGUAGE_LIKERT_SCALE	−.161	5.006	−.004	−.032	.974	.505	1.980
	OVERALL_EXPERIENCE_PROGRAMMING_LIKERT_SCALE	1.614	4.651	.035	.347	.729	.618	1.618
	EXPERIENCE_UNIT_TESTING_LIKERT_SCALE	−6.628	6.285	−.124	−1.055	.294	.451	2.216
	EXPERIMENT_IDE_USED_DUMMY	19.122	7.303	.243	2.619	.010	.728	1.374
	TDD_USED_DUMMY	−4.760	8.711	−.049	−.546	.586	.776	1.288
	TASK_ITLD	11.285	11.642	.138	.969	.335	.308	3.252
	SLICED_ITLD_DUMMY	31.924	11.771	.391	2.712	.008	.301	3.324

[a] Dependent Variable: PROD

Appendix 8: Residual Analysis by Experiment

Quality

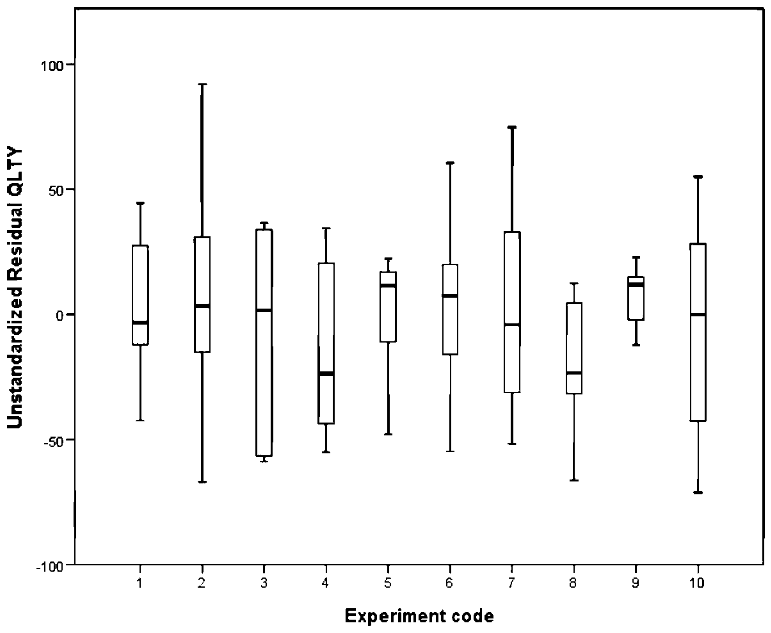


Fig. 10 Residual by Experiment – QLTY

Table 23 Effect of the experiment on Quality

Tests of Between-Subjects Effects

Source	Type III Sum of Squares	df	Mean Square	F	Sig.	Partial Eta Squared	Noncent. Parameter	Observed Power[b]
Dependent Variable: Unstandardized Residual QLTY								
Corrected Model	7137.509[a]	9	793.057	.652	.750	.053	5.866	.305
Intercept	370.491	1	370.491	.304	.582	.003	.304	.085
EXP_CODE	7137.509	9	793.057	.652	.750	.053	5.866	.305
Error	127758.601	105	1216.749					
Total	134896.110	115						
Corrected Total	134896.110	114						

[a] R Squared = .053 (Adjusted R Squared = −.028)

[b] Computed using alpha = .05

The results reported in Table 24 show that the model residuals plotted against the EXPERIMENT_CODE variable are significant (p-value = 0.006 < 0.05), which means that the variances are not homogeneous.

Table 24 Levene test for QLTY

Levene's Test of Equality of Error Variances[a]			
F	df1	df2	Sig.
Dependent Variable: Unstandardized Residual QLTY			
2.798	9	105	.006

Tests the null hypothesis that the error variance of the dependent variable is equal across groups
[a] Design: Intercept + EXP_CODE

Productivity

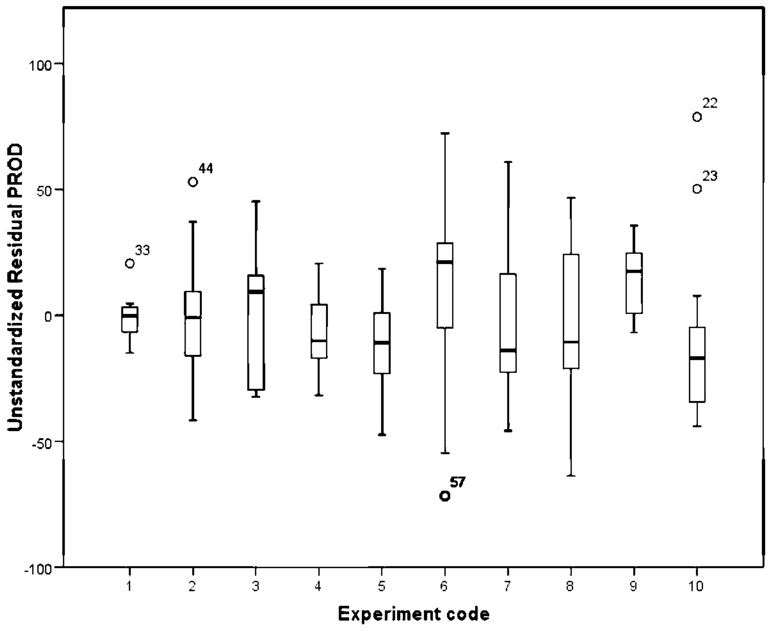


Fig. 11 Residual by Experiment – PROD

Table 25 Effect of the experiment on PRODUCTIVITY

Tests of Between-Subjects Effects

Source	Type III Sum of Squares	df	Mean Square	F	Sig.	Partial Eta Squared	Noncent. Parameter	Observed Power[b]
Dependent Variable: Unstandardized Residual PROD								
Corrected Model	9282.965[a]	9	1031.441	1.235	.282	.096	11.115	.578
Intercept	88.286	1	88.286	.106	.746	.001	.106	.062
EXP_CODE	9282.965	9	1031.441	1.235	.282	.096	11.115	.578
Error	87693.492	105	835.176					
Total	96976.457	115						
Corrected Total	96976.457	114						

[a] R Squared = .096 (Adjusted R Squared = .018)

[b] Computed using alpha = .05

The results reported in Table 26 show that the model residuals plotted against the EXPERIMENT_CODE variable are not significant (p-value = 0.155 > 0.05), which suggests that the residual variances are homogeneous.

Table 26 Levene test for PROD

Levene's test of equality of error variances[a]

F	df1	df2	Sig.
Dependent Variable: Unstandardized Residual PROD			
1.507	9	105	.155

Tests the null hypothesis that the error variance of the dependent variable is equal across groups

[a] Design: Intercept + EXP_CODE

Appendix 9: SPSS Scripts

Filter

```
USE ALL.
COMPUTE filter_$=(EXP_CODE ~= 11 and (TASK_ITLD = 1 or TASK_ITLD = 2)).
VARIABLE LABELS filter_$ 'EXP_CODE ~= 11 and (TASK_ITLD = 1 or TASK_ITLD = 2)
(FILTER)'.
VALUE LABELS filter_$ 0 'Not Selected' 1 'Selected'.
FORMATS filter_$ (f1.0).
FILTER BY filter_$.
EXECUTE.
```

Original MLR Model

```
REGRESSION
  /MISSING LISTWISE
  /STATISTICS COEFF OUTS CI(95) R ANOVA COLLIN TOL
  /CRITERIA=PIN(.05) POUT(.10)
  /NOORIGIN
  /DEPENDENT QLTY_ITLD
  /METHOD=ENTER SITE TRAINER CS_TITLE
UNIT_TESTING_FRAMEWORK2_ADAPTED
EXPERIENCE_UNIT_TESTING_FRAMEWORK_LIKERT_SCALE
EXPERIMENT_PROGRAMMING_LANGUAGE
EXPERIENCE_EXPERIMENT_PROGRAMMING_LANGUAGE_ACADEMY_YEARS
EXPERIENCE_EXPERIMENT_PROGRAMMING_LANGUAGE_INDUSTRY_YEARS
OVERALL_EXPERIENCE_PROGRAMMING_ACADEMY_YEARS
OVERALL_EXPERIENCE_PROGRAMMING_INDUSTRY_YEARS
EXPERIENCE_UNIT_TESTING_LIKERT_SCALE
EXPERIMENT_IDE_USED_DUMMY
TDD_USED_DUMMY
TASK_ITLD
SLICED_ITLD_DUMMY
  /SCATTERPLOT=(*ZPRED ,*ZRESID)
  /RESIDUALS DURBIN HISTOGRAM(ZRESID) NORMPROB(ZRESID)
  /SAVE RESID ZRESID.
```

MLR Results for QLTY

```
REGRESSION
  /MISSING LISTWISE
  /STATISTICS COEFF OUTS CI(95) R ANOVA COLLIN TOL
  /CRITERIA=PIN(.05) POUT(.10)
  /NOORIGIN
  /DEPENDENT QLTY_ITLD
  /METHOD=ENTER SITE CS_TITLE
EXPERIENCE_UNIT_TESTING_FRAMEWORK_LIKERT_SCALE
EXPERIENCE_EXPERIMENT_PROGRAMMING_LANGUAGE_ACADEMY_YEARS
EXPERIENCE_EXPERIMENT_PROGRAMMING_LANGUAGE_INDUSTRY_YEARS
OVERALL_EXPERIENCE_PROGRAMMING_ACADEMY_YEARS
OVERALL_EXPERIENCE_PROGRAMMING_INDUSTRY_YEARS
EXPERIENCE_UNIT_TESTING_LIKERT_SCALE
EXPERIMENT_IDE_USED_DUMMY
TDD_USED_DUMMY
TASK_ITLD
SLICED_ITLD_DUMMY
  /SCATTERPLOT=(*ZPRED ,*ZRESID)
  /RESIDUALS DURBIN HISTOGRAM(ZRESID) NORMPROB(ZRESID)
  /SAVE RESID ZRESID.
```

MLR Results for PROD

```
REGRESSION
  /MISSING LISTWISE
  /STATISTICS COEFF OUTS CI(95) R ANOVA COLLIN TOL
  /CRITERIA=PIN(.05) POUT(.10)
  /NOORIGIN
  /DEPENDENT PROD_ITLD
  /METHOD=ENTER SITE CS_TITLE
EXPERIENCE_UNIT_TESTING_FRAMEWORK_LIKERT_SCALE
EXPERIENCE_EXPERIMENT_PROGRAMMING_LANGUAGE_ACADEMY_YEARS
EXPERIENCE_EXPERIMENT_PROGRAMMING_LANGUAGE_INDUSTRY_YEARS
OVERALL_EXPERIENCE_PROGRAMMING_ACADEMY_YEARS
OVERALL_EXPERIENCE_PROGRAMMING_INDUSTRY_YEARS
EXPERIENCE_UNIT_TESTING_LIKERT_SCALE
EXPERIMENT_ IDE_USED_DUMMY
TDD_USED_DUMMY
TASK_ITLD
SLICED_ITLD_DUMMY
  /SCATTERPLOT=(*ZPRED ,*ZRESID)
  /RESIDUALS DURBIN HISTOGRAM(ZRESID) NORMPROB(ZRESID)
  /SAVE RESID ZRESID.
```

Decision Trees for the QLTY

```
TREE QLTY_ITLD [s] BY
SITE [n]
CS_TITLE [n]
EXPERIENCE_UNIT_TESTING_FRAMEWORK_LIKERT_SCALE [o]
EXPERIENCE_EXPERIMENT_PROGRAMMING_LANGUAGE_ACADEMY_YEARS [s]
EXPERIENCE_EXPERIMENT_PROGRAMMING_LANGUAGE_INDUSTRY_YEARS [s]
OVERALL_EXPERIENCE_PROGRAMMING_ACADEMY_YEARS [s]
OVERALL_EXPERIENCE_PROGRAMMING_INDUSTRY_YEARS [s]
EXPERIENCE_UNIT_TESTING_LIKERT_SCALE [o]
EXPERIMENT_IDE_USED_DUMMY [o] TDD_USED_DUMMY [o]
TASK_ITLD [n]
SLICED_ITLD_DUMMY [o]
  /TREE DISPLAY=TOPDOWN NODES=STATISTICS BRANCHSTATISTICS=YES NODEDEFS=YES SCALE=AUTO
  /PRINT MODELSUMMARY IMPORTANCE SURROGATES RISK TREETABLE
  /GAIN SUMMARYTABLE=YES TYPE=[NODE] SORT=DESCENDING CUMULATIVE=NO
  /METHOD TYPE=CRT MAXSURROGATES=AUTO PRUNE=NONE
  /GROWTHLIMIT MAXDEPTH=AUTO MINPARENTSIZE=12 MINCHILDSIZE=6
  /VALIDATION TYPE=NONE OUTPUT=BOTHSAMPLES
  /CRT MINIMPROVEMENT=0.0001
  /MISSING NOMINALMISSING=MISSING.
```

Decision Trees for the PROD

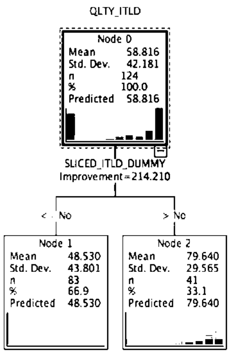
```
TREE PROD_ITLD [s] BY
SITE [n]
CS_TITLE [n]
EXPERIENCE_UNIT_TESTING_FRAMEWORK_LIKERT_SCALE [o]
EXPERIENCE_EXPERIMENT_PROGRAMMING_LANGUAGE_ACADEMY_YEARS [s]
EXPERIENCE_EXPERIMENT_PROGRAMMING_LANGUAGE_INDUSTRY_YEARS [s]
OVERALL_EXPERIENCE_PROGRAMMING_ACADEMY_YEARS [s]
OVERALL_EXPERIENCE_PROGRAMMING_INDUSTRY_YEARS [s]
EXPERIENCE_UNIT_TESTING_LIKERT_SCALE [o] EXPERIMENT_IDE_USED_DUMMY [o]
TDD_USED_DUMMY [o]
TASK_ITLD [n]
SLICED_ITLD_DUMMY [o]
  /TREE DISPLAY=TOPDOWN NODES=BOTH BRANCHSTATISTICS=YES NODEDEFS=YES SCALE=AUTO
  /PRINT MODELSUMMARY IMPORTANCE SURROGATES RISK TREETABLE
  /GAIN SUMMARYTABLE=YES TYPE=[NODE] SORT=DESCENDING CUMULATIVE=NO
  /PLOT IMPORTANCE MEAN INCREMENT=10
  /METHOD TYPE=CRT MAXSURROGATES=AUTO PRUNE=NONE
  /GROWTHLIMIT MAXDEPTH=AUTO MINPARENTSIZE=12 MINCHILDSIZE=6
  /VALIDATION TYPE=NONE OUTPUT=BOTHSAMPLES
  /CRT MINIMPROVEMENT=0.0001
  /MISSING NOMINALMISSING=MISSING.
```

Appendix 10: Decision Trees CART (CRT)

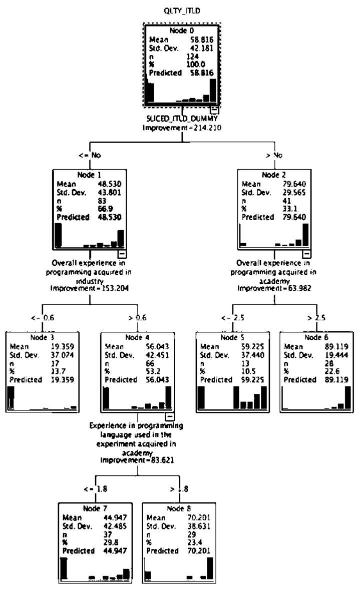
QLTY

Figure 12 shows the decision tree for the QLTY response variable with different number of cases for the parent node (N) and the child node (n).

N=20,
n=20

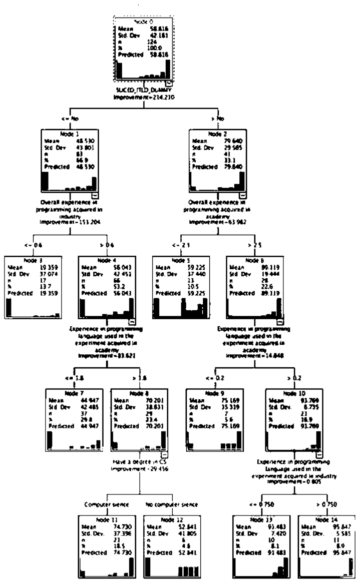


N=10,
n=10



N=5,
n=5

N=10,
n=5



N=12,
N=6

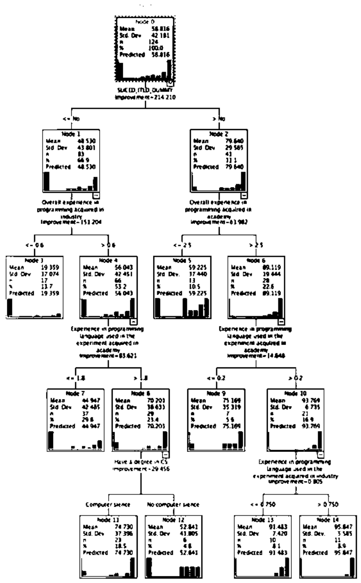
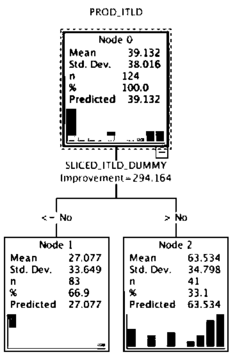
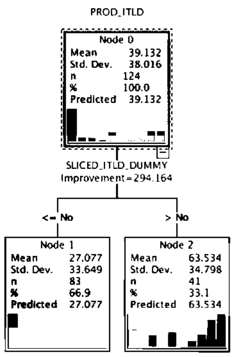

Node 0
SLICED_ITLD_DUMMY
Improvement=214.210
Overall experience in programming acquired in industry
Improvement=153.204
Overall experience in programming acquired in academy
Improvement=63.982
Experience in programming language used in the experiment acquired in academy
Improvement=83.621
Experience in programming language used in the experiment acquired in academy
Improvement=14.848
Have a degree in CS
Improvement=29.456
Experience in programming language used in the experiment acquired in industry
Improvement=0.805
Computer science
No computer science
<= 0.750
> 0.750

Fig. 12 CART decision tree for QLTY

Productivity

Figure 13 shows the decision tree for the PROD response variable with different number of cases for the parent node (N) and the child node (n).

N=20, n=20	 PROD_ITLD; Node 0 Mean 39.132 Std. Dev. 38.016 n 124 % 100.0 Predicted 39.132; SLICED_ITLD_DUMMY Improvement=294.164; <= No; > No; Node 1 Mean 27.077 Std. Dev. 33.649 n 83 % 66.9 Predicted 27.077; Node 2 Mean 63.534 Std. Dev. 34.798 n 41 % 33.1 Predicted 63.534	N=10, n=10	 PROD_ITLD; Node 0 Mean 39.132 Std. Dev. 38.016 n 124 % 100.0 Predicted 39.132; SLICED_ITLD_DUMMY Improvement=294.164; <= No; > No; Node 1 Mean 27.077 Std. Dev. 33.649 n 83 % 66.9 Predicted 27.077; Node 2 Mean 63.534 Std. Dev. 34.798 n 41 % 33.1 Predicted 63.534

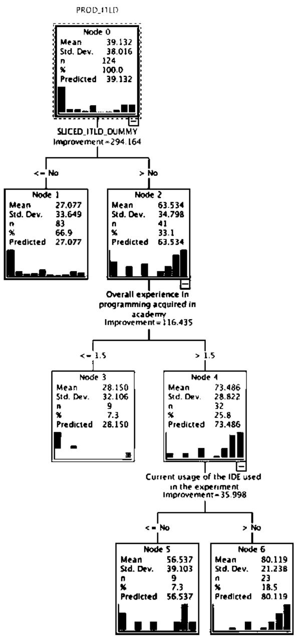
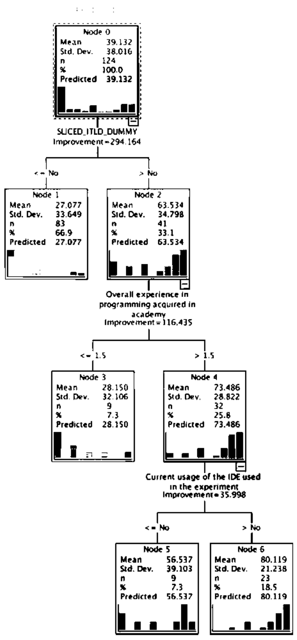
N=5, n=5	 PROD_ITLD; Node 0 Mean 39.132 Std. Dev. 38.016 n 124 % 100.0 Predicted 39.132; SLICED_ITLD_DUMMY Improvement=294.164; <= No; > No; Node 1 Mean 27.077 Std. Dev. 33.649 n 83 % 66.9 Predicted 27.077; Node 2 Mean 63.534 Std. Dev. 34.798 n 41 % 33.1 Predicted 63.534; Overall experience in programming acquired in academy Improvement=116.435; <= 1.5; > 1.5; Node 3 Mean 28.150 Std. Dev. 32.106 n 9 % 7.3 Predicted 28.150; Node 4 Mean 73.486 Std. Dev. 28.822 n 32 % 25.8 Predicted 73.486; Current usage of the IDE used in the experiment Improvement=35.998; <= No; > No; Node 5 Mean 56.537 Std. Dev. 39.103 n 9 % 7.3 Predicted 56.537; Node 6 Mean 80.119 Std. Dev. 21.238 n 23 % 18.5 Predicted 80.119	N=10, n=5
		
		N=12, N=6
			 Node 0 Mean 39.132 Std. Dev. 38.016 n 124 % 100.0 Predicted 39.132; SLICED_ITLD_DUMMY Improvement=294.164; <= No; > No; Node 1 Mean 27.077 Std. Dev. 33.649 n 83 % 66.9 Predicted 27.077; Node 2 Mean 63.534 Std. Dev. 34.798 n 41 % 33.1 Predicted 63.534; Overall experience in programming acquired in academy Improvement=116.435; <= 1.5; > 1.5; Node 3 Mean 28.150 Std. Dev. 32.106 n 9 % 7.3 Predicted 28.150; Node 4 Mean 73.486 Std. Dev. 28.822 n 32 % 25.8 Predicted 73.486; Current usage of the IDE used in the experiment Improvement=35.998; <= No; > No; Node 5 Mean 56.537 Std. Dev. 39.103 n 9 % 7.3 Predicted 56.537; Node 6 Mean 80.119 Std. Dev. 21.238 n 23 % 18.5 Predicted 80.119

Fig. 13 CART decision tree for PROD

References

Adelson B (1981) Problem solving and the development of abstract categories in programming languages. Mem Cogn 9(4):422–433

Adelson B (1984) When novices surpass experts: the difficulty of a task may increase with expertise. J Exp Psychol: Learn Mem Cogn 10(3):483

Agarwal R, Tanniru MR (1991) Knowledge extraction using content analysis. Knowl Acquis 3:421–441

Aranda A, Dieste O, Juristo N (2014) Evidence of the presence of bias in subjective metrics: analysis within a family of experiments. Proceedings of the 18th International Conference on Evaluation and Assessment in Software Engineering (EASE 2014). London, UK, pp 24–27

Arisholm E, Gallis H, Dyba T, Sjoberg DIK (2007) Evaluating pair programming with respect to system complexity and programmer expertise. IEEE Trans Softw Eng 33(2):65–86

Armour PG (2004) Beware of counting LOC. Commun ACM 47(3):21–24

Askar P, Davenport D (2009) An investigation of factors related to self-efficacy for java programming among engineering students. Turk Online J Educ Technol 8(1):26–32

Belsley DA (1991) Conditioning diagnostics: collinearity and weak data in regression. Wiley

Bob U (2005) The bowling game kata. Retrieved from http://butunclebob.com/ArticleS.UncleBob.TheBowlingGameKata

Brandmaier AM, von Oertzen T, McArdle JJ, Lindenberger U (2013) Structural equation model trees. Psychol Methods 18:71–86

Burkhardt J, Détienne F, Wiedenbeck S (1997) Mental representations constructed by experts and novices in object-oriented program comprehension. In: Howard S, Hammond J, Lindgaard G (eds) Springer US, pp 339–346

Burkhardt J, Détienne F, Wiedenbeck S (2002) Object-oriented program comprehension: effect of expertise, task and phase. Empir Softw Eng 7(2):115–156

Camerer CF, Johnson EJ (1997) 10 the process-performance paradox in expert judgment: How can experts know so much and predict so badly? Research on Judgment and Decision Making: Currents, Connections, and Controversies. 342

Campbell RL, Bello LD (1996) Studying human expertise: beyond the binary paradigm. J Exp Theor Artif Intell 8(3-4):277–291

Chase WG, Simon HA (1973) The mind's eye in chess

Chmiel R, Loui MC (2004) Debugging: from novice to expert. ACM SIGCSE Bull 36(1):17–21

Chulis K (2012) Optimal segmentation approach and application. clustering vs. classification trees. Retrieved from http://www.ibm.com/developerworks/library/ba-optimal-segmentation/

Cohen J (1988) Statistical power analysis for the behavioral sciences, 2nd edn. Lawrence Erlbaum Associates, Hillsdale

Colvin G (2008) Talent is overrated: What really separates world-class performers from Everybody Else. Penguin Publishing Group

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

Curtis B (1984) Fifteen years of psychology in software engineering: individual differences and cognitive science. IEEE Press, Orlando

Curtis B, Krasner H, Iscoe N (1988) A field study of the software design process for large systems. Commun ACM 31(11):1268–1287

Darcy DP, Ma M (2005) Exploring individual characteristics and programming performance: Implications for programmer selection. Proceedings of the 38th Annual Hawaii International Conference on System Sciences, 314a.

Daun M, Salmon A, Weyer T, Pohl K (2015) The impact of students' skills and experiences on empirical results: A controlled experiment with undergraduate and graduate students. Proceedings of the 19th International Conference on Evaluation and Assessment in Software Engineering, Art. No. 29.

De Groot AD (1978) Thought and choice in chess. Walter de Gruyter

Erdogmus H, Morisio M, Torchiano M (2005) On the effectiveness of the test-first approach to programming. Softw Eng IEEE Trans 31(3):226–237

Ericsson KA (2006a) The influence of experience and deliberate practice on the development of superior expert performance. The Cambridge Handbook of Expertise and Expert Performance, pp 683–703

Ericsson KA (2006b) An introduction to *cambridge handbook of expertise and expert performance*: Its development, organization, and content. In: Ericsson KA, Charness N, Hoffman RR, Feltovich PJ (eds) The cambridge handbook of expertise and expert performance. Cambridge University Press, pp 3–19

Ericsson KA, Charness N (1994) Expert performance: its structure and acquisition. Am Psychol 49(8):725

Ericsson KA, Lehmann AC (1996) Expert and exceptional performance: evidence of maximal adaptation to task constraints. Annu Rev Psychol 47(1):273–305

Ericsson KA, Krampe RT, Tesch-Römer C (1993) The role of deliberate practice in the acquisition of expert performance. Psychol Rev 100(3):363–406

Experience (2015) from http://www.merriam-webster.com/dictionary/experience. Retrieved 7 Oct 2015

Faul F, Erdfelder E, Lang A, Buchner A (2007) G* power 3: a flexible statistical power analysis program for the social, behavioral, and biomedical sciences. Behav Res Methods 39(2):175–191

Fenton N, Bieman J (2014) Software metrics: a rigorous and practical approach, third edition. CRC Press.

Field A, Miles J, Field Z (2012) Discovering statistics using R. SAGE Publications

Glenwick DS (2016) Handbook of methodological approaches to community-based research: Qualitative, quantitative, and mixed methods. Oxford University Press

Green SB (1991) How many subjects does it take to do A regression analysis. Multivar Behav Res 26(3):499–510

Hedges LV, Olkin I (1985) Statistical methods for meta-analysis. Academic Press

Heiberger RM, Holland B (2013) Statistical analysis and data display: an intermediate course with examples in S-plus, R, and SAS. Springer, New York

ISO I (2011) IEC25010: 2011 systems and software engineering–systems and software quality requirements and evaluation (SQuaRE)–System and software quality models. Int Organ Stand

Jeffries R, Turner AA, Polson PG, Atwood ME (1981) The processes involved in designing software. Cogn Skills Acquis 255:283

Jørgensen M, Faugli B, Gruschke T (2007) Characteristics of software engineers with optimistic predictions. J Syst Softw 80(9):1472–1482

Kitchenham B, Mendes E (2004) Software productivity measurement using multiple size measures. IEEE Trans Softw Eng 30(12):1023–1035

Larkin J, McDermott J, Simon DP, Simon HA (1980) Expert and novice performance in solving physics problems. Science (New York, NY) 208(4450):1335–1342

Lee WK, Chung IS, Yoon GS, Kwon YR (2001) Specification-based program slicing and its applications. J Syst Archit 47(5):427–443

Lui KM, Chan KCC (2006) Pair programming productivity: novice–novice vs. expert–expert. Int J Hum-Comput Stud 64(9):915–925

MacCallum R, Zhang S, Preacher K, Rucker D (2002) On the practice of dichotomization of quantitative variables. 7:10–40

MacDorman KF, Whalen TJ, Ho C, Patel H (2011) An improved usability measure based on novice and expert performance. Int J Hum-Comput Interact 27(3):280–302

Madeyski L (2005) Preliminary analysis of the effects of pair programming and test-driven development on the external code quality. Proceedings of the 2005 Conference on Software Engineering: Evolution and Emerging Technologies. pp. 113–123

Marakas GM, Elam JJ (1998) Semantic structuring in analyst and representation of facts in requirements analysis. Inf Syst Res 9(1):37–63

Mayer RE (1997) From novice to expert. In: Helander M, Landauer TK, Prabhu P (eds) Handbook of human-computer interaction, 2nd edn. Elsevier Science B.V, pp. 781–795

McDaniel MA, Schmidt FL, Hunter JE (1988) Job experience correlates of job performance. J Appl Psychol 73(2):327

McKeithen KB, Reitman JS, Rueter HH, Hirtle SC (1981) Knowledge organization and skill differences in computer programmers. Cogn Psychol 13(3):307–325

Miles J, Shevlin M (2001) Applying regression and correlation: A guide for students and researchers. SAGE Publications

Müller MM, Höfer A (2007) The effect of experience on the test-driven development process. Empir Softw Eng 12(6):593–615

Muller MM, Padberg F (2004) An empirical study about the feelgood factor in pair programming. Proceedings 10th International Symposium on Software Metrics. pp 151–158

Munir H, Moayyed M, Petersen K (2014) Considering rigor and relevance when evaluating test driven development: a systematic review. Inf Softw Technol 56(4):375–394

Nisbet R, Elder J, Miner G (2009) Handbook of statistical analysis and data mining applications. Academic Press

O'brien R (2007) A caution regarding rules of thumb for variance inflation factors. Qual Quant 41(5):673–690

Ricca F, Di Penta M, Torchiano M, Tonella P, Ceccato M (2007) The role of experience and ability in comprehension tasks supported by UML stereotypes. 29th International Conference on Software Engineering. pp 375–384

Riley RD, Lambert PC, Abo-Zaid G (2010) Meta-analysis of individual participant data: Rationale, conduct, and reporting. BMJ 340. doi:10.1136/bmj.c221

Runeson P (2003) Using students as experiment subjects – an analysis on graduate and freshmen student data. Proceedings 7Th International conference on empirical assessment & evaluation in software engineering. pp 95–102

Sheppard SB, Curtis B, Milliman P, Love T (1979) Modern coding practices and programmer performance. Computer 12:41–49

Siegmund J, Kästner C, Liebig J, Apel S, Hanenberg S (2014) Measuring and modeling programming experience. Empir Softw Eng 19(5):1299–1334

Sim SE, Ratanotayanon S, Aiyelokun O, Morris E (2006) An initial study to develop an empirical test for software engineering expertise. Institute for Software Research, University of California, Irvine, CA, USA, Technical Report# UCI-ISR-06-6

Soloway E, Ehrlich K (1984) Empirical studies of programming knowledge. IEEE Trans Softw Eng SE-10(5): 595–609

Soloway E, Bonar J, Ehrlich K (1983) Cognitive strategies and looping constructs: an empirical study. Commun ACM 26(11):853–860

Sonnentag S (1995) Excellent software professionals: experience, work activities, and perception by peers. Behav Inform Technol 14(5):289–299

Sonnentag S (1998) Expertise in professional software design: a process study. J Appl Psychol 83(5):703–715

Votta LG (1994) By the way, has anyone studied any real programmers, yet? Software Process Workshop, 1994. Proceedings., Ninth International. pp 93–95

Weisberg S (2005). Applied Linear Regression, third edition. John Wiley & Sons, Inc., Hoboken, NJ

Weiser M (1981) Program slicing. IEEE Press, San Diego

Weiser J, Shertz J (1984) Programming problem representation in novice and expert programmers. Int J Man-Mach Stud 19:391–398

Wiedenbeck S (1985) Novice/expert differences in programming skills. Int J Man-Mach Stud 23(4):383–390

Williams L, Kudrjavets G, Nagappan N (2009) On the effectiveness of unit test automation at microsoft. software reliability engineering, 2009. ISSRE '09. 20th International symposium on. pp 81–89

Winship C, Mare RD (1984) Regression models with ordinal variables. Am Sociol Rev 49(4):512–525

Ye N, Salvendy G (1994) Quantitative and qualitative differences between experts and novices in chunking computer software knowledge. Int J Hum-Comput Interact 6(1):105–118