Noname manuscript No.
(will be inserted by the editor)

Influence of confirmation biases of developers on software quality: an empirical study

Gül Çalıklı · **Ayşe Başar Bener**

Received: date / Accepted: date

Abstract The thought processes of people have a significant impact on software quality, as software is designed, developed and tested by people. Cognitive biases, which are defined as deviations of human mind from the laws of logic and mathematics, are likely to cause software defects. However, there is little empirical evidence to date to substantiate this assertion. In this research, we focus on a specific cognitive bias type called *confirmation bias*, which is defined as the tendency of people to seek for evidence to verify hypotheses rather than seeking for evidence to falsify them. Due to confirmation bias, developers might perform unit tests to make their program work rather than to break. Hence, confirmation bias is believed to be one of the factors that lead to increased software defect density. In this research, we present a metric scheme to explore the impact of developers' confirmation bias on software defect density. In order to estimate effectiveness of our metric scheme in quantification of confirmation bias within the context of software development, we performed an empirical study that addressed the prediction of the defective parts of software. In our empirical study, we used confirmation bias metrics on five datasets obtained from two companies. Our results provide empirical evidence that human thought processes and cognitive aspects deserve further investigation to improve decision making in software development for effective process management and resource allocation.

Keywords Human factors · Software psychology · Defect Prediction · Confirmation bias

G. Çalıklı
Department of Computer Engineering, Boğaziçi University, 34342, Bebek, Istanbul, Turkey
E-mail: gul.calikli@boun.edu.tr

A. B. Bener
Ted Rogers School of Information Technology Management, Ryerson University, M5B 2K3 Toronto, Canada
E-mail: ayse.bener@ryerson.ca

1 Introduction

Quality of software is often measured by the number of defects in the final product. In [8] Boehm and Basili indicate that about 40-50% of effort in software projects is spent for avoidable rework 80% of which is due to the 20% of the defects. Software testing is the critical process in software development life cycle (SDLC) to detect defects before the product is released. However, software testing is the most resource-consuming phase of SDLC, since approximately 50% of a project schedule is allocated to the testing phase [1], [2].

Defect predictors guide project managers for the effective allocation of resources during the testing phase by pointing out the defect-prone parts of the software. As a result, both increasing the efficiency of the software testing phase and delivering the software product to the market on time become possible. Reported results in software defect prediction literature suggest that further progress in defect prediction performance can be achieved by increasing the content of input data that defect predictors learn rather than using different algorithms or increasing the size of input data [17], [15], [16]. We can group some significant work in the literature in terms of their focus: algorithm driven approaches; data size driven approaches; and data content driven approaches.

Algorithms: In software defect prediction, various machine-learning algorithms have been employed by researchers. Munson and Khoshgoftaar [9] construct discriminant models by using static code metrics as independent data, where multicolinearity among static code metrics is eliminated by Principle Component Analysis. Bullard et. al. [3] propose a rule-based classification model for the prediction of defects in a large legacy Telecomunication system. In [10], Classification and Regression Trees (CART) algorithm is used to identify fault-prone modules in embedded systems. Neural networks is another machine-learning technique used by Khosgoftaar and Szabo [56] for learning defect predictors. Regression models have also been widely used [11] [12], [13], [14]. The model consisting of an ensemble of classifiers proposed by Tosun et. al. [7] combines three algorithms which are Naïve Bayes, Neural Networks and Voting Feature Intervals respectively. In his repeatable set of experiments, Menzies et. al. [15] discovered that the Naïve Bayes classifier with a log-filtering preprocessor on the numeric data outperforms methods such as OneR and J4.8. The results obtained by Menzies at. al. are in line with the results of the benchmark study by Lessmann et. al. [17]. In this benchmark study, Lessmann et. al. also found no significant difference between the performance of Naïve Bayes and more complex machine-learning algorithms.

Data Size: In order to find out whether the performance of defect predictors can be increased by sampling methods due to the unbalanced nature of the defect data, Menzies et. al. [16] performed a series of experiments. They used Naïve Bayes as their algorithm, since it was useful in their previous experiments [15] as well as J4.8 which was used in prior under-over sampling experiments [18], [19]. According to the results obtained, discarding data (i.e. undersampling) does not degrade the performance of the learner. For J4.8 algorithm discarding data improved the median performance

from around 40% to 70%, while under-sampling outperformed over-sampling for both J4.8 and Naïve Bayes. These results are consistent with those of Drummond et. al. [18] and Kamei et. al. [19].

Data Content: The literature also includes instances where metrics other than or in addition to static code attributes were used for defect prediction. Jiang et. al. [20] compared predictor performances that were learned from design metrics, static code features and both for 13 NASA projects. Design metrics were extracted from the requirements documents with a text miner. More accurate results were obtained by using both design and static code metrics rather than individual use. The results obtained were consistent with the results of similar experiments which were previously conducted by Zhao. et. al. [21] for the analysis of a real time Telecommunication system. Zimmerman and Nagappan [22] developed a metric suite that defines the dependency of binary files from a graph-theoretic point of view. The authors used these metrics as an input to linear and logistic regression models to predict the post-release failures of Windows Server 2003. Zimmerman and Nagappan reported a 10% increase in the defect prediction performance due to the inclusion of dependency graphs as input data. Following this research, Nagappan and Ball [23], combined dependency and churn metrics to predict post-release faults in the binary files of Windows Server 2003. The authors concluded that they could predict post-release failure using regression models at a statistically significant level. Tosun et. al. [24] used network and churn metrics as well as static code metrics in order to build defect predictors for different defect categories. According to their results, churn metrics gave the best result for predicting all types of defects. Turhan et. al. in another study [27] reduced the probability of false alarms by supplementing static code metrics by their Call Graph Based Ranking (CGBR) framework.

The above-mentioned research, which aims to improve the input data content, mainly focuses on product attributes and process attributes. However, the thought processes of people may have a significant impact on software defect density, as software is developed and tested by people. Therefore, it is highly likely that further progress in defect prediction performance may be achieved by taking into account information regarding people's thought processes. In this paper, to identify the defect prone parts of software, we address a specific aspect of people's thought processes, namely *confirmation bias*, which is defined as people's tendency to seek for evidence to verify hypotheses. Due to confirmation bias, developers might perform only the tests that make their program work, which in turn leads to an increase in software defect density. In the long run, our high-level research goal is to investigate how the confirmation biases of developers influence software quality. Our research questions we would like to investigate in this paper are:

RQ1: How can we identify measures of confirmation bias in relation to the software development process?

RQ2: How do measures of confirmation bias perform in predicting the defect prone parts of software?

To answer the research question *RQ*1, we propose a methodology to define and extract confirmation bias metrics in relation to the software development process.

We also investigate the effectiveness of these metrics in the prediction of software failure proneness, in order to answer our second research question *RQ*2. For this purpose, we conduct an empirical analysis, where we use five datasets collected from two of our industrial partners in the Telecommunications and Enterprise Resource Planning (ERP) domains respectively. For each dataset, we compare the prediction performance of confirmation bias metrics with the prediction performance of static code metrics and churn metrics respectively as well as all combinations of these three metric types. Our results show that by using only confirmation bias metrics, we obtain defect prediction results which are as good as the results obtained from defect predictors that are learned from *only* churn metrics and *only* static code attributes respectively.

We can summarize the contributions of this work as follows:

1. The definition of confirmation bias within software development/testing domain.
2. A methodology to define and extract confirmation bias metrics.
3. Collection of static code, confirmation bias and churn metrics from five software projects. One of these projects belongs to Turkey's largest Independent Software Vendor specialized in ERP domain, while the remaining four projects belong to Turkey's largest Telecommunication/GSM company.
4. An empirical study to evaluate the effectiveness of confirmation bias metrics in software defect prediction compared to static code and churn metrics as well as all combinations of these three metric types.

The rest of the paper is organized as follows: In Section 2, we cover existing research about the effects of cognitive bias types on software engineering in addition to a survey on people metrics that were used in software defect prediction. Section 3 contains detailed information about confirmation bias in cognitive psychology literature. In this section, we also define confirmation bias in relation with the software development process as well as explaining the analogy between Wason's experiments and unit testing. In Section 4, we explain our methodology to define and extract confirmation bias metrics. The details related to confirmation bias metrics are given in Section 5. In Section 6, we explain our experimental methodology and present our experimental results. The empirical results are discussed in Section 7. Section 8 addresses the threats to validity. Finally, in Section 9 we conclude our work after pointing out possible future directions.

2 Related Work

The notion of cognitive biases was first introduced by Tversky and Kahneman [28]. There are various cognitive bias types such as availability, representativeness, overoptimism, over-confidence, anchoring and adjustment, and confirmation bias. Although an intensive amount of research about cognitive biases exists in the field of cognitive psychology, interdisciplinary studies about the effects of cognitive biases in software development life-cycle are at an immature level. In this section, we mention existing research about the effects of some of these cognitive bias types on software development and effort estimation. This section also includes a survey of people-related metrics, which were used to identify defect-prone parts of software. As far as

we know, our research is the first one to build a defect prediction model that learns from metrics related to a cognitive bias type.

2.1 Effects of Cognitive Biases on Software Engineering

In [29], Stacy and MacMillian emphasize the fact that the thought processes of developers are a fundamental concern in software development. To the best of our knowledge, Stacy and MacMillian are the two pioneers who recognized the potential effects of cognitive biases on software engineering. The authors discuss how cognitive biases might show up in software engineering activities by giving examples from several contexts. However, this work contains no empirical investigations. The authors put forth some ideas with explanations and potential areas that require further research.

Another study that provides empirical evidence about the existence of another cognitive bias type (anchoring and adjustment) within the context of software development is conducted by Parsons and Saunders [31]. Parson and Saunders performed two experiments to investigate the existence of anchoring and adjustment in software artifact reuse. The first experiment they conducted, examined the reuse of object classes in a programming task, whereas their second experiment investigated how anchoring and adjustment bias affected reuse of software design artifacts.

Mair and Shepperd [32] discuss how the cognitive biases of software engineers such as over-optimism and over-confidence contaminate the results obtained by software effort predictors, making them far from being objective. Mair and Shepperd also emphasize that experiments on software developers in realistic settings must be conducted by interdisciplinary teams consisting of cognitive psychologists and computer scientists in order to discover de-biasing strategies. This work by Mair and Shepperd is in the form of a preliminary research and contains no empirical investigation.

On the other hand, Jørgensen et. al empirically investigate some cognitive bias types within the scope of software development effort estimation. According to the empirical findings of Jørgensen[35], an increase in the effort spent on risk identification during software development effort estimations leads to an illusion of control, which in turn leads to more over-optimism and over-confidence. Moreover, as a result of the cognitive bias type availability, risk scenarios which are more easily recalled are overemphasized so that inaccurate effort estimations are made. Jørgensen also empirically investigates how anchoring and adjustment heuristic lead to inaccurate effort estimates [34]. Jørgensen indicates that reasonable results can be obtained only if the reference value for the estimates (i.e. the anchor) is the typical effort of tasks of same category or the effort of the closest analogy.

Finally, empirical evidence which supports existence of confirmation bias among software testers is provided by Teasley et. al. in [30]. In their work, Teasley et. al conduct laboratory experiments as well as observing software testers in their naturalistic environment. The authors found that testers are four times more likely to choose positive tests than to choose negative ones and that the testers are just subject to confirmation bias as novices. Based on the empirical findings of Teasley et. al. and the fact that testing one's own code triggers confirmatory behavior, we can conclude that

developers also exhibit confirmatory behavior during unit testing. Therefore, in our study we focus on the confirmation bias of developers.

2.2 People Related Metrics in Software Defect Prediction

In the literature, various people-related metrics have been used to build defect predictors, yet these are not directly related to the thought processes of people or other cognitive aspects.

Nagappan et. al. [63] defined a metric suite to quantify the complexity of organizations consisting of many teams of software professionals working together. The authors built a model to predict the failure proneness of Windows Vista. They compared the performance of this defect predictor with the performance of models that are learned using code churn, code complexity, code coverage, pre-release bugs, and dependencies respectively. In terms of precision and recall values, their model outperformed all the other models.

Graves et. al. [36] also used metrics regarding development organization that worked on a specific code and number of developers who made changes on that code, as well as churn metrics for the prediction of defective modules. According to the results obtained by the authors, the number of developers who have changed a module did not improve the defect prediction performance. Weyuker et. al. [41] also found that the number of developers is not a major influence to increase the defect prediction performance.

On the other hand, Mockus et. al. [37] found that developer experience is essential to predicting failures. In [38], Weyuker et.al. used developer information that distinguishes developers who are new to a working file or who share the responsibility of that file with other developers, since it is more likely that changes made by such developers would result in faults. However, Weyuker et. al. detected no significant contribution of this kind of developer information to the defect prediction performance. Following this research, the authors later analyzed the effectiveness of individual developer performance on the defect prediction performance and found no evidence of a significant improvement in the defect prediction performance either [39].

Social interaction between developers who have collaborated on the same file during the same time period was modeled as social networks and it is used in defect prediction by Meneely et. al. [40]. The model constructed for an industrial product from Nortel was able to explain 60% of the variance of failures during the testing phase. Pinzger et. al. [42] formed a contribution network by combining modules with developers who contribute to those modules and defined centrality measures to quantify the number of developers contributing to a specific module. The empirical analysis of the data from the Windows Vista project showed that centrality metrics can predict software failures up to a significant extent. Bird et. al. [43] formed a network which is a combination of module dependency and contribution networks to predict fault prone modules. As a result, they were able to predict fault prone binary files with greater accuracy than prior methods which use dependency networks [22] or contribution networks [42] in isolation.

3 Confirmation Bias

In cognitive psychology, *confirmation bias* is defined as the tendency of people to seek for evidence that could verify their hypotheses rather than seeking for evidence that could falsify them. The term confirmation bias was first used by Peter Wason in his rule discovery experiment [44] and later in his selection task experiment [45].

3.1 Wason's Experiments

In order to form a confirmation bias metric suite, we prepared the interactive question and the written question set. Based on the outcomes of these questions, we evaluated metric values for each developer. Interactive question is Wason's Rule Discovery Task itself [44] and the set of written questions is based on Wason's Selection Task [45]. In the following subsection, we briefly explain Wason's two experiments, which he proposed to show the existence of confirmation bias among people.

Wason's Rule Discovery Task: In this experiment, Wason asked his subjects to discover a simple rule about triples of numbers [44]. The experimental procedure can be explained as follows: Initially, the subjects are given a record sheet on which the triple "2, 4, 6" is written. The subjects are told that "2 4 6" conforms to this rule. In order to discover the rule, they are asked to write down triples together with the reasons of their choice on the record sheet. After each instance, the examiner tells whether the instance conforms to the rule or not. The subject can announce the rule only when (s)he is highly confident. If the subject cannot discover the rule, (s)he can continue giving instances together with reasons for his/her choice. This procedure continues iteratively until either the subject discovers the rule or (s)he wishes to give up. If the subject cannot discover the rule in 45 minutes, the experimenter aborts the procedure.

Wason designed this experiment in such a way that subjects mostly showed a tendency to focus on a set of triples that is contained inside the set of all triples conforming to the correct rule. Due to this fact, the discovery of the correct rule was possible only by following a hypothesis testing strategy. Once the subject sees the triple "2 4 6", a set of hypotheses come to her/his mind. An ideal hypothesis testing strategy is to start by giving examples which do not refute all hypotheses the subject has in his/her mind at once. The examples of triples that refute more hypotheses should be given as the subject becomes sure about the rule to be discovered. The hypotheses in mind should be eliminated, modified and created in a strategic manner so that the subject can come up with a single hypothesis at the end. Once the subject is sure about the correct rule, (s)he may also give additional triple instances to verify his/her guess.

Wason's Selection Task: The written set of questions is based on Wason's Selection Task [45]. In the original task, the subject is given four cards, where each card has a letter on one side and a number on the other side. These four cards are placed on a table showing D, K, 4, 7 respectively. Given the rule: "Every card that has a D on one

side has a 3 on the other side", the subject is asked which card(s) must be turned over to find out whether the rule is true or false.

3.2 Confirmation Bias in Relation to Software Development

Due to confirmation bias, developers may perform only the tests that make their program work rather than break the code. This may lead to an increase in software defect density. On the other hand, during all levels of software testing, including unit testing, a systematic hypothesis testing procedure should be followed similar to the one followed by a scientist making experiments in his/her laboratory. In general, scientific inferences are based on the principle of eliminating hypotheses while provisionally accepting the remaining ones. Therefore, similar to a scientist, a software developer should try test scenarios starting from the ones that are less likely to fail the code and proceeding with test scenarios that aim the code to fail. In most cases, there are infinitely many test scenarios which require following a strategy to select the appropriate ones.

Hence, within the context of software development and testing, we extend the definition of confirmation bias to include one or both of the following: 1) The tendency to verify software code, 2) The incompetency to apply strategies to try to fail software code.

Wason's Rule Discovery Task in Relation to Unit Testing: There are similarities between Wason's Rule Discovery task and functional (black-box) testing that are performed by software developers to test functional units of their codes during unit testing. This similarity is also mentioned by Teasley et. al. [64]. According to the findings of Wason's Rule Discovery Task, the subjects have a tendency to select many triples (i.e. test cases) that are consistent with their hypotheses and few tests that are inconsistent with them. Similarly, program testers may select many test cases consistent with the program specifications (positive tests) and a few that are inconsistent with them (negative tests). Moreover, the state space of possible test cases is either infinite or too large to be tested within a limited amount of time. Hence, a strategic approach must be followed that covers both positive and negative test cases while trying to make the code fail during testing in order to find as many defects as possible.

Wason's Selection Task in Relation to Unit Testing: Wason's selection task measures the capability of the subject to use logical rules such as modus ponens and modus tollens as well as his/her tendency to refute the given statement. In unit testing when covering possible scenarios, logical reasoning is required. Moreover, testing correctness of conditional statements in the source code during white box testing also requires logical reasoning skills. In order to explain the analogy between Wason's Selection Task and white box testing, we extend the example given by Stacy and MacMiilian in [29] as follows: Suppose a developer wants to make sure that every instance of a class named "Controller" has been initialized throughout his/her code. Hence, in unit testing, the developer will perform a test that can be thought as checking the validity

of the following hypothesis: ”If an instance’s class is Controller, then it has been initialized”. In that case, we can categorize parts of the code that may need to be tested as follows:

- *category* #1: Parts of the code with instances of Controller that may or may not be initialized.
- *category* #2: Parts of the code with instances of a class other than Controller that may or may not be initialized.
- *category* #3: Parts of the code with initialized instances whose class is unknown.
- *category* #4: Parts of the code with uninitialized instances whose class is unknown.

Logical expression for the hypothesis ”If an instance’s class is Controller, then it has been initialized” would be ”if p then q”, where p stands for the phrase ”an instance’s class is Controller” and q stands for the phrase ”it (class) has been initialized”. According to *modus ponens* given that p is true, ”if p then q” is true only if q is true. Therefore, one must check all instances of the class ”Controller” to guarantee that they have all been initialized. This means that parts of the code which fall into *category* #1 must be tested. However, this is not adequate. Since ”if p then q” is equivalent to ”if not-q then not-p”, one must also check the validity of negated form of the hypothesis which is: ”If an instance has *not* been initialized, then the class of that instance is *not* Controller”. According to *modus tollens*, given that *not-q* is true, *not-p* must also be true. This means that every instance that has not been initialized must be checked to find out whether the class of that instance is ”Controller” or not. Hence, a developer must also test the parts of the code that fall into *category* #4. If we transform each category of the parts of the code into logical expressions, then p stands for *category* #1, *not-p* stands for *category* #2, q stands for *category* #3, and finally *not-q* stands for *category* #4. Although the correct choice for testing would be parts of the code that fall into *category* #1 and *category* #4, a developer might prefer to test the parts of the code that fall into *category* #3 in addition to the parts that fall into *category* #1, due to *confirmation bias*. To summarize, confirmation bias may lead to incomplete unit testing of a code which in turn may lead to overlooking most of the defects.

4 Methodology to Define and Extract Confirmation Bias Metrics

The overall methodology to define and extract confirmation bias metrics is shown in Figure 1, where thick arrows indicate the information flow. The preparation of the interactive question and the written question set was performed concurrently with the creation of the initial metric suite. In order to prepare confirmation bias questions, we performed an extensive survey in cognitive psychology literature. This survey also helped us to initiate the confirmation bias metric suite. There is a mutual information feedback between the preparation of questions and the metric suite update processes since the definition of a new metric sometimes required adding new questions into the written question set. This, in turn, often led to the introduction of more metrics. Having prepared confirmation bias questions and a metric suite, the interactive question

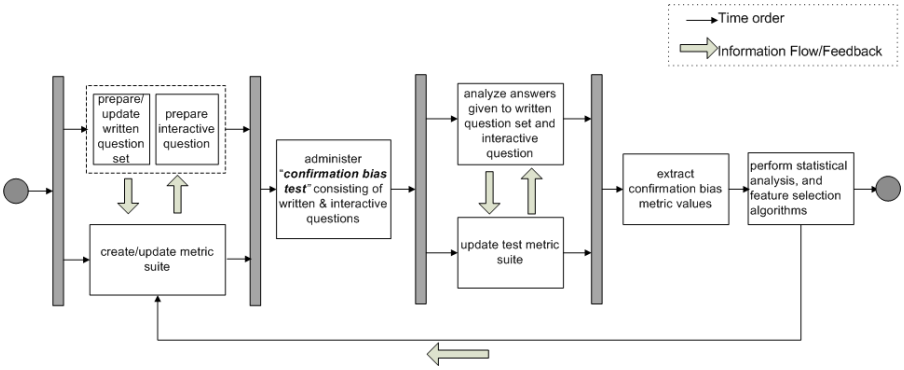

Fig. 1 : Methodology to define confirmation bias metrics and extract confirmation bias metric values.

and the written question set are answered by the participants (software professionals). During the evaluation and analysis of the answers of confirmation bias question given by the participants, new metrics are introduced into the metric suite. Statistical analysis and feature selection techniques help to eliminate metrics that have less significance in the measurement/quantification of confirmation bias. Our methodology for the definition of confirmation bias metric suite is an iterative process. Hence, this procedure is repeated for each new group of participants using confirmation bias questions and the metric suite has been modified at the end of the previous iteration. The extent of the changes regarding the content of the confirmation bias questions and the metric suite were much larger during the early stages of the metric definition process when confirmation bias questions consisting of the interactive question and the written question set were administered to pilot participant groups. In this paper, we present the latest version of the metric suite, for which only minor changes in the content are likely to occur.

4.1 Preparing the Interactive Question and the Set of Written Questions

The interactive question is Wason's Rule Discovery Task itself, whose details are explained in the previous section. The set of written questions is based on Wason's Selection Task and it consists of two parts. The first part contains abstract and thematic questions, whereas the second part contains thematic questions on software development and testing. Table 1 gives information about the distribution of the questions.

Abstract questions require *pure logical reasoning* to be answered correctly. In our question set, there are 8 abstract questions. Compared to thematic problems, it is easier to reason with problems that have thematic content, since real life experience may help to answer such questions easily [46]. For written question set, we prepared 7 thematic questions after having made a literature survey covering the major thematic variants of Wason's original selection task [47], [48], [49], [50],[51], [52], [53], [54],[55]. In written question set, there is one abstract-thematic question. Similar to an abstract question, an abstract-thematic question can be answered correctly

Table 1 Distribution of question types in written question set

Question Type	# of Questions
Part I	
Abstract	7
Abstract+Thematic	1
Thematic	7
Part II	
Software Development	9

by pure logical reasoning. Although such questions seem to have a thematic content, thematic facilitation effect does not take place [46].

4.2 Administration of the Confirmation Bias Test

In order to collect confirmation bias metrics in a controlled manner, we administered the *confirmation bias test*, which consists of the interactive question and the written question set, under a predefined standard procedure. The environment where the *confirmation bias test* was administered was isolated from noise and had adequate lighting. Both Turkish and English versions of the interactive question and the written question set were previously prepared. In this study, participants, who are software developers, took Turkish version of the questions, since their native language is Turkish. The English version of the questions was also required, as in our previous work some of the participants were software developers from North America [59]. Participants were informed about the fact that the results of the confirmation bias test would not be used in their company's performance evaluations and their identity would be kept anonymous. The goal was not to exert pressure on the participants in order not to affect their performance. The participants were also told that there was no time constraints for completing the questions. After the completion of both booklets, the participants were warned not to inform other software developers and testers in their company about the content of the questions.

Below we explain the standard procedures that are specific for the written question set and the interactive question respectively.

Written Question Set In Wason's studies related to his Selection Task, participants were allowed to inspect real packs of cards before the experimenter secretly selected four cards from the pack and placed them on a table so that only a single side of each card was visible. However, the most recent studies in this field rely on the description of the cards and the pictorial representations of the visible sides of the cards either with pencil and paper or on a computer screen. These procedural differences have made insignificant differences in the results of experiments[46].

Since it is possible to administer this part of the confirmation bias test for a group of participants at once, we preferred to use the pen and pencil approach rather than the traditional approach. Hence we prepared the written question set that consisted

of two booklets. The first booklet included abstract, thematic-abstract and thematic questions, while the second booklet consisted of thematic questions with software development/testing theme. Each group of participants consisting of developers, who took part in this research, answered the questions in the first and the second booklet altogether in a meeting/seminar room. Before starting to read and answer the questions, the participants were told to fill in their personal information (gender, education, experience in software development/ testing) on the form. This information was used in our previous research where we investigated the factors affecting confirmation bias as well as the effects of confirmation bias on the performance of software developers and testers [59], [60], [61]. Afterwards, the first booklet was given to the participants so that they started to answer the questions in the first booklet simultaneously.

Interactive Question Each participant answered the interactive question in a separate room and there was one examiner to guide and give feedback to each participant. Before the whole procedure started, the participants were asked whether they gave permission for their voices to be recorded during the session. The goal of voice recording was to catch every detail about the way a participant thought to discover the correct rule. Before starting the test, detailed information was given about the procedure to discover the correct rule.

5 Confirmation Bias Metrics

The metrics in the confirmation bias metric suite are extracted from the interactive question and the written question set respectively. In this research, our concern is unit testing performed by developers. We focus on functional and structural testing, since these are the two testing techniques both of which are used by all developer groups who took part in this research. As it is also stated by Teasley et. al. in [64], there is a similarity between Wason's Rule Discovery Task and functional testing. Since the interactive question is Wason's Rule Discovery Task itself, the hypotheses testing behavior of the developer gives us clues about the strategies employed by the developer to test his/her own code. On the other hand, the metrics extracted from the written test are designed to give clues about the way a developer performs structural testing on his/her code. Structural testing focuses on the logic of the program and its internal structure. Moreover, we need to decide which parts of the code should be tested. Therefore, knowledge about first order logics is required, just as it is required to solve the questions in the written question set correctly. In Table 5, the confirmation bias metrics used in this research are listed. Below, we give details about these metrics and explain how they can inform us about the effectiveness of the unit tests performed by developers. The more effective the unit testing process, the fewer defects will be overlooked by the developers. In that sense, the confirmation bias metrics we have defined are also related to software defect density. Therefore, we use them to build defect predictors.

Table 2 Comparison of average $Ind_{elim/enum}$ and $F_{negative}$ values of developers who announced the correct rule on first trial with remaining developers for each dataset

	$Ind_{elim/enum}$		$F_{negative}$	
Dataset	*Immediate Correct Rule*	*Incorrect Rule*	*Immediate Correct Rule*	*Incorrect Rule*
ERP	1.89	0.53	2.50	0.33
Telecom1	2.68	0.86	2.00	0.46
Telecom2	2.45	0.53	2.70	1.07
Telecom3	2.35	0.51	2.61	1.01
Telecom4	2.37	0.61	2.55	1.06

5.1 Interactive Question Metrics

Total Number of Rule Announcements (N_A): As one of the outcomes of his rule discovery task [44], Wason presents distribution of the participants with respect to the total number of rule announcements made. We defined the metric N_A to measure the total number of rules announced by a participant throughout the interactive question session.

Duration to Solve Interactive Question (T_I): As a performance metric, we defined the metric T_I, which measures total time duration for the interactive question session. Due to the similarity between unit testing and the interactive question, which is Wason's Rule Discovery Task itself, a developer who finds the correct rule in a reasonably short time is also very likely to perform effective unit testing. In commercial software development projects, it is crucial to perform effective testing at all levels in a reasonably short time to meet the release deadlines. However, by referring to only T_I, one cannot make a deduction if a participant employed an effective hypotheses testing strategy when (s)he is solving the interactive question. Therefore, we introduce additional metrics that are described below in this section. These metrics have been designed to contain information about developer's hypotheses testing behavior which in turn gives clues about the strategy the developer employs while testing his/her own code. Some of these metrics also include *time* in their formulation, and *time* is always measured in *minutes*.

Eliminative/Enumerative Index ($Ind_{elim/enum}$): The eliminative/enumerative index ($Ind_{elim/enum}$) was introduced by Wason to evaluate the results of his rule discovery task [44] with the aim of determining the proportion of the total number of instances that are incompatible with reasons to those that are compatible. In [44], Wason concluded that participants who announced the correct rule on first trial had higher $Ind_{elim/enum}$ values compared to the rest of the participants. Our results were also in line with Wason's findings. Developers who took part in this research and who announced the correct rule on the first trial, had an average $Ind_{elim/enum}$ value of 2.43; whereas this value was 0.75 for the rest of the developers. Kruskal-Wallis test results for the significance of the difference between average $Ind_{elim/enum}$ values is ($\chi^2 = 15.42$, p = 8.62E-5). The comparison of the average $Ind_{elim/enum}$ value of developers who announced the correct rule on first trial and those who made incorrect rule announcement(s) is given in Table 2 for each dataset that is used in this research.

The value of the eliminative/enumerative index being lower than 1 implies that participants are more inclined to use triples of numbers (i.e. test cases) that are compatible with their hypotheses. Our participants are developers, and as we mentioned previously, there is a similarity between software testing and Wason's Rule Discovery Task [64]. Hence, developers with $Ind_{elim/enum}$ values lower than 1 are more likely to be inclined to select positive test cases to verify their code. Some flaws in a program such as logical errors can be discovered by positive test cases. However, other flaws will not be discovered unless test cases, which aim to fail the code, are also used. As a result, effective unit testing would not be possible leading to an increase in the amount of defects overlooked during unit testing.

Frequency of Negative Instances ($F_{negative}$): Wason also classifies the results he obtained according to the frequency of negative instances given by the participants ($F_{negative}$). According to the definition by Wason, negative instances are triples of numbers which are incompatible with the correct rule to be discovered. Wason found that the mean frequency of negative instances given by the participants who discovered the correct rule at first announcement is significantly higher than that of the participants who found the correct rule after the announcement of an incorrect rule. Among developers who took part in this empirical study, the average $F_{negative}$ value of 2.31 belongs to those who announced the correct rule on the first trial. On the other hand, for developers who announced an incorrect rule, this value is equal to 0.81. According to Kruskal-Wallis test, the statistical significance of this difference is $\chi^2 = 10.59, p = 0.0011$. The results obtained within each dataset are in line with this cumulative result as shown in Table 2.

Wason obtained a highly significant correlation between $Ind_{elim/enum}$ and $F_{negative}$. We also obtained a Spearman correlation of 0.70 (p = 0.9193E-5) for developers who took part in our research. On the other hand, these two are not entirely the same, since negative instances don't necessarily imply an eliminative behavior. Negative instances might, on the other hand, help to identify the boundaries of the set of all instances that are compatible with the correct rule to be discovered. Similarly, during software testing some test cases may help to identify the missing parts of the software specifications. The more complete the specifications are, the higher the quality of the testing is. For this reason, in addition to $Ind_{elim/enum}$, we also include $F_{negative}$ in our confirmation bias metric suite.

Immediate Rule Announcements (F_{IR} *and* $avgL_{IR}$): When the interactive question was presented to the pilot group, we observed that 17.24% of the participants made immediate rule announcements. However, announcing consecutive rules without giving any instances in between is not part of the protocol. This was explained to each participant before they started to solve the interactive question. Immediate rule announcements were also observed among participants consisting of developers and testers both in this research and previous ones [59], [60], [61]. Immediate rule announcements are an indication of a participant's inadequate hypotheses testing strategies. As a result of such announcements, participants cannot come up with a single rule at the end by eliminating alternative hypotheses in their minds. Equivalence partitioning is a non-exhaustive functional testing technique that is applied to each functional unit mostly together with boundary testing. In equivalence partitioning, a set of dimensions of input data are identified for each functional unit, and a

set of equivalence classes are identified for each dimension. A developer who makes immediate rule announcements when solving the interactive question is very likely to fail to identify all dimensions of input data to be tested in the functional unit testing. Moreover, (s)he will probably fail to properly determine equivalence classes for each dimension. In order to quantify the extent of immediate rule announcements, we defined two metrics, which are immediate rule announcement frequency (F_{IR}) and average length of immediate rule announcements ($avgL_{IR}$) respectively. F_{IR} is the frequency of occurrences where each occurrence corresponds to a series of immediate rule announcements without giving any instances in between. The number of consecutive rule announcements within each rule announcement series may change. Moreover, we also need to discriminate participants who make more consecutive rule announcements within each immediate rule announcement series. This is due to the fact that the increase in the number of consecutive rule announcements implies an increase in the number of alternative hypotheses that the participant was unable to eliminate. $avgL_{IR}$ is the average number of rule announcements made within each series of consecutive rule announcements.

Total Number of Instances Given Per Unit Time ($Instances/Time$): Another metric is the number of instances given per unit time ($Instances/Time$). When solving the interactive question, some participants showed a tendency to guess the correct rule without giving any triple of numbers as instances. As a result, there were long pauses with no interaction between the participant and the experimenter. Such participants usually gave instances only after the experimenter reminded them to do so several times during the interactive question session. Therefore, the value of the metric $Instances/Time$ for such participants was low compared to the rest of the participants. Developers having significantly low $Instances/Time$ metric values are likely to have less tendency to make strategic unit tests. Instead, they have the tendency to consider their code ready for the testing phase, after having performed unit tests with a couple of randomly selected input data. On the other hand, a high value for the metric $Instances/Time$ does not necessarily imply the existence of an ideal hypotheses testing strategy employed by the participant to solve the interactive question. Moreover, a developer having high $Instances/Time$ metric value as an outcome of the interactive question does not necessarily follow a strategy when performing unit tests on his/her code. For instance, more than one instance may have been given for a reason for choice (i.e. to test an alternative hypothesis). This corresponds to selecting more than one test case from an equivalence class.

Total Number of Unique Reasons Given Per Unit Time ($UnqReasons/Time$): On the other hand, the basic assumption of equivalence partitioning is that if the program functions correctly for one test case selected from an equivalence class, then it will function correctly for any test case from that equivalence class. Therefore, we also included the metric $UnqReasons/Time$ into our metric suite. This metric measures the total number of unique reasons stated by a participant for the instances (s)he gives while solving the interactive question.

Total Number of Rules Announced Per Unit Time ($Rules/Time$): Unlike the metric $Instances/Time$, an increase in the value of the metric which measures the total number of rules announced per unit time $Rules/Time$ is not only an indication of the lack of a hypothesis testing strategy to find the correct rule in the interactive

question. A developer having a high $Rules/Time$ value as the outcome of the interactive question has the tendency to deliver his/her code to the testing phase without making adequate unit testing. For such a developer, the compilation of his/her code shall be enough. In other words, high $Rules/Time$ is a result of the developer's rush to solve the interactive question correctly, mostly without checking the correctness of the alternative hypotheses in his/her mind by giving instances.

Total Number of Unique Rules Announced Per Unit Time ($UnqRules/Time$): Among the groups of participants who solved the interactive question, we observed that some participants repeated or reformulated some of the rules (s)he already had announced. Participants exhibiting such behavior while solving the interactive question, are the ones who do not take into account the feedback given by the experimenter. In order to discriminate these developers from the rest, we included $UnqRules/Time$, which measures the unique rules announced per unit time into our confirmation bias metric suite.

5.2 Written Question Set Metrics

In order to quantify the extent of a participant's logical reasoning skills within the context of hypotheses testing, we introduced metrics extracted from the outcomes of the written question set into our confirmation bias metric suite as shown in Table 5. Below, we explain the confirmation bias metrics that are extracted from the answers developers gave to written questions.

Portion of Correctly Answered Questions: S_{ABS} and S_{Th} measure the portion of the correctly answered abstract and thematic questions respectively. A participant who has a low S_{ABS} and a high S_{Th} metric value compensates the lack of his/her logical reasoning skills with the thematic facilitation effects such as daily life experience or memory queuing. S_{SW} is the ratio of the correctly answered questions having software development and testing theme to the total number of such questions. We included S_{SW} into our confirmation bias metric suite in order to find out whether the lack of logical reasoning skills can be compensated through knowledge in software development and testing.

Duration to Solve Written Questions: T_{Th+ABS} is the total time it takes a participant to solve the first part of the written question set, while T_{SW} measures the time it takes a participant to solve the second part of the question set consisting of questions with software development/testing theme.

Insight Metrics: Among our participants, we observed that the majority selected the cards whose visible faces have symbols or words matching the ones in the rule. The information processing model proposed by Johnson-Laird and Wason [65] classifies the participant's performance on Wason's Selection Task as "no insight", "partial insight" and "complete insight" based on the kinds of systematic errors they make. According to the results obtained by both Matarasso Roth [66] and Evans and Lynch [67], participants performing at the level of "no insight" focus on cards mentioned in the rule whose validity is tested. The selection of cards by a participant with "no insight" might be due to the participant's tendency to verify the rule, or (s)he might just match the symbols or words on the cards with those mentioned in the rule. On

Table 3 Distribution of insights within each developer group

		Abstract Questions			Thematic Questions		
Developer Group #	**Dataset #**	**No Insight**	**Partial Insight**	**Complete Insight**	**No Insight**	**Partial Insight**	**Complete Insight**
1	ERP	26.14%	21.43%	7.14%	7.14%	0.00%	90.43%
2	Telecom1	28.57%	12.86%	21.43%	10.00%	2.86%	77.14%
3	Telecom1	31.14%	4.29%	15.57%	10.29 %	5.43%	74.29%
4	Telecom3	31.43%	4.71%	14.29%	10.43%	5.71%	73.29%
5	Telecom4	31.14%	4.14%	14.29%	9.29%	5.00%	74.00%

the other hand, participants performing at the level of "partial insight" or "complete insight" consider what symbols or words occur at the back of each card. In other words, such participants perform a systematic combinatorial analysis of the cards. The difference between these two performance levels is that the participants having "partial insight" select all cards that could either verify or falsify the rule, whereas the participants with "complete insight" select only the cards that have the potential to falsify the rule. Depending on whether the selection task in the written question set is abstract, thematic or thematic-abstract performance of a participant may vary [46]. According to the findings of experiments in cognitive psychology, participants usually perform poorly on abstract questions [46],[65]. This finding is also supported by our empirical results. Table 3 shows that for each project, answers given by the majority of the developers to abstract questions can be categorized as "no insight". On the other hand, the performance of developers on thematic questions is higher as shown in Table 3. When a statement is supposed to be tested for its validity and that statement has a theme consisting of experiences/familiarity from real life settings, then these factors may help the subject to find the correct answer. However, when the statement is purely abstract, one can answer that question by mere logical reasoning.

We introduced three metrics in order to determine participants' performance for both abstract and thematic question types in the written question set. Confirmation bias metrics $ABS_{CompleteInsight}$, $ABS_{PartialInsight}$ and $ABS_{NoInsight}$ measure the number of abstract questions that are answered with "complete insight", "partial insight" and "no insight" respectively. In other words, these metrics give us information about the number of abstract questions that are answered by selecting cards with the symbols that match the ones in the rule as well as the number of questions answered through a systematic combinatorial analysis of the cards. Similar metrics are defined to identify participants' performance on thematic questions that are $Th_{CompleteInsight}$, $Th_{PartialInsight}$ and $Th_{NoInsight}$ respectively. In the written question set, there is only one thematic-abstract question. Hence, instead of defining three separate metrics taking continuous values, we defined a single metric $ThABS_{Insight}$ that can take one of the three categorical values "Complete Insight", "Partial Insight" and "No Insight" respectively. We also defined derived insight metrics, which are given in Table 5.

Falsifier, Verifier and Matcher Categorization: Although insight metrics we defined give information about the existence of a systematic analysis of the cards, the distinction between verification, falsification and matching tendencies are not clear

Table 4 Reich and Ruth's categorization of response tendencies for the selection task

Question Type	Rule	Rule Formulation	Chosen Cards	Response Tendency
Type I	*"If a card has a D on one side, then it has a 7 on its other side."*	if p, then q	3 (not-q)	Falsifier
Type II	*"If a card has a D on one side, then it does not have a 7 on its other side."*	if p, then not-q	3 (not-q)	Verifier
Type III	*"If a card does not have a D on one side, then it has a 7 on its other side."*	if not-p, then q	D (p)	Matcher
Type III	*"If a card does not have a D on one side, then it has a 7 on its other side."*	if not-p, then q	3 (not-q)	Falsifier
Type IV	*"If a card does not have a D on one side, then it does not have a 7 on its other side."*	if not-p, then not-q	D (p)	Matcher
Type IV	*"If a card does not have a D on one side, then it does not have a 7 on its other side."*	if not-p, then not-q	3 (not-q)	Verifier

enough. Reich and Ruth [57] propose an alternative approach to the assessment of falsification, verification and matching tendencies in isolation from one another. For this purpose, they ask four questions to their subjects. In each question, the symbols on the cards and the symbols in the rule are the same. However, the rule whose validity to be tested is in one of the following forms: "if p, then q", "if p, then not-q", "if not p, then q" and "if not p, then not q". Reich and Ruth labels these four questions as TypeI, TypeII, TypeIII and TypeIV questions respectively. The response given to TypeI and TypeII questions help to identify tendencies for falsification and verification respectively. The responses to TypeIII and TypeIV questions are also indications of the existence of falsification and verification tendencies respectively. However, the responses given to these two questions may also hint the existence of a matching tendency. Based on the response tendencies given to these four questions, we defined six metrics, which are $N_{Falsifier}$, $N_{Verifier}$, $N_{Matcher}$, and N_{None} respectively. For instance, $N_{Falsifier}$ measures the total number of questions answered with *falsifying* tendency. The definitions of the rest of the metrics are given in Table 5.

6 Empirical Study

6.1 Datasets

In this study, we used datasets from five different projects as shown in Table 6. In order to build the defect prediction model, we took into account only the source code files whose development activities can be traced through the version control system. Only these active source code files were tested by the testing teams. Therefore, project

Table 5 List of confirmation bias metrics

Interactive Test Metrics	
Metric	**Explanation**
N_A	Number of rule announcements
T_I	Duration of interactive question session (in minutes)
$Ind_{elim/enum}$	Eliminative/enumerative index by Wason
$F_{negative}$	Frequency of negative instances
F_{IR}	Immediate rule announcement frequency
$avgL_{IR}$	Average length of immediate rule announcements
$Instances/Time$	Number of instances given per unit time
$UnqReasons/Time$	Number of unique reasons given per unit time
$Rules/Time$	Number of rules announced per unit time
$UnqRules/Time$	Number of unique rules announced per unit time
Written Test Metrics	
Metric	**Explanation**
S_{Abs}	Score in abstract questions
S_{Th}	Score in thematic questions
S_{SW}	Score in the second part of the written question set
T_{Th+Abs}	Time it takes to answer the first part of the written question set
T_{SW}	Time it takes to answer the second part of the written question set
$ABS_{CompleteInsight}$	Number of abstract questions answered with *complete insight*
$ABS_{PartialInsight}$	Number of abstract questions answered with *partial insight*
$ABS_{NoInsight}$	Number of abstract questions answered with *no insight*
$Th_{CompleteInsight}$	Number of thematic questions answered with *complete insight*
$Th_{PartialInsight}$	Number of thematic questions answered with *partial insight*
$Th_{NoInsight}$	Number of thematic questions answered with *no insight*
$N_{Falsifier}$	Total number of answers with *only* falsifying tendency
$N_{Verifier}$	Total number of answers with *only* verifying tendency
$N_{Matcher}$	Total number of answers with *only* verifying tendency
N_{None}	Total number of answers with no defined tendency

Table 6 Properties of datasets

Dataset	**# of active files**	**Defect rate**	**# of developers**
ERP	3199	0.07	6
Telecom1	826	0.11	9
Telecom2	1481	0.03	4
Telecom3	284	0.02	7
Telecom4	63	0.05	17

managers needed guidance about defect-prone parts of these files to efficiently allocate their testing resources within tight release deadlines. In Table 6, the total number of maintained/developed files, file types and defect rates are listed for each dataset. The defect rate is the ratio of the number of defective files to the number of active files.

Dataset ERP belongs to a project group that consists of 6 developers who are employees of the largest ISV (Independent Software Vendor) in Turkey. The software developed by this project group is an enterprise resource planning (ERP) software. The snapshot of the software that was retrieved from the version management sys-

tem comes from March 2011, and it consists of 3199 java files. The remaining four datasets come from the largest wireless telecom operator (GSM) company in Turkey. Dataset Telecom1 consists of four versions of a software product that is used to launch new campaigns. On average, 545 java files exist in a single version. They make modifications in 206 files per version on average. The rest of the datasets come from the billing and charging system. Among these three projects, the Telecom2 dataset is relatively a new one, and it consists of java and JSP files. The modification and updates involve all existing source code files in the project as well as the creation of new files. Therefore, dataset Telecom2 includes all source code files of the corresponding project. On the other hand, dataset Telecom3 consists of source code files of the revenue collection system. This software package has been developed and maintained since the inception of the GSM company in 1994. On average, there are 1092 java and JSP files in a single version of this software package. However, maintenance, development and software testing activities take place only for 284 files. Dataset Telecom4 is extracted from the database transactions system. It is as old as the revenue collection system and consists of PL/SQL files. Similar to Telecom3, only the files that are maintained and created are taken into account in the defect prediction analysis.

Dataset ERP consists of the single release of a software product; therefore, we did not use any merging process on the data that the defect prediction model would be using to learn and test from. On the other hand, datasets Telecom1 and Telecom2 are obtained by merging the files in four releases of the software. The remaining datasets Telecom3 and Telecom4 are obtained by merging files that come from two releases of corresponding software products. During the merging process, file entries with identical file names are assumed to be different files if and only if corresponding static code metrics are different (i.e. the said file is considered to be modified). Otherwise, such a file is included in the list only once.

6.2 Metric Extraction Process

We performed defect prediction analysis at the granularity level of 'file', since defect data was not available at the granularity level of 'method'. We used the Prest tool to extract static code metrics at 'file' level [58]. The list of the static code metrics that are used in this research are given in Table 7.

We parsed the log files that come from version control systems to extract the churn metrics. Table 8 consists of the list of churn metrics we used as input data to the defect prediction model. The log file for the first dataset contains file commit activities starting from the beginning of July 2007 till the end of February 2011. On the other hand, the log file for the second dataset covers file commit activities starting from the beginning of September 2001 till the end of December 2009. A single log file was retrieved for the third, fourth and fifth datasets covering commit activities starting from the beginning of December 2007 till the end of July 2011. We evaluated the outcomes of the interactive question and written question set to extract confirmation bias metrics. The details about the confirmation bias metric suite that is used to feed the defect prediction model are briefly explained in Section 4. In order to calculate the confirmation bias metrics corresponding to each file, we consolidated confirma-

tion bias metrics from individual developers into developer groups. We looked into individual files in each version and we marked the developers who created and/or modified that file before the code freeze date (i.e. dates when the development phase for that release is over and the testing phase starts) are the ones who are responsible for any defects found in that file. We made this match because some of the previously introduced defects may be overlooked during the testing phase of earlier versions due to defect propagation. Again, we looked into individual files in each file to examine the file commit information in the version control systems by taking code freeze dates into account. As a result, we formed a group of developers who are responsible for a particular file.

We used three different operators to calculate minimum and maximum values of the metrics of developers who committed code to the same source file. Assuming that A_{di} represents the i^{th} confirmation bias metric value of d^{th} developer, $d \in G_j$ means that d^{th} developer is among the group of developers who created and/or modified j^{th} source file, and finally, S_{ji}^{op} represents the resulting i^{th} confirmation bias metric value of j^{th} source file when operator op is applied. op can be one of the operators *min* or *max* which are used to find minimum and maximum values of the i^{th} confirmation bias metric respectively. We can formulize the definition for the *min* and *max* operators as follows:

$$S_{ji}^{max} = max(A_{di} | \forall d \in G_j) \tag{1}$$

$$S_{ji}^{min} = min(A_{di} | \forall d \in G_j) \tag{2}$$

6.3 Defect Matching

All datasets except for the first one were obtained from two project groups within the large scale Telecommunications company. As mentioned previously, the first dataset comes from an ERP software project developed by the ISV company. In order to match the defects in the first dataset, we had to understand the work flow followed by the ISV in their software development lifecycle. The company uses an issue management system. Each issue is stored in this system with a unique issue code. Issues can be a *new feature*, a regular *project item* or a *defect* that needs to be fixed. We managed to match the issue items that were labeled as *defects* with source code files. As per the company's software development policy, developers must write the corresponding unique issue code as a comment before they commit file(s) to the version control system. Therefore, it was possible to match the file committed to the version management system with the corresponding issue item in the issue management system. Figure 2 shows the methodology we followed to extract the list of the defective files. The company provided us with the list of issues extracted from the issue management system. We formed a "final issue list" by taking into account only issue entries with a request type of *defect* and issue status of *different than canceled*. An issue of request type *defect* and of status *canceled* corresponds to defects whose existence we can not verify. These defects have no impact on the customer versions of the software. We

Table 7 List of static code metrics used in experiments

Attribute	Description
McCabe Metrics	
Cyclomatic Complexity $v(G)$	number of linearly independent paths
Cyclomatic Density $vd(G)$	the ratio of the filess cyclomatic complexity to its length
Decision Density $dd(G)$	condition/decision
Essential Complexity $ev(G)$	the degree to which a file contains unstructured constructs
Essential Density $ed(G)$	$(ev(G)-1)/(v(G)-1)$
Maintenance Severity	$ev(G)/v(G)$
Lines Of Code Metrics	
Unique Operands Count	n_1
Unique Operators Count	n_2
Total Operands Count	N_1
Total Operators Count	N_2
Lines Of Code (LOC)	source lines of code
Branch Count	number of branches
Conditional Count	number of conditionals
Decision Count	number of decision points
Halstead Metrics	
Level (L)	$(2/n_1)/(n_2/N_2)$
Difficulty (D)	1/L
Length (N)	N_1+N_2
Volume (V)	$N*log(n)$
Programming Effort (E)	$D*V$
Programming Time (T)	$E/18$

Table 8 List of churn metrics used in experiments

Attribute	Description
commits	number of commits made for a file
committers	number of committers who committed a file
commitsLast	number of commits made for a file since last release
committersLast	number of developers who committed a file since last release
rmlLast	number of removed lines from a file since last release
alLast	number of added lines to a file since last release
rml	number of removed lines from a file
al	number of added lines to a file
topDevPercent	percentage of top developers who committed a file

mined the commit log file in the version control system to get a commit history file. The format of each commit log entry is shown in Figure 2. We, then, found the names of source files for each issue in the list and marked them as defective.

The second dataset comes from the project group with whom we have been doing collaborative research [26]. This project group provided us with a list of the source files that they found to have bugs during the testing phase of each release. However,

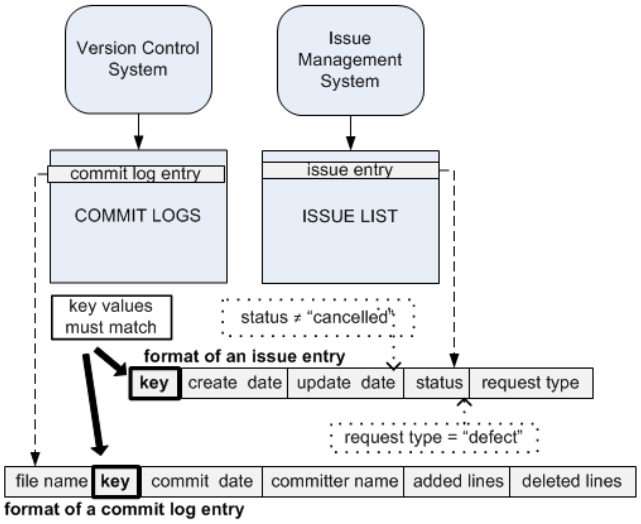

Fig. 2 Defect matching procedure of a file for the last dataset

the third, fourth and fifth datasets come from project groups with whom we have not worked before. They provided us with a list of file commit activities to fix defects. We were told that the second group labels its final release of the product under one single release number although the final product is composed of many releases. We were also a given release calendar that contains the code freeze and production release dates of each release. The release calendar also included information about the defect detection and defect fix dates for each defect. We considered that a file comes from a specific release if and only if the date when that defect is detected and/or fixed is later than the code freeze date and earlier than the production release date. As a result, we were able to match each file with a specific release number, in addition to labeling defective files for each release.

6.4 The Relationship Between Confirmation Bias Metrics and Defect Rate

In section 5, we made claims about the correlation between developers' lack of falsifying behavior in confirmation bias test and their tendency to validate their code during unit testing. In order to justify our claims, we made an empirical analysis where we used pre-release defect rates of developer groups as a measure of developers' confirmatory behaviors during unit testing. This empirical analysis was also essential to choose the appropriate confirmation bias metrics as an input for defect prediction models. The basic steps of our empirical analysis can be summarized as follows:

1. *Formation of Developer Groups*: For each project, we analyzed the log file obtained from the version control management system. Since the majority of the

files were committed by more than one developer, we decided to focus on developer groups who created and/or updated a certain set of files rather than individual developers.

2. *Estimation of Developer Groups' Confirmation Bias Metric Values*: Since the performance of a group or system is determined by its weakest component [80] , we selected the metric value that is the indication of the highest confirmation bias level (i.e. highest tendency for verification) in a given developer group. For instance, a low eliminative/enumerative index ($Ind_{elim/enum}$) value is among the indications of high confirmation bias. Therefore, to estimate $Ind_{elim/enum}$ for a developer group, we use *min* operator. On the other hand, to estimate N_A metric value for a developer group, we use *max* operator, since high N_A metric value is an indication of high confirmation bias.
3. *Estimation of Defect Rate for Each Developer Group*: We defined the defect rate for each developer group as the ratio of the total number of defected files created/updated by that group to the total number of files that group created/updated. We can formulate the defect rate dr^i for i^{th} developer group, where $N^i_{defectiveFiles}$ and $N^i_{allFiles}$ stand for number of defective files and number of all files, as follows:

$$dr^i = N^i_{defectiveFiles} / N^i_{allFiles} \tag{3}$$

4. *Estimation of the Correlation Between Developer Groups' Confirmation Bias Metrics and Defect Rate*: We calculated the Pearson Product Moment Correlation between the confirmation bias metrics of each developer group and the defect rate of the same group. Significant correlation results are given in Tables 10 and 11. These tables also include information about whether *min* or *max* operator is used to calculate the value of that metric for any developer group.

The results shown in Table 10 and Table 11 verify what we have claimed in section 5. Although the magnitude of the correlation values ranges between 0.10 and 0.50 for p values smaller than the significance level 0.05, these correlation results are quite significant in the field of behavioral sciences. In order to make an empirical analysis in the field of behavioral sciences, one needs to move from theoretical constructs such as hypothetically strong relationships to their operational realizations in measurement. The guidelines of magnitude of Pearson correlation for the behavioral sciences given by Cohen in [74] [75] are as follows:

- $\rho = 0.10 - 0.23$ is small effect size,
- $\rho = 0.24 - 0.36$ is medium effect size, and
- $\rho = 0.37 - larger$ is large effect size.

The power values of one tailed t test (i.e. the probability of rejecting the null hypothesis, $H_0 : \rho = 0$) for the significance criterion of $\alpha = 0.05$ are listed in Table 9 for $n = 120$ samples.

Interpretation of the Results As shown in Table 10, we could not find any correlation between metric T_I and defect rate. This result is in line with our claim about T_I in Section 5.1, as duration to solve interactive question does not alone give much information about the existence of a hypotheses testing strategy. According to Cohen

Table 9 Conventional definitions of effect size offered by Cohen

ρ	**Power Value**
0.10	0.29
0.20	0.71
0.30	0.96
0.40	≥ 0.995
0.50	≥ 0.995

Table 10 Results for the Pearson correlation between developer groups' interactive question metrics and defect rate

Metric Name	**min/max operator ?**	ρ	*pval*
N_A	max	+ 0.2092	0.0003
$Ind_{elim/enum}$	min	– 0.2547	1.1E-05
T_I	max	+ 0.0396	0.5014
$F_{negative}$	min	– 0.3546	4.8E-10
F_{IR}	max	+ 0.1252	0.0327
$avgL_{IR}$	max	+ 0.5297	1.9E-22
$Instances/Time$	min	– 0.2389	3.8E-05
$UnqReasons/Time$	min	– 0.2355	5E-05
$Rules/Time$	max	– 0.4493	7.2E-16
$UniqueRules/Time$	max	– 0.4510	5.5E-16

Table 11 Results for the Pearson correlation between developer groups' written question set metrics and defect rate

Metric Name	**min/max operator ?**	ρ	*pval*
S_{ABS}	min	– 0.2371	4.4E-5
S_{Th}	min	+ 0.0536	0.3622
S_{SW}	min	+ 0.0367	0.5332
T_{ThABS}	max	– 0.1396	0.0172
T_{SW}	max	+ 0.1553	0.0172
$AbsCompleteInsight$	min	– 0.1280	0.0290
$AbsPartialInsight$	max	+ 0.3572	3.5E-10
$AbsNoInsight$	max	+ 0.5364	4.4E-23
$ThCompleteInsight$	min	– 0.2851	7.7E-7
$ThPartialInsight$	max	+ 0.1128	0.0545
$ThNoInsight$	max	+ 0.1342	0.0220
$N_{Falsifier}$	min	– 0.0935	0.8738
$N_{Verifier}$	max	+ 0.3742	4.2E-11
$N_{Matcher}$	max	+ 0.1749	0.0027
N_{None}	max	+ 0.1852	0.0015

[75], the degree of correlation between F_{IR} and defect rate is low; whereas the degree of correlation between $avgL_{IR}$ and defect rate is high. Non-zero values for these two metrics are indications of the fact that a developer is unable to determine equivalence classes properly during unit testing. However, compared to F_{IR}, $avgL_{IR}$ is more informative, since high $avgL_{IR}$ value implies an increase in the number of alternative hypotheses that a developer is unable to eliminate due to his/her poor hypothesis test-

ing strategy. The degree of correlation between defect rate and N_A is low. N_A is an indication of whether developer succeeds in solving the interactive question (i.e. finds the correct rule on the first trial). However, this metric does not explain the hypotheses testing strategy employed by the developer. The degree of the correlation between the defect rate and $Ind_{elim/enum}$ is medium which is a meaningful result. One cannot expect a decrease in defect rate for very high $Ind_{elim/enum}$ values, since mere eliminative behavior does not help us to find the correct rule. While solving the interactive question, one must start with instances that conform to the alternative hypotheses (s)he has in his/her mind [79]. Similarly, during unit testing a developer may start with positive tests and evaluates the code executions, and then, (s)he must switch to negative test scenarios. The high frequency of negative instances $F_{negative}$ may help to identify the boundaries of instances that conform to the correct rule when solving the interactive question. However, $F_{negative}$ is a direct indicator of neither verifying nor falsifying tendency. Therefore, a medium degree of correlation between defect rate and $F_{negative}$ is acceptable. Our correlation results regarding metrics $Instances/Time$ and $UnqReasons/Time$ are also in line with our claims in Section 5. Although very low values of these two metrics imply a lack of unit testing strategy, high values do not necessarily imply an existence of a strategy in testing. A high level of correlation between defect rate and $Rules/Time$ as well as the $UniqueRules/Time$ support our claims in Section 5 since an increase in the number of announced rules directly imply a lack of testing strategy when solving the interactive question.

Table 11 summarizes the correlation results for written question set metrics. The degree of correlation between defect rate and S_{ABS} is close to medium, but we could not find statistically significant correlation values for S_{Th} and S_{SW}. Since daily life experiences and domain knowledge such as software engineering may help, the majority of developers answered the thematic questions correctly. Therefore, the score in thematic questions and the score in questions with software development/testing theme were not discriminative enough for developer groups with high and low defect rates. In order to test the validity of our claim, we also defined a derived metric to quantify the ratio of the score in thematic questions to the score in abstract questions: $(1+S_{Th})/(1+S_{ABS})$. We added 1 to the denominator to avoid division by zero and 1 to the numerator to obtain values greater than zero. The correlation between defect rate and the derived metric $(1+S_{Th})/(1+S_{ABS})$ is $\rho = +0.4655$ $p = 4.7E-17$. According to Cohen [75], the degree of this correlation is large. We found a negative correlation with the defect rate and the duration to solve the general part of the written test T_{ThAbs}. Developers who had a matching or verifying tendency answered the questions more quickly compared to developers with a falsifying tendency. Moreover, developers who could not even be categorized as matcher or verifier were the ones who answered these questions the fastest. However, there is a positive correlation between defect rate and T_{SW}, which also makes sense as expertise in software development/testing domain helps developers to answer questions correctly. In other words, abstract reasoning skills are not crucial to succeed in the second part of the written question set. When we examine insight metrics, we realize that insight metrics that are extracted from abstract questions are much more powerful indicators of defect rate. With an exception of the metric $N_{Falsifier}$, the results on the metrics that were categorized based on Reich and Ruth [57] are in line with our claims

```
M = 10
N = 10
DataSet = [ERP, Telecom1, Telecom2, Telecom3, Telecom4]
Algorithm = Naive Bayes
PreProcessing = UnderSampling
FeatureSelection = CorrelationAnalysis
FOR EACH data IN DataSet
    REPEAT M TIMES
       data'  ← randomizeOrderOfData(data)
       data'' ← generateBinsFromData(data',N)
       FOR i  ← 1,N
          TestSet  ← data''[i]
          TrainSet ← data'' - data''[i]
          Predictions[i,...] = apply(FeatureSelection, PreProcessing, Algorithm,
                  TrainSet, TestSet)
       END
    END
END
```

Fig. 3 Pseudocode for construction of the defect prediction model

in Section 5. On the other hand, the correlation value between the derived metric $(1+N_{Falsifier})/(1+N_{Verifier})$ and defect rate is $\rho = -0.2280$ $p = 8.7E-5$. This implies that as the number of questions answered with falsifying tendency increase with respect to the number of questions that are answered with verifying tendency, the defect rate decreases. According to Cohen, the degree of this correlation is low but close to medium.

6.5 Construction of the Prediction Model

In this study, we used the Naïve Bayes algorithm since it combines signals coming from different attributes [15]. In software defect prediction studies, it is also empirically shown that the performance of the Naïve Bayes is amongst the top algorithms [17]. As shown in Table 6, datasets are imbalanced. In other words, the number of defective files is far less than the number of defect-free files. Therefore, we use the under-sampling method, which is the most suitable sampling method for our datasets [16]. The pseudocode of the prediction model is given in Figure 3. In order to overcome ordering effects, we shuffled the data 10 times, and 10-fold cross validation was used for each ordering configuration of input data. In other words, for each ordering configuration, we create 10 stratified bins: 9 of these 10 bins are used as training sets, and the last one is used as the test set [62]. As a result, during each experiment, the Naïve Bayes algorithm with under-sampling is executed 10*10=100 times for each dataset.

6.6 Performance Measures

In order to evaluate the performance of the defect prediction models that are built by using different metric suite combinations, we used well-known performance measures that are a probability of detection, false-alarm rate and balance [15].

Probability of Detection (pd): Pd measures how good a predictor is in finding defective modules, where modules can be files, methods or packages depending on the granularity level. In the ideal case, we expect a predictor to catch all defective modules, which in turn, implies that pd is equal to 1.

Probability of False Alarms (pf): Pf measures false alarm rates, when the predictor classifies defect-free modules as defective. In the ideal case, we expect a predictor to classify none of the defect-free modules as defective. In other words, the value of pf is equal to 0.

Balance (bal): In practice, the ideal case where a defect predictor has a high probability of detecting defective modules and a low probability of false alarm is very rare. Therefore, we try to balance the pd and pf values. The notion of balance is formulized as the Euclidean distance from the *sweet spot* (pd = 1 and pf = 0) normalized by the maximum possible distance to this spot. It is desirable that predictor performance is close to the *sweet spot* as much as possible.

$$bal = 1 - \frac{\sqrt{(1-pd)^2 + (0-pf)^2}}{\sqrt{2}} \tag{4}$$

Pd and pf values are calculated using the Confusion Matrix that is given in Table 12. In the confusion matrix, TP is the number of correctly classified defective modules; FP is the number of non defective modules that are classified to be defective; FN is the number of defective modules that are classified to be non-defective; and finally, TN is the number of correctly classified non-defective modules. The formulations for pd and pf in terms of confusion matrix values is given below:

$$pd = TP/(TP+FN) \tag{5}$$

$$pf = FP/(FP+TN) \tag{6}$$

Table 12 Confusion matrix *TP:True Positives, FN:False Negatives, FP:False Positives, TN:True Negatives*

	Predicted	
Actual Case	Defected	Not-defected
Defected	TP	FN
Not-defected	FP	TN

6.7 Results of the Empirical Study

In this section, we discuss the performance results of the defect prediction models that are constructed by taking all seven combinations of static code, confirmation bias and churn metrics using the datasets ERP, Telecom1, Telecom2, Telecom3 and Telecom4 respectively. *Pd*, *pf* and *balance* values, which are listed in Tables 9-13, are the average performance values of these defect predictors.

Table 13 Experiment results for dataset ERP

Metric Types					
Confirmation Bias	Static Code	Churn	pd	pf	balance
-	+	-	0.72	0.29	0.69
+	-	-	0.91	0.31	0.74
-	-	+	0.81	0.38	0.66
+	+	-	0.93	0.30	0.76
-	+	+	0.71	0.15	0.74
+	-	+	0.77	0.27	0.69
+	+	+	0.93	0.32	0.74

The performance results for the dataset ERP are summarized in Table 13. The probability of the detection (*pd*) of the defect predictor that is built using only confirmation bias metrics is higher than the *pd* value of the predictor that is built using only static code metrics. According to the results of the Kruskal-Wallis test, the statistical significance of this difference is $\chi^2 = 52.84$, p = 3.62E-8. However, there is no statistically significant difference between false alarm rates ($\chi^2 = 0.36$, $p = 0.55$) or between balance values ($\chi^2 = 2.84$, $p = 0.092$). On the other hand, the defect prediction model that is learned using only confirmation bias metrics has lower false alarm rates (pf) and higher balance values (*bal*) when compared to the model that take only churn metrics as input. The Kruskal-Wallis test results indicating the statistically significant difference in pf values are $\chi^2 = 62.70$, $p = 0.0060$; whereas the results for the difference in *bal* values are $\chi^2 = 15.29$, p = 9.23E-5. When both static code and churn metrics are used, no statistically significant difference is observed between the average balance value of the resulting defect predictor and the balance value of the predictor built using only confirmation bias metrics ($\chi^2 = 0.85$, $p = 0.3563$). Using both static code and confirmation bias metrics leads to a significantly higher balance value compared to the balance value obtained from the individual usage of static code metrics ($\chi^2 = 27.26$, p = 1.78E-7). Supplementing churn metrics with confirmation bias metrics to learn the defect predictors also resulted in an improvement in defect prediction performance. The average balance value of the defect predictor that is constructed using only confirmation bias metrics is 0.66, and this value increases to 0.69 as a result of the inclusion of confirmation bias metrics. The difference between these two prediction performance values is significantly different as indicated by the Kruskal-Wallis test, $\chi^2 = 4.17$, p = 0.0412. However, using confirmation bias metrics in addition to static code and churn metrics did not result in a significant difference in

defect prediction performance compared to using both static code and churn metrics ($\chi^2 = 0.04$, $p = 0.8468$).

Table 14 Experiment results for dataset Telecom1

Metric Types					
Confirmation Bias	Static Code	Churn	pd	pf	balance
-	+	-	0.60	0.41	0.58
+	-	-	0.66	0.38	0.62
-	-	+	0.49	0.30	0.55
+	+	-	0.67	0.33	0.67
-	+	+	0.57	0.32	0.61
+	-	+	0.60	0.26	0.62
+	+	+	0.62	0.28	0.66

The defect prediction performance results obtained for the dataset Telecom1 are in line with the results obtained for dataset ERP. As it can be seen from Table 14, the individual usage of confirmation bias metrics leads to defect prediction performance ($balance = 0.62$) which is higher than the performance obtained by individual usage of static code metrics ($balance = 0.58$) and churn metrics ($balance = 0.55$). According to the Kruskal-Wallis test, the statistical significance of these differences are ($\chi^2 = 21.35$, p = 3.82E-6) and ($\chi^2 = 54.42$, p = 1.62E-8) respectively. There is no significant difference between the balance value of the defect predictor that is learned using both static code and churn metrics and the balance value that is obtained by using only confirmation bias metrics ($\chi^2 = 0.36$, $p = 0.55$). The defect prediction performance result obtained for this dataset by using both static code and confirmation bias metrics is also significantly higher than the prediction performance results (i.e. balance values) obtained from the individual usage of static code ($\chi^2 = 127.13$, p = 1.74E-29) and confirmation bias metrics respectively ($\chi^2 = 28.01$, p = 1.21E-2). The introduction of churn metrics in addition to confirmation bias metrics does not lead to a significant improvement in defect prediction performance. The Spearman correlation between churn metrics and 15.67% of confirmation bias metrics is higher than or equal to 0.50. The correlation between 6.72% of confirmation bias metrics with static code metrics is greater than or equal to 0.50. The highest Spearman correlation between churn and confirmation bias metrics is 0.58, p = 1.21E-75; whereas the corresponding value between static code and confirmation bias metrics is -0.53, p = 7.90E-62. Using static code, confirmation bias and churn metrics altogether result in an average balance value that is far better than the average balance values obtained from the individual usage of these three metrics types. Moreover, the resulting defect predictor outperforms the defect predictor which is learned from static code and churn metrics as well as exceeding the performance of the prediction model which is learned from confirmation bias and churn metrics. However, the highest defect prediction performance is obtained by using static code and confirmation bias metrics.

Unlike the results obtained for the datasets ERP and Telecom1, , the individual usage of confirmation bias metrics for dataset Telecom2 resulted in average defect prediction performance ($balance = 0.61$) which is lower than those of the defect pre-

Table 15 Experiment results for dataset Telecom2

Metric Types					
Confirmation Bias	Static Code	Churn	pd	pf	balance
-	+	-	0.63	0.33	0.63
+	-	-	0.60	0.35	0.61
-	-	+	0.70	0.32	0.64
+	+	-	0.69	0.29	0.69
-	+	+	0.68	0.26	0.68
+	-	+	0.64	0.35	0.62
+	+	+	0.70	0.32	0.67

dictors, which are learned by the individual usage of static code (*balance* = 0.63). According to the Kruskal-Wallis test, the difference is significant: $\chi^2 = 11.71$, p = 0.0006. However, no significant difference is detected between the defect prediction performance value obtained by using only churn metrics and the performance value obtained by using only confirmation bias metrics ($\chi^2 = 1.4$, p = 0.2368). Moreover, the defect prediction model that is learned by using both static code and churn metrics outperformed (*balance* = 0.68) all three prediction models that are learned from the individual use of static code, confirmation bias and churn metrics respectively ($\chi^2 = 110.48$, p = 7.69E-26). Using both static code and confirmation bias metrics also led to higher average defect prediction performance compared to the performance results obtained from individual usage of static code metrics ($\chi^2 = 21.97$, p = 2.77E-6). However, using confirmation bias metrics in addition to churn metrics gave a lower defect prediction performance result compared to the performance result obtained by using churn metrics only ($\chi^2 = 34.83$, p =3.6E-9). This is due to the high correlation between the churn and confirmation bias metrics. The Spearman correlation between churn metrics and 56.72% of confirmation bias metrics is higher than or equal to 0.70. The Spearman correlation between churn metrics and 23.12% of confirmation bias metrics is higher than or equal to 0.85. Moreover, the maximum Spearman correlation value is 0.94, p = 0. In contrast, the highest Spearman correlation value between static code and confirmation bias metrics is $0,37$, p = 2.81E-49. Therefore, the defect prediction performance improves significantly by supplementing the static code metrics with confirmation bias metrics, compared to the performance values obtained by using static code and confirmation bias metrics separately.

The experiment results for dataset Telecom3 are shown in Table 16. Using only confirmation bias metrics results in a better defect prediction performance than using churn metrics only ($\chi^2 = 9.2$, p = 0.0024). On the other hand, the improved performance results are obtained by individual usage of static code metrics compared to the results obtained from individual usage of confirmation bias metrics ($\chi^2 = 13.31$, p = 0.0003). Supplementing static code metrics with confirmation bias metrics leads to a defect prediction performance that is significantly lower than the performance obtained by using static code metrics only ($\chi^2 = 6.13$, p = 0.0133). This is due to the existence of the correlation between confirmation bias metrics and static code metrics. The Spearman correlation between 17.91% of confirmation bias metrics and static

Table 16 Experiment results for dataset Telecom3

Metric Types					
Confirmation Bias	Static Code	Churn	pd	pf	balance
-	+	-	0.83	0.12	0.81
+	-	-	0.90	0.23	0.78
-	-	+	0.75	0.24	0.67
+	+	-	0.87	0.20	0.78
-	+	+	0.85	0.14	0.81
+	-	+	0.87	0.24	0.76
+	+	+	0.87	0.25	0.75

code metrics is greater than or equal to 0.45. The maximum estimated Spearman correlation is 0.50, p = 1.49E-19 whereas the correlation between *cyclomatic complexity* and churn metric *rml* (i.e. total number of removed lines) is $\rho = 0.67, p = 0.0054$. The correlation between *Halstead length* and churn metric *al* (i.e. total number of added lines) is $\rho = 0.85, p = 0.0231$. Consequently, there is an improvement in the prediction performance when churn metrics are used with static code metrics. However, these results are not higher than the performance of the prediction model which is built using only static code metrics. Similarly, the Spearman correlation between 26.87% of confirmation bias metrics and churn metrics is higher than or equal to 0.45. The maximum Spearman correlation is $\rho = 0.60, p = 0.0077$. Hence, supplementing static code metrics with confirmation bias metrics leads to a degradation in prediction performance. As a result of the correlation among static code, confirmation bias and churn metrics , when metrics from all three metric types are used together to learn a defect prediction model, a degradation in defect prediction performance is observed.

Table 17 Experiment results for dataset Telecom4

Metric Types					
Confirmation Bias	Static Code	Churn	pd	pf	balance
-	+	-	0.91	0.08	0.88
+	-	-	0.93	0.15	0.85
-	-	+	0.90	0.11	0.86
+	+	-	0.93	0.21	0.81
-	+	+	0.83	0.04	0.85
+	-	+	0.94	0.11	0.88
+	+	+	0.94	0.10	0.89

Table 17 summarizes the experiment results for dataset Telecom4. The defect predictor that is built by using only static code metrics outperforms the prediction model that is built by using confirmation bias metrics only ($\chi^2 = 13.31$, p = 0.0003). The performance of the latter defect prediction model is also outperformed by the model that is built by using churn metrics only ($\chi^2 = 9.2$, p = 0.0024). Both of these results are in line with the corresponding results of dataset Telecom2. Supplementing static code metrics with confirmation bias metrics leads to a degradation in the defect prediction performance ($\chi^2 = 18.43$, p = 1.76E-5). The Spearman correlation

between 11.19 % of confirmation bias metrics and static code metrics is higher than or equal to 0.45, while the average Spearman correlation is 0.32. On the other hand, supplementing churn metrics with confirmation bias metrics does not cause a significant improvement in the defect prediction performance ($\chi^2 = 0.37$, $p = 0.544$). The Spearman correlation between 18.66 % of confirmation bias metrics and churn metrics is greater than or equal to 0.45, while the average Spearman correlation is 0.40. For similar reasons, the introduction of confirmation bias metrics to the metric set of static code and churn metrics does not lead to a statistically significant improvement in the defect prediction performance ($\chi^2 = 0.11$, $p = 0.7455$).

We can summarize the experiment results for all five datasets as follows:

- The performance of defect prediction models built by using only confirmation bias metrics is comparable with the performance of the defect prediction models that use static code metrics and churn metrics.
- Any combination of static code, churn and confirmation bias metrics may not lead to an increase in the defect prediction performance. A possible explanation is that the combination of metrics may not correspond to an increase in information content since there is as a high correlation between any two of the static code, confirmation bias and churn metrics. Our purpose in this research was not to find a better defect prediction model; we rather wanted to understand the impact of people-related attributes in determining the defect proneness of the software product. Many aspects of people as social actors or individuals may be measured. In this research, we focused only on the thought process of people since it was a mature research area in psychology. We built defect prediction models to validate our research questions. Our empirical results show us that the thought processes of people have a significant impact on the defect proneness of software since the defect prediction model built by using confirmation bias metrics gives as good results as other metrics sets in all datasets. The next section discusses in more detail the purpose of this research and its contributions.

7 Discussions

The first goal of this research was to identify the measures of confirmation bias in relation to software development process. We prepared a confirmation bias test that consists of an interactive question and a written question set. As mentioned previously, we used a collection of questions that were proposed by cognitive psychologists to prove the existence of confirmation bias among people [44], [45], [54], [53], [55], [47], [48], [49], [50], [51], [52]. Confirmation bias is a cognitive bias type, and cognitive biases are defined as the deviations of the mind from the laws of logic and mathematics. Hence, confirmation bias and intelligence as well as abstract/logical reasoning skills may be correlated. Moreover, abstract questions in the written question set require logical reasoning skills in addition to the tendency to refute hypotheses. A relation between intelligence and performance on Wason's Selection Task has been found by Valentine as a result of an experiment she conducted on 38 subjects [76]. However, there are no concrete results in the literature that can be generalized, and some findings are contradictory. Wason uses undergraduate psychology students

as subjects in his famous experiment Wason's Rule Discovery Task. While interpreting his findings, he does not explain that the success of 6 subjects out of 29 is based on intelligence. On the other hand, only less than 10% of the doctoral scientists who took part in Griggs and Ransdell's experiment, that is a variation of Wason's original Selection Task, could give the correct answer [77]. Jackson and Griggs [78]replicated Wason's Selection Task on mathematicians and they found that only slightly more than 50% gave the correct answer. One potential future work may be to investigate the extent of the correlation between confirmation bias test results and IQ test results.

The second goal of this paper was to investigate how well these measures perform in predicting defect prone parts of software. We found that defect prediction models that use only confirmation bias metrics as input are able to predict 60-93% of the files with defects, and these models report low false alarm rates at the same time. However, in some cases, factors other than confirmation bias might be the cause of software defects. These factors can be related to human aspects as well as factors that are directly related to the development process such as development methodologies, company culture, or frequency of software releases. Among human aspects, one can consider other cognitive bias types such as representativeness, availability, adjustment and anchoring. Moreover, widely studied concepts such as attention, memory, motivation, personality, and social cognition are likely to affect software defect density. Interaction among software professionals during the software development process is also very likely to affect software defect density [40], [42], [43], [22]. Since, developers are the ones who implement and test code, static code metrics are reflections of human aspects, as well as other factors that are directly related to the software development process. In some situations, it is possible that other human aspects may be more effective in the introduction of defects. Therefore, another area of future work would be to analyze up to what extent these human aspects effect software defect density.

This paper also serves for a high level goal that is to investigate the effect of confirmation bias on software defect density, and hence on software quality. We did a preliminary analysis to find the correlation between defect rate and confirmation bias metrics for each developer group. This gave us some clues about the influence of developers' confirmation bias on software defect rate, while it also guided us to select the appropriate metrics as input to defect prediction models. However, in order to gain more insight about the influence of developers' confirmation bias on software defect density, we need to do the following:

1. Identify a single metric to measure the confirmation bias level of developers, and
2. Analyze developers individually instead of analyzing developer groups.

We need a single confirmation bias metric to further analyze the relationship between the confirmation bias level of a developer and defect density. However, defining a set of confirmation bias metrics was a prerequisite for the formation of a single confirmation bias metric. This is the next step in our research program. As it is indicated by Cook et. al. [73], the construct validity of an empirical research requires each construct to be operationalized in a multiple manner. In order to avoid underrepresenting the effect of construct "confirmation bias" and to eliminate irrelevancies in the cause-

and-effect relationship between confirmation bias and defect rate, we used alternative measures of confirmation bias. The metrics we introduced in this paper are based on the quantitative and qualitative results of Wason's experiments about the existence of confirmation bias among people. These metrics also come from other significant experiments that have been conducted over the last sixty years in cognitive psychology literature. Further investigation of the relation between developers' confirmation bias and defect density also requires information about defects introduced by each developer so that we can perform individual-based analyses in addition to group-based analyses. Moreover, monitoring developers while they are performing unit testing on a piece of code that they have implemented and gaining insight about the types of defects they introduce, might be also be valuable.

8 Threats to Validity

Our study consists of two main parts: The first part of our study consists of the definition and extraction of confirmation bias metrics, and the second part includes an empirical analysis that consists of building defect prediction models using static code, churn and confirmation bias metrics. We address threats to validity for each part of our study in the form of two separate subsections.

8.1 Threats to Validity for Definition and Extraction of Confirmation Bias Metrics

In order to avoid mono-method bias that is one of the threats to construct validity, we used more than a single version of a confirmation bias measure. In other words, we defined a set of confirmation bias metrics. In order to form our confirmation bias metrics set, we made an extensive survey in cognitive psychology literature covering significant studies that have been conducted since the first introduction of the term "confirmation bias" by Wason in 1960 [44]. Moreover, we defined confirmation bias in relation to the software development life cycle. Since our metric definition and extraction methodology is iterative, we were able to improve the content of our metrics set through pilot study as well as datasets collected during our related previous research [59], [60], [61]. As a result, we were able to demonstrate that multiple measures of key constructs behave as we theoretically expect them to.

Another threat to construct validity is the interaction of different treatments. Before the administration of confirmation bias test to participants groups, we ensured that none of the participants were involved simultaneously in several other programs designed to have similar effects.

Evaluation apprehension is a social threat to construct validity. Many people are anxious about being evaluated. Moreover, some people are even phobic about testing and measurement situations. Participants may perform poorly due to their apprehension, and they may feel psychologically pressured. In order to avoid such problems, we informed the participants before the tests started that the questions they are about to solve do not aim to measure IQ or any related capability. Participants were also told that results would not be used in their performance evaluations and their identity

would be kept anonymous. Moreover, participants were told that there was no time constraint for completing the questions.

Another social threat to construct validity is the *expectancies of the researcher*. There are many ways a researcher may bias the results of a study. Hence, the outcomes of both the written question set and the interactive question were independently evaluated by two researchers, one of whom was not actively involved in the study. The said researcher was given a tutorial about how to evaluate the confirmation bias metrics from the outcomes of the written question set and the interactive question. However, in order not to induce a bias, she was not told about what the desired answers to the questions were. The inter-rater reliability was found to be high for the evaluation of each confirmation bias metric. The average value for Cohen's kappa was 0.92. During the administration of the confirmation bias test, explanations given to the participants before they started solving the questions did not include any clue about the ideal responses. Moreover, while the participants were solving the interactive question, an independent researcher attended the session in order to observe whether the researcher in charge influenced the response of the participants as a result of his/her gestures or facial expressions. The dialogues, that took place while the interactive question is being solved were also recorded. These recordings were later examined to find out whether the researcher in charge gave any clues to the participants about the expected result. The parts of the datasets, that were found to be influenced by the expectancies of the researcher were excluded from the empirical investigation.

In order to avoid internal threats to validity, we set the test dates for all project groups for a time when the workload of the developers was not intense. No event took place in between the confirmation bias tests that could influence the performance of the subjects in any of the groups. The members of the developer group corresponding to the first dataset within a week the confirmation bias test, consisting of the written question set and the interactive question. The remaining developer groups took the confirmation bias test in one day. As a result, we managed to create similar conditions for each member within a project group when administering the confirmation bias test. Our methodology would not have been reliable if we had tested one group member when his/her workload and time pressure were intense while testing another member of the same group under much more favorable and relaxed conditions. Another attempt to avoid internal validity was to administer the confirmation bias test in environments that were isolated from distraction factors such as noise.

To avoid external validity, we collected data from two different companies specialized in two different software development domains. We also selected different projects within a single company. In the short run, our goal is to expand our dataset to contain data from companies that are located in different countries, specialized in different domains and practicing various development methodologies. We are planning to use an in-house built automated web-based tool.

8.2 Threats to Validity for the Defect Prediction Study

We consider three major threats to the validity of our experiments: construct, internal and external. To avoid the construct validity threats in terms of measurement artifacts, we used three popular performance measures in software defect prediction research: the probability of detection (pd), the probability of false alarm rates (pf) and balance values (bal). In order to avoid internal validity threats, we shuffled data 10 times and used 10-fold cross validation for each ordering configuration of the input data to overcome ordering effects. Moreover, during undersampling, we shuffled each portion of the dataset, which is used as an input to the Naïve Bayes algorithm, 10 times. As a result, the Naïve Bayes algorithm with undersampling was executed 100 times for each dataset during each experiment.

In order to externally validate our results, we used datasets from 5 developer groups. 4 datasets come from a Telecommunication company, and 1 dataset comes from an ISV specialized in Enterprise Resource Planning (ERP) domain. Hence, our datasets cover two different software development domains. Moreover, we were able to collect datasets from two different project groups within the Telecommunications company. One project group develops software that is responsible for launching GSM tariff campaigns to its customers. Dataset Telecom1 has been extracted from this software, and it mainly consists of user interfaces. The remaining projects mainly consist of database transactions, and there is no direct interaction with the customer via user interfaces.

For statistical validity, we used the Kruskal-Wallis test to interpret our experimental results. The Kruskal-Wallis test is an alternative to the single factor ANOVA test that uses data from independent measures design. However, ANOVA assumes that data is normally distributed. On the other hand, the Kruskal-Wallis test solely requires that data be rank ordered. Since our data was not normally distributed, it was more appropriate to use the Kruskal-Wallis test.

9 Conclusion and Future Work

The overall aim of our research program is to explore the impact of cognitive biases in the development and testing of software, and it addresses two critical areas: (1) the prediction of the defective parts of the software, and (2) determining the right person to test the defective parts of the software. Software defect prediction models have tackled the problem of which parts of the software are likely to be defective to help managers effectively allocate resources during the testing phase of the product. However, these models only take into consideration the product (e.g., lines of code, code complexity, etc.) and process- (e.g., change history of code) related attributes of SDLC. Currently, there are very few attempts to understand the human aspects of defect detection, and and few ultimately recommend the best person to be assigned a specific task. Every phase of the SDLC requires analytical problem solving skills. Moreover, using everyday life heuristics instead of laws of logic and mathematics may influence the quality of the software product in an undesirable manner. This research aims to understand how the human mind works in solving problems.Therefore,

in this paper, we have investigated the effect of the thought processes of people on software quality in terms of defect density. Since the thought processes of people cover a wide range of aspects, we have focused on confirmation bias that is believed to be one of the factors that lead to increased software defect density. In this paper, we defined a metric scheme to quantify confirmation bias within the context of software development and testing. In order to validate how well our proposed metric scheme identifies the effect of developers' confirmation bias on software quality, we conducted experiments by constructing defect prediction models. We used confirmation bias metrics as input to defect prediction models that we extensively investigated in our past research [7], [26], [70], [71], [72].

In our empirical study, we used five datasets obtained from two industrial partners, which are from the telecommunications and ERP domains respectively. We compared predictors built using confirmation bias metrics with predictors built using static code and churn metrics. In the overall, the improvement in defect prediction performance as a result of using confirmation bias metrics was not significant. However, we can explain the importance of the results we obtained as follows: Static code metrics included all major metrics from the source code based on program flow and the readability of the code [69] [68]. The churn metrics set contains extensive information about the changes in source code during the implementation phase. We extracted a significant portion of information regarding code change history from version control systems. On the other hand, confirmation bias metrics represent only a single aspect about the thought processes of people. Despite this, our empirical findings showed that using only confirmation bias metrics to learn defect predictors reveals comparable performance results. In cognitive psychology, the causes of biases have been extensively investigated in various domains over the past three decades. There is extensive amount of findings in the field of cognitive psychology that can be employed to form a metric suite covering developers' cognitive aspects that may have a significant effect on software defect density. To summarize, we believe that the thought processes of people deserve further attention.

The objective of this research in the long run is to help software development managers make specific resource allocation decisions by considering metrics related to the thought processes of people. Such a metric scheme will help managers to determine the right person to test the defective parts of the software. As a result, the guidance of metrics related to the thought processes of people may decrease the uncertainty in Human Resource (HR) related decisions to a significant extent.

As future work, we aim to collect data from larger software development groups within different companies located in different countries. The collection of data from different contexts would be possible once we complete the implementation of our web-based tool. The said tool will be available for both other researchers and practitioners. Practitioners may also use the tool to assess employee performance, design training programs based on their assessment score, and hire new employees.

Acknowledgements We would like to thank Turkcell A. Ş. and; Turgay Ayta ç and Ayhan Inal from Logo Business Solutions for their support in sharing data.

References

1. Harrold, M., Testing: a roadmap. Proceedings of the Conference on the Future of Software Engineering, 61-72, (2000)
2. Tahat, L. H., Vaysburg, B., Korel, B. and Bader, A., Requirement based automated black-box test generation. Proceedings of the 25th Annual Int. Computer Software and Applications Conference, 489-495, (2001)
3. Bullard, L. A, and Gao, K., An application of a rule-based model in software quality classification, Proceedings of the 6^{th} International Conference on Machine Learning and Applications, pp.204-210, 2007.
4. Nagappan, N., Toward a software testing and reliability early warning metric suite. Proceedings of 26th Int. Conference on Software Engineering Conference, (2004)
5. Khoshgoftaar, T. M., Van Hulse, J. and Napolitano, A., Supervised neural network modeling: an empirical investigation into learning from imbalanced data with labeling errors. IEEE Transactions on Neural Networks, 21(5), 813-830, (2010)
6. Khoshgoftaar, T. M., Building decision tree software quality classification models using genetic programming.Proceedings of the Genetic and Evolutionary Computation Conference, (2003)
7. Tosun, A., Turhan, B. and Bener, A., Ensemble of software defect predictors: a case study. Proceedings of 2^{nd} International Symposium on Empirical Software Engineering and Measurement, (2008)
8. Boehm, B. and Basili, V. R., Software defect reduction top 10 list. IEEE Software, pp.135-137, (2001)
9. Munson, J. C. and Khoshgoftaar, T. M., Detection of fault prone programs, IEEE Transactions on Software Engineering, (18)5, pp.423-433, (1992)
10. Khoshgoftaar, T. M. and Allen, E. B., Predicting fault-prone software modules in embedded systems with classification trees, Proceedings of the 4^{th} IEEE International Symposium on High-Assurance Systems Engineering,(1999)
11. Nagappan, N., Toward a software testing and reliability early warning metric suite, Proceedings of the 26^{th} International Conference on Software Engineering, pp. 60-62, (2004)
12. Bell, R. M., Ostrand, T. J. and Weyuker, E. J., Looking for bugs in all the right places, Proceedings of the 2006 International Symposium on Software Testing and Analysis, pp.61-71, (2006)
13. Ostrand, T. J. and Weyuker, and Bell, R. M., Where the bugs are, Proceedings of the 2004 ACM SIGSOFT International Symposium on Software Testing and Analysis,pp. 86-96, (2004)
14. Ostrand, T. J. and Weyuker, and Bell, R. M., Automating algorithms for the identification of fault prone files, Proceedings of the 2007 International Symposium on Software Testing and Analysis, pp. 219-227, (2007)
15. Menzies, T. Z., Hihn, C. J. and Lum, K., Data mining static code attributes to learn defect predictors. IEEE Transactions on Software Engineering, 33(1): 2-13 (2007)
16. Menzies, T., Turhan, B., Bener, A., Gay, G., Cukic, B., and Jiang, Y., Implications of ceiling effects in defect predictors, Proceedings of the 3^{rd} Workshop on Predictive Models in Software Engineering, pp.47-54,(2008)
17. Lessmann, S., Baesens, B., Mues, C., and Pietsch, S., Benchmarking classification models for software defect prediction: a proposed framework and novel findings. IEEE Transactions on Software Engineering,34(4): 485-496, (2008)
18. Drummond, C. and Holte, R. C., C4.5, Class imbalance and cost sensitivity: why under-sampling beats over-sampling, Proceedings of 2^{nd} Workshop on Learning from Imbalanced Datasets, (2003)
19. Kamei, Y., Monden, A., Matsumoto, T. and Matsumoto, K., The effects of over and under-sampling on fault prone module detection, Proceedings of the 1^{st} International Symposium on Empirical Software Engineering and Measurement, pp.196-204, (2007)
20. Jiang, Y., Cuki, B., Menzies, T. and Bartlow, N., Comparing design and code metrics for software quality prediction, Proceedings of the 4^{th} International Workshop on Predictor Models in Software Engineering, (2008)
21. Zhao, M., Wohlin, C., Ohlsson, N. and Xie, M., A comparison between software design and code metrics for the prediction of software fault content, Information and Software Technology,(40)14, pp.801-809, (1998)
22. Zimmerman, T. and Nagappan, N., Predicting subsystem failures using dependency graph complexities, Proceedings of the 18^{th} IEEE International Symposium on Software Reliability, pp. 227-236, (2007)
23. Nagappan, N. and Ball, T., Using software dependencies and churn metrics to predict field failures: an empirical case study, Proceedings of the 1^{st} International Symposium on Empirical Software Engineering and Measurement, pp.364-373, (2007)

24. Misirli-Tosun, A., Caglayan, B., Mirasky, A., Bener, A., and Ruffolo, N., Different strokes for different folks: a case study on software metrics for different defect categories, Proceedings of the 2nd Workshop on Emerging Trends in Software Metrics, pp. 45-51, (2011)
25. Nagappan, N. and Ball, T., Using software dependencies and churn metrics to predict fied failures, Proceedings of the 1st Symposium on Empirical Software Engineering and Measurement, pp.364-373, (2007)
26. Tosun, A., Turhan, B. and Bener, A., Practical considerations in deploying AI for defect prediction: a case study within the Turkish Telecommunication industry. Proceedings of 5th International Conference on Predictor Models in Software Engineering, (2009)
27. Turhan, B., Kocak, G. and Bener, A., Software defect prediction using call graph based ranking (CGBR) framework., Proceedings of. 34th International EUROMICRO Software Engineering and Advanced Applications Conference, (2008)
28. Kahneman D., Slovic P., and Tversky, A., Judgment Under Uncertainty: Heuristics and Biases. Cambridge University Press, New York, (1982)
29. Stacy, W. and MacMillan, J., Cognitive bias in software engineering, Communication of the ACM, (38)6, (1995)
30. Teasley, B., Leventhal, L. M., and Rohlman, S., Positive test bias in software engineering professionals: What is right and what's wrong. Proceedings of the 5th Workshop on Empirical Studies of Programmers,(1993)
31. Parsons, J., and Saunders, C., Cognitive heuristics in software engineering: applying and extending anchoring and adjustment to artifact reuse, IEEE Transactions on Software Engineering, 30(12), pp. 873-888, (2004)
32. Mair, C. and M. Shepperd, Human judgment and software metrics: vision for the future, Proceedings of the 2nd International Workshop on Emerging Trends in Software Metrics, (2011)
33. Jørgensen, M., Identification of more risks can lead to increased over-optimism of and over-confidence in software development effort estimates, Journal of Information and Software Technology, (52)5, pp.506-516, (2010)
34. Jørgensen, M., Estimation on software development work effort: evidence on expert judgment and formal models, International Journal of Forecasting, 23(3), pp. 449-462, (2007).
35. Jørgensen, M., The effects of request formats on judgment-based effort estimation, Journal of Systems and Software, (83)1, pp. 29-36, (2010).
36. Graves, T. L., Karr, A. F.,Marron, J. S., and Siy, Harvey, Predicting fault incidence using software change history, IEEE Transactions on Software Engineering, (26)7, pp.653-661, (2000)
37. Mockus, A. and Weiss, D. M., Predicting risk of software changes, Bell Labs Technical Journal, pp.169-180, (2000)
38. Weyuker, E.J., Ostrand, T. J. and Bell,R. M., Using developer information as a factor for fault prediction, Proceedings of the 1st International Workshop on Predictor Models in Software Engineering, pp.1-7, (2007)
39. Ostrand, T., J., Weyuker, E. J. and Bell, R. M., Programmer-based fault prediction, Proceedings of the 3rd Workshop on Predictor Models in Software Engineering, pp.1-7, (2010)
40. Meneely, A., Williams, L, Snipes, W., and Osborne, J., Predicting failures with developer networks and social network analysis, Proceedings of the 16th ACM SIGSOFT International Symposium on Foundations of Software Engineering, pp.13-23, (2008)
41. Weyuker, E. J., Ostrand, T. J. and Bell, R. M., Do too many cooks spoil the broth? Using the number of developers to enhance defect prediction models, Journal of Empirical Software Engineering, 13, pp. 539-559, (2008)
42. Pinzger, M., Nagappan, N. and Murphy,B., Can developer-module networks predict failures?, Proceedings of the 16th ACM SIGSOFT International Symposium on Foundations of Software Engineering, pp. 13-23, (2008)
43. Bird, C., Nagappan, N., Gall, H., Murphy, B. and Devanbu, P., Putting it all together: Using socio-technical networks to predict failures, Proceedings of the 17th International Symposium on Software Reliability Engineering, (2009)
44. Wason, P. C., 1960. On the failure to eliminate hypotheses in a conceptual task, Quarterly Journal of Experimental Psychology, 12, pp.129-140, (1960)
45. Wason, P. C. 1968. Reasoning about a rule, Quarterly Journal of Experimental Psychology, 20, pp: 273-28, (1968)
46. Evans, J. St. B. T., Newstead, S. E. and Byrne, R. M., Human reasoning: the psychology of deduction. Lawrence Erlbaum Associates Ltd., East Sussex, U.K. (1993)

47. Cox, J. R. and Griggs, R. A., The effects of experience on performance in Wasons selection task, Memory and Cognition, 10, pp.496-502 (1982)
48. Griggs, R. A. and Cox, J. R., The elusive thematic materials effect in Wasons selection task, British Journal of Psychology, 73, pp.407-420 (1982)
49. Cheng, P. W. and Holyoak, K. J., Pragmatic reasoning schemas, Cognitive Psychology, 17, pp.391-416, (1985)
50. Cosmides, L., The logic of social exchange: Has natural selection shaped how humans reason? Studies with Wason's selection task, Cognition, 31, pp.187-276, (1989)
51. Manktelow, K. I. and Over, D. E., Inference and understanding: A philosophical and psychological perspective, (1990)
52. P. C. Wason and Shapiro, D., Natural and Contrived Experience in a Reasoning Problem, Quarterly Journal of Experimental Psychology, 23, pp.63-71, (1971)
53. Manktelow, K. I. and Evans, J. St. B. T., Facilitation of reasoning by realism: Effect or non-effect?, British Journal of Psychology, 70, pp.477-488, (1979)
54. Johnson-Laird, P.N. and Tridgell, J. M., When negation is easier than affirmation, Quarterly Journal of Experimental Psychology, 24, pp.87-91, (1972)
55. Griggs, R.A., The role of problem content in the selection task and in the THOG problem, Thinking and reasoning: psychological approaches. Routledge and Kegan Paul London, (1983).
56. Khoshgoftaar, T. M. and Szabo, R. M., Using neural networks to predict software faults during testing, IEEE Transactions on Reliability, 45, pp.456462, (1996)
57. Reich, S.and Ruth, P., Wason's selection task: verification, falsification and matching, British Journal of Psychology, 73, pp.395-405, (1982)
58. Kocaguneli, E., Tosun, A., Bener, A., Turhan, B., and Caglayan, B., Prest: an intelligent software metrics extraction, analysis and defect prediction tool, Proceedings of 21st International Conference on Software Engineering and Knowledge Engineering, pp.637-642, (2009)
59. Calikli, G., Bener, A., and Arslan, B., An analysis of the effects of company culture, education and experience on confirmation bias levels of software developers and testers. Proceedings of 32nd International Conference on Software Engineering, (2010)
60. Calikli, G., Arslan, B., and Bener, A., Confirmation bias in software development and testing: an analysis of the effects of company size, experience and reasoning skills. Proceedings of the 22nd Annual Psychology of Programming Interest Group Workshop, (2010)
61. Calikli, G. and Bener, A., Empirical analyses factors affecting confirmation bias and the effects of confirmation bias on software developer/tester performance. Proceedings of 5th International Workshop on Predictor Models in Software Engineering, (2010)
62. Hall, M. A. and Holmes, G. , Benchmarking attribute selection for discrete class data mining. IEEE Transactions on Knowledge and Data Engineering, 15, pp.1437-1447, (2003)
63. Nagappan, N., Murphy, B. and Basili, V. R., The influence of organizational structure on software quaity: an empirical case study, Proceedings of the 30th International Conference on Software Engineering, pp.521-530,(2008)
64. Teasley, B. F., Leventhal, L. M., Mynatt, C. R. and Rohlman D. S., Why software testing is sometimes ineffective: two applied studies of positive test strategy. Journal of Applied Psychology, 79, 1, pp.142-155, (1994)
65. Johnson-Laird, P.N. and Wason, P. C., A theoretical analysis of insight into a reasoning task, Cognitive Psychology, 1, pp.134-148, (1970)
66. Mataraso-Roth, E., Facilitating insight in a reasoning task, British Journal of Psychology, 70, pp.265-271, (1979)
67. Evans,J.,St.,B.,T. and Lynch, J.,S., Matching bias in the selection task, British Journal of Psychology, 64, pp.391-397, (1973)
68. McCabe, T., A complexity measure, IEEE Transactions on Software Engineering, 2, pp.308-320, (1976)
69. Halstead, M., Elements of software science, Elsevier, (1977)
70. Turhan, B. and Bener, A., A multivariate analysis of static code attributes for defect prediction, Proceedings of the 7th International Conference on Quality Software, pp. 231-237, (2007)
71. Turhan, B. and Bener, A., Weighted static code attributes for software defect prediction, Proceedings of the 20th International Conference on Software Engineering and Knowledge Engineering, pp.143-148, (2008)
72. Turhan, B., Bener, A. and Menzies, T., Nearest neighbor sampling for cross company defect predictors, Proceedings of the 1st International Workshop on Defects in Large Software Systems, (2008)

73. Cook, T.D. and Campbell, D.T. Quasi-experimentation: design and analysis issues for field settings. Houghton Mifflin, Boston, (1979)
74. Cohen, J., A power primer, Psychology Bulletin, 112:1, pp. 155-159, (1992)
75. Cohen, J. Statistical power analysis for the behavioral sciences. Lawrence Erlbaum Associates Publishers, Hillsdale, New Jersey, (1988)
76. Valentine, E. R., Performance on two reasoning tasks in relation to intelligence, divergence and interference proneness. British Journal of Educational Psychology, 45, pp. 198-205, (1975)
77. Griggs, R. A. and Ransdell, S. E., Scientists and the selection task. Social Studies of Science, 16, pp.319-330, (1986)
78. Jackson, S. L. and Griggs, R. A., Education and the selection task. Bulletin of Psychometric Society, 26, pp. 327-330.
79. Poletiek, F. Hypothesis-Testing Behaviour. Psychology Press, East Sussex, UK (2001)
80. Ye, MaoLiang, Does gradualism build coordination? Evidence from laboratory experiments. Job Market Paper, (2011)