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

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

Gül Çalıklı · Ayşe Başar Bener

Received: date / Accepted: date

Abstract People's thought processes 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*. Confirmation bias is believed to be one of the factors that lead to increased software defect density. Due to confirmation bias, developers might perform unit tests to make their program work. This results in the propagation of more defects to testing phase and hence probably an increase in 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 perform an empirical study which addresses prediction of defective parts of software. In our empirical study, we applied confirmation bias metrics to five datasets obtained from two industrial partners which are from Telecomunications and Enterprize Resource Planning (ERP) domains respectively. Our results provide empirical evidence that people's thought processes and cognitive aspects deserve further investigation to find out empirical evidence about their effectiveness in software defect prediction as well as their relation to software quality.

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 in avoidable rework 80% of which is due to 20% of the defects. Therefore, software testing is crucial for the detection of defects before the software product is released to the market. However, software testing is the most resource consuming phase of software development life-cycle, since approximately 50% of a project schedule is allocated to testing phase [1], [2].

Defect predictors provide guidance to project managers for effective allocation of resources in testing phase by pointing out defect-prone parts of the software. As a result, it is possible to increase efficiency of software testing phase as well as delivering the software product to market on time. 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].

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 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] to learn 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 Naïve Bayes classifier with a log-filtering preprocessor on the numeric data, outperforms methods such as OneR and J4.8. 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 performance of Naïve Bayes and more complex machine learning algorithms.

In order to find out whether 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. As algorithm, they used Naïve Bayes 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, throwing away data (i.e. undersampling) does not degrade the performance of the learner. For J4.8 algorithm throwing away data improved 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].

In the literature there are also instances where metrics other than or in addition to static code attributes have been used for defect prediction. Jiang et. al. [20] compared

predictor performances that were learnt from design metrics, static code features and both for 13 NASA projects. Design metrics were extracted from 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 results of the similar experiments which were previously conducted by Zhao. et. al. [21] for the analysis of a real time Telecomunication system. Zimmerman and Nagappan [22] developed a metric suite which defines dependency of binary files from a graph theoretic point of view. The authors used these metrics as input to linear and logistic regression models to predict post-release failures of Windows Server 2003. Zimmerman and Nagappan report 10% increase in 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 binary files of Windows Server 2003. The authors conclude that they can 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 to predict all types of defects. Turhan et. al. in another study [27] reduced probability of false alarms by supplementing static code metrics by their Call Graph Based Ranking (CGBR) framework.

The above mentioned research to improve input data content mainly focus on product attributes and process attributes. However, people's thought processes have a significant impact on software defect density, as software is being developed by people, and tested by people. In this paper, we address a specific aspect of people's thought processes, namely *confirmation bias*, to identify defect prone parts of software. In this study, we focus on confirmation bias of developers. 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. Hence, we can state our research question as follows:

RQ1: How do developers' confirmation biases influence software quality?

In order to propose a solution to research question *RQ1*, firstly one needs to answer the following:

RQ2: How can we identify measures of confirmation bias in relation to software development process?

RQ3: How well measures of confirmation bas do in predicting defect prone parts of software?

As a solution to research question *RQ*2, we present a methodology to define and extract confirmation bias metrics in relation to software development process. We also investigate effectiveness of these metrics in the prediction of software failure-proneness, in order to answer our final research question *RQ*3. For this purpose, we conduct a benchmark analysis. In this benchmark analysis, we use five datasets collected from two of our industrial partners in Telecomunications and Enterprize Resource Planning (ERP) domains respectively. For each dataset, we compare 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 comparable with results of defect predictors that are learnt from *only* churn metrics and *only* static code attributes respectively.

We can summarize the contributions of this work as follows:

1. 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 Telecomunication/GSM company.
4. A benchmark study to evaluate 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: We mention related work which is followed by detailed information about confirmation bias in cognitive psychology literature. In this section, we also make a definition of confirmation bias in relation to software development process as well as explaining the analogy between Wason's experiments and unit testing. We later explain our methodology to define and extract confirmation bias metrics. This is followed by details about the confirmation bias metrics. In the next section, having explained our experimental methodology, we present our experimental results and address threats to validity. We finally address threats to validity regarding definition of confirmation bias metrics and conclude 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, over-optimism, over-confidence, anchoring and adjustment; and confirmation bias. Although there has been intensive research in the field of cognitive psychology about cognitive biases, 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 which takes into account metrics related to a cognitive bias type to learn defect predictors.

2.1 Effects of Cognitive Biases on Software Engineering

In [29], Stacy and MacMillian emphasize the fact that thought process 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 forward some ideas as possible explanations and as potential areas that require further research.

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.

Another study which provides empirical evidence about the existence of another cognitive bias type anchoring and adjustment within the context of software development belongs to Parsons and Saunders [31]. Parson and Saunders conduct two experiments, which investigate the existence of anchoring and adjustment in software artifact reuse. The first experiment they conduct examines the reuse of object classes in a programming task, whereas their second experiment investigates how anchoring and adjustment bias affects reuse of software design artifacts. In their second experiment, Saunders and Parsons ask the participants to develop an Entity-Relationship (E-R) model for an airplane application.

Mair and Shepperd [32] discuss how software engineers' cognitive biases 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 should 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 it contains no empirical investigation.

On the other hand, Jørgensen et. al empirically investigates some cognitive bias types within the scope of software development effort estimation. According to empirical findings of Jørgensen [35], 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 over-emphasized so that inaccurate effort estimations are made. Jørgensen also empirically investigates how anchoring and adjustment heuristic leads 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 effort of the closest analogy.

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 people's thought processes 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 au-

thors built a model to predict failure proneness of Windows Vista. They compared the performance of this defect predictor with the performance of models that are learnt using code churn, code complexity, code coverage, pre-release bugs and dependencies respectively. In terms of precision and recall values, their model outperformed all these mentioned 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 prediction of defective modules. According to the results obtained by the authors, the number of developers who have changed a module did not improve defect prediction performance. Weyuker et. al. [41] also found that number of developers is not a major influence to increase 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 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 defect prediction performance. Following this research, the authors later analyzed the effectiveness of individual developer performance on defect prediction performance and no evidence of a significant improvement in defect prediction performance was found either [39].

Social interaction between developers who have collaborated to the same file during same period of time was modeled as social networks to be 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 making contribution to a specific module. Empirical analysis of the data from Windows Vista project showed that centrality metrics can predict software failures up to a significant extend. 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 interactive question and written question set and 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 written question set is based on Wason's Selection Task [45]. In the following subsection, we briefly explain these two experiments of Wason 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, 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 a way such 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, 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 does 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 more sure about the rule to be discovered. The hypotheses in mind should be eliminated, modified and created in a strategic manner, so that subject can come up with a single hypothesis at the end. Once the subject is sure about what the correct rule is, (s)he may also give additional triple instances to verify his/her guess.

Wason's Selection Task: Written question set 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 its other side. These four cards are placed on a table showing respectively D, K, 4, 7. 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 might perform only the tests which make their program work and this leads 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 ones that are less likely to fail the code and proceeding with test scenarios which 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 which 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 in his Rule Discovery Task, the subjects have tendency to select many triples (i.e. test cases) which are consistent with their hypotheses and few tests which are inconsistent with them. Similarly, program testers may select many test cases consistent with the program specifications (positive tests) and few which are inconsistent with them (negative tests). Moreover, the state space of possible test cases is either infinite or too large to be tested within limited amount of time. Hence, a strategic approach must be followed which covers both positive and negative test cases while trying to make the code fail during testing in order to find as much 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. During unit testing while covering possible scenarios logical reasoning is required. Moreover, testing correctness of conditional statements in source code during white box testing also requires logical reasoning skills.

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 information flow. Preparation of interactive question and written question set was performed concurrently with the creation of the initial metric suite. In order to prepare confirmation bias questions, we made an extensive survey in cognitive psychology literature. This survey also helped us to initiate confirmation bias metric suite. There is a mutual information feedback between preparation of questions and metric suite update processes since definition of a new metric sometimes required adding new questions into the written question set which in turn often led to the introduction of more metrics. Having prepared confirmation bias questions and a metric suite, interactive question and written question set are answered by the participants software professionals. During evaluation and analysis of the answers of confirmation bias question given by the participants, new metrics can be introduced, into the metric suite. Statistical analysis and feature selection techniques help to eliminate metrics which have less significance in

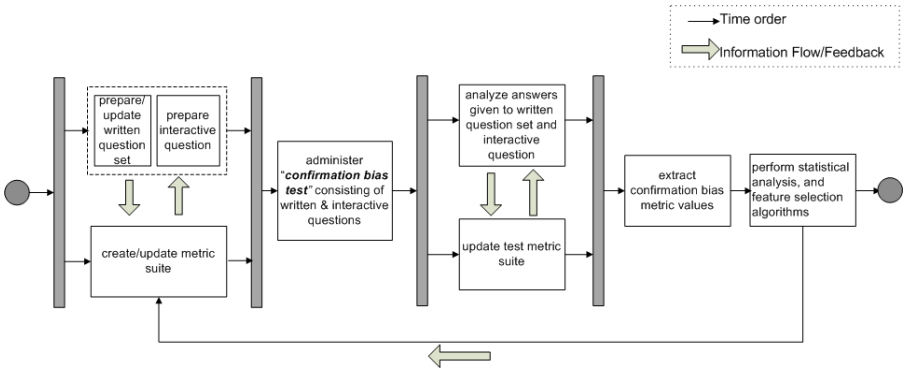

Fig. 1 : Methodology to define confirmation bias metrics and extract confirmation bias metric values.

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 metric suite which has been modified at the end of the previous iteration. The extend of the changes regarding content of the confirmation bias questions and metric suite were much larger during the early stages of the metric definition process when confirmation questions consisting of interactive question and 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 content are likely to occur.

4.1 Preparing Interactive Question and Written Question Set

Interactive question is Wason's Rule Discovery Task itself, the details of which are explained in the previous section. Written question set 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 with software development and testing theme. Table 1 give 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 which have thematic content, since real life experience may help to answer such questions correctly [46]. For written question set, we prepared 7 thematic questions after having made a literature survey covering major thematic variants of Wason's original selection task [47], [48], [49], [50],[51], [52], [53], [54],[55]. In written question set, there is 1 abstract-thematic question. Similar to an abstract question, an abstract-thematic question can be answered correctly by pure logical reasoning. Although such questions seem to have a thematic content, thematic facilitation effect does not take place [46].

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

4.2 Administration of Confirmation Bias Test

In order to collect confirmation bias metrics in a controlled manner, we administered *confirmation bias test*, which consist of interactive question and written question set, under a predefined standard procedure. The environment where *confirmation bias test* was administered was isolated from noise and had adequate lighting. Both Turkish and English versions of the interactive question and 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. 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 confirmation bias test results shall not be used in their company's performance evaluations and their identity shall be kept anonymous. The goal was not to exert pressure on participants which could affect participants' performance. Moreover, participants were told that there is no time constraint to complete the questions in order not exert time pressure. After the completion of both booklets, 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 which are specific for written question set and 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 is visible. However, most recent studies in this field rely on the description of the cards, and pictorial representations of cards' facing sides either on pencil and paper or on a computer screen. These procedural differences have made insignificant differences in the obtained results [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 pen and pencil approach rather than the traditional approach. Hence we prepared written question set which consists of two booklets. The first booklet includes abstract, thematic-abstract and thematic questions, while the second booklet consists of thematic questions with software development/testing theme. Each group of participants, corresponding to each data set listed in Table 6, answered questions in the first and second booklet altogether in a meet-

ing/seminar room. Before starting to read and answer the questions, the participants were told to fill in the form where personal information such as gender, age, education and experience in software development and/or testing were asked. This information was used in our previous research where we investigated factors affecting confirmation bias as well as effects of confirmation bias on software developer and tester performance [59], [60], [61]. Afterwards, first booklet was given to participants so that they started to answer questions in the first booklet simultaneously.

Interactive Question Each participant answered interactive question in a separate room and for each participant there was one examiner to guide and give feedback. Before the whole procedure started, participants were asked whether they gave permission to record their voices during the session. The goal of voice recording was to catch every detail about the way a participant thinks to discover the correct rule as well as eliminating ambiguities while interpreting reasons for choice participant wrote down for each instance they gave while evaluating the question result. Voice recording was made only if the participant gave us the permission. Before starting, detailed information was given about the procedure which should be followed to answer the interactive question (i.e. to discover the correct rule).

5 Confirmation Bias Metrics

The metrics in the confirmation bias metric suite are extracted from 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 that 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 interactive question is Wason's Rule Discovery Task itself, hypotheses testing behavior the developer exhibits while solving the interactive question has the potential to give us clues about the strategies employed by the developer to test his/her own code. On the other hand, 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 focusses on the logic of the program and its internal structure. Therefore knowledge about first order logics is required as it is also required to solve the questions in the written question set correctly. In Table 5 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. Since increase in the effectiveness of unit testing leads to a decrease in the number of defects overlooked by developers, the confirmation bias metrics we have defined are also related to software defect density. Therefore, they are used to build defect predictors.

5.1 Interactive Question Metrics

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 total number of rules announced by a participant throughout the interactive question session. However, the value of N_A alone does not give information about the number of rule announcements it takes the participant to find the correct rule. Moreover, this metric does not inform us about whether the participant finds the correct rule or not. As we mentioned previously, according to the interactive question protocol the participant can abort the session whenever (s)he wants. In addition to this, when total session time exceeds 45 minutes, the experimenter aborts the interactive question session. In order to discriminate these three different situations we defined the metric IND_{ABORT}. For a participant who eventually finds the correct rule, $IND_{ABORT} = 0$. $IND_{ABORT} = 1$ implies that the participant gives up the test on his/her own will, whereas $IND_{ABORT} = 2$ if and only if the session exceeds 45 minutes.

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 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 perform effective unit testing. It is very crucial for commercial software products to perform effective testing at all levels in a reasonably short time while rushing for the next release. However, metric T_I does not contain any information about whether the participant finds the correct rule or (s)he gives up the session, or the experimenter aborts the session. What is more, by referring to only T_I, one cannot make a deduction about the existence of an effective hypotheses testing strategy employed by the participant while (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 ($Ind_{elim/enum}$) was introduced by Wason to evaluate the results of his rule discovery task [44], in order to determine 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 made immediate correct rule announcement, had higher $Ind_{elim/enum}$ values compared to the rest of the participants. Our results were also in line with Wason's findings. For developers, who took part in this research, average $Ind_{elim/enum}$ value for developers, who made immediate correct announcement, is 0.75; whereas this value is 2.43 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). Comparison of average $Ind_{elim/enum}$ value of developers who made immediate correct rule announcement and those who made incorrect rule announcement(s) is given in Table 2 for each dataset that is used in this research. Details about datasets are listed in Table 6.

Table 2 Comparison of average $Ind_{elim/enum}$ and $F_{negative}$ values of developers who made immediate correct rule announcement 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

The value of eliminative/enumerative index being less 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 less 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 shall not be discovered unless test cases, which aim to fail the code, are also used. As a result, effective unit testing shall be hindered leading to an increase in the amount of defects overlooked during unit testing.

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 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 an incorrect rule announcement. Among developers, who took part in this empirical study, average $F_{negative}$ value of 2.31 belongs to developers, who made immediate correct rule announcement. On the other hand, for developers who made incorrect rule announcement, this value is equal to 0.81. According to Kruskal-Wallis test, statistical significance of this difference is $\chi^2 = 10.59, p = 0.0011$. The results obtained within each dataset are inline 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 which are compatible with the correct rule to be discovered. Similarly, during software testing some test cases may help to identify missing parts of the software specifications. Specifications play a crucial role in software testing. The more complete the specifications are, the more likely that the quality of the testing will be high. For this reason, in addition to $Ind_{elim/enum}$, we also include $F_{negative}$ to our confirmation bias metric suite.

While 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 a part of the protocol which was explained to each participant before that participant 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]. Table 3 shows the portion of participants who made immediate rule announcements among the developer groups which took part in this research. Immediate rule announcements are indication of participant's inadequate hypotheses testing strategies, as a result of which participant cannot come up with a single rule at the end by eliminating alternative hypotheses in his/her mind. Equivalence partitioning is a non-exhaustive functional testing technique which is applied to each functional unit mostly together with boundary testing. In equivalence partitioning, for each functional unit, a set of dimensions of input data are identified as stated in the specifications and for each dimension a set of equivalence classes are identified. A developer who makes immediate rule announcements while 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 extend 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. 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 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.

As the most interesting qualitative results of his rule discovery experiment, Wason indicates that some participants reformulated or just repeated their rules even after they were informed about the incorrectness of their rule [44] they had announced. We also observed such behavior among our participants as shown in Table 3. Therefore, we thought that rule repetitions which were qualitatively discussed by Wason in [44], should be quantified. For this purpose, we defined the metric $avgF_{RR}$ which measures average frequency of rule repetition. We can explain implication of this metric as follows: A developer, who makes rule repetitions while solving the interactive question, is likely to perform functional unit tests using input values such that not all equivalence classes are covered. Such behavior is usually in the form of a tendency to make the code run rather than finding defects. In addition to the rule repetitions, we observed reason repetitions as it can be seen in Table3. In order to quantify reason repetitions, we introduced the metric $avgF_{RsR}$ which measures average frequency of reason repetition. Unlike rule repetitions, reason repetitions may be due to the lack of strategic hypotheses testing and they do not necessarily imply a tendency to make the

Table 3 Distribution of developers who made immediate rule announcements, rule and reason repetitions while solving interactive question

Developer Group #	Dataset #	Immediate Rule Announcement %	Reason Repetition %	Rule Repetition %
1	ERP	50.00 %	50.00 %	16.67 %
2	Telecom1	30.00 %	50.00 %	20.00 %
3	Telecom2	6.67 %	33.33 %	20.00 %
4	Telecom3	6.25 %	37.50 %	18.75 %
5	Telecom4	5.88 %	41.18 %	17.64 %

code run during unit testing. Despite this, reason repetitions cause ineffective testing during which defects are overlooked.

Another metric is the number of instances given per unit time ($Instances/Time$). While 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 during which there was 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 value 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, 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 while 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. 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 total number of unique reasons stated by a participant for the instances (s)he gives while solving the interactive question.

Unlike the metric $Instances/Time$, increase in the value of the metric which measures 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 outcome of the interactive question has the tendency to deliver his/her code to the testing phase without making adequate unit testing. For such developer, compilation of his/her code shall be enough. In other words, high $Rules/Time$ is a result of 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. 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 unique rules announced per unit time into our confirmation bias metric suite.

While solving the interactive question, the instance generated by the participant after the announcement of an incorrect rule can either be compatible or incompatible with the incorrect rule announced and it can be either a negative or a positive positive instance of the correct rule. $n_{Pos+Comp}$ measures total number of positive and compatible instances announced by a developer. If the value of $n_{Pos+Comp}$ for a developer is greater than zero, then developer has a tendency to avoid test cases which can help to find defects. $n_{Neg+Comp}$ measures total number of negative and compatible instances. Existence of negative and compatible instances may help to find the correct rule. Moreover, a developer with a positive $n_{Neg+Comp}$ metric value has a tendency to select input data from different equivalence classes during functional testing. In other words such a developer does not stick to a single equivalence class. Positive $n_{Pos+Incomp}$ and/or positive $n_{Neg+Incomp}$ value imply the existence of incompatible instances. Incompatible instances given just after the announcement of an incorrect rule shows the participant's tendency to discover the correct rule rather than sticking to the incorrectly announced ones.

5.2 Written Question Set Metrics

In order to quantify the extend of participant's logical reasoning skills within the context of hypotheses testing, we introduced metrics extracted from outcomes of the written question set into our confirmation bias metric suite as shown in Table 5. 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 affects such as daily life experience or memory queueing. 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 lack of logical reasoning skills can be compensated by knowledge in software development and testing.

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.

Among our participants, we observed that majority selected the cards whose visible faces have symbols or words matching the ones in the rule. Information processing model proposed by Johnson-Laird and Wason [65], classifies 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 the results obtained by both Matarasso Roth [66] and Evans and Lynch [67] partic-

Table 4 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%

ipants performing at the level of "no insight" focus on cards mentioned in the rule whose validity is to be tested. Selection of cards by a participant with "no insight" might be due to the participants' 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 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 participants having "partial insight" select all cards that could either verify or falsify the rule, whereas 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 4 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, performance of developers on thematic questions is higher as shown in Table 4.

We introduced three metrics in order to determine participant's 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 which 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 which are answered by selecting cards having the symbols that match the ones in the rule as well as the number of those which are answered a systematic combinatorial analysis of the cards. Similar metrics are defined to identify participants' performance on thematic questions which 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 $ThABS_{Insight}$ which can take one of the three categorical values "Complete Insight", "Partial Insight" and "No Insight" respectively.

Although insight metrics we have defined give information about the existence of a systematic analysis of the cards, distinction between verification, falsification and matching tendencies are not clear enough. Reich and Ruth [57] proposes an alterna-

tive 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 those mentioned in the rule are same. However, the rule whose validity is 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. Response given to the TypeI and TypeII questions, help to identify tendencies for falsification and verification respectively. Responses to TypeIII and TypeIV questions are also indications of the existence of falsification and verification tendencies respectively. However, responses given these two questions may also give us clues about the existence of a matching tendency. The metrics $FVMN_{TypeI}$, $FVMN_{TypeII}$, $FVMN_{TypeIII}$ and $FVMN_TypeIV$, which we defined as a part of our confirmation bias metric set, take one or two of the categorical values such as ”Falsifier”, ”Verifier”, ”Matcher” as well as the categorical value ”None”. ”None” is used to label responses that cannot be categorized under the former three response types.

6 Empirical Study

6.1 Datasets

In this study, we used datasets from five different projects as shown in Table 6. In order to learn defect predictors, we took into account only source code files where development activities can be observed from the outcomes of the version management system. Only these active source code files were tested by testing teams, hence project teams needed guidance about defect prone parts of these files in order to allocate their testing resources efficiently, while rushing for the next release. In Table 6 total number of maintained/developed files, file types and defect rate are listed for each dataset. 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 enterprize resource planning (ERP) software. The snapshot of the software that was retrieved from the version management system belongs to period of March 2011 and it consists of 3199 java files. The remaining four datasets belong to the Telecomunications company which is Turkey’s largest GSM operator. Dataset Telecom1 consists four versions of a software product responsible for launching new campaigns. There are 545 java files in a single version on average and modifications are made on average in 206 files per version. The rest of the datasets belong to projects responsible from billing and charging functionalities. Among these three projects, the project to which dataset Telecom2 belongs consists of java and JSP files, and is not as old as the remaining to projects. Hence, modification and updates cover all source code files within the project as well as creating 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

Table 5 List of confirmation bias metrics

Interactive Test Metrics		
Metric	**Explanation**	**Value Type**
N_A	Number of rule announcements	continuous
IND_{ABORT}	Indicates whether participant aborts interactive question session or not	categorical
T_I	Duration of interactive question session (in minutes)	continuous
$Ind_{elim/enum}$	Eliminative/enumerative index by Wason	continuous
$F_{negative}$	Frequency of negative instances	continuous
F_{IR}	Immediate rule announcement frequency	continuous
$avgL_{IR}$	Average length of immediate rule announcements	continuous
$avgF_{RsR}$	Average frequency of reason repetition	continuous
$avgF_{RR}$	Average frequency of rule repetition	continuous
$Instances/Time$	Number of instances given per unit time	continuous
$UnqReasons/Time$	Number of unique reasons given per unit time	continuous
$Rules/Time$	Number of rules announced per unit time	continuous
$UnqRules/Time$	Number of unique rules announced per unit time	continuous
$n_{Pos+Comp}$	Number of positive and compatible instances	continuous
$n_{Pos+InComp}$	Number of positive and incompatible instances	continuous
$n_{Neg+Comp}$	Number of negative and compatible instances	continuous
$n_{Neg+InComp}$	Number of negative and incompatible instances	continuous
Written Test Metrics		
Metric	**Explanation**	**Value Type**
S_{Abs}	Score in abstract questions	continuous
S_{Th}	Score in thematic questions	continuous
S_{SW}	Score in the second part of the written question set	continuous
T_{Th+Abs}	Time it takes to answer the first part of the written question set	continuous
T_{SW}	Time it takes to answer the second part of the written question set	continuous
$ABS_{CompleteInsight}$	Number of abstract questions answered with *complete insight*	continuous
$ABS_{PartialInsight}$	Number of abstract questions answered with *partial insight*	continuous
$ABS_{NoInsight}$	Number of abstract questions answered with *no insight*	continuous
$Th_{CompleteInsight}$	Number of thematic questions answered with *complete insight*	continuous
$Th_{PartialInsight}$	Number of thematic questions answered with *partial insight*	continuous
$Th_{NoInsight}$	Number of thematic questions answered with *no insight*	continuous
$ThABS_{Insight}$	Insight according to the answer given to thematic-abstract question	categorical
$FVMN_{TypeI}$	Reich and Ruth's categorization with respect to Type I question	categorical
$FVMN_{TypeII}$	Reich and Ruth's categorization with respect to Type II question	categorical
$FVMN_{TypeIII}$	Reich and Ruth's categorization with respect to Type III question	categorical
$FVMN_{TypeIV}$	Reich and Ruth's categorization with respect to Type IV question	categorical

software package responsible from monitoring collection of revenues. This software package has been developed and maintained since the inception of the GSM company in 1994. There are 1092 java and JSP files in a single version of this software package on average. However, maintenance, development and software testing activities take place only for 284 files. Finally, dataset Telecom4 is extracted from a project that is as old as the project to which dataset Telecom3 belongs. This project is a software package responsible from database transactions and it consists of PL/SQL files. Similar to Telecom3, only files which are being maintained and created are taken into account during defect prediction analysis.

Dataset ERP consists of a single release of the software product, therefore no merging process was used to learn and test defect prediction models. On the other

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

hand, datasets Telecom1 and Telecom2 are obtained by merging files in four releases of the software being developed. The remaining datasets Telecom3 and Telecom4 are obtained by merging files which belong to 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. that file has been modified). Otherwise, such a file is included to the list only once.

6.2 Metric Extraction Process

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

In order to extract churn metrics, we parsed the log files which are obtained from version control systems. Table 8 consists of the list of churn metrics we used as input data to learn defect predictors. 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, log file for the second dataset covers file commit activities starting from the beginning of September 2001 till the end of December 2009. Finally, 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 outcomes of the interactive question and written question set to extract confirmation bias metrics. Details about the confirmation bias metric suite used to learn defect predictors are briefly explained in Section 4. In order to calculate confirmation bias metrics corresponding to each file, we consolidated confirmation bias metrics from individual developers to developer groups. For each file in each version, the developers who created and/or modified that file before the code freeze date (i.e. dates when development phase for that release is over and testing phase starts) are considered to be responsible from any defects found in that file. This is due to the fact that some previously introduced defects can be overlooked during testing phase of earlier versions resulting in the propagation of defects. For each file in each version, while examining file commit information retrieved from version control systems, we took into account code freeze dates. As a result, we obtained a group of developers responsible from that file.

As shown in Table 5, some confirmation bias metrics take continuous values while some take categorical values. For each confirmation bias metric, which take continuous values, we applied three different operators to calculate minimum, maximum and average of the metric values of developers who contributed to the same source file. Results are assigned to be the source file's corresponding feature. 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* , *max* or *avg* which are used to find minimum, maximum and average values of the i^{th} confirmation bias metric respectively. We can formulize the definition for the *min*, *max* and *avg* operators as follows:

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

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

$$S_{ji}^{avg} = \sum_{d}(A_{di} | \forall d \in G_j) / \sum_{d}(1 | \forall d \in G_j) \tag{3}$$

We follow a different procedure while consolidating categorical confirmation bias metrics of each developer $d \in G_j$ to find the corresponding metric value for the group of developers who created and/or modified j^{th} source file. Assume that C_{di}^{j} is the value of the i^{th} categorical confirmation bias metric of developer d in group G_j and it can take one of the values $c_1, ..., c_n$. Then, for each c_k, where $k = 1, ..., n$, we can calculate the portion of developers who created and/or modified j^{th} source file such that value of the $i^{t}h$ categorical confirmation bias metric is equal to c_k, as follows:

$$Sc_{ji}^{k} = \sum_{d}(1 | \forall C_{di}^{j} = c_k) / \sum_{d}(1 | \forall d \in G_j) \tag{4}$$

Hence, consolidating categorical metrics which can take one of the n values of $c_1, ..., c_n$ for each developer $d \in G_j$, results in the formation of n metrics for the j_{th} source file.

6.3 Defect Matching

All datasets except for the first one were obtained from two project groups within the large scale Telecomunication company. As mentioned previously, first dataset belongs to an ERP software project developed by the ISV company. In order to match defects for the first data-set, we had to learn about the work-flow followed by the ISV during their software development life-cycle. The company uses an issue management system. Each issue is stored in this system with a unique issue code and it can be a *new feature* to be added to the software being developed, a regular *project item* or a *defect* that needs to be fixed. We managed to match issue items that were labeled as *defect* with source code files. According to the company's software development

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

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 defective files. The company provided us with the issue list extracted from the issue management system. We formed a ”final issue list” by

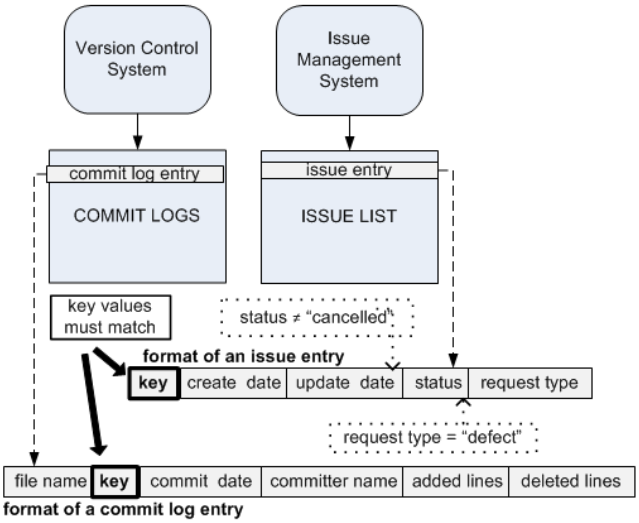

Fig. 2 Defect matching procedure of a file for the last dataset

taking into account only issue entries where request type is *defect* and issue status is *different than canceled*. An issue of request type *defect* and status *canceled* corresponds to defects whose existence could not be verified. They are mostly due to the factors related to the testing environment of the tester and thus they do not affect the customer. We mined the commit log file obtained from the version control system to get a commit history file where format of each commit log entry is in the form as shown in Figure 2. Finally, for each issue in our final issue list we found names of source files in commit history file and marked those files as defective.

Second dataset belongs to the project group which is among the project groups with whom we have been doing a collaborative research for a couple of years [**?**]. Hence, this project group already has an existing infrastructure to list source files where bugs are detected during testing phase for each release of the software product. However, this was the first time we collaborated with the company's second project group to which third, fourth and fifth datasets belong. This project group provided us with a list of file commit activities which took place to fix defects. We were informed about the fact that all software products of the second project group are released to the market at the same time labeled with a single release number. We were also given release calendar which contains code freeze and production release dates of each release in addition to the information about dates when a particular defect was detected and/or fixed. We took into account the fact that a file belongs to a specific release if and only if the date when that defect is detected and/or fixed, is later than code freeze date and earlier than production release date of that release. 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 Construction of the Prediction Model

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

6.5 Performance Measures

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

Probability of Detection (pd): Pd measures how good a predictors 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 implies that pd is equal to 1.

Probability of False Alarms (pf): Pf measures false alarm rates, when 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 high probability of detecting defective modules and low probability of false alarm is very rare. Therefore, we try to balance between pd and pf values. The notion of balance is formulized to be 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.

$$bal = 1 - \frac{\sqrt{(1-pd)^2 + (0-pf)^2}}{\sqrt{2}} \tag{5}$$

Pd and pf values are calculated using Confusion Matrix that is given in Table 9. 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. Formulations for pd and pf in terms of confusion matrix values is given below:

$$pd = TP/(TP+FN) \tag{6}$$

$$pf = FP/(FP+TN) \tag{7}$$

Table 9 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

6.6 Results of the Empirical Study

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

Performance results for the dataset ERP are summarized in Table 10. Probability of detection (*pd*) of the defect predictor which is built using only confirmation bias metrics is higher than *pd* value of the predictor which 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, defect prediction model that is learnt 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. Kruskal-Wallis test results indicating the statistically significant difference in pf values is $\chi^2 = 62.70$, $p = 0.0060$; whereas the results for the difference in *bal* values is $\chi^2 = 15.29$, 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 defect predictors also resulted in in an improvement in defect prediction performance. Average balance value of the defect predictor which 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 10 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

Defect prediction performance results obtained for the dataset Telecom1 is inline with the results obtained for dataset ERP. As it can be seen from Table 11, 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 learnt using both static code and churn metrics and the balance value that is obtained by using only confirmation bias metrics ($\chi^2 = 0.36$, $p = 0.55$). 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). 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 metric 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 results in an average balance value that is far better than the average balance values obtained from individual usage of these three metric types. Moreover, the resulting defect predictor outperforms the defect predictor which is learnt from static code and churn metrics as well as exceeding performance of the prediction model which is learnt from confirmation bias and churn metrics. However, the highest defect prediction performance is obtained by using static code and confirmation bias metrics.

Table 11 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

Unlike the results obtained for the datasets ERP and Telecom1, for dataset Telecom2 individual usage of confirmation bias metrics resulted in average defect prediction performance ($balance = 0.61$) which is lower than those of the defect predictors which are learnt by individual usage of static code ($balance = 0.63$) According to Kruskal-Wallis test the difference is significant: $\chi^2 = 11.71$, p = 0.0006. However, no significant difference is detected between 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 learnt by using both static code and churn metrics outperformed ($balance = 0.68$) all three prediction models which are learnt 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 lower defect prediction performance result compared to the performance result obtained by using only churn metrics ($\chi^2 = 34.83$, p =3.6E-9). This is due to the high correlation between 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 23.12% of confirmation bias metrics is higher than or equal to 0.85. Moreover, 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, defect prediction performance improves significantly by supplementing static code metrics with confirmation bias metrics, compared to the performance values obtained by using static code and confirmation bias metrics separately.

Experiment results for dataset Telecom3 are shown in Table 13. Using only confirmation bias metrics results in a better defect prediction performance rather than using only churn metrics ($\chi^2 = 9.2$, p = 0.0024). On the other hand, 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 which is significantly lower than the performance obtained by using only static code metrics ($\chi^2 = 6.13$, p = 0.0133). This is due to

Table 12 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

the existence of correlation between confirmation bias metrics and static code metrics. Spearman correlation between 17.91% of confirmation bias metrics and static code metrics is greater than or equal to 0.45. 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 $\rho = 0.85, p = 0.0231$. As a result of this, there is an improvement in prediction performance when churn metrics are used with static code metrics. However, the obtained performance is not higher than the performance of the prediction model which is built using only static code metrics. Similarly, Spearman correlation between 26.87% of confirmation bias metrics and churn metrics is higher than or equal to 0.45. 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. Finally, 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 13 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

Table 14 summarizes experiment results for dataset Telecom4. The defect predictor which is built by using only static code metrics outperforms the prediction model which is built by using only confirmation bias metrics ($\chi^2 = 13.31$, $p = 0.0003$). The performance of the latter defect prediction model is also outperformed by the model which is built by using only churn metrics ($\chi^2 = 9.2$, $p = 0.0024$). Both of these

results are inline with the corresponding results of dataset Telecom2. Supplementing static code metrics with confirmation bias metrics does leads to a degradation in defect prediction performance ($\chi^2 = 18.43$, p = 1.76E-5). Spearman correlation between 11.19 % of confirmation bias metrics and static code metrics is higher than or equal to 0.45, while average Spearman correlation is 0.32. On the other hand, supplementing churn metrics with confirmation bias metrics does not cause a significant improvement in 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, introduction of confirmation bias metrics to the metric set of static code and churn metrics does not lead to a statistically significant improvement in defect prediction performance ($\chi^2 = 0.11$, p = 0.7455).

Table 14 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

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

- Performance of defect prediction models built by using only confirmation bias metrics is comparable with performance values obtained by individual usage of static code metrics and churn metrics.
- Any combination of static code, churn and confirmation bias metrics may not lead to an increase in defect prediction performance. This is due to the fact that combination of metrics may not correspond to an increase in information content as a result of the high correlation between any two of the static code, confirmation bias and churn metrics.

6.7 Threats to Validity of Empirical Study

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

each experiment Naïve Bayes algorithm with under-sampling is executed 100 times, for each dataset.

In order to externally validate our results, we used datasets from 5 developer groups. 4 datasets belong to a Telecomunication company, while 1 dataset belongs to an ISV specialized in Enterprize 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 Telecomunications company. One project group develops software which is responsible from launching GSM tariff campaigns to company's customers. Dataset Telecom1 has been extracted from this software and it mainly consists of user interfaces. Remaining projects mainly consist of database transactions and there is no direct interaction with the customer via user interfaces.

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

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

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

Another threat to construct validity is interaction of different treatments. Before administration of confirmation bias test to groups of participants, 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. In order to avoid participants' poor performance due to their apprehension and not to exert psychological pressure on them, before solving both written question set and interactive question, participants were informed about the fact that the questions they are about to solve do not aim to measure IQ or any related capability. Participants were also told that results shall not be used in their company's performance evaluations and their identity shall be kept anony-

mous. Moreover, participants were told that there is no time constraint to complete the questions, although some of our metrics are requires measurement of time it takes to answer questions.

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

In order to avoid internal threats to validity, for all project groups we selected test dates, when workload of developers are not intense. Within any of the groups, there was no event in between the confirmation bias tests that can affect subjects' performance. Members of the developer group corresponding to the first dataset took confirmation bias test, which consists of written question set and interactive question, within a week. Remaining developer groups took the confirmation bias test in a single day. As a result, within a project group for each member we managed to create similar conditions while administering confirmation bias test. If one group member were tested when work load and time pressure were intense, whereas another member of the same group were tested under much more suitable and relaxed conditions, then our methodology would not have been reliable. Another attempt to avoid internal validity was to administer confirmation bias test in environments, which are isolated from distraction factors such as noise.

Finally, to avoid external validity we tried to collect data from two different companies specialized in two different 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, which are located in different countries, specialized in different domains and practicing various development methodologies through a web based application.

8 Conclusion and Future Work

Our main goal in this research is to investigate people's thought processes on software quality. Since people's thought processes cover a wide range of aspects, we have focussed on confirmation bias which 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 investigate how well our metric scheme identifies the effect of developers' confirmation bias on software quality, in our empirical study, we used confirmation bias metrics as input to defect prediction models which we have investigated extensively during our past research [7], [26], [71], [72], [73].

In our empirical study, we used five datasets obtained from two industrial partners which are from telecommunications and ERP domain respectively. We have compared predictors built using confirmation bias metrics with predictors built using static code and churn metrics. Static code metrics we used in this research include Lines of Code (LOC), Halstead [69] and McCabe [68] metrics. In other words, we have covered the set of all major metrics which can be extracted from source code regarding code complexity based on program flow and readability of the code. Similarly, the churn metric set, which we have employed in our analysis, contains extensive information about the changes in source code during the implementation phase. We extracted significant portion of information regarding code change history from version management systems.

On the other hand, confirmation bias metrics represent only a single aspect about people's thought processes. Despite this, according to our empirical findings using only confirmation bias metrics to learn defect predictors yields comparable performance results. Moreover, the phenomena of cognitive biases, which is only one dimension regarding people's thought processes, comprise other bias types such as representativeness, availability, adjustment and anchoring in addition to confirmation bias. In cognitive psychology, the causes of biases have been extensively investigated in various domains over the past three decades since the introduction of the concept of bias by Kahneman and Tversky [28]. In addition to cognitive biases, concepts widely studied in cognitive science such as attention, memory, reasoning, motivation, social cognition are also among the cognitive aspects which require special attention. Hence, there is extensive amount of findings in the field of cognitive psychology which can be employed to form a metric suite covering developers' cognitive aspects which have significant effect on software defect density, hence on software quality.

We are aware of the fact that our research is empirical and as stated by Popper, we cannot verify a theory with limited number of empirical findings, yet falsify it [70]. Hence, we need to be careful while generalizing our experimental results in order not to be subject to confirmation bias we have been discussing in this paper. However, our empirical findings suggest that the effects of cognitive aspects and people's thought processes on software quality deserves to be investigated. Moreover, obtaining comparable performance results in software defect prediction by confirmation bias metrics implies that further investigation of people's thought processes may help us to overcome the ceiling effect in defect prediction performance.

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 people's thought processes. Such a metric scheme will help managers to determine the right person to test the defective parts of the software. As a result, guidance of metrics related to people's thought processes may decrease the uncertainty in Human Resource (HR) related decisions up to a significant extent.

As future work, we aim to collect data from larger software development groups from different companies located in different countries. Collection of data from different contexts shall be possible once we complete implementation of our web based software product. This software will help us to improve our metric suite to cover other relevant cognitive aspects that are briefly mentioned above. Since our software has been designed to be a decision support tool, it shall also be able to analyze the metrics and make recommendations to software professionals.

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 rel iability 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 ciling effects in defect predictors, Proceedings of the 3rd 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 2nd 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 1st 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 4th 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 18th 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 1st 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 Telecomunication 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., Proceding 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 judgement 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 judgement and formal models, International Journal of Forecasting, 23(3), pp. 449-462, (2007).
35. Jørgensen, M., The effects of request formats on judgement-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, Proceedingds 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., Programer-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 shemas, Cognitive Psychology, 17, pp.391-416, (1985)
50. Cosmides, L., The logic of social exchange: Has natural selection shaped how humans reason ? Studies with Wasons 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 Contrieved 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?, Biritish 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, falsificatin 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, Procedings 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 30^{th} 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. Popper, K.R., The logic of scientific discovery. Hutchinson, London, (1959/1974)
71. Turhan, B. and Bener, A., A multivariate analysis of static code attributes for defect prediction, Proceedings of the 7^{th} International Conference on Quality Software, pp. 231-237, (2007)
72. Turhan, B. and Bener, A., Weighted static code attributes for software defect prediction, Proceedings of the 20^{th} International Conference on Software Engineering and Knowledge Engineering, pp.143-148, (2008)
73. Turhan, B., Bener, A. and Menzies, T., Nearest neighbor sampling for cross company defect predictors, Proceedings of the 1^{st} International Workshop on Defects in Large Software Systems, (2008)
74. Cook, T.D. and Campbell, D.T. Quasi-experimentation: design and analysis issues for field settings. Houghton Mifflin, Boston, (1979)