City Research Online

City St George's, University of London

Citation: Huang, F. & Strigini, L. (2023). HEDF: A Method for Early Forecasting Software Defects Based on Human Error Mechanisms. IEEE Access, 11, pp. 3626-3652. doi: 10.1109/access.2023.3234490

This is the published version of the paper.

This version of the publication may differ from the final published version. To cite this item please consult the publisher's version.

Permanent repository link: https://openaccess.city.ac.uk/id/eprint/30602/

Link to published version: https://doi.org/10.1109/access.2023.3234490

Copyright and Reuse: Copyright and Moral Rights remain with the author(s) and/or copyright holders. Copies of full items can be used for personal research or study, educational, or not-for-profit purposes without prior permission or charge, unless otherwise indicated, provided that the authors, title and full bibliographic details are credited, a hyperlink and/or URL is given for the original metadata page and the content is not changed in any way. For full details of reuse please refer to City Research Online policy.

City Research Online: http://openaccess.city.ac.uk/ publications@citystgeorges.ac.uk

IEEE Access
Multidisciplinary : Rapid Review : Open Access Journal

Received 16 December 2022, accepted 1 January 2023, date of publication 5 January 2023, date of current version 12 January 2023.

Digital Object Identifier 10.1109/ACCESS.2023.3234490

RESEARCH ARTICLE

HEDF: A Method for Early Forecasting Software Defects Based on Human Error Mechanisms

FUQUN HUANG[1], (Member, IEEE), AND LORENZO STRIGINI[2], (Member, IEEE)
[1]Centre for Informatics and Systems of the University of Coimbra, Department of Informatics Engineering, University of Coimbra, 3030-790 Coimbra, Portugal
[2]Centre for Software Reliability, City, University of London, EC1V 0HB London, U.K.

Corresponding author: Fuqun Huang (huangfuqun@dei.uc.pt)

This work was supported in part by the Foundation for Science and Technology (FCT), I.P./MCTES through National Funds (PIDDAC), within the Scope of CISUC Research and Development Unit - UIDB/00326/2020 or project code UIDP/00326/2020.

ABSTRACT As the primary cause of software defects, human error is the key to understanding, and perhaps to forecasting and avoiding defects. Little research has been done to forecast defects on the basis of the cognitive errors that cause them. The existing ''defect prediction'' models are applied to code once it has been produced: therefore, their ''predictions'' have little implications for preventing the defects. This paper proposes an approach, ''Human-Error-based Defect Forecast'' (HEDF), to forecasting the exact defects at early stages of software development, before the code is produced, through knowledge about the cognitive mechanisms that cause developers' errors. This approach is based on a model of human error mechanisms underlying software defects: a defect is caused by an *error-prone scenario* triggering *human error modes*, which psychologists have observed to recur across diverse activities. Software defects can then be forecast by identifying such error-prone scenarios the in requirements and/or design documents. We assessed this approach empirically, with 55 programmers in a programming competition and four representative analysts serving as the users of the approach. Impressively, the approach was able to forecast, at the requirement phase, 75.7% of the defects later committed by all of the programmers. When considering just the *defect forms*, which may manifest as distinct defects even in the same program, the proposed method predicted 31.8% of them. This approach substantially improved the defect forecasting performances for analysts of various expertise, with a minimum of 100% improvement, compared to forecasts without the approach. If the forecast had been used to prevent the defects, it could have saved an estimated 46.2% of the debugging effort and increased the fraction of programmers delivering an acceptable program by 32.6%. The observed excellent performance of HEDF in forecasting (early at requirement stage) the exact forms and locations of defects that may be later introduced by developers into code makes it a promising candidate for preventing the defects, worthy of further study.

INDEX TERMS Defect forecast, defect prevention, human error, programming cognition, software quality assurance.

I. INTRODUCTION

Software defects (''incorrect or missing steps, processes, or data definitions in a computer program'' [1]) are a primary threat to the reliability and safety of computer systems in many safety-critical domains, such as aerospace systems [2], [3], energy systems [4], and medical devices [5]. Finding and fixing defects were estimated to cost trillions of dollars, worldwide, in 2018 [6], and yet the residual defects are still causing accidents and threatening human lives. In many safety-critical domains, preventing defects from occurring is especially desirable, because once defects are created in the code, it is hard to guarantee all of them will be found by tests and removed.

As a primary cause of software defects, human error is a key to forecasting and preventing software defects. Programming is a special type of writing, performed by programmers [7]; a computer program is a pure cognitive product that describes its designers' thoughts [8], [9], [10]. Software defects are the manifestations of cognitive errors by

The associate editor coordinating the review of this manuscript and approving it for publication was Liang-Bi Chen.

This work is licensed under a Creative Commons Attribution-NonCommercial-NoDerivatives 4.0 License.
For more information, see https://creativecommons.org/licenses/by-nc-nd/4.0/

individual software practitioners and/or of miscommunication between them [8], [9]. However, there is a lack of theory on how software defects are produced by cognitive error mechanisms, on which methods can be based to prevent software defects.

This paper proposes to forecast software defects early, before the code is produced, aiming to provide actionable inputs for defect prevention activities. The approach is based on understanding of the primary cause of software defects: the human error mechanisms affecting software developers. "**H**uman-**E**rror-based **D**efect **F**orecast" (**HEDF**) is an approach for forecasting software defects that may arise when programmers translate requirements specification and/or design documents into code, through identifying, in these documents, conditions that tend to trigger human errors, and the forms that the errors tend to take. While coding errors are only one cause of defects, they are an important cause, attracting much effort for preventing or finding them.

The proposed objective and method are superficially related to, but fundamentally different from, the existing models in the area of "defect prediction" [11], [12], [13], [14]. Predictions in the "defect prediction" area are performed on code. The predictors can be categorized into three groups [11]: program metrics such as program size and complexity, testing metrics, and software development process metrics. These predictors are then related to defect density by various methods, which have been evolving from simple correlation functions [15] to multivariate approaches such as regression analysis [16], data mining [17] and machine learning algorithms [18]. Graphic methods such as Bayesian Belief Networks (BBNs) [11], [12] and dependency graphs [19] have been used to analyze the dependencies between various metrics. The outputs of the models in the "defect prediction" area usually take two forms: 1) classifying a program module into defective or defect-free, 2) providing a rank list of modules that likely contain defects. These methods help to allocate testing resources more effectively.

What we propose here is fundamentally different from the existing models in the "defect prediction" (DP) area, in terms of: 1) forecast phase — HEDF is performed before code is produced, while DP is performed after code is produced. 2) Inputs—HEDF is performed based on requirements and the knowledge about human cognitions, while DP uses code-based and/or project-based metrics). 3) Outputs—HEDF provides the locations and forms of defects that developers may introduce, while DPs classify or rank code modules' likelihood of containing defects. 4) Purpose—HEDF is devoted to providing an actionable checklist for defect prevention, while DPs help allocate testing resources. For these reasons, we deliberately use the word "forecast" instead of "prediction", to avoid any confusion or involving the readers in unnecessary comparisons.

The rest of this paper is organized as follows: Section II reviews the literature; Section III presents the proposed approach; Section IV presents the research design, including research questions, metrics, the task and the overview of the process; Section V presents the real programming data produced by 55 programmers for a given specification, while Section VI presents the case study of representative users performing defect forecast using HEDF; Section VII presents the results analysis by comparing the real programming data and the forecasting results; Section VIII provides discussions; Section IX concludes the paper.

II. LITERATURE REVIEW

We have found no literature on approaches similar to that studied here, that is, aiming at pinpointing which defects are likely in a program *before* the program is written, using human error theories. We briefly outline the huge literature in the defect prediction area, which has a similar name but is only marginally relevant to our method. We also review a much more relevant area, studies of human errors in software engineering.

A. "DEFECT PREDICTION" METHODS

The defect prediction area has been continuously developing since the early 1970s, when Akiyama first built a correlation model between lines of code (LOC) and number of defects in program modules. More complex metrics such as Cyclomatic complexity [20] and Halstead complexity [21] were then proposed to represent the complexity of a software system or module for such correlation models. Since then, both the predictors and the methods used to model the relation between predictors and software defects have evolved.

Various metrics have been used as predictors for defect prediction, such as process metrics [22], testing metrics, design metrics [23], organizational structure metrics [24], code change metrics [25], [26], dependency metrics [19] and social network measures [27]. With so many metrics, the problem arises of how to choose the most effective ones. A series of studies try to rank or simplify the metrics using statistical analysis [28], while others try to integrate various metrics [29], or summarize and compare various metrics through literature review [30], [31]. A variety of modeling methods have also been proposed to relate the predictors and software defects, such as Bayesian Network [11], [32], and various machine learning algorithms [18], [33], [34], [35].

Researchers proposed to predict software defects in various phases of software development [12], [36], [37]. These methods generally use Bayesian networks [38] and other graphical models [37] to integrate different kinds of metric data. The most difficult problem in current software defect prediction studies is that the current methods do not work well on novel projects or projects for which historical data for comparable projects are lacking [39]. To explore these issues, various cross-project defect prediction models are proposed [39], [40], [41]; these studies basically propose to consider more contextual factors (such as whether the products are from the same domain and from the same company) to characterize software projects.

More recently, the importance of software developers' contribution to software defects has been recognized [42],

[43], metrics such as the number of low-expertise developers [44] and developer-module networks [45] are proposed to enhance defect prediction models. There is still debate whether the metrics used for human factors are too simplistic [43], [45], [46].

Despite the significant progress made, the current prediction models are generally applied after code is available, and their purposes are prioritizing test case [47], [48], [49] and planning maintenance [50]. Researchers have reported some limitations such as very few uses in industrial projects [49], [51], [52], and lack of "actionable" advice for developers [49], [52].

The method explored in this paper is radically different from existing methods in the "defect prediction" area. We explore whether defects are forecastable in the early phases before software implementation, and how such forecast can be achieved through understanding of how developers commit errors.

B. HUMAN ERROR CAUSE OF SOFTWARE DEFECTS

To the best of our knowledge, globally there are four research teams that have studied the human error causes of software defects from a psychological perspective.

Ko and Myers [53] first introduced J. Reason's human error theory [54] to study software errors of a programming environment. Ko and Myers' research [53] was from a human-computer interaction perspective: the "software errors" are the defects in a programming environment, programmers are the users of the programming environments, and human errors are the errors of the programmers in using these tools.

Huang (one of the present authors) and Liu [55] first proposed the interdisciplinary area "Software Fault Defense based on Human Errors" [55], [56], which aims to systematically reduce software defects based on an understanding of the cognitive errors of software practitioners. After that, Huang et al. have conducted a series of in-depth studies on various topics: defect prevention based on human errors [9], [57], which includes a comprehensive human error taxonomy for root cause analysis [9], and an approach for promoting software developers' knowledge, awareness and abilities to prevent defects through cognitive training [57]; fault tolerance based on human errors [58], which identified several dimensions of cognitive styles and performance levels that can be used to seek fault diversity; the new controlled experimental method for studying the cause-effect relationships between the cognitive error mechanisms of software engineers and software defects [59]; human error modeling tool, which is used to represent the interactions between human errors and various context factors [60]; code review based on human errors [61], in which code reviewers' performance in finding defects in code is improved through "cognitive training".

Anu, Hu, Carver, Walia, and Bradshaw [62] focused on using a human error taxonomy to improve quality of requirements. Their research also found that students who received training on human error taxonomy wrote requirements with fewer defects [63]

Nagaria and Hall [64] recently interviewed developers about the situations of the skill-based errors and how they mitigate such errors.

Despite the progress made, this emerging area still lacks a causal mechanism model explaining how cognitive errors interact with context factors to trigger the production by software developers of software (design and/or coding) defects in code. No method has been published for forecasting software defects early (before code is produced) for focused preventative action.

III. THE PROPOSED APPROACH

Though human errors appear to be diverse across different activities, a proportion of human errors are predictable, in the sense that human errors take a limited number of recognizable patterns [59]. The proposed approach HEDF, is therefore based on analyzing a programming task for patterns that match the conditions known to trigger erroneous human behaviors; patterns that psychologists have observed to recur across diverse activities [59], [60]. We call this **E**rror-**P**rone **S**cenario **A**nalysis (EPSA).

In this chapter, Section A defines the concepts used in HEDF; Section B extracts a set of human error patterns and the general conditions that tend to trigger these error patterns; Section C presents EPSA, which is the core process of the HEDF approach; Section D presents a detailed demonstration on how to use EPSA on a specific requirement, and/or design if the design document is available.

A. TERMINOLOGY

The concepts used in HEDF are defined as follows:

Defect:an incorrect or missing step, process, or data definition in a computer program (adopted from "fault" in [1]). Note that "defects" in this paper are manifested in the computer program, but are not limited to those caused by errors in coding; they may originate as defects in a design, and may originate even further upstream, during the process of understanding requirements. A defect can be described by the difference between the incorrect and correct program at the specific defect location. A defect has two properties:

Defect Form, referring to a category of defects caused by the same human error mechanisms, that is, the way how a snippet of step, process, or data definition in a computer program is incorrect (where "being absent" is a form of "being incorrect"). Note that the Defect Form here is not the same as the "Defect Type" concept commonly used in software engineering, e.g. in "Orthogonal defect classification" [65]. Defect Form describes "what" a defect is, in more detail than "Defect Type". Still, one defect form can manifest itself as diverse unique defects. For instance, the defect form "the size of a variable is assigned without considering its usage requirement" may manifest itself as the defective code "char map [100], [100]" in one program but as "char a [50], [50]"

in another program, where both two arrays (''map'' and ''a'') should be larger than 512*512.

Defect Location is a logic place in the program that realizes a specific function. That is to say, the Defect Location here is defined through functionality rather than common ways such as ''lines 22-25 in file x.c''; a defect can be limited to a line of code, but it can also be a set of related code fragments in different modules or even several sub-systems. Defining a defect location from a functional viewpoint makes it possible to connect ''what a snippet of program does'' to ''how a programmer thinks''; by contrast, conventional location identifiers such as lines of code can hardly fulfill this purpose. For instance, the same number for a line of code (e.g. Line #22) in a program version developed by programmer A could do completely different things from that of another version developed by programmer B, even when the two versions have the same total number of lines of code and implement the same requirement. Examples of defect locations are demonstrated in tables from Table 5 to Table 11.

Defect early forecasting: the activity of a person to forecast, at requirement and/or design stage, defects that may be later introduced by programmers into code.

Error: an erroneous human behavior that leads to a software defect. Errors are classified at a finer-grained level in psychology as mistakes, slips or lapses [54]. Mistakes affect the analysis of a problem or conscious choice of action to perform; slips and lapses are involuntary deviations or omissions in performing the intended action.

Human Error Mode (HEM):a particular pattern of erroneous behavior that recurs across different activities, due to a cognitive weakness shared by all humans, e.g. applying ''strong-but-now-wrong'' rules (see Table 1) [54].

Error-Prone Scenario (EPS): A set of conditions under which a HEM tends to occur. An EPS encompasses not only the exterior conditions surrounding an individual (e.g. the most important factor–task), but also the interior cognitive conditions relevant to an individual's performance, e.g. his/her knowledge relevant to the task.

Error-Prone Scenario Analysis (EPSA): the process of an analyst identifying Error-Prone Scenarios. HEDF forecasts software defects by identifying EPSA, that is, HEDF is an EPSA-based approach.

Error Mechanism: how an error is formed; the way causal factors (e.g. the scenario and the error mode) interact to form an error. *The mechanism underlying a software defect is that the Error-Prone Scenarios have triggered one or more Human Error Modes.* Some defects can be caused by one single error mode, while others may be introduced by a combination of several error modes.

In summary, ''Error Mode'' concerns ''***why***'' a defect is introduced; ''Error-Prone Scenario'' concerns ''***when***'' (under what circumstances) a defect is introduced; ''Error Mechanism'' integrates all the aspects concerning ''***how***'' a defect is introduced.

Several psychological concepts will also be used in the paper:

Rasmussen's performance level [66]: A classification of cognitive activities into three ''levels'': Skill-based (SB) level, Rule-based (RB) and Knowledge-based (KB) level. We recall the three definitions below. Different performance levels have different cognitive characteristics, thus have different error modes [54].

Skill-based performance follows from the forming of an intention and ''rolls along'' automatically without conscious control. Skill-based activities in programming include typing a text string, compiling a program by pressing a button in the programming environment.

Rule-based performance is applicable for tackling familiar problems. It is typically controlled by stored rules or procedures that have been derived from a person's experiences. The mind matches patterns in the situation at hand to the preconditions for such stored rules, allowing quick selection of actions. In programming, there are many rule-based performances, such as programming the printing of a string line, and defining a variable in one's familiar programming language.

Knowledge-based performance comes into play when one faces novel situations, and no rules are available from previous experiences. At this level, actions must be planned using an analytical process. Errors at this level arise from resource limitations and from incomplete or incorrect knowledge. In programming, cognitive performances such as constructing the mental model of the system so as to understand a specific programming task, and trying to figure out a solution for a novel problem are knowledge-based performance.

Schema: ''A schema is a modifiable information structure that represents generic concepts stored in memory. Schemata represent knowledge that we experience–interrelationships between objects, situations, events, and sequences of events that normally occur. In this sense, schemata are prototypes in memory of frequently experienced situations that individuals use to interpret instances of related knowledge.'' [67]

B. HUMAN ERROR MECHANISMS

We propose a model of the Human Error Mechanism (shown in Figure 1) for describing the process of how a software defect is caused by human errors. This model includes the main causal factors that determine a human error [54]: the nature of a task (including both its content and the form in which it is represented), the nature of the programmer (mainly including their current available knowledge base [10]), and human error modes, which are the general mechanisms governing humans' erroneous cognitive performance.

A software defect is caused when the conditions of the programming task and/or programmer trigger one or more human error modes. To predict an error, the task and individual should be analyzed together because these two factors interact. For instance, the same task could be very easy for one person while difficult for another person. We call this combination between the features of a task and of a human individual a ''scenario''.

Based on the human error mechanism model in Figure 1, software defects can then be predicted through identifying in

TABLE 1. A pool of human error mechanisms.

Human Error Modes	Descriptions of the error modes	Error-Prone Scenario
Applying "strong-but-now-wrong" rule	In a context that is similar to past circumstances, people tend to behave in the same way, neglecting the signs of exceptional or novel circumstances. The typical scenarios include: the person encounters a new feature for the first time ("First Exception"); the competing rule is very "strong", in that has been successfully used many times before ("Rule Strength"); and the competing rule is a general rule ("General Rules") [54]. We should expect that programmers tend to prefer rules that have been successfully used in the past. The more frequently and successfully the rule has been used before, the more likely it is to be recalled and used.	***IF*** *Current task requires* Rule X <Feature FeX >, ***WHEN*** *There Exists* Rule A <Feature *Fe*A, *F*requency of successful *u*sage *Fu*A>, ***AND*** *There Exists* Rule B <Feature FeB, *F*requency of successful *u*sage *Fu*B >; $\{FeX \cap FeB\} \supseteq \{FeX \cap FeA\} \neq \emptyset$; FuA>> FuB, ***OR*** Fu $((FeX \cap FeB)_i)$=0, ***OR*** $FeB \subset FeA$; ***THEN*** The person tends to retrieve Rule A.
Rule encoding [a] deficiencies	Features of a particular situation are either not encoded at all or misrepresented in the conditional component of the rule [54].	***IF*** *Current task requires* Rule X <Feature FeX >; ***WHEN*** *There Exists* Rule $\tilde{X}$<Feature Fe$\tilde{X}$ >; $FeX - Fe\tilde{X} \neq \emptyset$ ***IF*** The person uses Rule $\tilde{X}$; ***THEN*** The person commits an error of misusing Rule $\tilde{X}$ for the current situation that requires Rule X.
Lack of knowledge	Software defects are introduced when one lacks knowledge, or even does not realize some extra knowledge is required. This error mode is liable to appear especially when the problem belongs to an unfamiliar application domain.	***IF*** *Current task requires* Rule X; ***WHEN*** Rule X *does not exist*; ***THEN*** The person tends to fail the task.
Difficulties with exponential developments [54].	Psychologists find that humans to underestimate the rate of change, either growth or decline. Humans tend to construct linear models (whose growth rate is lower than exponential models) when exponential models are required to understand a situation in reality	***IF*** *Current task requires* extracting a relation between independent variable x and dependent variable y according to a sample data; ***WHEN*** The actual relation belongs to models in the families of "$y = x^p$" or "$y = d^x$"; ***THEN*** People tend to construct wrong models in the family of "$y = ax$".
Selectivity	Psychologically salient, rather than logically important task information is attended to [54]. "Selectivity" in programming means that when a programmer is solving problems, if attention is given to the wrong features or not given to the right features, mistakes will occur, resulting in wrong problem presentation, or selecting wrong rules or schemata to develop solutions. For example, if there are several requirement items scattered at different places in a document, for a single task, programmer may fail to notice some information and perceive a wrong representation of the task.	***IF*** *Current task contains* features FeT_i ={$Salience_i$, $LogicImportance_i$} and FeT_j ={$Salience_j$, $LogicImportance_j$} ***WHEN*** $Salience_j > Salience_i$; ***THEN*** People tend to notice a FeT_j; ***WHEN*** $LogicImportance_i > LogicImportance_j$; ***THEN*** An error tends to be introduced due to the omission of FeT_i.
Biased review	Humans tend to believe that all possible courses of action have been considered, when in fact only a subset have been considered. When programmers generate test cases in debugging, they may fail to take all conditions into consideration, e.g. exception and boundary conditions [54].	***IF*** *Current task requires* a person to review one's own work X; ***WHEN*** X contains N courses or conditions; ***THEN*** The person tends to review n<N courses or conditions.
Post-completion error [68]	If the ultimate goal is decomposed into sub-goals, a sub-goal is likely to be omitted under the following conditions: the sub-goal is not a necessary condition for the achievement of its super-ordinate goal, and the sub-goal is to be carried out at the end of the task.	***IF*** Task A ={Task A.1, Task A.2, …., Task A.n}; ***WHEN*** <Task A.1 is the main subtask>, ***AND*** <Task A.n is not a necessary condition to Task A.1>, ***AND*** <Task A.n is the last step of Task A >; ***THEN*** Humans tend to omit Task A.n.

[a] Encoding is the process of extracting features of a situation and mentally representing the situation based on the extracted features.

some part of the current programming contexts (e.g., a program specification) conditions that are likely to trigger human error modes. To conduct such prediction, the first step is to build a pool of human error mechanisms that contains the error modes and their associated general scenarios.

We developed a pool of human error mechanisms (shown in Table 1) based on the error patterns that have been widely accepted in the psychological community, such as the error patterns summarized by Reason [54] and Byrne and Bovair [68]'s theory on "post-completion errors". These error modes are included because they are reported to reoccur across diverse activities. Three strategies are used to build the pool of human error mechanisms:

1) ERROR MODES ARE ONLY INCLUDED IF THEY ARE SUITABLE FOR EPSA

Some of the error modes identified in psychology are not. For example, when an individual's working memory is

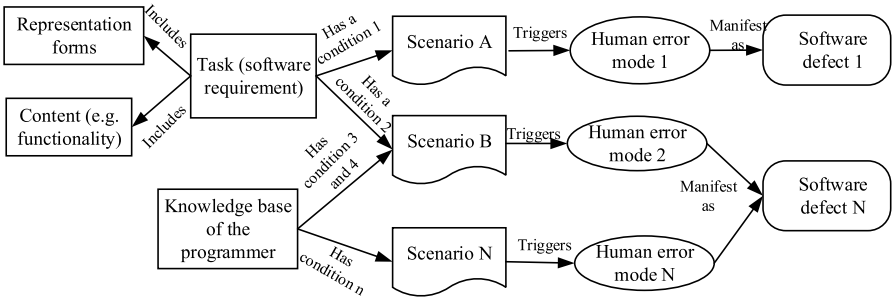


FIGURE 1. A model of human error mechanisms underlying software defects.

overloaded, he/she is likely to commit an error; this is called "Workspace Limitations" by Reason [54]. However, working memory load is closely related to one's working memory capacity, one's expertise on the problem (intrinsic load), the format of the problem representation materials (extraneous load) and one's schema construction processes (germane load) [69]. Working memory overload is a real-time cognitive state that is thus hard to forecast. Therefore, we do not include it in our initial pool of human error mechanisms; adding it may become possible with further study of cognitive load in programming tasks.

2) THE ERROR MODE TAXONOMY IS ADAPTED IN ORDER TO RETAIN A SYSTEMATIC RELATION BETWEEN THE ERROR MODES INCLUDED

For example "Countersigns and non-signs", "Rule strength", and "General rules", and "Redundancy" are four sub-modes under the mode "misapplication of good rules" in.Reason [54]'s work. These modes describe different situations in which people are prone to apply "strong-but-now-wrong" rules, but they pertain to the same mechanism. Therefore, we consider them not as error modes but as scenarios in which the error mode of "applying 'strong-but-now-wrong' rule" is prone to occur.

3) THE ERROR MODES ARE EXTRACTED AND REPRESENTED BY PSEUDO CODES, SHOWN IN THE THIRD COLUMN OF TABLE 1

The fundamental psychological theories tend to be somewhat vague and thus difficult to apply for practical purposes. We specified the preconditions (using "IF"), the situations under which an error tends to occur (using "WHEN"), and the final manifestation of the error (using "THEN"). Notations such as "AND" and "OR" are used to combine multiple situations. The definitions of the specific notations other than natural language are provided in Table 2.

Table 1 is ***not*** meant to exhaustively enumerate all the human error modes described in cognitive psychology. This pool is used for the practical purpose of exploring whether and how software defects can be predicted based on human error mechanisms; therefore, it only includes a selected set of human error modes that the authors consider systematic, valid, typical and suitable for performing EPSA, based on the current understanding of human error theories. The pool can be extended with the progress of research in cognitive psychology and with extensions of this study in the future. For instance, while above we listed a human error mode "difficulties with exponential developments", because this has been documented by psychology researchers, we would not be surprised if experience showed this to belong to a broader category of mathematical functions and series that programmers find it difficult to extract from informal specifications.

TABLE 2. A sample of notations used to represent Human Error Scenarios.

Notation	Meaning
IF	Starting a clause defining the precondition of an Error-Prone Scenario based on the requirement specification, before she had access to any information about the programs developed to satisfy
WHEN	Starting a clause describing the conditions that form an Error-Prone Scenario
THEN	Starting a clause describing the manifestation form of an Error
AND	Logic AND, collecting two conditions, both of which need to be satisfied for forming an Error-prone Scenario.
OR	Logic OR, collecting two conditions, either of which needs to be satisfied for forming an Error-prone Scenario.
$\cap$	Intersection
$\subset$	Belongs to
$\emptyset$	Empty set
>	More than
<	Less than
=	Equal
$\neq$	Not equal
>>	Far more than
,	Conjunction between conditions that constitute an error-prone scenario
;	Separation between two independent clauses.
.	The end of the representation of a human error mode.

C. ERROR-PRONE SCENARIO ANALYSIS

This is the process to match the specific contexts of the task to the general features that tend to trigger a human error mode. For instance, for the post-completion error, an analyst can review the requirement and/or design document (if available), and see if there is a snippet of items forming a task, and then check whether these items form the EPSA of post-completion

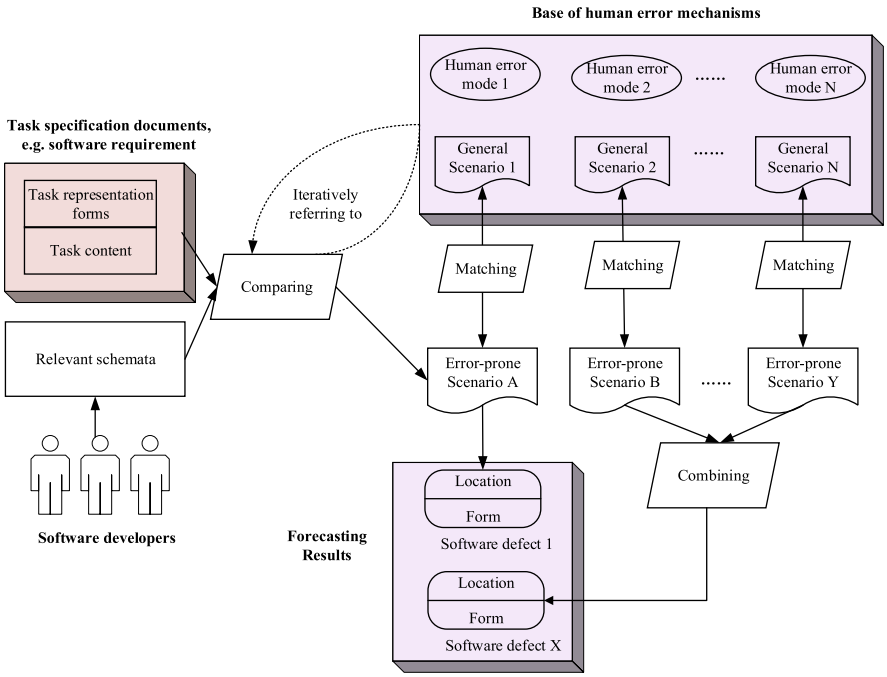


FIGURE 2. The Error-Prone Scenario Analysis process.

error (the last row of Table 1): there is a main item and several non-main items, and a non-main item is not a necessary for completing the main item, but logically at the last step of the whole task. For the human error mode "difficulties with exponential developments", an analyst can simply check if there are places in the requirement involving quantitative relationships between variables, it is an error-prone scenario if a quantitative relationship is exponential. Considering the error mode "applying 'strong-but-now-wrong' rule", the analyst needs to check if there are some new or exceptional features contained in the documents; relevant domain and programming knowledge is expected to help in identifying EPSAs of this HEM.

EPSA is performed in an iterative manner, shown in Figure 2. The scenario analysis itself is an analyst's cognitive process, just like any other types of software reviews, such as requirement review and code review. The contribution of HEDF is to tell an analyst "what to look at" (HEM) and "how to look at" (EPSA). Furthermore, we designed a set of graphic symbols to guide the scenario analysis and record the results, shown in Figure 3. We expect such graphical representations to help. Symbols could remind one of the essential elements that constitute an error-prone scenario, so as to promote the thinking process as well as represent the mental model produced during the analysis [10].

D. DEMONSTRATION OF ERROR-PRONE SCENARIO ANALYSIS

The first author (Huang) performed an error-prone scenario analysis on the "jiong" requirement task specification. The EPSA was performed in a forecasting manner, that is, solely based on the requirement specification, before she had access to any information about the programs developed to satisfy those requirements; the EPSA was recorded as an exploratory study and was formally reviewed by multiple independent academic committee members [56]. The defects that were forecast were indeed found to be introduced by some programmers in a programming contest. These are indicated by grey background in the first column of Table 13. However, to avoid possible researcher bias, in this paper Huang's EPSA is used for demonstration purpose only, rather than for validating HEDF. The forecast processes and data in this paper remain the same as that presented in Huang's PhD dissertation [56], while the EPSA diagrams are updated with the new notation for the Human Error Analysis [60]. The purpose here is to show how to perform EPSA in a specific programming task, so interested readers can better apply HEDF in their own contexts.

1) THE REQUIREMENT

The programming task used in this study is called the "jiong" problem. "Jiong" is a simplified Chinese character shown in Table 3. The requirement specification of the task is shown in Table 3. The line numbers in the left column was added by the authors to facilitate the analysis in this paper.

For a given programming problem, we expect that the programs written by different programmers may differ in detail; but the problem to be solved poses a set of identical demands to all, so that some similar or common parts exist between different programs. For instance, Table 4 shows the two typical solutions of the "jiong" problem and the elements likely to be shared by programs implementing either solution.

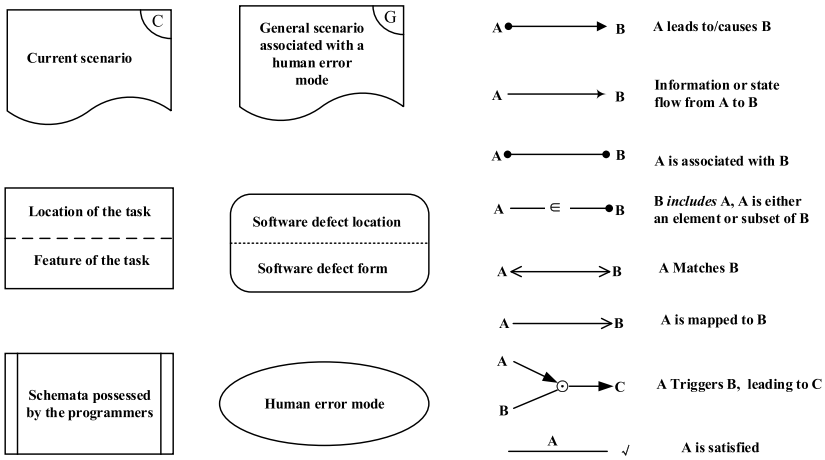


FIGURE 3. Symbols used to aid Error-Prone Scenario Analysis.

TABLE 3. The requirement specification of the "jiong" task.

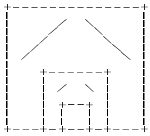
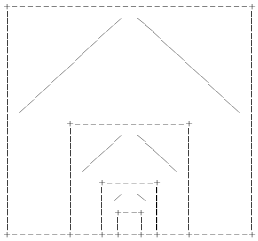
Line1	Print a Chinese word "jiong" in a nested structure.
L2	***Inputs***
L3	There is an integer in the first line that indicates the number of input groups.
L4	Each input group contains an integer n (1≤n≤7).
L5	***Outputs***
L6	Print a word "jiong" after each input group, and then print a blank line after each "jiong" word.
L7	***Sample Inputs***
L8	3
L9	1
L10	2
L11	3
L12	***Sample Outputs***
L13	
L14	
L15	
L16	
L17	
L18	

Table 4 is comparable to design documents in organizational software development contexts.

2) ERROR-PRONE SCENARIO ANALYSIS EXAMPLES

In total, seven error-prone scenarios were identified, with seven scenario forms produced, shown from Table 5 to Table 11. Each form described the location of the scenario within the task, the error modes that were likely to be triggered, the defect forms that were likely to occur, and how this scenario was identified–scenario analysis. Each scenario analysis process produced an error mechanism model, describing possible interaction mechanisms between the task, individuals and error modes.

TABLE 4. Two typical solutions of the "jiong" problem.

Solution A	1) Define and initialize an array to store the symbols (+, -, /, |, blank space) to form the smallest "jiong" as indicated by the image; 2) Read in the number of "jiong"s, and the nesting levels for each "jiong"(indicated as n); 3) Compute the symbols of the "jiong" at the *n*-th nesting level by recursion according to the relationship revealed in the images, and store in another array; 4) Print a "jiong" line by line by two nesting loops, where the number of iterations is related to the width and height of the "jiong". Model the relationship between the structure of the "jiong" and its nesting level: the width=2^{n+2}, height = width; 5) Print a blank line after the "jiong"; 6) Return to the loops and process the next "jiong", until all the "jiong" images are printed.
Solution B	1) Define and initialize an array to store the symbols (+, -, /, |, blank space) to form the smallest "jiong" as indicated by the image; 2) Read in the number of "jiong"s, and the nesting levels for each "jiong"(indicated as n); 3) Model the relationship between the structure of the "jiong" and its nesting level: width=2^{n+2}, height = width; 4) Compute and print each symbol of the "jiong" at the *n*-th nesting level by iteration, where the type of a symbol is determined by the height of the symbol's location in the "jiong" image. The number of iterations is determined by the width and height of the "jiong" at the *n*-th nesting level; 5) Print a blank line after the "jiong"; 6) Return to the loops and process the next "jiong", until all the "jiong" images are printed.

IV. RESEARCH DESIGN

Since HEDF focuses on forecasting the code locations where defects tend to be created due to common cognitive error

TABLE 5. Error-prone scenario analysis (1).

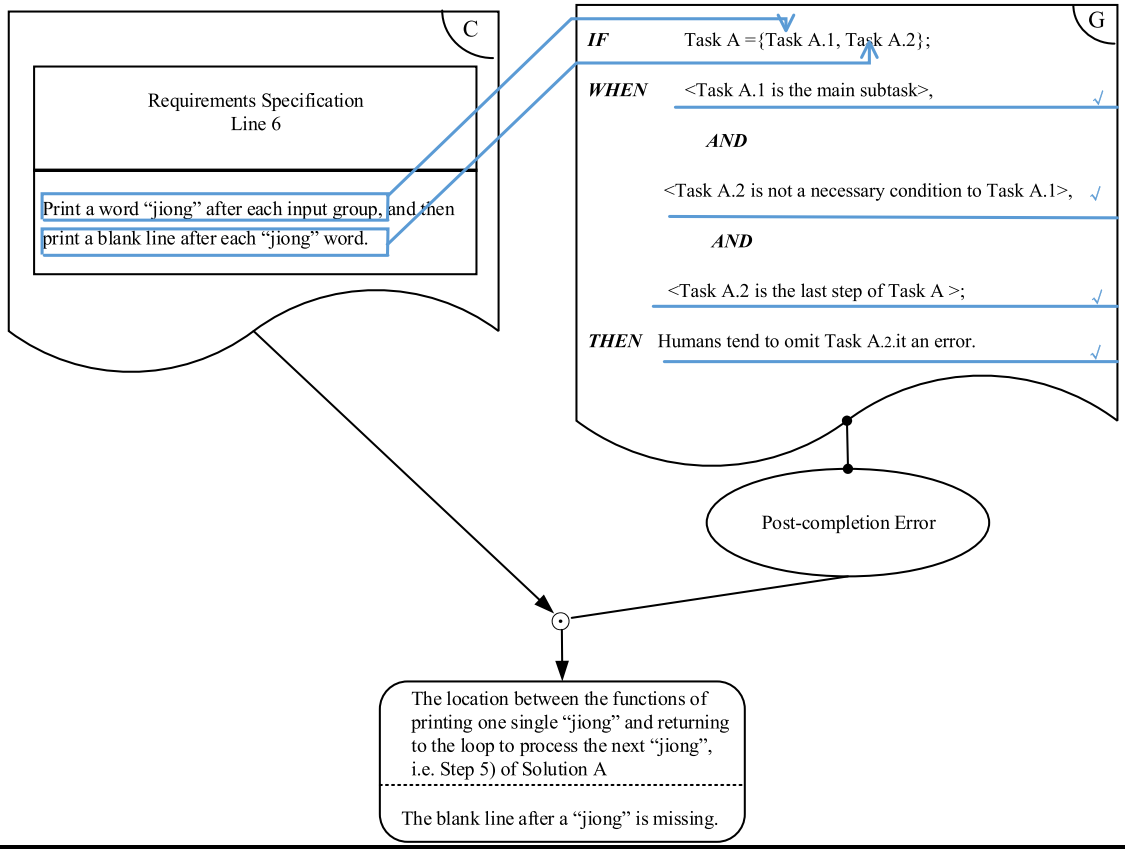
Scenario ID: *ES1*

Defect location: *The location between the functions of printing one single "jiong" and returning to the loop to process the next "jiong", i.e. Step 5) of Solution A in Table II.*

Defect form: *the blank line after a "jiong" is missing.*

Error modes: *Post-completion error*

Scenario analysis: In the process of solving the "jiong" problem, the programmers' ultimate aim (superordinate goal) is to compute and print the image of the "jiong". The task of printing a blank line after the "jiong" is a subgoal, but it is carried out at the end of the task. This is a typical scenario of post-completion error, where the subgoal is likely to be omitted, thus, programmers may forget to print the blank line after the "jiong". The graphic representation of the Error-prone Scenario Analysis is as follows:



mechanisms that are shared by all humans, we validate the approach through testing whether the forecasted defects are really introduced by programmers, in a study of multiple programmers, each independently developing software based on the same requirement.

We designed a case study in which independent analysts performed defect forecasting using HEDF. The forecast was performed solely based on a requirement specification. Afterwards, the forecasting results were compared to the real programming defects found in the multiple programs.

A. RESEARCH QUESTIONS

The study aims to explore the following four research questions (RQ):

RQ1: Can HEDF forecast the forms and locations of software defects, before code is produced?

RQ2: How effective is HEDF in forecasting software defects? This is evaluated by the data collected in the Case Study using a set of new metrics for defect early forecasting (defined in Section IV.B.2).

RQ3: To what extent can HEDF improve users' performances in defect early forecasting?Because there is no existing method for the same purpose of HEDF (that is, forecasting from requirement specifications the defects that may be later introduced by developers in code), it is inappropriate and impossible to compare HEDF with those predictions performed on code. Alternatively, a conjecture is that people could perform such early forecasting adequately using just their own educational and professional experience, even without HEDF. This also concerns the internal validity of HEDF: the proportion of defects forecast due to an analyst's own experience versus that of HEDF. Based on these

TABLE 6. Error-prone scenario analysis (2).

Scenario ID: *ES2*

Defect location: *The locations where array or variables are defined, e.g. Step 1) of Solution A in Table II.*

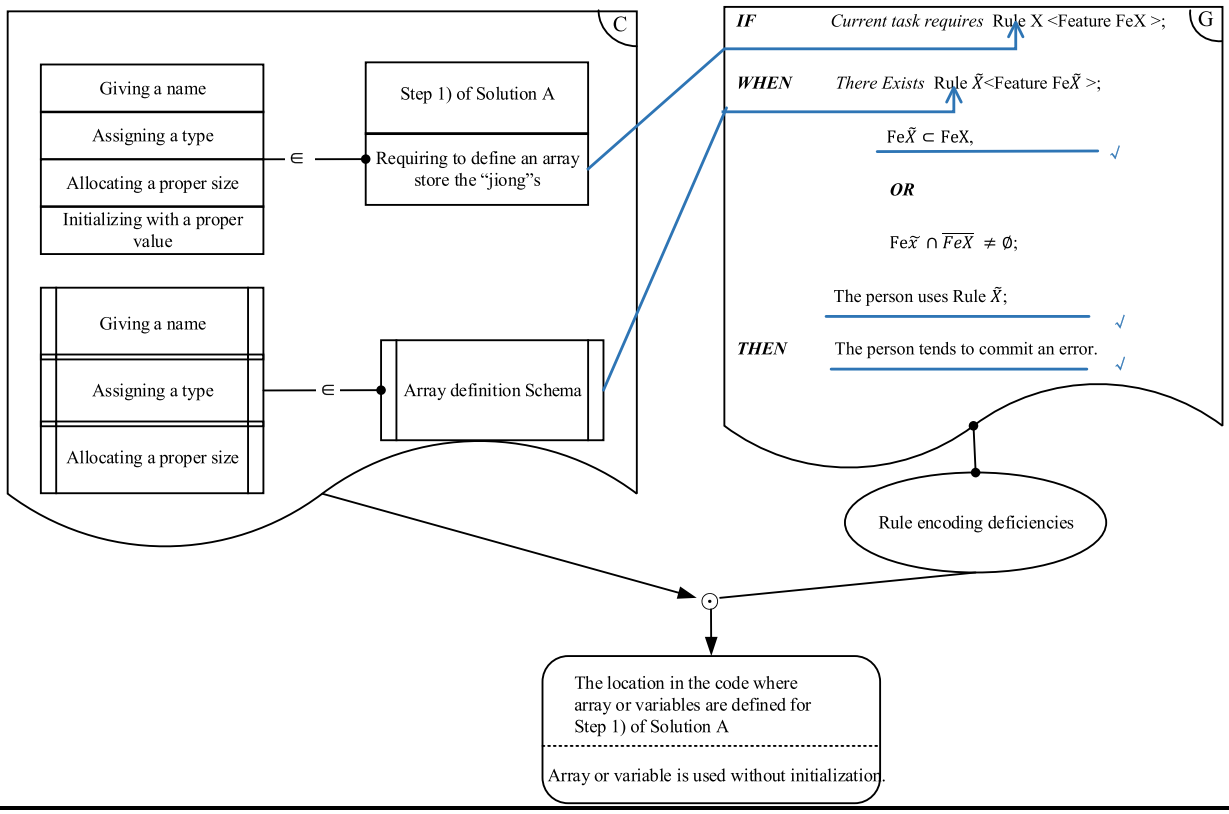
Defect form: *Array or variable is used without initialization*

Error modes: *Rule encoding deficiencies*

Scenario analysis:
Huang checked the textbook [70] (which was used as the textbook in most universities across China at the time of analysis, including the programmers involved in the exploratory study) about the contents of variables and arrays and found that the author represents the contents relevant to variable in several separate sections. The textbook first introduces the structure of a variable very simply with a small image, including *name, value and size.* Then it takes several long sections to describe different *types of variables*. After that, a section about variable assignment is presented, followed by several sections about operators.

No passage in the textbook explicitly represents the full structure of schema for array or variable definition. For the general schema of variable definition, it contains a total of four sub-rules: giving a name to the variable, defining the type of the variable, allocating the variable with a proper size, and initializing the variable. The students need to acquire these sub-rules from different places at different times, and then encode them together to form the complete schema for variable definition. Therefore, the analyst anticipates that some students may have not encoded a few sub-rules or not integrated all the sub-rules to the general rule. If the sub-rule of "initialization" has not been integrated to the general rule of variable definition, the programmer may forget initializing an array or variable when he defines it. Then, a fault is likely to be introduced in the form of "using array or variable without initialization".

The graphic representation of this Error-prone Scenario Analysis is as follows:



considerations, we recruited four analysts as the representative users of HEDF at different expertise levels: Expert, High, Intermediate and Entry expertise level. The analysts performed two rounds of defect forecasting. In the initial round, they forecast defects based on their own experience, without having been introduced to the HEDF techniques, with unlimited time. Then, a training of HEDF was performed by the first author. After that, the analysts used HEDF to forecast defects. The performance difference between these two rounds of forecasting were calculated.

RQ4: What are the potential benefits of HEDF for debugging and defect prevention? Since our forecasting is performed before the code is produced, its outputs could be used to prevent defects, thus saving programmers' efforts in finding and fixing these defects. We explore this question using the metrics a set of new metrics for defect early forecasting (defined in Section IV.B.3).

B. METRICS

Because HEDF is performed before code is produced, the metrics for evaluating HEDF are naturally different from that for conventional predictions performed on code. For instance, code-based defect prediction methods usually classify code units (e.g. modules) into two categories: defective

TABLE 7. Error-prone scenario analysis (3).

Scenario ID: *ES3*

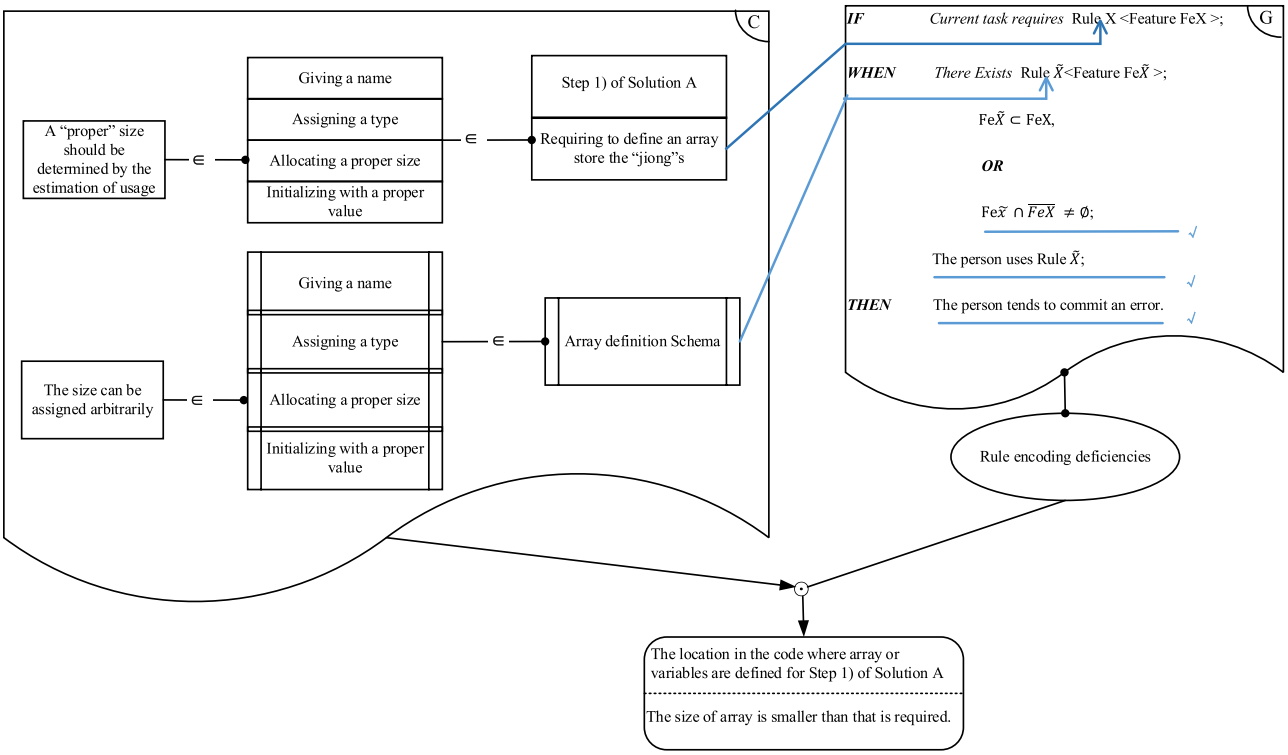
Defect location: *The locations where arrays are defined, e.g. Step 1) of Solution A in Table IV.*

Defect form: The size of array is smaller than that is required.

Error modes: *Rule encoding deficiencies*

Scenario analysis: The analysis of this scenario is similar to the scenario of ES2. Allocating a variable or array with a proper size is an important sub-rule of the general rule of variable or array definition. The size of the array is determined by the specific application requirement. If the allocated size is too large, it is a waste of memory. If the allocated size is too small, an overflow fault can occur.

Similar as the situations described in ES2: no place in the textbook explicitly presents the complete schema of variable/array definition. The sub-rule may not be encoded at all or not integrated to the general rule for some students. Thus, a fault may occur in the form that the size of array defined is smaller than that is required. The graphic representation of ES3 is as follows:



or defect-free. The performance of such predictions are commonly evaluated by a confusion matrix, which consists of four cells: true positives (TP), false positives (FP), true negatives (TN), and false negatives (FN), and derived metrics based on these four cells, such as Precision (TP/ (TP+FP)) and Matthews correlation coefficient (MCC) [71].

For HEDF, the individual event predicted concerns one possible defect form, rather than one module. All the usual measures, above, are meaningful. However, some extra notes are required about assessing them through experiment. HEDF focuses only on positives (what errors developers may commit), because HEDF is performed before code is produced. HEDF does not forecast "what error a programmer won't commit", because such a forecast has little practical implications for defect prevention. Instead, we developed metrics to estimate how much development effort could be saved if HEDF results were exploited for defect prevention.

1) METRICS FOR DEFECTS

The following metrics are proposed to measure how common a software defect is among a set of programs independently developed by separate programmers[1] for a given set of requirements or specifications.

Occurrences (OC) of a defect: the number of programmers in a study who introduced that defect in at least one version of their respective submissions.

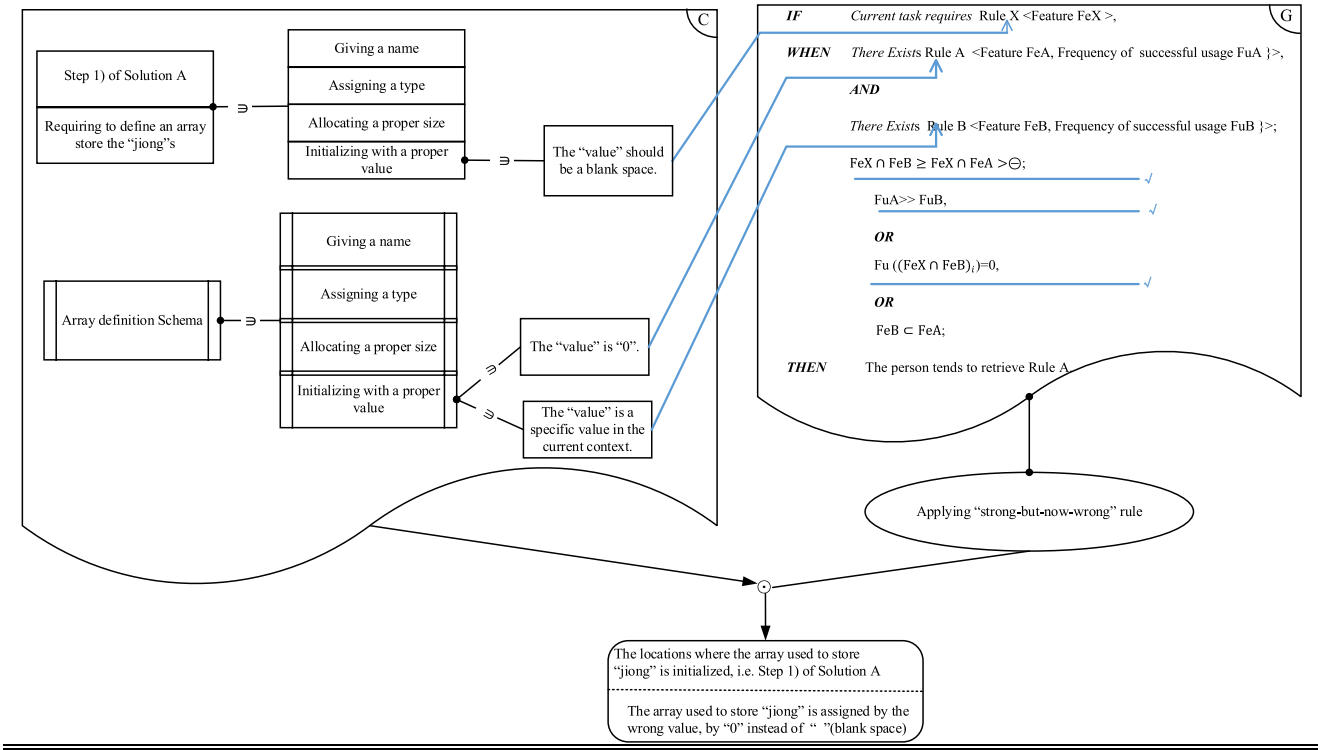
The ***Prevalence of Occurrence (POC)*** of a defect: the percentage of programmers who introduced the corresponding defect, defined as:

$$\text{POC} = \frac{OC}{P} \times 100\% \tag{1}$$

[1]Or programming teams. In the study we report, each programmer worked alone, so there were as many programs as programmers.

TABLE 8. Error-prone scenario analysis (4).

Scenario ID: *ES4*
Defect location: *The locations where the array used to store "jiong"is initialized, i.e. Step 1) of Solution A in Table IV.*
Defect form: *The array used to store "jiong" is initialized to the wrong value, by "0" instead of " "(blank space).*
Error modes: *Applying "strong-but-now-wrong" rule under the "first exception"*
Scenario analysis: In the array used to store the symbols of the image "jiong", many elements should be assigned a "blank space" value. On the other hand, the programming examples and exercises in the textbook used by the participants generally require them to initialize array contents with numbers or strings. The students have never experienced initializing a variable to blank spaces before. This is a typical scenario of "first exception". The "first exception" means that on the first occasion an individual encounters a significant exception to a general rule, particularly if that rule has repeatedly shown itself to be reliable in the past, that "strong-but-now-wrong" rule is likely to be applied. Here, the "strong-but-now-wrong" rule is: initialize the array to "0" (instead of "blank space"). The detailed graphic representation of ES4 is as follows:



where P is the total number of programmers who submitted code for the task. POC describes how common a defect is, that is, how likely it is to be introduced by different programmers. POC in an experiment is an estimator for the probability of the defect being inserted by a randomly chosen programmer from the population sampled for the experiment.

Coincident Defect: a defect whose Occurrence is two or more, i.e. that was introduced by at least two programmers.

2) METRICS FOR DEFECT EARLY FORECASTING

a: SENSITIVITY TO DEFECT FORMS (SDF)

Sensitivity to **Defect Forms (DF)** refers to the proportion of defect forms that are correctly forecast, out of all the defect forms found in the programs, given a specific piece of software requirement and a group of programmers. As defined before, a defect form is the manifestation of a human error. Essentially, SDF describes effectiveness in predicting human errors. SDF is calculated as:

$$\text{SDF} = \frac{\text{Number of correctly forecasted DF}}{\text{Number of all DF}} \times 100\% \quad (2)$$

b: TRUE POSITIVES (TP)

are those defect forms that are forecast and really introduced by at least one programmer. We call their number TP.

c: FALSE POSITIVES (FP)

are those defect forms that are forecast but not introduced by any programmer. We call their number FP. FP is also important, since these predictions would encourage effort to be spent in preventing errors that did not occur.[2]

d: PRECISION

is the proportion of forecast defect forms that actually occur:

$$\text{Precision} = \frac{TP}{TP + FP} \times 100\% \quad (3)$$

[2]We note that the "false positive predictions" observed in an experiment are not necessarily defect forms that would never occur; and predictions of defects that are rare enough not to occur in a specific study, but frequent, and severe enough when present, to entail serious expected loss, may well be valuable. Estimates of specificity (true negative rate) and, for that matter, of sensitivity(recall, hit rate, or true positive rate) in an experiment are inevitably limited by which defects do occur in the experiment. This limitation is not specific to HEDF or to the study we report.

TABLE 9. Error-prone scenario analysis (5).

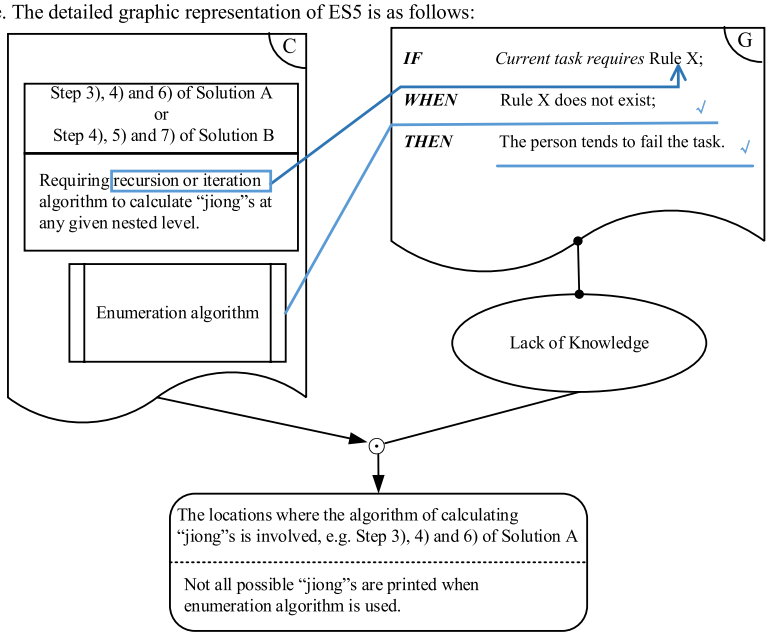
Scenario ID: *ES5*

Defect location: *The locations where the algorithm of calculating "jiong"s is involved, e.g. Step 3), 4) and 6) of Solution A in Table IV.*

Defect form: *Not all possible "jiong"s are printed when enumeration algorithm is used.*

Error modes: *Lack of knowledge*

Scenario analysis: The programming schema of either recursion or iteration is essential to solve the "jiong" problem. If one has not mastered these schemata, one can only enumerate all the "jiong" words (specify and print each character of the arrays used to store the "jiong" words) for all nesting levels. As the nesting level becomes high (e.g.7), the corresponding "jiong" word becomes so large that a person would need more time than was allowed in the contest to produce all this code. The detailed graphic representation of ES5 is as follows:



e: SENSITIVITY TO DEFECT OCCURRENCE (SDOC)

SDOC refers to the proportion of the occurrences for the forecast defects out of the occurrences for all the defects. This measure is an estimate of the expected fraction of defects that this method can forecast in the development of a single program. SDOC is calculated as:

$$\text{SDOC} = \frac{Sum(\text{OC}_{TP})}{Sum(OC_i)} \times 100\% \tag{4}$$

where OC_{TP} is the number of Occurrences for a correctly forecast defect, $Sum(\text{OC}_{TP})$ is the sum of correctly forecast defects; OC_i is the number of occurrences for a defect i, while $Sum(OC_i)$ is the sum of occurrences of all the defects.

3) METRICS FOR POTENTIAL BENEFITS FOR DEFECT PREVENTION

The value of defect early prediction is greater if it is effective on against defects that are likely, and/or that are difficult to eliminate, once accidentally created. We propose a metric to reflect these dimensions of the benefit.

a: AVERAGE PERSISTENCE OF FORECAST DEFECTS (APFD)

APFD represents how likely the defects that were forecast are to persist through successive versions of a program, once they have occurred. APFD is proposed to estimate how difficult the forecast defects are for programmers to debug, once the defects are introduced into programs. APFD is calculated as:

$$\text{APFD} = \frac{\sum_{i=1}^{I} DP_i}{I} \tag{5}$$

where DP_i is the Degree of Persistence (DP) of the defect i, and I is the total number of defects forecast by HEDF. DP_i is in turn estimated by (6).

$$DP_i = \sum_{n=1}^{N_i} \frac{VR_{n,i}}{V_{n,i}} / N_i \tag{6}$$

where $V_{n,i}$ is the total number of versions submitted by programmer $n$who introduced the defect i; $VR_{n,i}$ is the number of these versions in which the defect i is still present; N_i is the total number of programmers who introduced defect i. The fraction $\frac{VR_{n,i}}{V_{n,i}}$ describes the extent to which the defect i tended to remain in the versions generated by programmer n, named the ***Persistence*** of defect i for programmer n ($P_{n,i}$).

b: AVERAGE SAVING OF DEBUGGING EFFORT (ASDE)

ASDE is proposed for estimating how much debugging effort HEDF can save for one programmer. ASDE is

TABLE 10. Error-prone scenario analysis (6).

Scenario ID: *ES6*

Defect location: *The places where the inputs and outputs are addressed, e.g. Step 2) and 6) of Solution A in Table IV.*

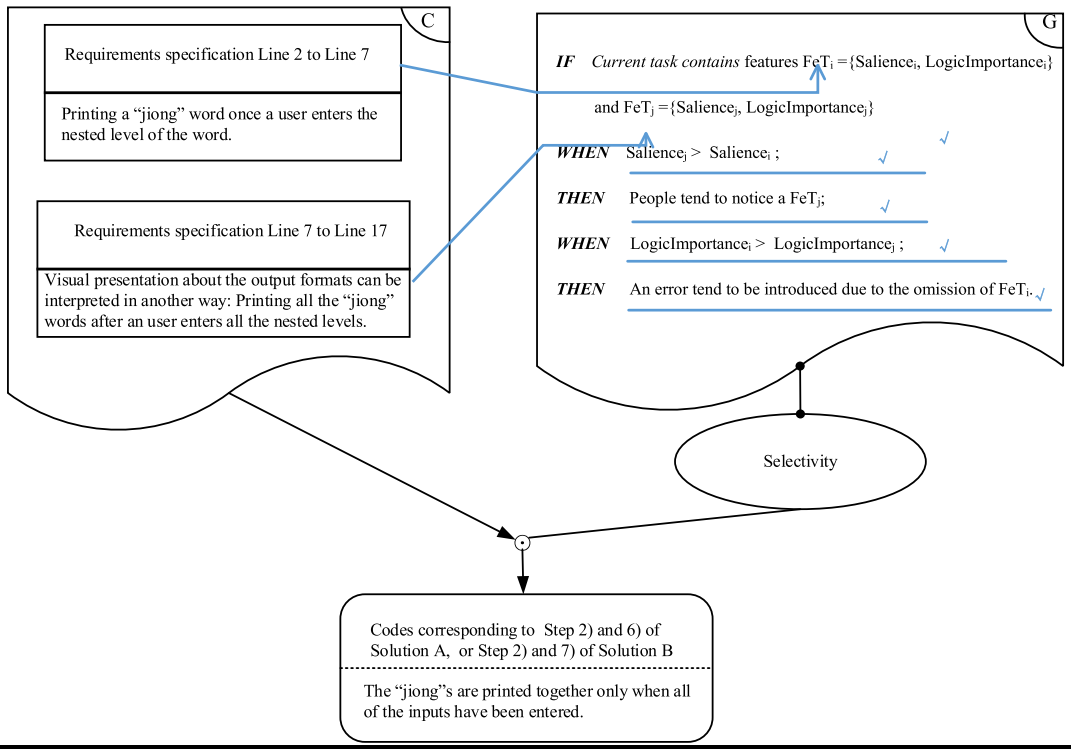
Defect form: *The "jiong"s are printed together only after all of the inputs have been entered.*

Error modes: *Selectivity*

Scenario analysis: There are two passages in the requirement specification that contain information on the formats of inputs and outputs. The first passage is located under the bullets "**Inputs**" and "**Outputs**". Under the bullet "Input", the format of input is specified as "there is an integer in the first line, which indicates the number of input groups. Each input group contains an integer n (1≤n≤7)." Under the bullet "Output", the format of output is specified as "print out a word 'jiong' after each input group, and then print out a blank line."

The other passage relevant to the formats of inputs and outputs is located under "**Sample inputs**" and "**Sample outputs**". Using "Sample inputs" and "Sample outputs" is a conventional part of specification in the programming contest, widely used across different tasks and different annual contests. The "Sample inputs" and "Sample outputs" here are static: they cannot reveal the required sequencing of inputs and outputs. However, if the participants only notice information under the "Sample inputs" and "Sample outputs", they may interpret the specification in two ways: print all the "jiong" words together after reading in all the inputs; print each "jiong" word once an integer indicating the nesting level of the corresponding "jiong" word is input.

This typical scenario is prone to trigger the error mode of "Selectivity". For the same task content, pieces of information are scattered at different places. Some pieces of information are redundant or incomplete. Under such circumstances, it is hard for a person to collect all the information, exclude redundant information and form an accurate mental representation. As a result, one may selectively notice, out of the whole body of information, only a fraction that is psychological salient, and thus commit errors. Thus, some programmers may just notice the information of "Sample inputs" and "Sample outputs" as they are visually presented, and visual representations are generally more likely to capture people's attention, and thus, wrongly interpret the specification as "print all the 'jiong's together after reading all of the inputs." The detailed graphic representation of ES6 is as follows:



calculated as:

$$ASDE = \sum_n SDE_n / N \tag{7}$$

where SDE_n is the Saving of Debugging Effort for programmer n, and N is the total number of programmers who introduced one or more defects. SDE_n is in turn estimated by Formula (8):

$$SDE_n = \frac{V'_n}{V_n} \times 100\% = \frac{\sum_k V'_{n,k}}{V_n} \times 100\% \tag{8}$$

where V_n is the total number of successive versions submitted by programmer n; V'_n is the number of versions that would not have been needed (the effort saved) if the HEDF forecasting results had been provided and used to avoid the defect. $V'_{n,k}$ is the number of iterations that directly precede the fixing of a predicted defect k. We counted V'_n in a conservative manner: when one or more predicted defects are fixed concurrently with an unpredicted defect in a version, all those iterations that directly precede this version are not

TABLE 11. Error-prone scenario analysis (7).

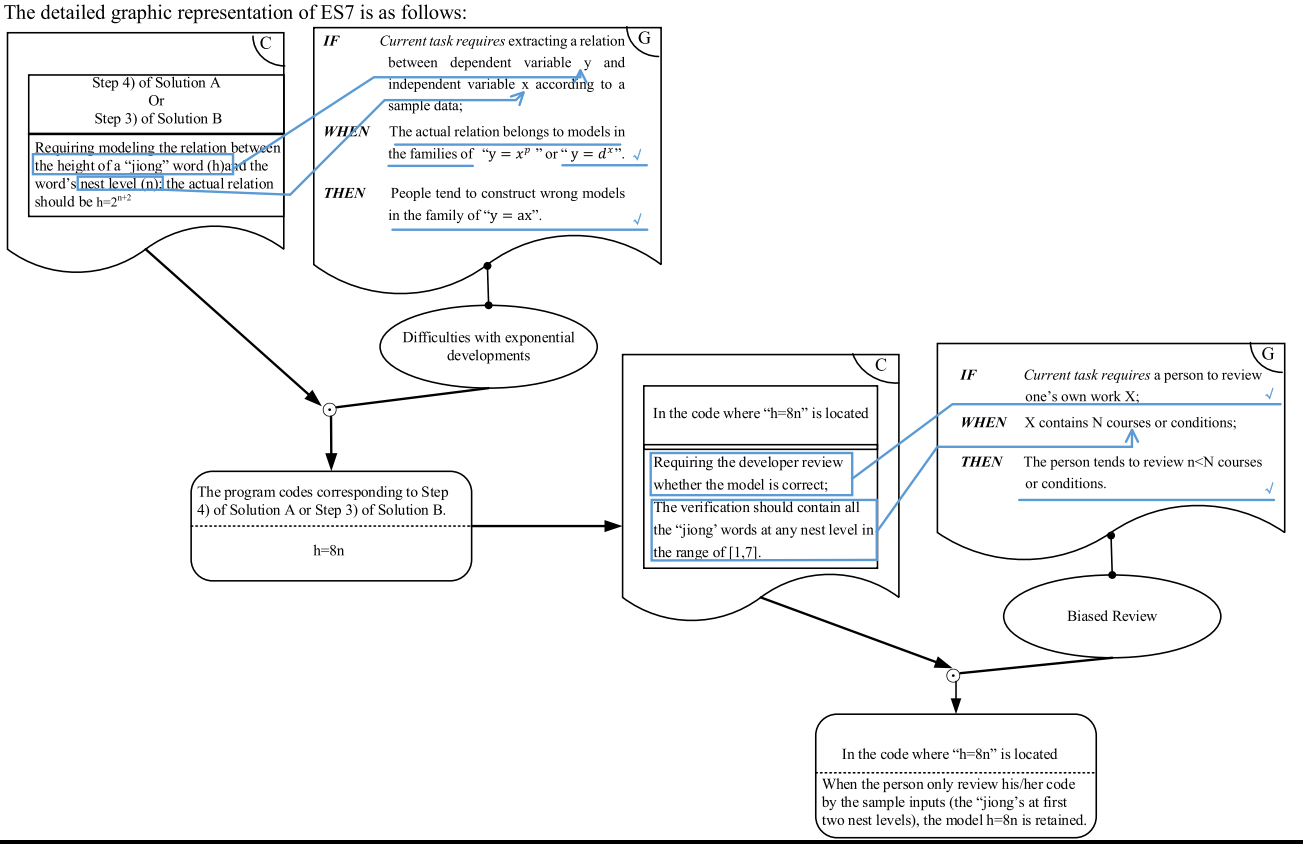
Scenario ID: ES7

Defect location: *In the algorithm where it requires modeling the relation between the height of a "jiong" and the "jiong"'s nesting level, i.e. Step 4) of Solution A or Step 3) of Solution B in Table IV.*

Defect form: *The relationship between the height and nesting level is modeled wrongly as h=8n, instead of* $h=2^{n+2}$

Error modes: *Difficulties with exponential developments and biased review*

Scenario analysis :
In the "jiong" task, programmers need to model the relationship between the height (h) and nesting level of the "jiong" (n). The right model should be $h=2^{n+2}$. This is an exponential model which must be built if a subject wants to solve the "jiong" problem successfully. This is a typical scenario for the error mode "difficulty with exponential developments" described in Table 1 [54].
Meanwhile, in problem solving, people always evaluate their solutions after constructing an initial solution [9]. When evaluating solutions, humans tend to "biased review" (the error mode described in the third row in Table 1): believing that all possible cases have been checked while in fact very few have been considered.
Notice that there are three groups of sample inputs (nesting level n=1, 2, 3, respectively). The tendency to "biased review" of solution implies that some people may choose a subset of the sample inputs (instead of all the three sample inputs) to check their solutions. But for n=1 or n=2, there is a linear model h=8n that produces the same value as the exponential model $h=2^{n+2}$. Therefore, the wrong model may escape one's self review if one uses only the first two sample inputs to test his solution. Based on these scenarios, the analyst anticipates that some programmers may finally model the relationship between the height and nesting level wrongly as h=8n, instead of $h=2^{n+2}$.
The detailed graphic representation of ES7 is as follows:



counted in the estimate of saving, V'_n. In the special case where a programmer submitted only 1 version with defects, we assume the programmer's debugging effort are all contained in that version: 1) if the defects in this version are all predicted, we assume the Saving of Debugging Effort is 1; 2) if this version contains any unpredicted defect, we assume the saving is 0.

For instance, a programmer submitted 6 versions in total, and his final version is correct. Our code inspector found that 3 defects (F2, F6, and F15) were present and were removed across these 6 versions. The programmer's debugging history was recorded as the sequence "N F15 **F2** N N **F6**" shown in Figure 4, where N denotes a version in which no defect (present in the previous version) is fixed, while a defect ID (e.g. F15 in Figure 4) indicates this defect is fixed in the corresponding version (e.g. F15 is fixed in the 2nd version). F2 and F6 were predicted defects while F15 was unpredicted, therefore, only the 4th and 5th version which directly preceded F6 were counted as saved debugging effort–V'_n.

c: IMPROVEMENT ON ACCEPTANCE RATE (IAR)

Acceptance Rate (AR)is defined as the percentage of programmers who got their submissions accepted in the programming contest (meaning that their programs were

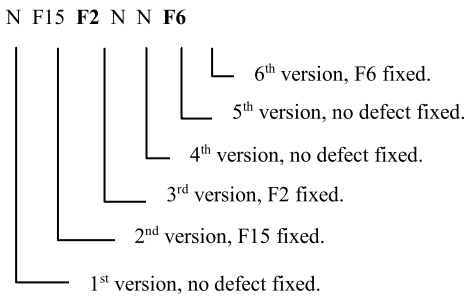


FIGURE 4. An example programmer's debugging history.

TABLE 12. Statistics for the Complexity metrics of the "jiong" programs.

Program Metrics	Maximum	Minimum	Mean	Std. deviation
Lines of Code	706	64	208	134.9
Cyclomatic complexity	69	5	20.5	15.1
Halsteads volume	4021	290	1587.9	766.9
Nesting depth	5	1	2.9	0.8

judged correct, as described in Section 5) among the total number of programmers (P) who submitted code on the task.

Improvement on Acceptance Rate concerns how many more programmers would produce correct programs if actionable preventions were derivable from the defects forecast via HEDF. IAR in this study is estimated in (9):

$$\text{IAR}=\frac{AC'}{P}\times 100\% \tag{9}$$

where AC' is the number of programmers who failed to get their submissions accepted in the programming contest (that is, their programs were incorrect), but would have their programs accepted if HEDF forecasts guided effective prevention actions. A program is counted towards the total AC' only if *all of the defects* introduced by a programmer are covered by the set of HEDF forecast defects *AND* all of the corresponding forecasts have *actionable meanings* for defect prevention. For instance, suppose that a programmer introduced two defects and failed to get his submissions accepted, and one of his defects is in the set of HEDF forecast defects; then, this programmer does not count towards AC'. For another example, suppose that a programmer had one defect in his/her submissions and failed to get the submissions accepted; this defect is in the set of HEDF forecast defects; but the forecast does not imply actionable strategies (e.g. due to lack of knowledge for F17 in Table 12), and thus does not lead to preventing the defect; this programmer does not count towards AC' either.

The**Improvement Ratio for Acceptance Rate (IRAR)** is calculated in (10):

$$\text{IRAR}=\frac{IAR}{AR}\times 100\% \tag{10}$$

V. THE PROGRAMMING DATA

A. THE TASK AND ITS SUITABILITY

As the first fundamental step to establish and test a new theory that involves SE and psychology, it is important to select a suitable task that is representative for programming performances and at a controllable scale. Because our research aims to test a new causal theory, we believe it is more suitable to test it in a controlled-experiment fashion, rather than field observations in industrial settings that have many confounding factors threatening to the internal validity.

We selected the "jiong" problem to conduct the case study, due to two reasons. The first reason is that the first author had done an exploratory study on this task, and it was intriguing for us to see what performance in forecasting other users of HEDF would achieve. The second reason is that the programming data of the "jiong" problem were already made public in another experiment examining the correlations between personality traits, cognitive styles and the number of defects a programmer introduced [58]. The defects were collected by an independent code inspector Reusing these data would minimize the researchers' bias, and allow any interested readers to do replicated trials and compare their forecasting results with the results presented in this paper.

We consider the "jiong" task comparable to a piece of requirement for a module or a story in agile development. According to Rasmussen's classification of "performance levels" (skill-based, rule-based and knowledge-based level) [66], which is widely accepted in human error studies [54], the "jiong" task contains sub-tasks at all the three performance levels: skill-based, rule-based and knowledge-based levels.

Therefore, before the case study, we did a rough estimation by offering the "jiong" task to a group of 15 professional software engineers. Six of them (40%) submitted within 2 hours a program that was complete enough to run, while 9 of them did not finish it. Overall, these professional engineers performed no better than the programming contestants did on the "Jiong" task. Among the 55 contestants in our study, 44 submitted a correct version, with an acceptance rate 72.7%. This is just a rough estimation, nevertheless, the estimation corroborates for this special case the well-established theory in programming cognition: an "expert" perform no better than "novice" if a task is completely new to the expert and he has no pre-existing knowledge and skills to migrate to this task [58], [72]. To complete the "jiong" task, one needs to "observe the structure features of "jiong" words and extract the mathematical model that describes the relationships between width, height and nesting level". This is a new feature of the task that requires "knowledge-based" performances with considerable mental effort for anyone who had never been exposed to the "jiong" task, no matter whether he/she has 10 years' industrial experience or is an undergraduate student.

In summary, for studying human error mechanisms in controlled experiments, the criteria for selecting a suitable

programming task are not simply "large" vs "small", nor "experts" vs "novices". Instead, we consider whether the task contains functional features that cover all of Rasmussen's performance levels for the participants, because different levels of performance determine the different mechanisms of human errors [54]. The "jiong" task met these criteria, making it suitable to the research purposes of the case study.

B. THE PARTICIPANTS

The participants were 55 undergraduate students in Computer Science who submitted programs for the "jiong" requirement in a context of a programming contest.

C. THE DATA COLLECTION PROCESS

The programming data was collected through a programming contest that was similar to the ACM International Collegiate Programming Contest (ACM-ICPC). The contest was held in the form of on-site testing in a computer room. Each contestant was assigned one computer.

There were strict precautions against cheating: 1) each room was monitored by two supervisors to prevent the contestants from copying others' code; 2) the programming environment was cut off from the external internet to prevent the contestants from learning or copying code from the internet. With such precautions, and in a competitive contest where the contestants were highly motivated to win, we believe that the contestants worked independently; any similarity between errors by different contestants is unlikely to be due to copying or collaborating.

The contest scores were announced in real time by the Online Judge System, which was similar to the system used in the ACM-ICPC. Each contestant can first compile and run the program in his/her local environment, then submit it to the Online Judge System on a server. For each problem, contestants can submit to the Online Judge System as many versions of programs as they wish, until the system "accepts" one version or the contestant quits. After each submission, the Online Judge System fed back to the contestants these types of results:

Accepted (AC). The output of the program matches what the Online Judge expects.

Wrong Answer (WA). The output of the program does not match what the Online Judge expects.

Presentation Error (PE). The program produces correct output matching the Online Judge's secret data, but does not produce it in the correct format.

Runtime Error (RE). This error indicates that the program performs an illegal operation when running on the Online Judge's input. Some illegal operations include invalid memory references such as access outside an array boundary. There are also a number of common mathematical errors such as "divide by zero" error or "overflow".

Time Limit Exceeded (TL). The Online Judge has a specified time limit for every problem. When the program does not terminate within that time limit, this error will be generated.

Compile Error (CE). The program does not compile with the specified language's compiler.

The programming contest encouraged the contestants to compile, debug and fix the defects on their computers before submitting to the Online Judging System by a rule on penalty time: each unaccepted submission produced 20 minutes' penalty time. The penalty time affects the rank list: if two contestants solved the same number of problems, the one with less penalty time ranks higher on the list.

A software engineer independent of this paper performed code inspection to identify the defects introduced by the contestants on the "jiong" task. As the contestants were allowed to debug and re-submit their programs, each person can submit more than one version. For each contestant, the code inspector was asked to review all the versions he/she submitted, and record all of the defects they contained. This is because we were concerned with the error-proneness of programming activities; hence, we were interested in all of the errors a contestant made during the entire process of solving the "jiong" problem. However, *for each contestant, each defect was counted only once* in the statistics that follow, even if it appeared in several versions.

D. THE ACTUAL PROGRAMMING DATA

A total of 55 programmers submitted a total of 192 programs for the "jiong" problem, because some programmers submitted more than one version due to defects. The complexity metrics of the programs are shown in Table 12.

The defects found in the code inspection are summarized in the first four columns of Table 13. The code inspector found that 22 defect forms (described in the second column in Table 13) constituted *the set of all defects forms* in the case studies. Among the 22 defect forms, 9 were *Coincident Defects*.

The *Occurrences (OC)* of a defect (shown in the third column of Table 13) is the number of programmers who introduced that defect in at least one version of their respective submissions. The sum of the *Occurrences* for *the set of all defects* (sum of the third column of Table 13) is **70**. The sum of the *Occurrences* for coincident defects is **57.**

The *Prevalence of Occurrence (POC)* of a defect (in the fourth column of Table 13 is the percentage of participants who introduced the corresponding defect. For instance, defect F2 "the blank line after the 'jiong' is missing" was introduced by 23 programmers, that is, 42.1% of the total 55 contestants.

VI. THE CASE STUDY

The study aimed to explore whether people at various expertise levels in software quality assurance and CS could forecast

TABLE 13. The actual defects (1st - 4th COLUMN) and defect forecasting data (5th-14th COLUMN).

Defect Form ID	Description	OC	POC	Analyst #1		Analyst #2		Analyst #3		Analyst #4	
				PE	HEDF	PE	HEDF	PE	HEDF	PE	HEDF
F1	The size of an array is smaller than that is required (e.g. 100x100), where 512x512 is needed.	7	12.7%	-	-	-	Yes	-	-	-	-
F2	The blank line after the "jiong" is missing	23	41.2%	Yes	Yes	-	Yes	-	Yes	-	-
F3	Mistaking the symbol "!" for "|"when printing the "jiong"	1	1.8%	-	-	-	-	-	-	-	-
F4	The symbol '+'in the last line of "jiong" is missing	1	1.8%	-	-	-	-	-	-	-	-
F5	A number indicating the height of the word "jiong" was printed out by mistake.	1	1.8%	-	-	-	-	-	-	-	-
F6	Array or variable is used without being initialized.	14	25.5%	Yes	Yes	-	-	-	Yes	-	-
F7	Array is initialized to the wrong value, by "0" instead of " "(blank space).	3	5.5%	Yes	Yes	-	Yes	-	-	-	Yes
F8	A variable "n" is used as the upper limit in the "for" loop, without initialization. The "n" may have a large value, which causes the iteration time to exceed the allowed ceiling.	1	1.8%	-	-	-	-	-	-	-	-
F9	The array used to store the "jiong" is initialized by a 2-depth loop, leading to a problem that the program execution time exceeding the time limit of 1000ms.	2	3.6%	-	-	-	-	-	-	-	-
F10	The function "pow ()" (a C Standard Library function computing the power of a number) is used many times in "for" loop and they are called in every iteration, resulting the program running time exceeding the allowed time limit.	2	3.6%	-	-	-	-	-	-	-	-
F11	Using "n++" instead of "n--" in "for" loop.	1	1.8%	-	-	-	-	-	-	-	-
F12	The program is in c++, but the file name is ended ".c".	1	1.8%	-	-	-	-	-	-	-	-
F13	Array referencing mistake: char c[1000][1000]='0', where the programmer has unintentionally initialized the first value of the array to be "0".	1	1.8%	-	-	-	-	-	-	-	-
F14	The "memset" function is misused.	1	1.8%	-	-	-	-	-	-	-	-
F15	Mistaking the "\" for "\\".	1	1.8%	-	-	-	-	-	-	-	-
F16	"y<=m && m<y+b/2" is expressed as "y<=m<y+b/2" by mistake.	1	1.8%	-	-	-	-	-	-	-	-
F17	Enumerate the "jiong" words one by one, but the code is incomplete for printing all of the "jiong" words from nesting level 1 to 7.	2	3.6%	-	Yes	-	-	-	-	-	Yes
F18	Misunderstanding the requirement: printing out all the "jiong"s after all the inputs are entered by the user, while the requirement specifies that the program should print each "jiong" after the nest-level is entered by the user.	2	3.6%	-	Yes	-	-	-	-	-	-
F19	Mistaking the nesting level iteration "n=n/2" for "n=n-1" in "for" loop.	1	1.8%	-	-	-	-	-	-	-	-
F20	The location for the symbol "\" is wrong, mistaking f4 [32-i][65-i]='\\' for f4[32-i][63-i]='\\' .	1	1.8%	-	-	-	-	-	-	-	-
F21	The relationship between the height and nest level of the "jiong" is deduced wrongly as h=8n, which is supposed to be $h=2^{n+2}$.	2	3.6%	-	Yes	-	Yes	-	Yes	-	Yes
F22	Slips in indexing array elements, mistaking map[l][1] for map[l][i].	1	1.8%	-	-	-	-	-	-	-	-

[a] – means the defect was not forecast. [b] Yes means the defect was forecast.
OC: Occurrence
POC: Prevalence of Occurrence
PE: forecasted in the first round based on personal experience (PE)
HEDF: forecasted in the second round with HEDF

TABLE 14. The background and experience of the independent analysts.

Analyst #	Expertise level	Occupation (s)	Highest Degree	Years in Higher edu.	Years in SD	Years in SQA
1	Expert	Professor & Industrial Consultant	PhD in CS	11	10	20
2	High	Young Researcher	Ph.D. in CS	11	4	0
3	Intermediate	Graduate student	Master in CS	7	4	0
4	Entry	Graduate student	Bachelor in CS	4	4	0.5

the defects without using HEDF, and how many they could forecast without HEDF.

Four participants (Analysts #1, #2, #3, #4) independent of this paper served as the users of HEDF. Since using HEDF mainly involves Error-prone Scenario analysis, we call the users of HEDF "analysts" in the remaining of paper. The first author served as the trainer of HEDF.

A. THE ANALYSTS

The information on the backgrounds of the analysts were collected at the beginning of the study, using a survey shown in Part I of Appendix A. The backgrounds and experience of the analysts are shown in Table 14.

We consider analysts 1, 2, 3 and 4 are representative at different expertise levels: expert, high, mediate and entry-level, respectively. The analysts are independent of this study and had never been exposed to the "jiong" problem nor knowledge of human errors in psychology.

B. THE PROCESS

The study was conducted in the following process:

Step 1. The "jiong" requirement was provided to the analysts. The analysts were asked to forecast, based on their personal experience, the possible defect forms in the programs written by first-grade undergraduates who just completed their C language course. Step 1 was fulfilled by filling Survey 1 in Appendix A. There was no time limit, but the analysts were asked to try their best to predict as many defects as possible.

Step 2. The first author of this paper provided the analysts with a training session through Zoom Cloud Meetings. The training session lasted for 75 minutes. The training was given by presentation, focusing on Human Error Mechanisms described in Section III.B and the defect forecasting process described in Section III.C. No example of defect nor human error concerning the "Jiong" problem was given.

Step 3. The analysts were asked to forecast defects using HEDF. The time limit for the prediction is 45 minutes. The analysts were asked to only record the unique defects newly forecast by HEDF. They were encouraged not to

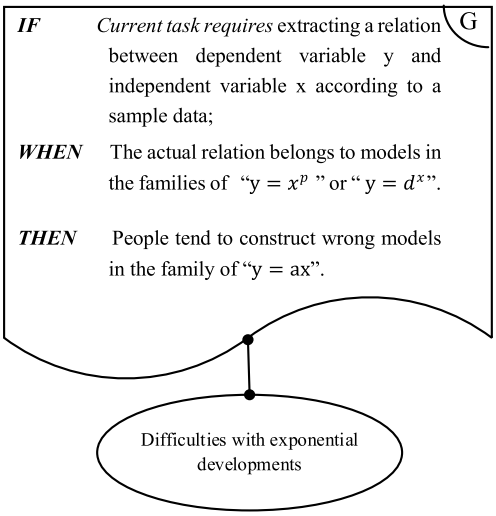


FIGURE 5. An example HEM and EPS for HEDF forecast.

①Misunderstanding on the second input parameter, which may lead to undifferentiated sizes of "jiong" words, plus, lack of design for interactions with the users.
②Lack of rationale assessment on the input parameters, including:

- The range of integers
- The constraint relationship between two parameters

③ Forgetting the blank line
④Forgetting minus 1 or the equivalent symbol in the first input parameter of the outer nesting level or the loop condition statement.
⑤Undifferentiated symbols in the printing statements.
⑥Arrays are not initialized by "void"
⑦Variables are not initialized.

FIGURE 6. An example results (Analyst #1) of defect forecasting based on personal experience.

repeat those defects which had already been predicted in Step 1 using personal experience, unless one achieved substantial improvement in forecasting accuracy with respect to defect forms and/or locations. Seven sheets printed with the seven diagrams were distributed to the analysts, with an example shown in Figure 5. Each sheet contained a Human Error Mode and EPS in Table 1, represented by the notations described in Table 2. These sheets were provided to the analysts because the descriptions on HEM and EPS can be stored in a database, and notations for EPSA can also be implemented as a tool in the near future, serving like a special "dictionary". The analysts would not need to remember every detail of these diagrams in a limited time, they can check and review the "dictionary" anytime they needed. However, we encouraged the analysts to use the sheets to forecast and record defects. In any case when they did not want to draw diagrams, they were asked to refer to the Human Error Modes and EPS during the forecasting session, and describe how they forecast the defects.

C. THE DATA OF DEFECT FORECASTING

Each analyst produced two sets of results of defect forecasting: the defects forecast based on Personal Experience (PE), and the defects forecast using HEDF.

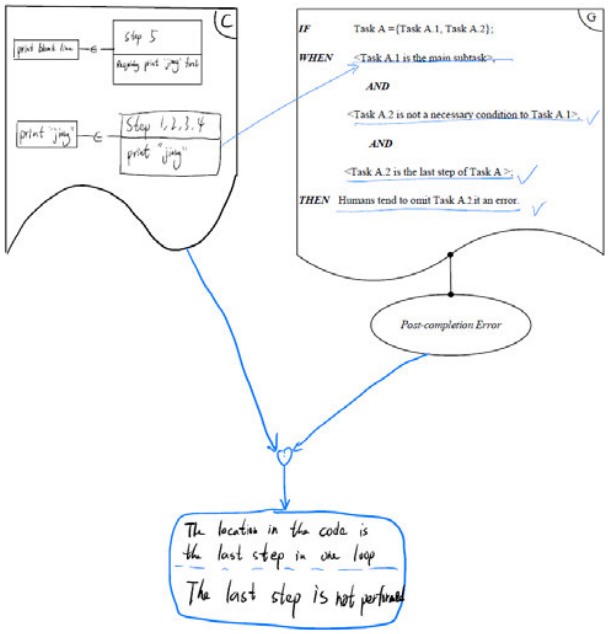


FIGURE 7. An example result (Analyst #3) of defect forecasting using HEDF.

TABLE 15. The defect forms forecast by all the analysts using HEDF.

Defect Form ID	Type of defects (activities in which the defect was introduced)	Analysts forecasted this defect using HEDF	Human Error Mode (s) contributed to the forecasting
F1	Coding	#2	Rule encoding deficiencies
F2	Requirement understanding	#1, #2, #3	Post-completion Error
F6	Coding	#1, #3	Rule encoding deficiencies
F7	Coding	#1, #2, #4	Applying "strong-but-now-wrong" rule
F17	Design	#1, #4	Lack of knowledge
F18	Requirement understanding	#1	Selectivity
F21	Design	#1, #2, #2, #4	Difficulties with exponential developments

Among all the analysts, Analyst #1 provided the best forecasting results; the original outputs and a translated version (by the first author) are shown in Figure 6. Analyst #1 is a professor and an expert consultant with 20 years of experience in industrial software quality assurance and has reviewed over 8000 real defects introduced by other people according to his responses to the survey. An example of HEDF forecasting results is shown in Figure 7.

The defects that were forecast in the first round (without HEDF) ***and*** really occurred (**$N_{PE\&OC}$**) are detailed in the 5th, 7th, 9th, and 11th columns, labeled "PE", of Table 13 The defects forecast by HEDF ***and*** really occurred in one or more programmers' submission (s) (**$N_{HEDF\&OC}$**) for the Case Study are shown in the 6th, 8th, 10th and 12th columns, labeled "HEDF", of Table 13. The types of defects and the human error modes contributed to both rounds of forecasting by all the analysts are summarized in Table 15.

TABLE 16. The summary of defect early forecasting data.

Analyst#	N_{PE}	$N_{PE\&OC}$	N_{HEDF}	$N_{HEDF\&OC}$	$N_{HEDF-PE}$	$N_{HEDF\&OC-PE\&OC}$
1	9	**3**	9	**6**	3	**3**
2	8	**0**	4	**4**	4	**4**
3	6	**0**	4	**3**	4	**3**
4	5	**0**	4	**3**	4	**3**

N_{PE}--Number of defects forecast by personal experience.
$N_{PE\&OC}$--Number of defects forecast by personal experience ***and*** actually occurring in one or more programmers' submission (s).
N_{HEDF}—Number of defects forecast by using HEDF.
$N_{HEDF\&OC}$-- Number of defects forecast by HEDF ***and*** actually occurring in one or more programmers' submission (s).
$N_{HEDF-PE}$-- Number of defects forecast by using HEDF ***but not*** forecast by personal experience.
$N_{HEDF\&OC-PE\&OC}$-- Number of defects forecast by using HEDF ***but not*** forecast by personal experience, ***and*** actually occurring in one or more programmers' submission (s).

The summary of the defect early forecasting data is shown in Table 16. The results of the defect forecasting metrics of the Application Case Study are shown in Table 17.

VII. RESULT ANALYSIS

This section answers the research questions based on an integrated analysis of the results obtained in the case study.

RQ1: CAN HEDF FORECAST THE FORMS AND LOCATIONS OF SOFTWARE DEFECTS, BEFORE CODE IS PRODUCED?

Yes. By using HEDF, all the analysts involved in Case Study successfully forecasted the exact locations and forms of defects solely based on requirement, without access to any code. In comparison, without the guidance of HEDF, only the professor who had 20 years of experience in industrial consulting on software quality assurance forecasted a small proportion of true positives; all the other analysts at high, intermediate and entry expertise levels did not forecast any true positives. This suggests that at requirement stage forecasting the exact forms of defects developers may introduce to code is extremely challenging (yet significantly beneficial, when such forecasts allow defect prevention), it could hardly be achieved solely based on one's experience without specialized knowledge and guidance provided in HEDF.

HEDF seems very beneficial for analysts at various levels of experience in computer science (high, intermediate and entry levels). Analyst #2 (PhD in CS), 3 (Master in CS) and 4 (Bachelor in CS) have considerable experience in software development (4 years) but little experience in software quality assurance. They all made more than 5 forecasts based on their previous experiences; however, none of these defects were really introduced by programmers (Precision 0). Using HEDF, they were able to forecast 3-4 true positives, and reached very high precision (66.7%-100% of their forecasts were defects that did occur). On average, HEDF enabled

them to forecast 41.7% ((4+3+3)/8) coincident defects, and captured an average of 36.2% defect occurrences. These are noticeable improvements compared to 0 true positive based on personal experience.

HEDF also seems very helpful for experts in software quality assurance in improving their accuracy and precision of defect forecasting. Analyst #1, despite vast experience (as a professor in a research university with over 20 years' consulting experience in software quality assurance), had 3 true positives in the first-round forecasting based on his experience, with a precision of 33.3%. In the round using HEDF, he forecast 3 new true positives, and detailed the forms for the 3 true positives forecast in the first round. For instance, based on experience, he successfully pointed out that someone may forget the blank line, but he found it difficult to explain how he reached this forecast. Later using HEDF, he drew a perfect diagram (almost the same to the one in Table 5) detailed how the contexts of "jiong" requirement matches to the "post-completion error" error-prone scenario, and (in his own description) "reached the 'aha' moment that transformed his intuitive judgment into a structural reasoning on why and how he reached this forecast". Excluding those forecasts with improved accuracy, only counting new true positives, HEDF has improved analyst 1's performances by 100% in True Positives, Precision, and Sensitivity to Defect forms. With the help of HEDF, he was able to forecast 75% (6/8) coincident defects, and 65.7% occurrences of all the defects.

RQ2: HOW EFFECTIVE IS HEDF IN FORECASTING SOFTWARE DEFECTS?

Among the 22 total defect forms present in the code delivered, 7 (31.8%) were forecast by HEDF. We found it impressive that by using information about human error mechanisms the analysts were able to predict the accurate forms of these seven defects just on the basis of the requirement specification and design analysis. This seems a much more valuable result than can be achieved by other approaches such as program metric-based models.

An interesting finding is that HEDF was especially effective in forecasting the high-probability defects. The 31.8% forecast defect forms accounted for 75.7% of the total defect occurrences introduced by all the programmers. The Occurrence suggests how common the predicted defects are, that is, by implication, how useful it is to take measures to prevent them, both for standard software development and for *multiple-version* development, as used for some critical applications [73], where avoiding common defects is the main concern.

Another interesting finding is that these 31.8% defect forms constitute 93.0% of the Occurrences of Coincident defects. This suggests that HEDF has significantly captured human error mechanisms widely shared among the participant programmers.

In summary (Table 18), HEDF has reached an average Precision 79.2% (Max 100%, Min 66.7%, SD 14.4%), an average Sensitivity to Defect Forms 18.2% (Max 27.3%, Min 13.6%, SD 6.4%), average Sensitivity to Defect Occurrence 43.6% (Max 65.7%, Min 10%, SD 24.3%). These are impressive achievements, in comparison to those predictions without the knowledge and application of HEDF, further detailed in RQ3.

TABLE 17. The results of the defect forecasting metrics.

Metrics	Analyst 1		Analyst 2		Analyst 3		Analyst 4	
	PE[a]	HE DF[b]	PE	HE DF	PE	HE DF	PE	HE DF
TP	3	6	0	4	0	3	0	3
FP	6	3	8	0	6	1	5	1
Precision (%)	33.3	66.7	0	100	0	75.0	0	75.0
SDF (%)	13.6	27.3	0	18.2	0	13.6	0	13.6
SDOC (%)	57	65.7	0	42.9	0	55.7	0	10.0

[a] PE-Defect Forecast based on Personal Experience

[b] Analyst 1 had 3 new True Positives forecast by HEDF, and 3 True Positives that had been forecast by PE. He refined the descriptions on the three forecasts by PE, as he considered the accuracy had significantly improved after using HEDF. However, we do not count these refined forecasts as the improvement made by HEDF in Table 19. Other analysts had no overlapping defects in the two rounds' forecasting.

TABLE 18. The Statistics for the metrics of defect early forecasting performances.

Metrics	Average		Max		Min		SD	
	HEDF	PE	HE DF	PE	HE DF	PE	HE DF	PE
TP	4	0.75	6	3	3	0	1.4	1.5
FP	1.25	6.25	3	8	0	5	1.26	1.26
Precision (%)	79.2	8.3	100	33.3	66.7	0	14.4	16.7
SDF (%)	18.2	3.4	27.3	13.6	13.6	0	6.5	6.8
SDOC (%)	43.6	14	65.7	57	10	0	24.3	28.5

RQ3: TO WHAT EXTENT CAN HEDF IMPROVE USERS' PERFORMANCES IN DEFECT EARLY FORECASTING?

We are interested in how many extra new defects can be forecast by HEDF, in addition to one's own experience. For this study, HEDF performances are compared to defect forecasting based on Personal Experience in Table 14.

With HEDF *all four analysts, at various expertise levels, significantly* and *consistently* improved their performance, adding on average 4 new True Positives per analyst, compared to an average of 0.8 True positives per analyst without HEDF.

If this study were indicative of general performance, it would seem that only an analyst with an extremely high level of expertise in software quality assurance could forecast, at the requirement stage, a portion of specific program defects that is comparable to what HEDF is capable of. In our case, Analyst 1 is a Ph.D. and professor in software quality assurance, and with over 20 years' industrial consulting experience. He was able to forecast 3 true positives, while

TABLE 19. HEDF performances compared to Defect forecast based on Personal Experience.

Metrics	Analyst 1		Analyst 2		Analyst 3		Analyst 4	
	Imp[a]	IR[b]	Imp	IR	Imp	IR	Imp	IR
TP	3	100	4	∞	3	∞	3	∞
FP	3[c]	100	8	100	5	83.3	4	80
Precision (%)	33.3	100	100	∞	75.0	∞	75.0	∞
SDF (%)	13.6	100	18.2	∞	13.6	∞	13.6	∞
SDOC (%)	8.7	15.3	42.9	∞	55.7	∞	10.0	∞

[a]Imp: The improvement of HEDF comparing to that of PE, counts only those defects that were forecast by HEDF AND occurred BUT not forecast by PE (HEDF&OC-PE&OC).

[b]IR: Improvement Ratio, calculated by $\frac{Imp}{PEDF}$ for each metric.

[c] Improvement of False Positives is calculated by the FP of PE minus the FP of HEDF.

TABLE 20. Descriptive statistics for the Persistence of all the defects.

Df. ID	Occurrence[1]	Total versions in which the defect is present	Persistence of the defect (P_i)		
			Min.	Max.	Mean (DP_i)
F1	7	33	0.17	0.67	0.50
F2	23	98	0.17	1	0.63
F3	1	1	0.09	0.09	0.09
F4	1	1	0.25	0.25	0.25
F5	1	3	0.75	0.75	0.75
F6	14	73	0.33	1.00	0.83
F7	3	36	0.25	1.00	0.75
F8	1	1	0.17	0.17	0.17
F9	2	5	0.17	0.22	0.19
F10	2	2	0.50	1.00	0.75
F11	1	1	0.50	0.50	0.50
F12	1	18	0.72	0.72	0.72
F13	1	4	1.00	1.00	1.00
F14	1	2	0.67	0.67	0.67
F15	1	1	0.17	0.17	0.17
F16	1	1	0.17	0.17	0.17
F17	2	2	1.00	1.00	1.00
F18	2	23	0.94	1.00	0.97
F19	1	1	0.50	0.50	0.5
F20	1	1	0.33	0.33	0.33
F21	2	27	0.76	1.00	0.88
F22	1	1	0.50	0.50	0.50

people with little experience in software quality assurance (Analyst 2, 3 and 4) had zero true positives, even though they had considerable experience in software development (≥4 years). By using HEDF, Analyst 2, 3 and 4 were able to forecast 3 or 4 true positives, which reached or exceeded what Analyst 1 achieved using personal experience, that is, 20 years' experience in software quality assurance, having reviewed over 8000 defects introduced by other people in over 80 diverse projects. Even for the most experienced analyst, HEDF advanced his performance by 100%.

RQ4: WHAT ARE THE POTENTIAL BENEFITS OF HEDF FOR DEBUGGING AND DEFECT PREVENTION?

To answer this question, we first evaluate how difficult the forecast defects are for the programmers to debug, once the defects are introduced into programs.

1) Average Persistence of Forecast Defects (APFD)

The descriptive statistics *Persistence* for all the defects are summarized in Table 20.

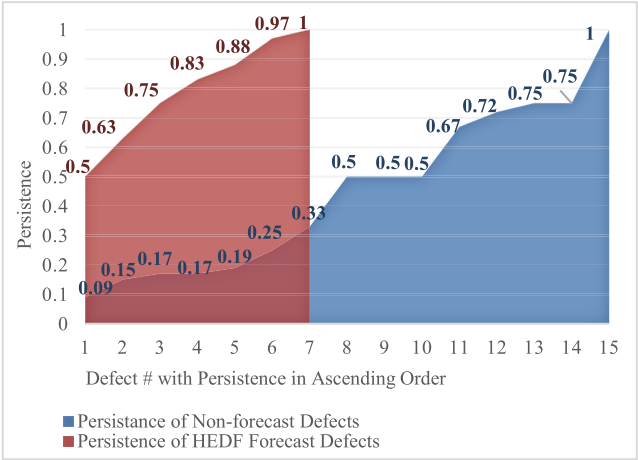


FIGURE 8. The persistence of HEDF forecast defects compared to that of non-forecast defects.

The results for the Average Persistence of HEDF forecast defects is shown in Figure 8. Among the forecast defect forms, F1 is the easiest for debugging (DP=0.50) and yet persists through a half of programmers' debugging process, while F17 is the most difficult for debugging and remains until the final versions (DP=1.00). Overall, **the*Average Persistence of Forecast Defects*is 0.79**, which means these forecast defects were highly persistent through the debugging process, contrasting to **the average persistence 0.45 for the non-forecast defects**. This suggests that our forecast defects tended to be more difficult for the programmers themselves to debug than other defects, as shown in Figure 8.

2) Average Saving of Debugging Effort (ASDE)

The result led to an estimate that HEDF would have saved a programmer **46.2% debugging effort on average** (Minimum = 0%, Maximum = 100%, Standard Deviation = 31.4%).

3) Improvement on Acceptance Rate (IAR)

Results show that HEDF could also significantly increase a programmer's chance of submitting a correct version. Among the total of 55 programmers, 40 programmers got their final versions accepted by the Online Judging System, with an Acceptance Rate of 72.7% (40/55). Among the 15 programmers who failed in submitting any correct versions, 13 programmers' defects could have been prevented if the HEDF forecast defects were provided, because all of their defects belonged to the set of HEDF forecast defects. There were 2 programmers having a defect that were forecast by HEDF but could not be prevented because of "lack of knowledge". This Acceptance Rate could be increased by 23.7% (IAR), up to 96.4% (53/55), if the HEDF were used to avoid or remove predicted defects. **The Improvement Ratio for Acceptance Rate (IRAR) is 32.6%.**

VIII. DISCUSSION

A. CONTRIBUTIONS

This paper proposed an approach to forecast the locations and forms of software defects on the basis of human error mechanisms, before code is produced. In this study, we were

able to predict the locations and forms of 75.7% of defect occurrences, and 31.8% of the total defect types, in the "jiong" example. 75% of the defects actually present in the 55 programs were predicted. Furthermore, these predicted defects were highly persistent through the debugging process: the predicted defects persist through 79% of the debugging process, contrasting to 45% for the non-predicted defects. Most importantly, the prediction was achieved at the requirement phase, using only the information of software requirement specification and the programmers' background knowledge. HEDF could save 46.2% of the debugging effort, using our indicative measure.

The proposed approach demonstrated impressive value in forecasting the locations and forms of software defects. We point at two examples. According to standard defect prediction approaches such as program-metric based models, or one's intuitive judgment, "printing a blank line after each 'jiong' word" should be the location where the defects are least likely to be introduced, since it is the simplest and smallest piece of requirements. The fact is, however, that 41.2% participants introduced the same defect at this place, which HEDF predicted to be a high-risk location: an instance of a scenario liable to trigger "post-completion error" (Table 5). Another example: based on complexity metrics or intuitive judgment, one may anticipate that the location in a program where the relation between the height of a "jiong" word and its nesting level is dealt with (i.e. Step 4 of Solution A in Table 4) is error-prone, as this is a relatively difficult task point involving mathematical modeling. However, one cannot explain why and how the error takes the form of h=8n (while the correct expression would be $h=2^{n+2}$) rather than any other forms, and how multiple programmers could make the exact same error at this place. Using the Error-Prone Scenario Analysis shown in Table 11, one can clearly forecast why and how this location would trigger a defect in this specific form.

B. RELATED WORK

As discussed at the beginning of the paper, the proposed approach, HEDF, is fundamentally different from "defect prediction" performed after code has been produced. HEDF is used in requirement phase (and/or design phase), which is two-three steps earlier than "defect prediction" models.

There are two series of studies relevant to the proposed approach of this paper. The first is Anu, Hu, Carver, Walia, and Bradshaw's research, which proposes a human error taxonomy for requirement review [62] and training people to write requirements specifications [63]. The "faults (or defects)" in their studies are the "manifestation of an error recorded in a document (e.g. requirements specification or use case)" [63]. Though the authors did not explicitly clarify, a "fault in a requirement document" commonly refers to an inappropriate specification according to requirements quality criteria such as incorrectness, inconsistency, incompleteness or ambiguousness in Requirement Engineering [74], [75], [76]. HEDF is distinguished from the studies in [62] and [63]:

- The "defects" in HEDF are the "incorrect or missing steps, processes, or data definitions in a computer program; whereas "faults" in [62] and [63] are the inappropriate specifications in a requirements document according to existing requirements quality criteria.
- The "human errors" of concern in HEDF are the errors committed by software developers in implementing a design or specification; whereas the "human errors' in [62] and [63] are the errors of people who write a requirements document.
- HEDF focuses on forecasting the locations and forms of defects developers may later introduce to code, on condition that the requirements are already appropriately specified according to traditional requirements quality criteria.

The second set of relevant works are Huang (one of the present authors) et al.'s previous studies on software defects prevention [9], [57]. The former study [9] proposes a human error taxonomy to identify the root causes of software defects. Note that Root Cause Analysis (a key process activity in Capability Maturity Model Integration (CMMI) Level 5) is a retrospective process, that is, investigating what factors may had caused a defect, after a defect has already been introduced and found. In contrast to this, HEDF forecasts defects that may be introduced by developers in future. The work [57] presents a method, "defect prevention based on human error theories"(DPeHE), for improving software developers' cognitive ability to prevent software defects based on the knowledge of human errors. The users of the DPeHE reflected that the method has promoted their knowledge and awareness in defect prevention, while difficulties exist in relating the psychological theories to their specific contexts of software development [57]. That is the challenge being addressed in HEDF by the new concept "Error-prone Scenario" and corresponding process of "Error-Prone Scenario Analysis". In a world, DPeHE [57] is a cognitive training method for promoting software engineers' general awareness and knowledge of human errors, while HEDF forecasts the specific defects that may be introduced for a specific piece of requirements of a project.

In summary, HEDF is a unique innovative method that forecasts the exact locations and forms of defects may be introduced by developers, before code is produced, based on the new "Error-Prone Scenario Analysis" process built on the psychological theories on cognitive errors.

C. COST-EFFECTIVENESS

An undoubted cost of HEDF, just like any other software review methods, is that it requires the analysts to receive training on the method. In our case study, the training session was 75 minutes, which seemed to have achieved adequate effects. Note that in the case study we only observed the effects on one forecasting session lasting for 45 minutes, which does not mean the training only benefits a performance of 45 minutes. Training is a one-off cost. Once an analyst

masters the method after the training, he/she can reapply the method in many projects in one's career life time.

The overall cost-effectiveness of the approach is determined by the sum of the costs of analysis (which would be determined by the size and complexity of the software specification to be examined) and the cost of the preventative measures adopted, while the effectiveness is measured by the cost reductions achieved by the approach. The cost reductions involve 1) the cost of finding and removing those defects that are present in the code, but the proactive measures would instead avoid altogether; or, 2) for defects that are not avoided, the cost of finding the defects without the benefit of the prediction that helps to focus inspection and testing; or 3) the cost of the defects remaining in the software in operation. This latter cost (of residual defects in the deployed software) involves, the cost of in-operation removal of the defects, plus client aggravation, for much commercial software, and the harm, potentially much greater, caused by failures in operation.

HEDF does not promise to forecast all potential defects, but is targeted at forecasting the common defects caused by human cognitive error mechanisms, at early stages of software development, with sufficient accuracy to allow actionable suggestions for prevention. Preventing defects is especially significant for safety-critical software, as once a defect has been introduced into a program, it is hard to guarantee the defect would be found and fixed, while a residual defect could lead to a catastrophic accident.

D. LIMITATIONS AND FUTURE STUDIES

The experimental study of this paper is limited to a small-sized programming task, comparable to a component in a larger development project. The focus here is developing the fundamental theories and the approach for the first human-error-based defect early forecast method, and validate it at the laboratory testing stage. The requirements used in the study covered skill-based, rule-based and knowledge-based activities, and involved requirement understanding, program design and coding. Trials in industrial development settings could both assess the value of HEDF in those specific environments and enrich the pool of human error modes.

The other future work is to develop a tool, including a database of Human Error Modes, Error-prone Scenarios, and Error-Prone Scenario Analysis notations, for aid analysts in performing defect early forecasting using HEDF.

E. IMPLICATIONS FOR DEFECT PREVENTION

Software defect early forecast based on human error mechanisms appears a very promising technique: once the location and form of a software defect can be forecast before the program is produced, the defect can be avoided or prevented in a real sense.

Once the error-prone locations are simply flagged to developers (architects, programmers, code reviewers, testers) in advance, they can allocate more attention resources to these locations, thus adjusting the cognitive process to prevent or detect the errors. For instance, with the information provided by our Error-Prone Scenario Analysis ES6, one can simply add one notice in a Defect Prevention Strategy like "***Correct:*** *printing a "jiong" word once a user enters the nesting level of the word.* ***Wrong:*** *the "jiong"s are printed together only after all of the inputs have been entered."* Such a message is highly actionable and would instantly prevent developers from committing this error.

HEDF produces not only information on the error-prone locations in the program but also possible error forms, thus it is helpful for test engineers to conduct focused checks and design test cases. For instance, with the information provided by our error-prone scenario analysis ES7, a code inspector can pay special attention to check whether the mathematical relation between the height of a "jiong" word (h) and its nesting level (n) is correctly implemented as $h=2^{n+2}$. That is, the analysis would highlight that this is a requirement for "exponential" programming and thus subject to specific difficulties.

Another strategy is to prevent defects by improving the representation of the specifications. For instance, at a place where post-completion error is likely to happen, software requirements/specifications can be revised to avoid putting the sub-goal in the last step, or highlighting the sub-goal in the last step to capture developers' attention. For instance, the requirement writer could highlight (e.g. using bright colors and/or bold font) the places of post-completion tasks in the requirement documents ("printing a blank line after each word" in the "jiong" case), since visual cues are an effective way to reduce post-completion errors [77]. The contribution here is to tell the writer exactly what should be highlighted to counteract the readers' error-proneness.

IX. CONCLUSION

Accuracy in forecasting an event often depends on the extent to which the causal mechanisms underlying the event are understood. This paper proposes an approach, HEDF, to forecast software defects early by considering the human error mechanisms that cause them. Compared to established prediction models relying on the code that has already been produced, the proposed HEDF approach emphasizes identifying, at requirement and/or design stage, scenarios that tend to trigger human error modes which psychologists have observed to recur across various activities. HEDF emphasizes detailed forecasting of specific forms of defect at specific locations in a program, before the code is produced, aimed to proactively prevent software defects.

Our case study, which involved 55 programmers and four representative analysts, suggested that defect early forecasting is very challenging without specialized training: only an extremely experienced expert forecast a small proportion of the defects with a high false positive rate while none of the analysts at high, intermediate and entry levels forecast any true positive. HEDF has *significantly* and *consistently* improved the performance of defect forecasting for *all users at various expertise levels* (a minimum of 100%

ratio of improvement on precision and Sensitivity to defect forms). The results suggest that HEDF is highly effective in forecasting software defects at the early phases of software development.

The forecast results of HEDF can be directly used to prevent defects from occurring, because the exact locations and forms have been pinpointed before programming. A rough estimate suggests 46% average savings in debugging or testing effort on the example program in this study. This study is limited to a small-sized programming task in a non-industrial setting.

In the future, we plan to explore how to extend the application of the approach to large-scale industrial projects, and develop a tool set to support the forecasting process.

ACKNOWLEDGMENT

The authors would like to thank Dr. You Song for giving them access to the programming contest data for the study and also would like to thank the participants of the research.

APPENDIX A

Survey #1 Defect Prediction based on Personal Experience

This survey aims to explore how software engineers predict software defects based on personal experience. Your defect prediction results will be used as data of this research. Your personal information will remain confidential and anonymous. Would you volunteer to participate?

( ) Yes ( ) No Name_____ Date _____

Part I. Background

1. **Occupation**_____ (you can choose multiple answers; please put your current primary occupation in the first place)
   A. Professor B. Expert in industry C. Young Researcher D. Professional engineer in industry E. Manager in industry F. Graduate Student G. Undergraduate Student H. other (please specify)_____
2. **Major(s) of degrees**_____(e.g. Bachelor in Computer Science)
3. Years of experience in software **development**_____
4. Please estimate the number of software **development projects** that you have participated so far_____
   A. 0-10 B. 10-30 C. 30-50 D. 50-80 E. > 80
5. Please estimate **the number of defects you have introduced** since your first time of programming_____
   A. 0-1000 B. 1000-3000 C.3000-5000 D. 5,000-8,000 E. >8,000
6. Years of experience in software verification, validation, and other **software quality assurance** activities_____
7. Please estimate **the number of projects in software quality assurance** that you have participated so far_____
   A. 0-10 B. 10-30 C. 30-50 D. 50-80 E. > 80
8. Please estimate the number of **other people's defects** you have reviewed during the course of all the projects on software quality assurance
   A. 0-100 B. 100-1000 C.1000-5000 D. 5,000-8,000 E. >8,000
9. Years of experience in teaching and mentoring students_____
10. Years of experience in providing consulting services to software industry_____
11. Years of **full-time** working in software industry_____
12. Knowledge of Human Errors in Psychology_____
    A. None or Scarce B. Basic C. Abundant/expert

Part II. Defect Prediction based on Personal Experience

The Requirement Specification of the "Jiong" Programming Task is provided here.

Predicted Defects:

➢ Please predict the defects that undergraduates (1st grader in Computer Science who have completed C-language) may introduce, based on the requirement specification on Page 2 and your experience. Please specify your predicted defects below:

(A blank page remains here.)

Part II. Reflection on your prediction

Please reflect what sources have led to your prediction results. Please first label your predicted defects (#1, #2, etc.) on page 3, and answer the following questions by choosing multiple answers from the A,B,C, D, or specify in details.

A. I have introduced similar defects in my previous development experience

B. I have seen such defects in my previous quality assurance activities, such as software testing, verification, validation, managing software V&V projects, or reviewing other people's projects

C. Intuition: I know it came from my previous experience, but could not tell where it came from

D. Unknown

Source(s) for Predicted **Defect #1** (if applicable):_____ Other (please specify)_____.

Source(s) for Predicted **Defect #2** (if applicable):_____ Other (please specify)_____.

Source(s) for Predicted **Defect #3** (if applicable):_____ Other (please specify)_____.

Source(s) for Predicted **Defect #4** (if applicable):_____ Other (please specify)_____.

****You are free to add more below if applicable****

REFERENCES

[1] *IEEE Standard Glossary of Software Engineering Terminology*, Institute of Electrical and Electronics Engineers, New York, NY, USA, Standard IEEE 610.121990, 1990.

[2] M. Ben-Ari, "The bug that destroyed a rocket," *ACM SIGCSE Bull.*, vol. 33, no. 2, pp. 58–59, Jun. 2001.

[3] M. Grottke, A. P. Nikora, and K. S. Trivedi, "An empirical investigation of fault types in space mission system software," in *Proc. IEEE/IFIP Int. Conf. Dependable Syst. Netw. (DSN)*, Jun. 2010, pp. 447–456.

[4] S. Wander, "Powerless. System failure case studies," Nat. Aeronaut. Space Admin., 2007, pp. 1–4, vol. 1, no. 10. [Online]. Available: https://sma.nasa.gov/docs/default-source/safety-messages/safetymessage-2008-03-01-northeastblackoutof2003.pdf

[5] *Actual and Potential Harm Caused by Medical Software*, Therapeutic-Goods-Admin., Austral. Dept. Health, Canberra, 2020.

[6] H. Krasner. (2020). *The Cost of Poor Software Quality in the US: A 2020 Report*. [Online]. Available: https://www.it-cisq.org/pdf/CPSQ-2020-report.pdf

[7] G. M. Weinberg, *The Psychology of Computer Programming*. Santa Fe Springs, CA, USA: Van Nostrand Reinhold, 1971.

[8] F. Détienne, *Software Design—Cognitive Aspects*. New York, NY, USA: Springer-Verlag, 2002.

[9] F. Huang, B. Liu, and B. Huang, "A taxonomy system to identify human error causes for software defects," in *Proc. 18th Int. Conf. Rel. Quality Design*, Boston, MA, USA, 2012, pp. 44–49.

[10] F. Huang, B. Liu, and Y. Wang, "Review of software psychology," *Comput. Sci.*, vol. 40, pp. 1–7, Mar. 2013.

[11] N. E. Fenton and M. Neil, "A critique of software defect prediction models," *IEEE Trans. Softw. Eng.*, vol. 25, no. 5, pp. 675–689, Sep./Oct. 1999.

[12] N. Fenton, M. Neil, W. Marsh, P. Hearty, D. Marquez, P. Krause, and R. Mishra, "Predicting software defects in varying development lifecycles using Bayesian nets," *Inf. Softw. Technol.*, vol. 49, no. 1, pp. 32–43, Jan. 2007.

[13] X. Yang, K. Tang, and X. Yao, "A learning-to-rank approach to software defect prediction," *IEEE Trans. Rel.*, vol. 64, no. 1, pp. 234–246, Mar. 2015.

[14] T. T. Nguyen, T. N. Nguyen, and T. M. Phuong, "Topic-based defect prediction (NIER track)," in *Proc. 33rd Int. Conf. Softw. Eng.*, May 2011, pp. 932–935.

[15] B. A. Kitchenham, L. M. Pickard, and S. J. Linkman, "An evaluation of some design metrics," *Softw. Eng. J.*, vol. 5, no. 1, pp. 50–58, Jan. 1990.

[16] N. Nagappan and T. Ball, "Use of relative code churn measures to predict system defect density," in *Proc. 27th Int. Conf. Softw. Eng.*, 2005, pp. 284–292.

[17] T. Menzies, J. Greenwald, and A. Frank, "Data mining static code attributes to learn defect predictors," *IEEE Trans. Softw. Eng.*, vol. 33, no. 1, pp. 2–13, Jan. 2007.

[18] K. O. Elish and M. O. Elish, "Predicting defect-prone software modules using support vector machines," *J. Syst. Softw.*, vol. 81, no. 5, pp. 649–660, 2008.

[19] T. Zimmermann and N. Nagappan, "Predicting defects using network analysis on dependency graphs," in *Proc. 13th Int. Conf. Softw. Eng. (ICSE)*, 2008, pp. 531–540.

[20] T. J. McCabe, "A complexity measure," *IEEE Trans. Softw. Eng.*, vol. SE-2, no. 4, pp. 308–320, Dec. 1976.

[21] M. H. Halstead, *Elements of Software Science*, vol. 7. New York, NY, USA: Elsevier, 1977.

[22] L. Madeyski and M. Jureczko, "Which process metrics can significantly improve defect prediction models? An empirical study," *Softw. Quality J.*, vol. 23, no. 3, pp. 393–422, 2014.

[23] K. El Emam, W. Melo, and J. C. Machado, "The prediction of faulty classes using object-oriented design metrics," *J. Syst. Softw.*, vol. 56, no. 1, pp. 63–75, Feb. 2001.

[24] N. Nagappan, B. Murphy, and V. Basili, "The influence of organizational structure on software quality: An empirical case study," in *Proc. 13th Int. Conf. Softw. Eng. (ICSE)*, 2008, pp. 521–530.

[25] K. Herzig, S. Just, and A. Zeller, "The impact of tangled code changes on defect prediction models," *Empirical Softw. Eng.*, vol. 21, no. 2, pp. 303–336, Apr. 2016.

[26] R. Moser, W. Pedrycz, and G. Succi, "A comparative analysis of the efficiency of change metrics and static code attributes for defect prediction," in *Proc. 13th Int. Conf. Softw. Eng. (ICSE)*, 2008, pp. 181–190.

[27] M. S. Zanetti, I. Scholtes, C. J. Tessone, and F. Schweitzer, "Categorizing bugs with social networks: A case study on four open source software communities," in *Proc. Int. Conf. Softw. Eng.*, 2013, pp. 1032–1041.

[28] P. He, B. Li, X. Liu, J. Chen, and Y. Ma, "An empirical study on software defect prediction with a simplified metric set," *Inf. Softw. Technol.*, vol. 59, pp. 170–190, Mar. 2015.

[29] Y. Peng, G. Kou, G. Wang, W. Wu, and Y. Shi, "Ensemble of software defect predictors: An AHP-based evaluation method," *Int. J. Inf. Technol. Decis. Making*, vol. 10, no. 1, pp. 187–206, 2011.

[30] D. Radjenović, M. Heričko, R. Torkar, and A. Živkovič, "Software fault prediction metrics: A systematic literature review," *Inf. Softw. Technol.*, vol. 55, no. 8, pp. 1397–1418, 2013.

[31] B. Kitchenham, "What's up with software metrics?—A preliminary mapping study," *J. Syst. Softw.*, vol. 83, pp. 37–51, Jan. 2010.

[32] G. J. Pai and J. B. Dugan, "Empirical analysis of software fault content and fault proneness using Bayesian methods," *IEEE Trans. Softw. Eng.*, vol. 33, no. 10, pp. 675–686, Oct. 2007.

[33] M. Liu, L. Miao, and D. Zhang, "Two-stage cost-sensitive learning for software defect prediction," *IEEE Trans. Rel.*, vol. 63, no. 2, pp. 676–686, Jun. 2014.

[34] I. H. Laradji, M. Alshayeb, and L. Ghouti, "Software defect prediction using ensemble learning on selected features," *Inf. Softw. Technol.*, vol. 58, pp. 388–402, Feb. 2015.

[35] H. Lu, E. Kocaguneli, and B. Cukic, "Defect prediction between software versions with active learning and dimensionality reduction," in *Proc. IEEE 25th Int. Symp. Softw. Rel. Eng.*, Nov. 2014, pp. 312–322.

[36] J. Ekanayake, J. Tappolet, H. C. Gall, and A. Bernstein, "Time variance and defect prediction in software projects," *Empirical Softw. Eng.*, vol. 17, nos. 4–5, pp. 348–389, Aug. 2012.

[37] P. Bhattacharya, M. Iliofotou, I. Neamtiu, and M. Faloutsos, "Graph-based analysis and prediction for software evolution," in *Proc. 34th Int. Conf. Softw. Eng. (ICSE)*, Jun. 2012, pp. 419–429.

[38] N. Fenton, M. Neil, W. Marsh, P. Hearty, Ł. Radliński, and P. Krause, "On the effectiveness of early life cycle defect prediction with Bayesian nets," *Empirical Softw. Eng.*, vol. 13, no. 5, pp. 499–537, Oct. 2008.

[39] T. Zimmermann, N. Nagappan, H. Gall, E. Giger, and B. Murphy, "Cross-project defect prediction: A large scale experiment on data vs. domain vs. process," in *Proc. 7th Joint Meeting Eur. Softw. Eng. Conf. ACM SIGSOFT Symp. Found. Softw. Eng. Eur. Softw. Eng. Conf. Found. Softw. Eng. Symp. (ESEC/FSE)*, 2009, pp. 91–100.

[40] F. Rahman, D. Posnett, and P. Devanbu, "Recalling the 'imprecision' of cross-project defect prediction," in *Proc. ACM SIGSOFT 20th Int. Symp. Found. Softw. Eng. (FSE)*, 2012, p. 61.

[41] Y. Kamei, T. Fukushima, S. McIntosh, K. Yamashita, N. Ubayashi, and A. E. Hassan, "Studying just-in-time defect prediction using cross-project models," *Empirical Softw. Eng.*, vol. 21, no. 5, pp. 2072–2106, 2016.

[42] E. J. Weyuker, T. J. Ostrand, and R. M. Bell, "Do too many cooks spoil the broth? Using the number of developers to enhance defect prediction models," *Empirical Softw. Eng.*, vol. 13, no. 5, pp. 539–559, Oct. 2008.

[43] R. M. Bell, T. J. Ostrand, and E. J. Weyuker, "The limited impact of individual developer data on software defect prediction," *Empirical Softw. Eng.*, vol. 18, no. 3, pp. 478–505, Jun. 2013.

[44] C. Bird, N. Nagappan, B. Murphy, H. Gall, and P. Devanbu, "Don't touch my code!: Examining the effects of ownership on software quality," in *Proc. 19th ACM SIGSOFT Symp. 13th Eur. Conf. Found. Softw. Eng. (SIGSOFT/FSE)*, 2011, pp. 4–14.

[45] M. Pinzger, N. Nagappan, and B. Murphy, "Can developer-module networks predict failures?" in *Proc. 16th ACM SIGSOFT Int. Symp. Found. Softw. Eng. (SIGSOFT/FSE)*, 2008, pp. 2–12.

[46] S. Matsumoto, Y. Kamei, A. Monden, K.-I. Matsumoto, and M. Nakamura, "An analysis of developer metrics for fault prediction," in *Proc. 6th Int. Conf. Predictive Models Softw. Eng. (PROMISE)*, 2010, p. 18.

[47] P. L. Li, J. Herbsleb, M. Shaw, and B. Robinson, "Experiences and results from initiating field defect prediction and product test prioritization efforts at ABB Inc," in *Proc. 28th Int. Conf. Softw. Eng.*, 2006, pp. 413–422.

[48] S. Yoo and M. Harman, "Regression testing minimization, selection and prioritization: A survey," *Softw., Test. Verification Rel.*, vol. 22, no. 2, pp. 67–120, 2012.

[49] J. Nam, "Survey on software defect prediction," Dept. Comput. Sci. Eng., Hong Kong Univ. Sci. Technol., Tech. Rep., 2014.

[50] P. Li, M. Shaw, and J. Herbsleb, "Selecting a defect prediction model for maintenance resource planning and software insurance," in *Proc. EDSER Affiliated ICSE*, 2003, pp. 32-37.

[51] E. Engström, P. Runeson, and G. Wikstrand, "An empirical evaluation of regression testing based on fix-cache recommendations," in *Proc. 3rd Int. Conf. Softw. Test., Verification Validation*, 2010, pp. 75–78.

[52] C. Lewis, Z. Lin, C. Sadowski, X. Zhu, R. Ou, and E. J. Whitehead Jr., "Does bug prediction support human developers? Findings from a Google case study," in *Proc. Int. Conf. Softw. Eng.*, 2013, pp. 372–381.

[53] A. J. Ko and B. A. Myers, "A framework and methodology for studying the causes of software errors in programming systems," *J. Vis. Lang. Comput.*, vol. 16, nos. 1–2, pp. 41–84, Feb. 2005.

[54] J. Reason, *Human Error*. Cambridge, U.K.: Cambridge Univ. Press, 1990.

[55] F. Huang and B. Liu, "Systematically improving software reliability: Considering human errors of software practitioners," in *Proc. 23rd Psychol. Program. Interest Group Annual Conf. (PPIG)*, New York, NY, USA, 2011, pp. 1–5. [Online]. Available: https://www.researchgate.net/publication/271517817_Systematically_Improving_Software_Reliability_Considering_Human_Errors_of_Software_Practitioners
[56] F. Huang, "Software fault defense based on human errors," Ph.D. thesis, School Rel. Syst. Eng., Beihang Univ., Beijing, China, 2013.
[57] F. Huang and B. Liu, "Software defect prevention based on human error theories," *Chin. J. Aeronaut.*, vol. 30, no. 3, pp. 1054–1070, Jun. 2017.
[58] F. Huang, B. Liu, Y. Song, and S. Keyal, "The links between human error diversity and software diversity: Implications for fault diversity seeking," *Sci. Comput. Program.*, vol. 89, pp. 350–373, Sep. 2014.
[59] F. Huang, "Post-completion error in software development," in *Proc. 9th Int. Workshop Cooperat. Human Aspects Softw. Eng.*, Austin, TX, USA, May 2016, pp. 108–113.
[60] F. Huang, "Human error analysis in software engineering," in *Theory and Application on Cognitive Factors and Risk Management-New Trends and Procedures*. Rijeka, Croatia: InTech, 2017.
[61] F. Huang and H. Madeira, "Targeted code inspection based on human errors," in *Proc. IEEE Int. Symp. Softw. Rel. Eng. Workshops (ISSREW)*, Wuhan, China, Oct. 2021, pp. 274–275.
[62] V. Anu, W. Hu, J. C. Carver, G. S. Walia, and G. Bradshaw, "Development of a human error taxonomy for software requirements: A systematic literature review," *Inf. Softw. Technol.*, vol. 103, pp. 112–124, Nov. 2018.
[63] W. Hu, J. C. Carver, V. Anu, G. S. Walia, and G. L. Bradshaw, "Using human error information for error prevention," *Empirical Softw. Eng.*, vol. 23, no. 6, pp. 3768–3800, Dec. 2018.
[64] B. Nagaria and T. Hall, "How software developers mitigate their errors when developing code," *IEEE Trans. Softw. Eng.*, vol. 48, no. 6, pp. 1853–1867, Jun. 2022.
[65] R. Chillarege, I. S. Bhandari, J. K. Chaar, M. J. Halliday, D. S. Moebus, B. K. Ray, and M.-Y. Wong, "Orthogonal defect classification—A concept for in-process measurements," *IEEE Trans. Softw. Eng.*, vol. 18, no. 11, pp. 943–956, 1992.
[66] J. Rasmussen, "Skills, rules, and knowledge; signals, signs, and symbols, and other distinctions in human performance models," *IEEE Trans. Syst., Man, Cybern.*, vols. SMC–13, no. 3, pp. 257–266, May 1983.
[67] R. Glaser, "Education and thinking: The role of knowledge," *Amer. Psychologist*, vol. 39, no. 4, pp. 93–104, 1984.
[68] M. D. Byrne and S. Bovair, "A working memory model of a common procedural error," *Cognit. Sci.*, vol. 21, no. 1, pp. 31–61, Jan. 1997.
[69] W. Schnotz and C. Kürschner, "A reconsideration of cognitive load theory," *Educ. Psychol. Rev.*, vol. 19, no. 4, pp. 469–508, Oct. 2007.
[70] H. Tan, *C Programming* 3rd ed. Beijing, China: Tsinghua Univ. Press, 2005.
[71] J. Yao and M. Shepperd, "The impact of using biased performance metrics on software defect prediction research," *Inf. Softw. Technol.*, vol. 139, Nov. 2021, Art. no. 106664.
[72] N. Ye and G. Salvendy, "Quantitative and qualitative differences between experts and novices in chunking computer software knowledge," *Int. J. Hum.-Comput. Interact.*, vol. 6, no. 1, pp. 105–118, Jan. 1994.
[73] L. Strigini, "Fault tolerance against design faults," in *Dependable Computing Systems: Paradigms, Performance Issues, and Applications*, H. Diab and A. Zomaya, Eds. Hoboken, NJ, USA: Wiley, 2005, pp. 213–241.
[74] P. Heck and A. Zaidman, "A systematic literature review on quality criteria for agile requirements specifications," *Softw. Quality J.*, vol. 26, no. 1, pp. 127–160, Mar. 2018.
[75] N. Kiyavitskaya, N. Zeni, L. Mich, and D. M. Berry, "Requirements for tools for ambiguity identification and measurement in natural language requirements specifications," *Requirements Eng.*, vol. 13, pp. 207–239, Jul. 2008.
[76] M. Sabetzadeh and S. Easterbrook, "View merging in the presence of incompleteness and inconsistency," *Requirements Eng.*, vol. 11, no. 3, pp. 174–193, Jun. 2006.
[77] P. H. Chung and M. D. Byrne, "Cue effectiveness in mitigating postcompletion errors in a routine procedural task," *Int. J. Hum.-Comput. Stud.*, vol. 66, no. 4, pp. 217–232, Apr. 2008.

FUQUN HUANG (Member, IEEE) received the Ph.D. degree in systems engineering from Beihang University, in 2013. She was a Visiting Scholar at the Centre for Software Reliability, City, University of London, in 2011. She was a Postdoctoral Researcher with The Ohio State University, from 2014 to 2016. She is currently a Research Assistant Professor (a Principal Investigator) at the Centre for Informatics and Systems, University of Coimbra, where she leads the Research Group "Human Errors in Software Engineering" (HESE). She gives the first university course in HESE to master's students with the University of Coimbra. She is also the Founder of the Institute of Interdisciplinary Scientists, a Federal 501(c)(3) non-profit research institute located at Seattle, where she has been initiated the "Software Engineering & Psychology" Interdisciplinary Research Program, since 2016, dedicated to defend against software defects through a deep understanding of the psychological mechanisms of how software practitioners commit human errors. She has regularly served as a Reviewer for reputed journals, such as *Reliability Engineering and Systems Safety*, IEEE Transactions on Reliability, and *Software Testing, Verification and Reliability*. She is a member of IEEE Standards and a Program Committee Member for conferences, such as the Annual Conference on Innovation and Technology in Computer Science Education (ITiCSE 2022 and 2023), the IEEE International Conference on Software Quality, Reliability and Security (QRS 2021 and 2022), and IEEE International Workshop on Software Certification (2015–2021). She is also a Founder Member of the Interdisciplinary Area "Human Errors in Software Engineering." She is also the Publicity Chair of The 53rd Annual IEEE/IFIP International Conference on Dependable Systems and Networks (DSN 2023).

LORENZO STRIGINI (Member, IEEE) is currently a Professor in systems engineering and the Director of the Centre for Software Reliability, City, University of London, which he joined, in 1995. Previously, he was a Researcher with the National Research Council of Italy and has been a Visiting Scholar with various leading research laboratories. He has worked for some 40 years on problems of reliability, safety, and security of software and systems, including leading research projects and consulting, and has published widely in these areas. His research interests include improving the scientific credibility of claims about dependability, using probabilistic modeling both for insight and for inference from data, to help design and deployment decisions as well as assessing working systems. Specific topics include fault tolerance through diversity, software testing, the reliability of human-computer systems, problems in acceptance decisions for critical systems, and interactions between safety and security. His recent application areas include nuclear power and autonomous driving. He has served as an Associate Editor for the IEEE Transactions on Software Engineering, an Associate Editor-in-Chief for IEEE Transactions on Dependable and Secure Computing. He is a member of IFIP Working Group 10.4 on Dependable Computing and Fault Tolerance; the European Workshop on Industrial Systems Reliability, Safety and Security; the ACM; and the IEEE Computer Society.