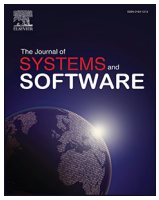
The Journal of Systems and Software 214 (2024) 112060

Contents lists available at ScienceDirect

The Journal of Systems & Software

journal homepage: www.elsevier.com/locate/jss



Advancing modern code review effectiveness through human error mechanisms

Fuqun Huang [a,*], Henrique Madeira [b]

[a] Department of Computer Science, Western Wasington University, Bellingham, WA 98225, USA
[b] University of Coimbra, Centre for Informatics and Systems of the University of Coimbra, 3030-790 Coimbra, Portugal

ARTICLE INFO

Keywords:
Code review
Human error
Developer
Software quality assurance
Cognitive psychology

ABSTRACT

Modern code reviews tend to take a lightweight process, in which the accuracy and efficiency of identifying defects rely heavily on code reviewers' experience. The human errors of developers, as a significant cause of software defects, is a key to identifying defects. However, there is a lack of understanding of the human error mechanisms underlying defects in code. This paper proposes an innovative code review method for identifying defects by pinpointing the scenarios that developers tend to commit errors. The method was validated by two experimental studies that involved 40 participants of about 5 years' programming experience and modest code review experience. The experiment shows that the proposed method has significantly improved True Positives and Sensitivity by about 400 %, improved Precision by approximately 200 %, and reduced around one-third of False Positives. The effects were consistent across different tasks and different code reviewers.

1. Introduction

Code review has become a widely used approach for software quality assurance, representing a significant part of software engineers' daily work (Bacchelli and Bird, 2013; Wilkerson et al., 2011). Since Fagan first proposed a formal procedure for code inspection (Fagan, 1999), modern code reviews (Bacchelli and Bird, 2013) have evolved to take a lightweight and more flexible process (Baum et al., 2016; Sadowski et al., 2018) of ad hoc reading by one reviewer with asynchronous comments to the author of the code, instead of formal review meeting involving multiple reviewing roles.

Finding defects is one of the primary goals of code review (Bacchelli and Bird, 2013). Classic code reviews (i.e., Fagan style) are reported to find about sixty percent defects on average, with a high deviation (Shull et al., 2002). In a recent study conducted by Khoshnoud et al. (2022), 77 open-source GitHub projects were evaluated, leading to the identification of 187 missed bugs within 173 buggy pull requests. This highlights the persistence of software defects in contemporary code review processes. Empirical research by McIntosh et al. (2016) revealed that software systems continue to exhibit defects in 11–19 % of their components, even when subjected to rigorous review. These findings collectively underscore the ongoing need for enhancements in defect identification within modern code review practices.

The effectiveness of modern code reviews heavily rely on developers' expertise (Baum et al., 2016; Baysal et al., 2016) and understanding of the code (Ebert et al., 2021). Ad-hoc reading, checklist reading and stepwise abstraction are the most widely used code review methods (Aurum et al., 2002). Perspective-based reading (Laitenberger and DeBaud, 1997) provides a code reviewer with a set of questions based on defect types, from the "perspective" (type of testing) the code review is involved, e.g., unit testing or integration testing. Recently, researchers explored new strategies for modern code review, such as how to recommend appropriate code reviewers based on file location (Thongtanunam et al., 2015), and biofeedback has been used to remind code reviewers whether a snippet of code is sufficiently read (Hijazi et al., 2021), or to score the quality of the review based on the assessment of the reviewer's comprehension of the code (Haytham Hijazi, 2022). Despite the progress, the reading techniques for code reviewers have changed little in modern code reviews, and ad hoc reading is still the most widely used approach in the software industry (Baum et al., 2016; Sadowski et al., 2018).

The human errors of developers are a significant cause of software defects in code, since computer programs are primarily written by individual developers. The human error taxonomies have been recently used for requirements review (Anu et al., 2016). Huang and Strigini (2023) proposed a method to forecast defects in programs based on

* Corresponding author.
E-mail address: huangf2@wwu.edu (F. Huang).

https://doi.org/10.1016/j.jss.2024.112060
Received 16 February 2023; Received in revised form 17 February 2024; Accepted 8 April 2024
Available online 15 April 2024

reviewing requirements and design documents. However, to the best of our knowledge, the psychological mechanisms by which programmers commit errors have not yet been used to improve the effectiveness of code reviews in identifying software defects.

Equipping code reviewers with the knowledge and skills to identify snippets of code associated with human errors is an intriguing and significant topic to explore. One rationale is that programmers' human errors are a primary cause of defects in code. Psychologists have found that although errors appear enormously diverse in different contexts, there are a limited number of cognitive patterns (to be defined later in this paper as "Human Error Mode") behind them (Reason, 1990). Once a code reviewer is trained in one human error mode and equipped with methods to identify this pattern, he/she can detects many defects in different programs/projects that are caused by the same cognitive pattern. Furthermore, because these cognitive patterns are shared by all humans, they are found to trigger multiple programmers to introduce the same defects in the same ways, with higher likelihoods than defects triggered by random factors, such as typos (Huang and Strigini, 2023). Therefore, training code reviewers to identify such "hot spots" in code could substantially improve the efficiency of code review performances.

We conducted a pilot study (Huang and Madeira, 2021) to explore the feasibility of this idea. In the small experiment, we conducted a 20-min brief training on two human error modes to two participants and asked them to conduct code reviews on a program with multiple seeded defects. The results showed that the participants took an average of 2 min to find the defects that associated with the trained human error modes and 17 min to find other defects. The pilot study indicates that training code reviewers on human error theories has great potential to improve code review performance, but it was limited by a very small number of human error modes, few participants, and the lack of a controlled group.

This paper proposes a code review method based on the knowledge of how programmers commit human errors; we name this method **H**uman **E**rror-based **C**ode **R**eview (HECR). The goal is to provide awareness and to train code reviewers on the identification of code areas that represent human error prone scenarios based on established human error modes, to maximize the quality of code reviews in terms of bug finding and reduction of false positives. We also contribute a comprehensive experimental study to evaluate the proposed method.

2. Related work

A variety of topics studied in the area of modern code reviews, such as reviewer recommendations (Chen et al., 2022; Mirsaeedi and Rigby, 2020), review prioritization (Fan et al., 2018), tool support (Kalyan et al., 2016; Perry et al., 2002), and process improvement (Thongtanunam and Hassan, 2020). Empirical studies on the difficulties encountered in industrial companies, e.g. Doğan and Tüzün (2022) identified eight types of "code review smells" (difficulties in code review) using a survey, including lack of reviewers, the author assigning the same reviewer (s), etc. Pascarella et al. (2018) identified seven high-level information needed for modern code review, such as knowing the uses of methods and variables declared/modified in the code under review.

There have been several studies on code review automation based on machine learning. The most advanced deep learning method can achieve about 14 % perfect accuracy–fully matching the correct outputs of human code review results (Tufano et al., 2022). Hong et al. (2022)'s machine learning approach to recommend the files that are likely to receive comments, and achieved 81 % "Top-10 Accuracy", a special metric defined by the authors as "the percentage of files where at least one of the top-10 lines ranked by REVSPOT (the proposed approach) is one of the actual lines that receive comments"; this accuracy is essentially about file recommendation accuracy rather than the fine-grained defect level in Tufano et al. (2022) and human code reviews. As we can see, such automatic tools have much lower accuracy than the average 60% accuracy achieved by human reviewers (Shull et al., 2002).

2.1. Human aspects of code reviewers

Human aspects of reviewers have been studied recently, but little research was found on using human error theories to promote reviewers' performances. In a recent comprehensive literature review conducted by Davila and Nunes (2021), nine publications were identified relevant to human aspects of reviewers: five of them explored reviewer's code review behaviors using biometric devices such as eye tracker and functional magnetic resonance imaging (fMRI) (Duraes et al., 2016; Floyd et al., 2017). The other four publications include: Kitagawa et al. (2016) proposed a game theory to model the situation when reviewers tend to participate in code review and reach agreements. Baum et al. (2019) conducted an experimental study, which found that there is a moderate association between reviewers working memory capacity and the effectiveness of finding delocalized defects, whereas the association with other defect types is almost non-existing. Spadini et al. (2019)'s empirical study found that most developers prefer to review changed production code rather than changed test code. Dunsmore et al. (2000) conducted an empirical study on code review on object-oriented programs, and found that reviewers have difficulty in finding defects that consist of information scattered at different locations of the program.

In recent two years, several new studies were found in the area of human aspects of code review. Wurzel et al. (2022) interviewed 22 developers about how they perceive and address interpersonal conflicts during code reviews. Gonçalves et al. (2022) conducted an experimental study to compare the novice reviewers' performances of ad hoc reviewing, checklist, and guided checklist, and found that guided checklist is a more effective aid for a simple review, but no statistically significant differences were found between these three types of reviews. Hijazi et al. (2022) proposed a method to assess whether lines of code are sufficiently reviewed and understood based on biometric measures such as Heart Rate Variability and task-evoked pupillary response.

2.2. Code reading techniques

Despite the progress made on diverse topics of modern code reviews, surprisingly, no systematic new code reading method was found in Davila and Nunes (2021)'s literature review nor our literature review conducted in January 2023, since the code reading methods summarized by (Aurum et al., 2002), including the ad-hoc reading, checklist reading, and stepwise abstraction (Aurum et al., 2002).

Ad hoc reading offers no procedure nor training for the reviewers. Reviewers use their own knowledge and experience to identify defects in the program or documents. Checklist reading is a procedure proposed by Fagan (1999). A checklist includes a list of questions on predefined issues derived from the project or the product itself, and reviewers answer the questions through inspecting the documentation or code. In stepwise abstraction reading (Basili and Selby, 1987), a reviewer first identifies subprograms in the software and determines their functions, then derives the specification and compares it to the official specification, thus inconsistencies can be identified.

Aurum et al. (2002) summarized two requirements reading methods: scenario-based reading (Porter et al., 1995) and perspective-based reading (Laitenberger and DeBaud, 1997), which focus on identifying flaws in requirements, rather than defects in code addressed in this paper.

3. HECR concepts and scope

Several new concepts are proposed for building the HECR method, and a couple of existing concepts are clarified as follows:

Defect: an incorrect or missing step, process, or data definition in a computer program (adopted from (IEEE, 1990)). Note that a defect in HECR has manifestation in the computer program, but it is not limited to

coding defect; it can also be a design defect or a defect originated during the process of understanding requirements.

Human 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 (Reason, 1990). 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 the cognitive weakness shared by all humans, e.g. applying "strong-but-now-wrong" rules (see Table 1) (Reason, 1990).

Error-Prone Scenario (EPS): A set of conditions under which a HEM tends to occur. An EPS in code inspection mainly includes the conditions around a programming task.

In addition to human errors by software developers, software defects can stem from a multitude of factors, including inappropriate requirements (which falls within the purview of Requirement Engineering), unsuitable organizational processes (the primary concern of Software Process Improvement), miscommunications among colleagues (emphasized in Engineering Management), and improper tool utilization (addressed in Human-Computer Interaction).

Table 1
A sample of Human Error Modes used in HECR.

Human Error Mode	Descriptions	Cognitive level
Perceptual confusion	People tend to mistakenly use one item for another because these two items share some similarities.	Skill-based
Omissions following Interruption	Original sequence of activity is picked up one or two steps further along after an interruption.	Skill-based
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. People tend to prefer rules that have been successfully used in the past, and apply them without noticing they are wrong in the new circumstances. The more frequently and successfully the rule has been used before, the more likely it is to be recalled and used.	Rule-based
Rule Encoding Deficiencies	Features of a particular situation are either not encoded at all or misrepresented in the conditional component of the rule (Reason, 1990).	Rule-based
Lack of knowledge	Software defects are introduced when one lacks knowledge, or even does not realize some other knowledge is required. This error mode is liable to appear especially when the problem belongs to an unfamiliar application domain.	Knowledge-based
Difficulties with exponential developments	Humans tend to underestimate the rate of change, either growth or decline, and tend to construct linear models, when exponential models are required to understand a situation in reality.	Knowledge-based
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, they may fail to take all conditions into consideration, e.g. exception and boundary conditions (Reason, 1990).	Knowledge-based
Post-completion Error (Byrne and Bovair, 1997)	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.	Meta-cognition

The scope of human errors focused by HECR (in this paper) include errors committed by individual developers, in three typical types of cognitive activities involved in software development: 1) understanding requirements provided that the requirements are correct, unambiguous, and consistent, 2) designing a program, and 3) coding. We provide examples of defects associated with each of these activities in Table 3 later in this paper.

It is essential to recognize that human errors can manifest at any stage of the software development lifecycle. For instance, during the requirement stage, an error by a requirement analyst can result in an inaccurate or ambiguous requirement specification, which, in turn, propagates into the design stage, eventually materializing as a design defect in the program. Miscommunications among software engineers can also lead to software defect (Huang et al., 2012). Our intention is to delve deeply into the mechanisms of cognitive errors made by individual developers. This interdisciplinary approach to code review is an area that has been underexplored in existing practices. Moreover, since individual developers are directly responsible for writing programs, their errors represent a primary category among all the potential causes of program defects.

4. The HECR method

The proposed method improves code review performances by assuring the reviewers understand the cognitive error mechanisms, and train them to identify the scenarios that tend to trigger developers to commit errors.

4.1. The components of HECR

The HECR method consists of a base Human Error Modes, the Error-prone Scenarios associated with these HEMs, and example defects for each of the HEMs and EPSs.

1) Human Error Modes

Rasmussen's performance framework classifies cognitive activities into three levels: Skill-based (SB) level, Rule-based (RB) and Knowledge-based (KB) level. These three levels have been used as a fundamental framework to study cognitive mechanisms of various human errors (Reason, 1990). We included the representative Human Error Modes for all of the three cognitive levels, as described in Table 1.

Furthermore, we include an error mode called "Post-completion Error", which was first discovered by Byrne and Bovair (Byrne and Bovair, 1997), and Huang found it prevalent in software development as well (Huang, 2016). Post-completion error does not necessarily belong to one of the three cognitive levels. Instead, it involves inappropriate organization of the overall cognitive process, or lack of evaluation on whether the goals of an episode of cognitive processes are achieved. Therefore, we classify it as a **metacognitive error:** lapses of monitoring and/or regulating one's own cognitive process. Metacognition is the"-cognition of cognition"(Huang and Liu, 2017).

2) Error-Prone Scenarios

Software development is a context-dependent activity: a developer could commit diverse errors on different given requirements, while different developers could commit diverse errors on the same given requirement. Meanwhile, psychologists find that human errors take a limited number of patterns across diverse contexts (Reason, 1990).

"Error-Prone Scenario" (EPS) serves as a "bridge" which allows us to transfer the general understanding of human errors in psychology to software engineering contexts. A sample of the EPSs are shown in Table 2. Each EPS models the conditions that tend to trigger a HEM, and

Table 2
A sample of the Error-Prone Scenarios for code review.

Human Error Mode	Error-Prone Scenario
Perceptual Confusion	***IF*** *Current task requires* object *A* <Appearance A, Function A, Location *A*>; ***WHEN*** *There Exists* object B ⟨ Appearance B, Function B, Location B ⟩, ***AND*** {Appeance A ∩ Appeance B} >> ∅, ***OR*** {Function A ∩ Function B} >> ∅, ***OR*** {Location A ∩ Location B} >> ∅; ***THEN*** The person may use/apply/fetch object B.
Applying"strong-but-now-wrong" Rule	***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 ∩ FeB} ⊇ {FeX ∩ FeA} ≠ ∅; FuA>> FuB, ***OR*** Fu ((FeX ∩ FeB)$_i$)=0, ***OR*** FeB⊂FeA; ***THEN*** The person tends to retrieve Rule A.
Rule Encoding Deficiencies	***IF*** *Current task requires* Rule *X* <Feature FeX >; ***WHEN*** *There Exists* Rule $\widetilde{X}$<Feature Fe$\widetilde{X}$ >; FeX − Fe$\widetilde{X}$ ≠ ∅; ***AND*** The person uses Rule $\widetilde{X}$; ***THEN*** The person commits an error of misusing Rule $\widetilde{X}$ for the current situation that requires Rule X.
Lack of knowledge	***IF*** *Current task requires* Rule X; ***WHEN*** Rule X *does not exist*; ***THEN*** The person tends to fail the task.
Difficulties with Exponential Developments	***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 " $y = a^x$"; ***THEN*** People tend to construct wrong models in the family of"$y = ax$".
Biased Review	***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	***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.

then represent them in pseudo codes. For instance, the preconditions are specified by"IF", the situations under which an error tends to occur are specified by "WHEN", and the final manifestation of the error is specified after "THEN". Notations such as "AND" and "OR" are used to combine multiple situations.

3) Example defects

We build an initial data set of example defects, and corresponding programs and requirements. Such examples are used for demonstrating (and training code reviewers on) how to map EPSs to the detailed defects in computer programs. These initial examples are adopted from a program called"Jiong" problem and another program called "Fractal Problem", with the requirements and programs provided in Appendix A–D.

The defects in the "Jiong" task are the real defects introduced by multiple programmers (Huang and Strigini, 2023). A defect in the Fractal program is similar to the defect in the Jiong program with the same identification number. For instance, defect "F1" mimics J1 in Table 3. J1 is an error that a programmer uses the symbol"!" instead of "|" to print an image called "jiong" word, while the human error mode behind this defect is "perception confusion" because "!" looks like "|". Similarly, F1 is an error that a programmer uses '0′ for 'o', which should be printed in the outputs of the Fractal program, shown in Appendix D.

4.2. The process of HECR

The main purpose of HECR is to educate code reviewers on Human Error Modes (HEM) and Error-Prone Scenarios (EPS) and then train them in identifying EPS in source code, thereby enhancing their skills in detecting defects during code reviews. **The training should be repeated with different source codes to ensure that the reviewers consolidate their skills using the proposed approach.** HECR comprises three stages: 1) Knowledge training on HEMs and EPSs, 2) Practicing the identification of EPS in a program, and 3) using HECR to **identify defects in another program with distinct functionality and requirements**.

1) Knowledge Training of HEMs and EPSes: Provide code reviewers with the knowledge of "why" software developers commit errors (Human Error Modes), "what" general conditions tend to trigger a human error (Error-prone Scenarios), and "how" a human error interact with a development context to form a defect (Error-prone

Table 3
The example defects and corresponding HEMs.

Program	Defect ID	Defect Description	Defect Type	Human Error Mode (s)
Jiong	J1	Mistaking '!' for '|'	Coding	Perception Confusion
	J2	'0′ should be' ';	Coding	Strong-but-now-wrong
	J3	N 100 should be N 512 or more;	Coding	Rule encoding deficiencies
	J4	The relationship between the height and nest level is modeled wrongly as t = 8n-2, instead of $t = 2^{n+2}$–2	Design	Difficulties with exponential developments
	J5	missed printf("\n")	Omitting Requirement	Post-completion error
	J6	In the"for" loop for calculating each "jong", t = 8n-2 should be $t = 2^{level+2}$–2	Design and Coding	Difficulties with exponential developments and Strong-but-now-wrong
	J7	map[di][dj+j]=2 should be map[di][dj+j]='-'	Coding	Omissions following interruption
Fractal	F1	Mistaking '0′ for 'o'	Coding	Perception Confusion
	F2	o[x][y]='o'; should be initialized with ' ' instead of 'o'	Coding	Strong-but-now-wrong
	F3	array size should >=729	Coding	Rule encoding deficiencies
	F4	w = 6*n-9; should be pow(3,n-1)+1e-5	Design	Difficulties with exponential developments
	F5	h = 6*n-9; should be pow(3,n-1)+1e-5	Design	Difficulties with exponential developments
	F6	Should not print spaces at the end of each line	Omitting Requirement	Post-completion error
	F7	Defined a variable int K never used	Coding	Omissions following interruption

Scenario Identification). The materials used in the knowledge training are those presented in Section 4.1. In addition to the concepts, the training focused on how to map a program context to an Error-Prone Scenario, which was done through diagram examples shown in Fig. 1.

2) Practicing identifying Error-Prone Scenarios: The skills of identifying defects are gained through practices. We provide a set of example programs associated with their requirements (described in Sections 4.1–3) for the code reviewers to practice. The practice includes three steps:
   a) Requirements (or functionality) explanation: the trainer explains the requirements or the functionalities of the program to be inspected. This is a step to make sure the code reviewers understand what the program is expected to"do".
   b) Identifying defects with EPS in mind: The code reviewers are trained to identify defects contained in the program, with the HEMs and EPSs in mind. The emphasis here is to find the conditions in the program that form an EPS.
   c) Disclosing the defects and EPSs: Once the code reviews are finished, the trainer discloses the defects contained in the program, and reviews the HEMs underlying them. This step is expected to reinforce the reviewers' understanding and awareness of HECR. An example slide is shown in Fig. 2.
3) HECR Application: Once the knowledge training HEMs and EPSes and the practices of EPS identification are completed, the code reviewers apply HECR on a new code review task. The HEMs are provided as a checklist, so in addition to Ad hoc reading, an code reviewer can direct some of his/her attention to those locations that tend to be error-prone, based on our knowledge of developers' human errors that are derived from the general cognitive error patterns discovered by psychologists (Byrne and Bovair, 1997; Reason, 1990).

4.3. Metrics

We propose a set of metrics for evaluating the effectiveness of HECR, defined as follows.

1) True Positives (TP): The defects identified by a code reviewer that are in fact real defects existing in the program under review. We call the number of those defects TP. We use $\boldsymbol{TP_n}$ to denote the number of true defects identified by code reviewer $\boldsymbol{n}$. The TP of the HECR on task $\boldsymbol{t}$ is then estimated by the average of $\boldsymbol{TP_{t,\,n}}$ in (1):

$$\mathrm{TP}_t = \frac{\sum_n TP_{t,n}}{N} \times 100\% \tag{1}$$

2) False Positives (FP): Defects identified by a code reviewer that do not correspond to real defects in the program under review. We call their number FP. FP is also important, since they would encourage effort to be spent in confirming and fixing defects that did not occur. We use $\boldsymbol{FP_n}$ to denote the number of false positives identified by code reviewer $\boldsymbol{n}$. The FP of the HECR on a task $\boldsymbol{t}$ is then estimated by the average of $\boldsymbol{FP_{t,\,n}}$ in (2):

$$\mathrm{FP}_t = \frac{\sum_n FP_{t,n}}{N} \times 100\% \tag{2}$$

3) Precision: The proportion of correctly identified defects out of all the identified defects. The Precision achieved by code reviewer $\boldsymbol{n}$ on program $\boldsymbol{t}$ is calculated in (3):

$$\mathrm{P}_{t,n} = \frac{TP_{t,n}}{TP_{t,n} + FP_{t,n}} \times 100\% \tag{3}$$

The precision of HECR on program t is then estimated by the average of the precisions achieved by all the code reviewers, in (4):

$$\mathrm{P}_t = \frac{\sum_n P_{t,n}}{N} \times 100\% \tag{4}$$

1) Sensitivity: The proportion of defects that are correctly identified out of all the defects contained in a program. We use $\boldsymbol{S_n}$ to denote the sensitivity achieved by code reviewer $\boldsymbol{n}$. Sensitivity reflects the extent to which the ways of human errors are predicted. The Sensitivity of the HECR method on task t is then estimated by the average of $\boldsymbol{S_n}$ in (5):

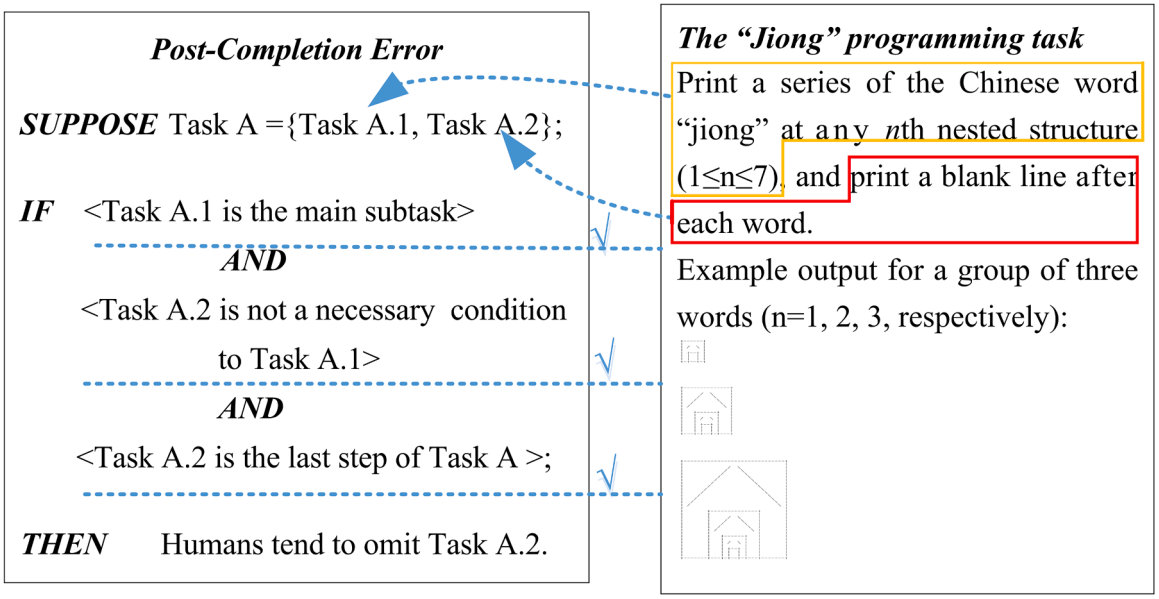


Fig. 1. An example of how to map program contexts to Error-Prone Scneario.

```
while(T--)
{
   scanf("%d",&n);
   memset(map,'0',sizeof(map));    // * '0' should be ' '      First exception, strong-but-now-wrong
   di=0,dj=0;
   for(level=n;level>=0;level--)
```

Fig. 2. An example snappet of code used during at the time of disclosing the defects (HECR training Step 2-c).

$$S_t = \frac{\sum_n S_{t,n}}{N} \times 100\% \quad (5)$$

where $S_{t,n}$is the proportion of defects found by code reviewer n($DF_{t,n}$) out of all the defects containing in the program t (D_t), shown in (6)

$$S_{t,n} = \frac{DF_{t,n}}{D_t} \times 100\% \quad (6)$$

1) Improvement of a code review performances (IMP): the performance of reviewers on the second task (t_2) using HECR compared to that of Ad hoc review on the initial task (t_1). The improvement can be calculated in all of the four dimensions defined above:

Improvement on True Positives:

$$IMP_{TP} = TP_{t_2} - TP_{t_1} \quad (7)$$

where positive IMP_{TP} indicates HECR is effective, or vice versa.

Ratio of Improvement on True Positives ($RIMP_{TP}$) is then calculated in (8):

$$RIMP_{TP} = \frac{TP_{t_2} - TP_{t_1}}{TP_{t_1}} \times 100\% \quad (8)$$

Improvement on False Positives:

$$IMP_{FP} = FP_{t_2} - FP_{t_1} \quad (9)$$

where negative IMP_{FP} indicates HECR is effective, or vice versa.

Ratio of Improvement on False Positives is then calculated in (10):

$$RIMP_{FP} = \frac{FP_{t_2} - FP_{t_1}}{FP_{t_1}} \times 100\% \quad (10)$$

Improvement on Precision is defined in (11):

$$IMP_P = P_{t_2} - P_{t_1} \quad (11)$$

where positive IMP_P indicates HECR is effective, or vice versa.

Ratio of Improvement on Precision is then calculated in (12):

$$RIMP_p = \frac{P_{t_2} - P_{t_1}}{P_{t_1}} \times 100\% \quad (12)$$

Improvement on Sensitivity is defined in (13):

$$IMP_S = S_{t_2} - S_{t_1} \quad (13)$$

where positive IMP_S indicates HECR is effective, or vice versa.

Ratio of Improvement on Sensitivity is then calculated in (14):

$$RIMP_S = \frac{S_{t_2} - S_{t_1}}{S_{t_1}} \times 100\% \quad (14)$$

5. Experimental study

This study was designed as a controlled experiment (Wohlin et al., 2012), in which we compared the code review performances of participants who received HESR training (Experimental Group) to that of participants who did not receive the training (Control Group). Participants in the Experimental Group and the Control Group have similar backgrounds in related experience and skills. Detailed designs and considerations are provided in Sections 5.1–5.3.

5.1. Participants

5.1.1. Considerations for selecting participants

We were fully aware that code review performance can be affected by the experience of the reviewer, in both programming and code review aspects. On one hand, the participants should have some experience in programming because they need to read and understand the programs. On the other hand, minimum code review experience is preferred in this experiment, because the participants' previous code review experiences would affect the outcomes of the experiment, as a threat to internal validity (Wohlin et al., 2012) (detailed discussion in Section 7). Finally, we were most interested in the professional software developers at entry level, because these developers need training on code reviewing the most.

Therefore, our targeted participants are software developers who have considerable experience in programming, but no or little experience in code reviewing. Meanwhile, it was difficult to find tens of professional developers at entry level to fully commit to this experiment. We considered graduate students in computer science would be the most suitable and feasible participants for this study, as they have considerable programming experience but little code review experience. Detailed information about the participants of this study is provided below in Section 5.1.2.

5.1.2. The participants of the experiment

A total of 49 graduate students majoring in Computer Science participated in the experiment. The participants were naturally divided into two groups (Group A and B) based on their class schedule.

There were 31 participants in Group A, among which 2 students attended only one of the two weeks. Therefore, the remaining 29 participants who attended both two weeks constituted Group A.

Group B initially had a total of 18 participants, among which 5 participants only attended in Week 2 (evaluation sessions), and 2 participants attended only in Week 1. Therefore, we have 11 participants.

In summary, we have a total of 40 participants who attended the completed process of the experiment. We have asked all participants to fill out a survey at the beginning of the experiment. The survey, shown in Appendix E, included questions to collect their experience in C/C++ development and code review, along with an ethical question asking whether they consent to participate in this study, and allow us to include their anonymized data for publications, following the ethical procedures of our organization.

Based on the survey data, we performed independent Sample t tests, results show that no significant differences between the two groups on average C/C++ experience, (A = 18.0 months, B = 18.6 months, t=−0.20, p = 0.84) and average code review experience (Average A = 1.9 months, Average B = 1.8 months, t = 0.28, p = 0.79).

All the participants had never been exposed to the programs used in the experiment of this paper.

5.2. The tasks

We used two tasks in the experiment: the Jiong task and the Fractal task. The requirements for the Jiong task is shown in Appendix A.

The "Jiong" task was originally used by Huang (the first author of this paper) and her collaborators (Huang and Strigini, 2023) to investigate whether and how human error modes manifest as software defects in programming. The advantage of using the "Jiong" task is that it contains real defects introduced by programmers in the previous experiment (Huang and Strigini, 2023), and to our best knowledge, it is the only dataset that has been validated in terms of mapping specific human error modes to specific defects in the program (Huang and Strigini, 2023).

The "Jiong" program (shown in Appendix B) used in this paper's study was a version adapted from a submission generated by a programmer in [2] and integrates 7 defects into a single program.

The Fractal problem was designed specifically for this study, with its requirements and program of the Fractal problem are shown in Appendix C and D, respectively.

We consider these two programs' complexity to be at the same level, but the Fractal program (Lines of code=86, Cyclomatic Complexity=16) is more complex than the Jiong program (Lines of code=55, Cyclomatic Complexity=10). As the tasks for a controlled experiment which is limited by time, the sizes of both programs were not large, but they are comparable to a"function" programmers develop and submit for code review in agile programming.

The whole set of defects embedded in these two programs are summarized in Table 3.

5.3. Experiment procedures

The experiment was dedicatedly designed to evaluate HECR, allowing us not only to observe the effects of HECR on a group of code reviewers longitudinally, but also to compare the effects between Experimental Group and Controlled Group. The experiment consisted of 8 sessions, in a span of two weeks, as illustrated in Fig. 3.

We had two independent participant groups (Participants A and B) and two tasks (Jiong task and Fractal Task). Because the two participants groups were independent from each other, and the evaluations were made on different tasks; the experiment of this study was essentially an integration of two"conventional" controlled experiments:

Experiment I (in the lower left and upper right of Fig. 3, in the scope of red lines): Participants A as the Experimental Group evaluated on the Jiong task (after HECR training); Participants B as the Controlled Group on the Jiong task (without HECR training).

Experiment II (highlighted in pattern background in Fig. 3): Participants B as the Experimental Group evaluated on the Fractal task (after HECR training); Participants A as the Controlled Group on the Fractal task (without HECR training).

The integrated design of this experiment allowed us to make full use of the data. It also allows us to investigate the effects of HECR treatment in more comprehensive perspectives, i.e. whether the HECR effects can be reproduced on different tasks.

The two participants Groups followed the same detailed procedures, but were separated from each other. The differences lay in: Group A used the Fractal program, its defects and requirements for the **E**ntry Ad hoc code review and **p**ractices (E&P program in Fig. 3) and used the Jiong program and its defects and requirements for the HECR code review (Evaluation program in Fig. 3); Participants Group B used Jiong as the

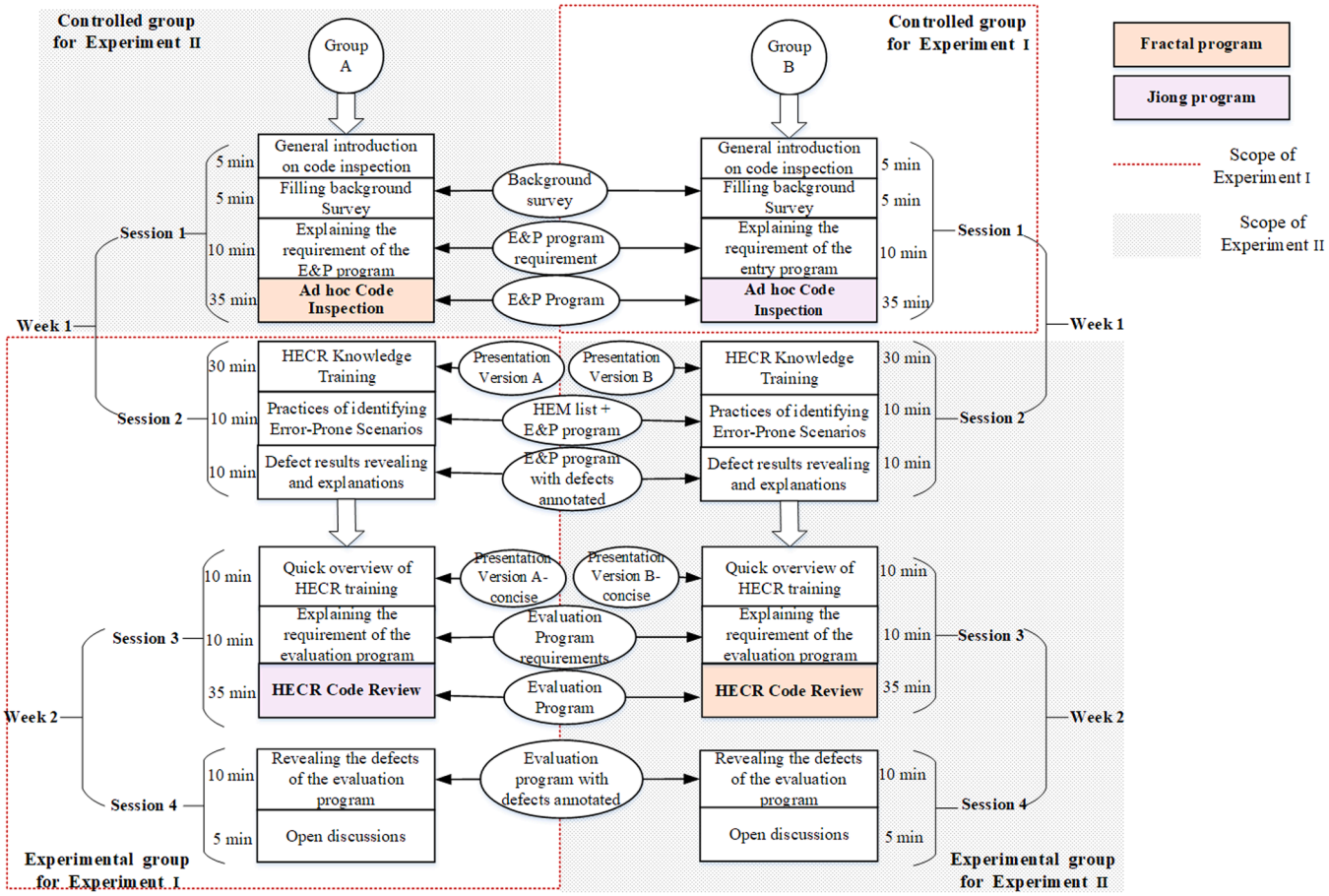


Fig. 3. The procedures of the experiments.

E&P program, while used Fractal program as the evaluation task for HECR code review.

Each experiment consisted of two weeks, and each week included two sessions. In Week 1, we did 35 minutes' Ad hoc code review (the focus of Session 1), and then, performed 50 min' HECR training (Session 2): knowledge training for 30, and practices of identifying EPSs on the "E&P program" for 10 min, and revealing and discussing the defects for 10 min.

Week 2 focused on applying HECR on a new task. Week 2 consisted of two sessions. In Session 3, we first quickly reviewed the HEMs and EPSs for 10 min, then took 10 min to explain the requirements for the Evaluation program, followed by HECR code review on the Evaluation program. After finishing the code reviews and collecting the results, we revealed the defects contained in the evaluation program in Session 4, and followed by a promoted discussion. Note that Session 4 was not necessary to the experiment of this paper, but it was designed for educational purposes.

6. Data and results analysis

The dedicated design of the experiment allows us to collect code review data on two tasks for two groups of participants independently. We analyze the data from multiple perspectives: first, we compare the code review performances after HECR training to that of the entry Ad hoc review for Group A and Group B, in Sections 6.1 and 6.2, respectively; second, we compare whether there are significant differences between these two groups, in terms of performance improvement in Section 6.3. Third, we perform the statistical tests on the effectiveness of HECR in Section 6.4, comparing the code review performances of an experimental group to the controlled group. Finally, we summarize the effects of HECR in Section 6.5.

6.1. Effects of HECR on group A

We evaluate the effects of HECR by comparing the HECR code review performances on the evaluation program t_2 (at the end of the experiment), to the Ad hoc code review performances on the entry program (t_1) performed at the beginning of the experiment. The statistics of the code review data on both programs for Group A participants are shown in Table 4.

The results (Table 4) show that Group A performed much better using HECR than Ad hoc code review: True Positives, Precision and Sensitivity were all largely improved, while False Positives were slightly reduced.

We further analyzed whether these effects were significant by Paired Samples t tests. Paired samples *t*-test is an appropriate statistical test here, as it is suitable to examine the effects of an intervention by comparing the means of two measurements (pre-intervention and post-intervention) from the same group of people. The statistical hypotheses are as follows:

H1.A. The TP of HECR is the same as that of Ad hoc code review for Group A.

Table 4
The statistics of the code review performances for Group A.

Metrics[a]	Mean	Min.	Max.	Std. Deviation	IMP	RIMP
TP_{t_1,n_A}	0.79	0	3	0.98	2.93	371 %
TP_{t_2,n_A}	3.72	1	6	1.22		
FP_{t_1,n_A}	2.07	0	6	1.76	−0.14	−6.8 %
FP_{t_2,n_A}	1.93	0	4	1.33		
P_{t_1,n_A}	25 %	0 %	75 %	30 %	44 %	176 %
P_{t_2,n_A}	69 %	33 %	100 %	19 %		
S_{t_1,n_A}	11 %	0 %	43 %	14 %	42 %	382 %
S_{t_2,n_A}	53 %	14 %	86 %	17 %		

[a] t_1 is the "Fractal" program; t_2 is the "Jiong" program.

H2.A. The FP of HECR is the same as that of Ad hoc code review for Group A.

H3.A. The Precision of HECR is the same as that of Ad hoc code review for Group A.

H4.A. The Sensitivity of HECR is the same as that of Ad hoc code review for Group A.

The results of the statistical tests (shown in Table 5) show that HECR has significantly improved the code review performances of Group A: increased True Positives by 371% ($p \leq 0.01$), increased Precision by 176 %, and increased Sensitivity by 382 % ($p \leq 0.01$). HECR has slightly decreased False Positives by 6.8 %, but the effect was not statistically significant.

6.2. Effects of HECR on group B

The statistics of the code review data on both programs for Group A participants are summarized in Table 6.

The results in Table 6 show that Group B also performed much better using HECR than Ad hoc code review: True Positives, Precision and Sensitivity were all largely improved, while False Positives were also significantly reduced.

Following the same analysis methods used for Group A, we performed Paired Samples t tests to exam the statistical hypotheses:

H1.B. The TP of HECR is the same as that of Ad hoc code review for Group B.

H2.B. The FP of HECR is the same as that of Ad hoc code review for Group B.

H3.B. The Precision of HECR is the same as that of Ad hoc code review for Group B.

H4.B. The Sensitivity of HECR is the same as that of Ad hoc code review for Group B.

The statistical tests (in Table 7) show that HECR has significantly improved the code review performances of Group B: increased True Positives by 425 % ($p \leq 0.01$), increased Precision by 400 %, increased Sensitivity by 433 % ($p \leq 0.01$), and decreased False Positives by 70 % (($p \leq 0.05$).

6.3. Performance differences between tasks and groups

We were interested in whether HECR affected Group A and B differently, because these two groups switched the entry and evaluation programs. Intuitively, the differences of difficulty between the two evaluation programs can also affect the improvement ratio. As we stated before, the Fractal program, used as the evaluation program for Group B, is considered a little more difficult than the Jiong program, the evaluation program for Group A.

Since HECR is based on the general human error mechanisms, if the method is reliable enough, it should allow code reviewers to reuse it across different tasks and achieve considerable consistent benefit. On the other hand, if HECR is not robust enough, the effects of the training can be easily counteracted by a more difficult code review task.

Table 5
Results of the statistical tests on the effects of HECR for Group A.

Hypothesis		Paired Samples *t* Test				
		Pairs	*Differences (Mean)*	*t*	*df*	*p-value (2-tailed)*
H1.A	Rejected	TP2 - TP1	2.93	10.80	28	0.00**
H2.A	Retained	FP2 - FP1	−0.14	−0.27	28	0.79
H3.A	Rejected	P2 - P1	44 %	6.58	28	0.00**
H4.A	Rejected	S2 - S1	42 %	10.83	28	0.00**

** statistically significant: $p \leq 0.01$.

Table 6
The statistics of the code review performances for participants in Group B.

Metrics[a]	Mean	Min.	Max.	Std. Deviation	IMP	RIMP
TP_{t_1,n_B}	0.64	0	2	0.81	2.72	425 %
TP_{t_2,n_B}	3.36	1	6	1.43		
FP_{t_1,n_B}	3.00	0	8	1.95	−2.10	−70 %
FP_{t_2,n_B}	0.91	0	3	1.14		
P_{t_1,n_B}	16 %	0 %	50 %	20 %	64 %	400 %
P_{t_2,n_B}	80 %	40 %	100 %	24 %		
S_{t_1,n_B}	9 %	0 %	29 %	12 %	39 %	433 %
S_{t_2,n_B}	48 %	14 %	86 %	20 %		

[a] t_1 is the "Fractal" program; t_2 is the "Jiong" program.

Table 7
Results of the statistical tests on the effects of HECR for Group B.

Hypotheses		Paired Samples t Test				
		Pairs	*Differences (Mean)*	*t*	*df*	*p-value (2-tailed)*
H1.B	Rejected	TP2 - TP1	2.73	5.39	10	0.00**
H2.B	Rejected	FP2 - FP1	−2.10	−3.07	10	0.01**
H3.B	Rejected	P2 - P1	64 %	6.83	10	0.00**
H4.B	Rejected	S2 - S1	39 %	5.39	10	0.00**

** statistically significant: $p \leq 0.01$.

The cross-sectional analysis was intended to compare the improvements between the two independent samples: Group A and Group B. The statistics of the improvements in TP, FP, Precision and Sensitivity are summarized in Table 8.

We further performed independent Samples t tests to compare whether the HECR observed on Group A and Group B are different.

H5. The performance improvement of TP is the same between Group A and Group B.

H6. The performance improvement of FP is the same between Group A and Group B.

H7. The performance improvement of Precision is the same between Group A and Group B.

H8. The performance improvement of Sensitivity is the same between Group A and Group B.

The results of the statistical tests are shown in Table 9.

Results show that improvements achieved by HECR for Group A and B were not significant in terms of True Positives, Precision and Sensitivity. Surprisingly, Group B, who performed HECR on a more difficult program, even performed significantly better in terms of reducing False Positives. These results suggest that HECR is very robust, allowing different people to replicate the same effects on a different task, and even a slightly more difficult task.

6.4. The effectiveness of HECR

Because the controlled group and experimental group in each experiment are independent, independent sample *t*-test is a suitable statistical test. We compared the differences between the code review performances of people had received HECR training (the experimental group) to that of people who had not (controlled group), for each experiment.

Table 8
The Statistics of the Improvements for both Groups.

Metrics	Group	*N*	Mean	Std. Deviation
IMP_TP	A	29	2.93	1.46
	B	11	2.73	1.68
IMP_FP	A	29	−0.10	2.08
	B	11	−2.09	2.26
IMP_Precision	A	29	44%	36%
	B	11	64%	31%
IMP_Sensitivity	A	29	42%	21%
	B	11	39%	24%

Table 9
The statistical test results for the differences of improvements between Group A and B.

Hypotheses		*t*	*df*	Mean Difference	*p*-value (2-tailed)
H5	Retained	0.36	16.11	0.20	0.73
H6	Rejected	2.54	16.83	1.99	0.02*
H7	Retained	−1.70	21.02	−20 %	0.10
H8	Retained	0.36	16.11	3 %	0.73

*Statistically significant: $p \leq 0.05$.

The tests are according to the hypotheses as follows:

H9. The TP of HECR is the same as that of Ad hoc code review, including:

- H9.I. The TP of the Experimental Group is the same as that of the Controlled Group in Experiment I.
- H9.II. The TP of the Experimental Group is the same as that of the Controlled Group in Experiment II.

H10. The FP of HECR is the same as that of Ad hoc code review, including:

- H10.I. The FP of the Experimental Group is the same as that of the Controlled Group in Experiment I.
- H10.II. The FP of the Experimental Group is the same as that of the Controlled Group in Experiment II.

H11. The Precision of HECR is the same as that of Ad hoc code review, including:

- H11.I: The Precision of the Experimental Group is the same as that of the Controlled Group in Experiment I.
- H11.II: The Precision of the Experimental Group is the same as that of the Controlled Group in Experiment II.

H12. The Sensitivity of HECR is the same as that of Ad hoc code review, including:

- H12.I. The Sensitivity of the Experimental Group is the same as that of the Controlled Group in Experiment I.
- H12.II. The Sensitivity of the Experimental Group is the same as that of the Controlled Group in Experiment II.

The results of the statistical tests for the effectiveness of HECR are shown in Table 10. The results show that the code reviewers who received HECR training and using HECR perform significantly better than those who did not. The decrease of False Positives is consistently

Table 10
The statistical test results for the Effectiveness of HECR (Independent sample T tests).

Hypotheses		*t*	*df*	Mean difference	*p*-value (2-tailed)
H9.I	Rejected	7.73	38	3.09	0.00**
H9.II	Rejected	6.51	38	2.57	0.00**
H10.I	Rejected	−1.99	38	−1.07	0.05*
H10.II	Retained	−1.96	38	−1.13	0.06
H11.I	Rejected	7.65	38	53%	0.00**
H11.II	Rejected	5.49	38	55%	0.00**
H12.I	Rejected	7.73	38	44%	0.00**
H12.II	Rejected	6.51	38	37%	0.00**

** Statistically significant: $p \leq 0.01$.
* Statistically significant: $p \leq 0.05$.

observed in both experiments; this effect has reached statistical significance ($p \leq 0.05$) in Experiment I, while it slightly missed the statistical significance in Experiment II ($p = 0.06$). This could be due to the sample size of the controlled group in Experiment II ($N = 11$) is smaller than that of Experiment I ($N = 29$).

The effectiveness is consistent in all other three aspects of code review performance metrics, including the increase of True Positives, and increase of Precision and Sensitivity, in both experiments. Overall, the tests suggest that HECR was very effective in improving coder reviewers' performances.

6.5. Summary of the overall effects of HECR

Since the positive effects of HECR on code review performances are consistently observed in Group A and B, we can integrate the data of the two groups, and provide an overview on the effects of HECR, by averaging the improvements achieved by all the 40 participants in this study, shown in Table 11, and the differences of means further illustrated in Fig. 4.

Overall, compared to Ad hoc code review, HECR has improved all the aspects of code review performances: increased True Positives by 384 %, decreased False Positives by 29 %, increased Precision by 217 %, and increased Sensitivity by 373 %.

7. Discussions

7.1. Cost-effectiveness

HECR requires the code reviewers to receive training on Human Error Modes, and on identifying Error-prone Scenarios in software code. In our experiment, the training session was 50 min, which has achieved adequate effects. The knowledge and skills gained in one time of training that is measured by hours could benefit many projects in a code reviewer's daily work activities. Most importantly, the extra defects found by HECR in modern code review (usually used as the initial verification strategy) could significantly reduce the cost of finding the defects at later verification & validation stages or the harm caused by the defects remaining in the software in operation. This is especially significant for safety-critical software, for which a residual defect could lead to a catastrophic accident.

An important advantage of HECR is that the method can be adapted to the different flavors of modern code reviews used by the software industry without requiring changes in the established code reviewing practices of the companies. HECR relies on reviewers' training on the Human Error Modes and the identification of Error-Prone Scenarios in code, without requiring any further change in the code reviewing workflow, which makes the future adoption of HECR by software development companies easy and cost effective.

7.2. Threats to validity

We discuss four types of validity concerns that apply to experimental studies in Software Engineering: conclusion validity, internal validity, external validity and construct validity (Wohlin et al., 2012). **Construct validity** concerns the degree to which the specific variables of a study represent the intended constructs in the conceptual or theoretical model. **Conclusion validity** refers to the degree to which conclusions about the relationship among variables based on the data are reasonable. We were not aware of any threat to construct validity and conclusion validity.

Table 11
The overall effects of HECR.

Metrics [a]	Min.	Max.	Std. Deviation	IMP	RIMP
TP_{Adhoc}	0	3	0.93	2.88	384 %
TP_{HECR}	1	6	1.27		
FP_{Adhoc}	0	8	1.84	−0.67	−29 %
FP_{HECR}	0	4	1.35		
P_{Adhoc}	0 %	75 %	28 %	50 %	217 %
P_{HECR}	33 %	100 %	21 %		
S_{Adhoc}	0 %	43 %	13 %	41 %	373 %
S_{HECR}	14 %	86 %	18 %		

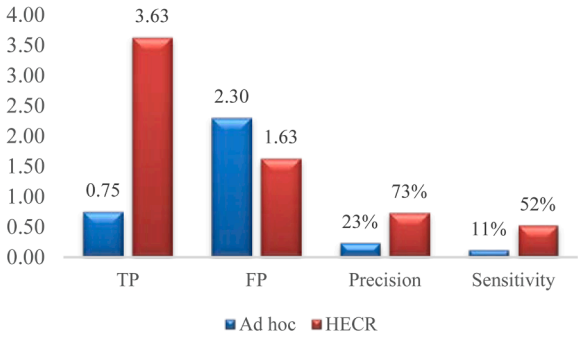


Fig. 4. The average performance improvement of HECR vs. Ad hoc review.

Internal validity concerns the extent to which other factors, besides the investigated factors, affect the outcomes. We were well aware of two factors that may threaten internal validity: 1) the variety in the expertise levels of the code reviewers, and 2) the differences of difficulty levels between the entry task and evaluation task. We minimized these threats by a dedicated design of the experiment procedures. To minimize the first threat, the participants we recruited to the experiment were at the same expertise level with similar background and experience in C/C++ development. The second threat was addressed by switching the entry and evaluation tasks between the two independent groups of participants, and making the comparison between these two groups. This approach excluded the possibility of observing a code-review improvement due to a less difficult evaluation task.

External validity concerns the extent to which it is possible to generalize findings and to what extent the findings are of interest to people outside the experiment. The participants in this study were limited to developers who had around 5 years' study in computer science, and had around 18–19 months' software development experience in C/C++, and modest experience in code review. Therefore, we expect the results of the study are extendable to developers at similar expertise level, i.e. upper-level students in computer science and/or entry-level professional developers. The extent to which HECR benefits people beyond this expertise range requires more future studies.

7.3. Future studies

The first future study is to investigate the effects of HECR on professional software engineers at intermediate and high expertise levels. These engineers have more experience in code review, so they could perform better in the ad hoc code review. Meanwhile, they would probably benefit from HECR as well, because HECR provides them with new perspectives and techniques that have not been covered by any existing software engineering practices.

The second future study is to extend the HECR method for larger size of programs. In this paper, HECR was evaluated by a controlled experiment. A good control experiment has the advantage of minimizing threats to internal, construct and conclusion validities by an appropriate experimental design, while it is limited by time, thus the size of the tasks. The programs used in our experiments are comparable to a function developed by a programmer in daily basis commonly encountered in agile programming. Our next future study is to investigate the application of HECR on large-scaled code review using empirical methodologies.

8. Conclusions

This paper proposes a new code review method called **H**uman **E**rror-based **C**ode **R**eview (HECR). The method takes advantage of the fact that humans tend to make errors according to a limited number of cognitive error patterns called Human Error Modes (HEMs). HECR improves the effectiveness of finding defects by training code reviewers to identify conditions in the code that tend to match error-prone scenarios associated with HEMs. The proposed method was validated by a comprehensive controlled experiment. Results show that HECR significantly improves various measures of code review effectiveness, particularly it improves the number of correctly found defects by nearly 400 %, compared to the most widely used modern code review method. HECR can be used by any variant of modern code reviews, as the method does not require any change in the reviewing process (it only needs specific training to the reviewers), which makes HECR quite easy to be adopted by the software industry.

CRediT authorship contribution statement

Fuqun Huang: Conceptualization, Data curation, Formal analysis, Investigation, Methodology, Project administration, Resources, Software, Supervision, Validation, Visualization, Writing – original draft, Writing – review & editing. **Henrique Madeira:** Data curation, Investigation, Methodology, Project administration, Resources, Writing – review & editing.

Declaration of competing interest

The authors declare that they have no known competing financial interests or personal relationships that could have appeared to influence the work reported in this paper.

Data availability

Data will be made available on request.

Acknowledgment

This work was supported in part by the Grant CISUCUID/CEC/00326/2020, funded in part by the European Social Fund, through the Regional Operational Program Centro 2020. We are especially thankful to all the participants of the study.

Appendix A. The requirements for the “Jiong” problem

Print a Chinese word “Jiong” in a nested structure.

Inputs

There is an integer in the first line that indicates the number of input groups.

Each input group contains an integer n ($1 \leq n \leq 7$).

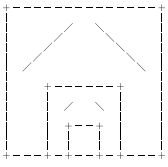
Outputs

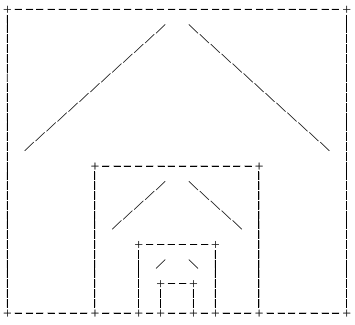
Print a word “jiong” after each input group, and then print a blank line after each “jiong” word.

Sample Inputs

3
1
2
3

Sample Outputs





Appendix B. The source code of the "Jiong" program for the code review

```
#include<stdio.h>
#include<stdlib.h>
#include<string.h>
#include<math.h>
#define N 100 //* should be N 512 or more
char map[N][N];
void out(int n)
{
  /* the variable i for row, j for column, t for height*/
  int i,j,t;
  t=8*n-2; //* should be: t=int(pow(2.0,n+2)-2)
  for(i=0;i<=t+1;i++,printf("\n"))
    for(j=0;j<=t+1;j++)
    printf("%c", map[i][j]);
}
int main()
{
  /*n is the nesting level of the "jiong" given by the user; T is the total number of input groups; t is for height; "level" is the nesting level of a "jiong" that constitutes the "jiong" at n^th nesting level; "di" and "dj" constitute the array index to indicate the location of the upper left "+" symbol at a nesting "level" .*/
int i,j,n,T,t,di,dj,level;
  scanf("%d",&T);
while(T--)
{
  scanf("%d",&n);
  memset(map,'0',sizeof(map)); // * '0' should be ' '
  di=0,dj=0;
  for(level=n;level>=0;level--)
  {
    t=8*n-2; //* should be t=int(pow(2.0,level+2)-2);
    map[di][dj]='+';
    for(j=1;j<=t;j++) {
    map[di][dj+j]=2;map[di][dj+t+1]='+'; //* map[di][dj+j]=2 should be map[di][dj+j]='-'
    }
    for(i=1;i<=t;i++) {
    map[di+i][dj]=map[di+i][dj+t+1]='!'; //* '!' should be '|' perceptual confusion
    }
    map[di+t+1][dj]=map[di+t+1][dj+(t+1)]='+';
    for(j=1;j<=(t/2+1)/2;j++){
    map[di+t+1][dj+j]=map[di+t+1][dj+(t+1)-j]='-';
    }
    if(level!=0)
    {
      for(i=2;i<t/2;i++)
      {
        map[di+i][dj+t/2+1-i]='/';
        map[di+i][dj+t/2+i]='\\';
      }
      di=di+t/2+1;
      dj=dj+(t/2+1)/2;
    }
    else {
    break;
    }
    }
    out(n); //* missed printf("\n")
  }
  return 0;
}
```

Appendix C. The requirements of the "Fractal" problem

The background

Fractal refers to a geometric shape containing detailed structure at arbitrarily small scales. Fractals are infinitely complex patterns that are self-similar across different scales.

There is a fractal defined as follows:

When level $n = 1$:

0

When level $n = 2$

```
0 0
 0
0 0
```

When level $n = 3$:

```
0 0    0 0
 0      0
0 0    0 0
   0 0
    0
   0 0
0 0    0 0
 0      0
0 0    0 0
```

If we use X (n-1) to represent a fractal at level (n-1), then the Fractal X (n) is represented as:

```
X(n-1) X(n-1)
    X(n-1)
X(n-1) X(n-1)
```

Specifications of the Fractal Problem

For a given character "o" and the fractal at level (n) 2, print out the fractals at required levels in the range of n (n<=7).

Inputs

The inputs include multiple groups of samples.

The first line is the groups of fractals *m*.

The second line and after are sample fractals.

Each sample fractal is described in four lines: the first line is the level of the fractal (n) to be printed, and 2nd −4th lines are its fractal at level 2. The fractal at level 2 is fixed to take 3 columns and 3 rows.

Outputs

For each set of characters specifying a fractal, print out corresponding fractal at n level; print a blank line after each fractal; and don't keep extra empty space in any line.

Sample Inputs

2

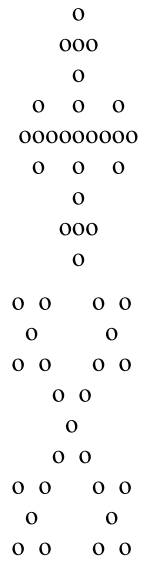
3

```
 o
ooo
 o

3

o o
 o
o o
```

Sample Outputs



Appendix D. The source code of the “Fractal” program for the code review

```
#include<stdio.h>
#include<stdlib.h>
#include<string.h>
#include<ctype.h>
#include<math.h>
int m,n;
int w,h;
char o[5][5],ans[200][200]; // array size should >=729
void draw(int n,int sw,int sh,int w,int h);
void print();
int main()
{
  scanf("%d",&m);
  int i,j,k;
  for(i=0;i<m;i++)
  {
    scanf("%d",&n); // need to initialize ans with value ' ' .
    int x,y;
    for(x=0;x<3;x++)
      for(y=0;y<3;y++)
      o[x][y]='o'; // should initialize with ' ' instead of 'o'.
    getchar();
    for(j=0;j<3;j++)
    {
      gets(o[j]);
    }
    if(n==1)
    {
      printf("o\n");
    }
    else if(n==2)
    {
      for(j=0;j<3;j++)
      {
        printf("%s\n",o[j]);
      }
    }else{
      w=6*n-9; // should be pow(3,n-1)+1e-5
        h=6*n-9; // should be pow(3,n-1)+1e-5
        draw(n,0,0,w,h);
```

```
        print();
      }
      printf("\n");
    }
    return 0;
  }
  void draw(int n,int sw,int sh,int w,int h)
  {
    int i,j;
    if(n==2)
    {
      for(i=0;i<3;i++)
    {
       for(j=0;j<strlen(o[i]);j++)
       {
         ans[sh+i][sw+j]=o[i][j];
       }
      }
      return;
    }
      for(i=0;i<3;i++)
      {
      for(j=0;j<3;j++)
      {
       if(o[i][j]=='0') // should be 'o' instead of '0'.
       {
         int ww,hh;
         ww=sw+w/3*j;
         hh=sh+h/3*i;
         // should be draw(n-1,ww,hh,w/3,h/3)
         draw(n-1,ww,hh,w-6,h-6);
       }
      }
    }
  }
  void print()
  {
    int i,j;
    for(i=0;i<h;i++)
    {
      // should not print spaces at the end of each line.
      for(j=0;j<w;j++)
      {
        printf("%c",ans[i][j]);
      }
    printf("\n");
    }
}
```

Appendix E. Participants Background Survey

1. **Occupation**_____ (you can choose multiple answers; please put your current primary occupation in the first place)
A. Undergraduate students B. Graduate students C. Researchers D. Professional engineer in industry E. other (please specify)____
2. **Major(s) of degrees obtained**_____ (e.g. Bachelor in Computer Science)
3. **Years** of experience in software d**evelopment**_____
4. Please estimate the number of software **development projects** that you have participated so far___
A. 0–5 B 6–10 C. 11–20 C. 21–30 D. > 30
5. Years of experience in software verification, validation, and other **software quality assurance** activities_____
6. Experience in Code Review_____(Months)
7. Experience in C/C++ _____(Months)

References

Anu, V., Walia, G., Hu, W., Carver, J.C., Bradshaw, G., 2016. Using a cognitive psychology perspective on errors to improve requirements quality: an empirical investigation. In: Paper presented at the 2016 IEEE 27th International Symposium on Software Reliability Engineering (ISSRE).

Aurum, A., Petersson, H., Wohlin, C., 2002. State-of-the-art: software inspections after 25 years. Softw. Test., Verif. Reliab. 12 (3), 133–154.

Bacchelli, A., Bird, C., 2013. Expectations, outcomes, and challenges of modern code review. In: Paper presented at the 2013 35th International Conference on Software Engineering (ICSE).
Basili, V.R., Selby, R.W., 1987. Comparing the effectiveness of software testing strategies. IEEE Trans. Softw. Eng. (12), 1278–1296.
Baum, T., Liskin, O., Niklas, K., Schneider, K., 2016. Factors influencing code review processes in industry. In: Paper presented at the Proceedings of the 2016 24th Acm Sigsoft International Symposium on Foundations of Software Engineering.
Baum, T., Schneider, K., Bacchelli, A., 2019. Associating working memory capacity and code change ordering with code review performance. Empir. Softw. Eng. 24, 1762–1798.
Baysal, O., Kononenko, O., Holmes, R., Godfrey, M.W., 2016. Investigating technical and non-technical factors influencing modern code review. Empir. Softw. Eng. 21 (3), 932–959.
Byrne, M.D., Bovair, S., 1997. A working memory model of a common procedural error. Cogn. Sci. 21 (1), 31–61.
Chen, Q., Kong, D., Bao, L., Sun, C., Xia, X., Li, S., 2022. Code reviewer recommendation in tencent: practice, challenge, and direction. In: Paper presented at the Proceedings of the 44th International Conference on Software Engineering: Software Engineering in Practice.
Davila, N., Nunes, I., 2021. A systematic literature review and taxonomy of modern code review. J. Syst. Softw. 177, 110951.
Doğan, E., Tüzün, E., 2022. Towards a taxonomy of code review smells. Inf. Softw. Technol. 142, 106737.
Dunsmore, A., Roper, M., Wood, M., 2000. Object-oriented inspection in the face of delocalisation. In: Paper presented at the Proceedings of the 22nd International Conference on Software Engineering.
Duraes, J., Madeira, H., Castelhano, J., Duarte, C., Branco, M.C., 2016. WAP: understanding the brain at software debugging. In: Paper presented at the 2016 IEEE 27th International Symposium on Software Reliability Engineering (ISSRE).
Ebert, F., Castor, F., Novielli, N., Serebrenik, A., 2021. An exploratory study on confusion in code reviews. Empir. Softw. Eng. 26 (1), 1–48.
Fagan, M.E., 1999. Design and code inspections to reduce errors in program development. IBM Syst. J. 38 (2.3), 258–287.
Fan, Y., Xia, X., Lo, D., Li, S., 2018. Early prediction of merged code changes to prioritize reviewing tasks. Empir. Softw. Eng. 23, 3346–3393.
Floyd, B., Santander, T., Weimer, W., 2017. Decoding the representation of code in the brain: an fMRI study of code review and expertise. In: Paper presented at the 2017 IEEE/ACM 39th International Conference on Software Engineering (ICSE).
Gonçalves, P.W., Fregnan, E., Baum, T., Schneider, K., Bacchelli, A., 2022. Do explicit review strategies improve code review performance? Towards understanding the role of cognitive load. Empir. Softw. Eng. 27 (4), 99.
Haytham Hijazi, J.D., Couceiro, R., Castelhano, J., Barbosa, R., Medeiros, J., Castelo-Branco, M., De Carvalho, P., Madeira, H., 2022. Quality evaluation of modern code reviews through intelligent biometric program comprehension. IEEE Trans. Softw. Eng. 49 (2), 626–645. https://doi.org/10.1109/TSE.2022.3158543.
Hijazi, H., Cruz, J., Castelhano, J., Couceiro, R., Castelo-Branco, M., de Carvalho, P., Madeira, H., 2021. iReview: an intelligent code review evaluation tool using biofeedback. In: Paper presented at the 2021 IEEE 32nd International Symposium on Software Reliability Engineering (ISSRE)..
Hijazi, H., Duraes, J., Couceiro, R., Castelhano, J., Barbosa, R., Medeiros, J., Madeira, H., 2022. Quality evaluation of modern code reviews through intelligent biometric program comprehension. IEEE Trans. Softw. Eng. (01), 1.
Hong, Y., Tantithamthavorn, C.K., Thongtanunam, P.P., 2022. Where should I look at? Recommending lines that reviewers should pay attention to. In: Paper presented at the 2022 IEEE International Conference on Software Analysis, Evolution and Reengineering (SANER).
Huang, F., 2016. Post-completion error in software development. In: Paper presented at the The 9th International Workshop on Cooperative and Human Aspects of Software Engineering, ICSE 2016. Austin, TX, USA.
Huang, F., Liu, B., 2017. Software defect prevention based on human error theories. Chin. J. Aeronaut. 30 (3), 1054–1070. https://doi.org/10.1016/j.cja.2017.03.005.
Huang, F., Liu, B., Huang, B., 2012. A taxonomy system to identify human error causes for software defects. In: Paper presented at the 18th International Conference on Reliability and Quality in Design. Boston, USA.
Huang, F., Madeira, H., 2021. Targeted code inspection based on human errors. In: Paper presented at the 32nd International Symposium on Software Reliability Engineering (ISSRE 2021). Wuhan, China.
Huang, F., Strigini, L., 2023. HEDF: a method for early forecasting software defects based on human error mechanisms. IEEE Access 11, 3626–3652.
IEEE, 1990. IEEE Standard Glossary of Software Engineering Terminology (Vol. lEEE Std 610.121990). The Institute of Electrical and Electronics Engineers, New York, USA.
Kalyan, A., Chiam, M., Sun, J., Manoharan, S., 2016. A collaborative code review platform for github. In: Paper presented at the 2016 21st International Conference on Engineering of Complex Computer Systems (ICECCS).
Khoshnoud, F., Nasab, A.R., Toudeji, Z., Sami, A., 2022. Which bugs are missed in code reviews: an empirical study on SmartSHARK dataset. Proceedings of the 19th International Conference on Mining Software Repositories, pp. 137–141.
Kitagawa, N., Hata, H., Ihara, A., Kogiso, K., Matsumoto, K., 2016. Code review participation: game theoretical modeling of reviewers in gerrit datasets. In: Paper presented at the Proceedings of the 9th International Workshop on Cooperative and Human Aspects of Software Engineering.
Laitenberger, O., DeBaud, J.-M., 1997. Perspective-based reading of code documents at Robert Bosch GmbH. Inf. Softw. Technol. 39 (11), 781–791.
McIntosh, S., Kamei, Y., Adams, B., Hassan, A.E., 2016. An empirical study of the impact of modern code review practices on software quality. Empir. Softw. Eng. 21, 2146–2189.
Mirsaeedi, E., Rigby, P.C., 2020. Mitigating turnover with code review recommendation: balancing expertise, workload, and knowledge distribution. In: Paper presented at the Proceedings of the ACM/IEEE 42nd International Conference on Software Engineering.
Pascarella, L., Spadini, D., Palomba, F., Bruntink, M., Bacchelli, A., 2018. Information needs in contemporary code review. Proc. ACM Hum.-Comput. Interact. 2 (CSCW), 1–27.
Perry, D.E., Porter, A., Wade, M.W., Votta, L.G., Perpich, J., 2002. Reducing inspection interval in large-scale software development. IEEE Trans. Softw. Eng. 28 (7), 695–705.
Porter, A.A., Votta, L.G., Basili, V.R., 1995. Comparing detection methods for software requirements inspections: a replicated experiment. IEEE Trans. Softw. Eng. 21 (6), 563–575.
Reason, J., 1990. Human Error. Cambridge University Press, Cambridge, UK.
Sadowski, C., Söderberg, E., Church, L., Sipko, M., Bacchelli, A., 2018. Modern code review: a case study at google. In: Paper presented at the Proceedings of the 40th International Conference on Software Engineering: Software Engineering in Practice.
Shull, F., Basili, V., Boehm, B., Brown, A.W., Costa, P., Lindvall, M., Zelkowitz, M., 2002. What we have learned about fighting defects. In: Paper presented at the Proceedings Eighth IEEE Symposium on Software Metrics.
Spadini, D., Palomba, F., Baum, T., Hanenberg, S., Bruntink, M., Bacchelli, A., 2019. Test-driven code review: an empirical study. In: Paper presented at the 2019 IEEE/ACM 41st International Conference on Software Engineering (ICSE).
Thongtanunam, P., Hassan, A.E., 2020. Review dynamics and their impact on software quality. IEEE Trans. Softw. Eng. 47 (12), 2698–2712.
Thongtanunam, P., Tantithamthavorn, C., Kula, R.G., Yoshida, N., Iida, H., Matsumoto, K.-i., 2015. Who should review my code? a file location-based code-reviewer recommendation approach for modern code review. In: Paper presented at the 2015 IEEE 22nd International Conference on Software Analysis, Evolution, and Reengineering (SANER).
Tufano, R., Masiero, S., Mastropaolo, A., Pascarella, L., Poshyvanyk, D., Bavota, G., 2022. Using pre-trained models to boost code review automation. In: Paper presented at the Proceedings of the 44th International Conference on Software Engineering.
Wilkerson, J.W., Nunamaker, J.F., Mercer, R., 2011. Comparing the defect reduction benefits of code inspection and test-driven development. IEEE Trans. Softw. Eng. 38 (3), 547–560.
Wohlin, C., Runeson, P., Höst, M., Ohlsson, M.C., Regnell, B., Wesslén, A., 2012. Experimentation in Software Engineering. Springer, New York.
Wurzel Gonçalves, P., Çalikli, G., Bacchelli, A., 2022. Interpersonal conflicts during code review: developers' experience and practices. Proc. ACM Hum.-Comput. Interact. 6 (CSCW1), 1–33.

Fuqun Huang: Dr. Huang is an Assistant Professor in Department of Computer Science at Western Washington University. She also serves as the President of the Board of Directors for the 501(c)(3) non-profit research organization, the Institute of Interdisciplinary Scientists in Seattle. Prior to joining WWU, she served as a researcher (Principal Investigator) at the Centre for Informatics and Systems at the University of Coimbra in Portugal and completed a two-year postdoctoral research position at The Ohio State University. Her academic journey began at Beihang University, where she earned both her Bachelor of Science in Engineering and her PhD in Systems Engineering.

Dr. Huang is dedicated to advancing the trustworthiness (reliability, safety, and security) of computer systems through an understanding of the psychological mechanisms behind how software engineers commit errors. She first initiated the interdisciplinary area of "Software Defect Defense Based on Human Errors" in 2011, which has since expanded into "Human Errors in Software Engineering." This interdisciplinary field encompasses topics such as software fault early forecasting based on human errors, software defect prevention based on human error theories, defect detection based on human errors, software fault tolerance design based on human errors, in-depth experimental studies on the cognitive mechanisms of human errors in programming, and advancing Computer Science education using human error theories. Dr. Huang has designed and delivered the first university course in this area, namely "Human Errors in Software Engineering," an undergraduate and graduate stack course first offered at Western Washington University in early 2024.

Henrique Madeira: Henrique Madeira is full professor at the University of Coimbra, where he has been involved in research on dependable computing since 1989. His-research interests include software quality and software reliability, experimental evaluation and benchmarking of dependability and security, and fault injection techniques. His-recent research projects involve two research directions: a) Assured AI, focusing on providing safety and security guaranties in critical applications that use AI and b) human factors in software engineering, particularly on the use of biometrics to improve software quality. He has co-authored more than 220 papers in refereed conferences and journals and has co-ordinated and/or participated in more than 30research projects funded by the European Commission and the Portuguese Science Foundation. Henrique Madeira was co-founder of the company Critical Software, S.A. (www.criticalsoftware.com).