A Proposal of Code Amendment Problem in Java Programming Learning Assistant System

Htoo Htoo Sandi Kyaw, Nobuo Funabiki, and Wen-Chung Kao

***Abstract*—To enhance Java programming educations, we have proposed a Web-based *Java Programming Learning Assistant System (JPLAS)* that provides a variety of programming assignments to cover different learning levels. As fundamental programming exercises for novice students, JPLAS offers the *Code Fill-in-blank Problem (CFP)* and the *Code Fixing Problem (CXP)*, to learn Java grammar and basic programming skills through *code* reading. A CFP instance requires filling in the *blank* elements in the *problem code* generated by applying the *coding rule check function* and the *blank element selection algorithm*. A CXP instance involves correcting the *error* elements made by the *error injection algorithm*. In both problems, all answers from the students will be marked through *string matching* with the stored correct one. In this paper, we propose the *Code Amendment Problem (CAP)* as a practical problem for learning the debugging process by combining CFP and CXP in JPLAS. As a mixture of CFP and CXP instances, a CAP instance is generated by randomly selecting either *blank* or *error* for each element with a given blank probability *BP*. For evaluations, we apply 12 CAP instances to 21 students in Japan and Myanmar, where the results show that *BP* = 50% offers the highest difficulty level, and CAP is harder than CFP and CXP.**

***Index Terms*—Blank element selection algorithm, code amendment problem, coding rule check function, error injection algorithm, Java programming, JPLAS.**

I. Introduction

Nowadays, the object-oriented programming language Java has been widely applied in various systems in societies and industries due to its exceptional reliability, portability, and scalability. Java was selected as the most popular programming language in 2015 [1], and still remains as the mainstream [2]. To respond to the strong demand from industries for Java programming educations, a plenty of universities and professional schools have introduced various Java programming courses to meet this challenge.

A typical Java programming education consists of two elements, namely, grammar study using a textbook and programming exercises with a computer. That is, we have developed a Web-based *Java Programming Learning Assistant System (JPLAS)* [3]-[7], which provides several types of programming exercises to support self-studies of students at different learning stages.

Among them, *Code Fill-in-blank Problem (CFP)* [1] [8] and the *Code Fixing Problem (CXP)* [9] are elementary programming exercises designed for novice, to learn Java grammar and basic programming skills through *code reading*. A CFP instance requires filling in the blank elements in the *problem code* that is generated by applying the *blank element selection algorithm* to a high-quality source code. This code is also generated by applying the *coding rule checking function*. A CXP instance needs correcting the error elements that are made by applying the *error injection algorithm* to the blank elements selected by the *blank element selection algorithm*. In both problems, all answers will be marked through *string matching* with the stored correct one.

In addition, the readability of a source code plays an important role in achieving the maintainability and the uniformity of the code for corrections, modifications, and extensions. A *readable code* can be realized by following *coding rules*, which may be composed of *naming rules, coding styles, and potential problems* [10]. The *coding rule check function* will examine the adherence of the coding rules of a given Java source code, then return the locations that do not follow them. By applying this function, only a readable source code is used to generate CFP and CXP instances.

The *blank element selection algorithm* selects as many blank elements as possible that have the unique answers from a given Java source code. An *element* represents the minimum unit in a code, which includes a *reserved word*, an *identifier*, and a *control symbol*. A *reserved word* signifies a fixed sequence of characters that has been defined in the Java grammar to imply a specific function. An *identifier* is a sequence of characters defined in the code by the author to represent a variable, a class, or a method. A *control symbol* in this paper indicates other grammar elements such as "." (dot), ":" (colon), ";" (semicolon), "(,)"(bracket), "{,}" (curly bracket).

The *error injection algorithm* injects errors into the source code by changing elements with similar but incorrect ones. The algorithm injects errors into the source code by changing the data type, the name of a class, a method or a variable, and important Java Keywords. It may also change an operator, a constant number, or a constant name in an equation or a conditional expression, to inject errors into the behavior of the code.

In this paper, we propose the *Code Amendment Problem (CAP)* in JPLAS to offer feasible exercises on code debugging process. As a mixture of CFP and CXP instances, a CAP instance is generated by randomly selecting either *blank* or *error* for each element that is selected by the *blank element selection algorithm*. For a generated CAP instance, the *difficulty level* can be controlled by adjusting the probability of selecting blank, which is specified as the *blank probability (BP)* for convenience in this paper.

Manuscript received April 6, 2020; revised June 22, 2020.

H. H. S. Kyaw and N. Funabiki are with Okayama University, Okayama, Japan (e-mail: pxs93q36@s.okayama-u.ac.jp, funabiki@okayama-u.ac.jp).

W-C. Kao is with the Department of Electrical Engineering, National Taiwan Normal University, Taipei, Taiwan (e-mail: jungkao@ntnu.edu.tw).

[1] In this paper, we use Code Fill-in-blank Problem (CFP) instead of Code Completion Problem (CCP) in previous papers to clarify the problem nature.

For evaluations, we generated 12 CAP instances using source codes in [11] and [12] with 30%, 50%, and 70% for *BP*, and assigned them to 21 students in Japan and Myanmar. The results show that *BP* = 50% offers the highest difficulty level among them, and CAP is harder than CFP and CXP.

The rest of this paper is organized as follows: Section II shows a brief survey of related works in literature. Section III reviews our related previous works to this paper. Section IV presents the code amendment problem. Section V evaluates the proposal. Finally, Section VI concludes this paper with future works.

II. Related Works in Literature

In this section, we overview the literature related to the programming study through code debugging and code reading.

In [13], Ahmandzadeh *et al*. showed that even students with knowledge of programming do not acquire the skills of debugging codes effectively. Skills at debugging seems to make the programmer confident and they suggested that more emphasis should be placed on debugging skills in the teaching of programming.

In [14], Ichinco *et al*. performed an exploratory study of novices using examples to complete programming tasks. To analyze programming behaviors, they defined the realization point at the time when a participant discovered the crucial concept in an example. It is observed that it may take a participant much time to reach the realization point because the time he/she spent on executing the example code was longer than on reading the example code.

In [15], Griffin discussed several lines of research, in order to support the premise that people learning programming can do more effectively and efficiently if they spend as much time on *deconstructing* codes as on writing codes. The term *deconstruction* is referred to as reading, tracing, and debugging a code.

In [16], Kakeshita *et al*. developed a programming tool supporting education called *Pgtracer*. *Pgtracer* utilizes fillin-the-blank questions composed of a source code and a trace table. The blanks in the code and the trace table must be filled by the students to improve the code reading while solving the questions.

III. Previous Works

In this section, we review our previous works related to this paper.

A. Coding Rule Check Function

Coding rules [17] represents a set of rules or conventions for producing high quality source codes. By following coding rules, the uniformity of the code will be maintained, which enhances the readability, maintainability, and scalability. *Coding rules* consist of *naming rules*, *coding styles*, and *potential problems*.

1) *Naming Rules: Naming rules* describe the rules for detecting the naming errors in the source code. Here, the *Camel case* [18] is adopted as the common Java naming rule. For an identifier representing a variable, a method, or a method argument, the first character should be a lower case, where the delimiter character between two words should be an upper case. For an identifier indicating a class, both of them should be an upper case. For an identifier signifying a constant, any character should be an upper case. An English word should be used as an identifier name, whereas Japanese or Roman Japanese should not be used.
2) *Coding Styles: Coding styles* indicate the rules for detecting the layout errors in the source code. They include the position of an indent or a bracket, and the existence of a blank space. By following coding styles, the layout of a source code will become more consistent and readable.
3) *Potential Problem: Potential problems* illustrate the rules for discovering the portions of the source code that can pass the compilation but may include functional errors or bugs with high possibility. They include a *dead code* and *overlapping codes.* A *dead code* represents the portion of the source code not executing, and *overlapping codes* indicate the multiple portions of the source code with similar structure and functions to each other. By solving potential problems, the code can not only improve the maintainability and scalability, but also speed up the execution.

B. Blank Element Selection Algorithm

The *blank element selection algorithm* [7] uses the *constraint graph* that is generated to describe the constraints in the blank element selection. Then, the fill-in-blank problem will be generated through the following five steps:

1) Vertex *generation for constraint graph:* each vertex represents a candidate element for being blank.
2) *Edge generation for constraint graph:* an edge is generated between any pair of two vertices or elements that should not be blanked at the same time.
3) *Compatibility graph generation:* by taking the complement of the constraint graph, the *compatibility graph* is generated to represent the pairs of elements that can be blanked simultaneously.
4) *Clique extraction:* a maximal clique of the compatibility graph is generated by a simple greedy algorithm to identify the maximal number of blank elements with unique answers from the given Java code. This greedy algorithm repeats to: 1) select the vertex that has the *largest degree* in the compatibility graph for the clique, 2) remove this vertex and its non-adjacent vertices of the graph, until the graph becomes null.
5) *Control symbol limitation:* the ratio between the number of blanks for control symbols and that for other elements is controlled.

C. Error Injection Algorithm

The error injection algorithm injects errors into the source code by changing elements with similar but incorrect ones. To be specific, this algorithm injects the source code with errors by changing the access modifier of a class or a method, the data type of a method or a variable, and the name of a class, a method, or a variable. Besides, it injects errors into the behavior of the source code by changing the operator, the constant number, the constant variable name in an equation or a conditional expression, or Java keywords defined in the *keyword list*. The procedure of the algorithm is as below:

1) Access modifier: An access modifier, *public*, *protected*, and *private*, of a class or a method is randomly changed to another one among them.
2) Data type: For the data type of a method or a variable, any of *byte*, *short*, *int*, and *long* is changed to *double*, or *float* randomly, and vice versa. Besides, *String* is changed to *char*, and vice versa. Here, *void* is not changed.
3) Loop: For the loop statement, any of *for* and *while* is changed to *if*.
4) Control: For the control statement, *if* is changed to *while*, and *switch* is changed to *do*.
5) Java keywords in list: The keywords in the keyword list are changed as follows:
   - *import* is changed to *implements*, and vice versa.
   - *extends* is changed to *instanceof*, and vice versa.
   - *Scanner* is changed to *System*, and vice versa.
   - *static* is changed to *super*, and vice versa.
   - *continue* is changed to *break*, and vice versa.
6) Behavior: An operator, a constant number, or a constant name in an equation or a conditional expression is changed:
   - an arithmetic operator, such as +, *, -, or /, is randomly changed to another one.
   - a conditional operator, such >, <, &&, ==, or !=, is randomly changed to another one.
   - a constant number is randomly changed to the similar number.
   - a constant name is changed to another name by applying the *error name generation method*.
7) Name: A name of a class, a method, or a variable will be switched to another name by applying the *error name generation method* in the next sub section.

D. Error Name Generation Method

The following procedure illustrates the *error name generation method.*

1) Error name generation using dictionary: A set of candidates for error names that have similar meanings as the original name, are extracted from the dictionary, *WordNet*, using the similar word estimating function. Then, one candidate will be randomly selected from this set for the error name.
2) Error name generation using word list: If a proper error name is not found by 1), the word whose spelling is most similar to the original name among the word in the *word list* is selected for the error name. The *word list* needs to be prepared by the user of the algorithm.
3) Error name random generation: If a proper error name is still not found in the second step, the error name will be generated by randomly adding or removing one character, or changing to another character, in the original name.

It is noted that a combined name using *Camel case* is first divided into a set of individual names, and then, the above procedure is applied to each individual name. Subsequently, the individual generated error names will be combined into one name.

E. Answer Interface of JPLAS

Fig. 1 demonstrates the answer interface for answering a CXP instance. The interface for CXP shows the problem code that has erroneous elements, which should be corrected by the students.

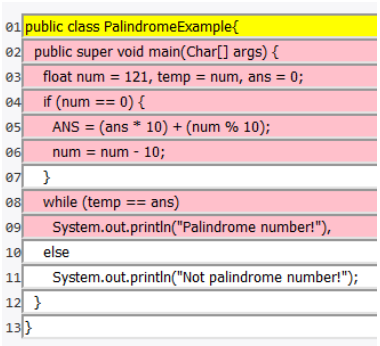

Fig. 1. Answer interface for CXP.

IV. Proposal of Code Amendment Problem

In this section, we present the *code amendment problem (CAP)* with the generation procedure.

A. Overview of Code Amendment Problem

In a CAP instance, a Java source code that has several missing or error elements, called a *problem code*, is shown to students, where one input form corresponds to one whole statement or line in the code. A student needs to identify the locations of missing or error elements in the code, then fill in or correct them with the correct elements. The correctness of the answer will be marked through *string matching* of the whole statement with the corresponding original one in the code.

B. Generation Procedure of CAP Instance

A CAP instance can be generated through the following steps:

1) Select a Java source code related to the current topic, from a website or a textbook.
2) Apply the coding *rule check function* to this source code for readable code.
3) Register each statement in the source code as the correct answer unit for string matching.
4) Apply the blank *element selection algorithm* to the source code to select blank elements from the code.
5) Select randomly blank elements found by the algorithm for error elements.
6) Apply the *error injection algorithm* to each selected element to make an error element.
7) Remove the remaining blank elements for generating the problem code.

For the automatic execution of this procedure, we implemented the necessary programs in Java and the script by Bash.

C. Instance Example

To clarify the CAP instance generation procedure, we explain the details by using the source code for class PalindromeExample.

1) *Source Code Selection*: This class classifies whether the given number is a *palindrome number* or not. A palindrome number is a number that becomes the same number after reversing the digits. For example, 121, 34543, 343, 131, and 48984 are palindrome numbers.
2) *Application of Coding Rule Check Function*: The *coding rule check function* is applied to the source code for class *PalindromeExample*. **code 1** shows the *source cod*e after the application.

code 1

```
01: public class PalindromeExample {
02:   public static void main(String[] args) {
03:     int num = 121, temp = num, ans = 0;
04:     while (num != 0) {
05:       ans = (ans * 10) + (num % 10);
06:       num = num / 10;
07:     }
08:     if (temp == ans)
09:       System.out.println("Palindrome number!");
10:     else
11:       System.out.println("Not palindrome number!");
12:   }
13: }
```

3) *Application of Blank Element Selection Algorithm*: The *blank element selection algorithm* is applied to this code, to select the blank elements from the code. That is, 9 blank elements are selected shown in **code 2**.

code 2

```
01: public class PalindromeExample {
02:   public _1_ void main(_2_[] args) {
03:     _3_ num = 121, temp = num, ans = 0;
04:     _4_ (num _5_ 0) {
05:       _6_ = (ans * 10) + (num % 10);
06:       num = num _7_ 10;
07:     }
08:     _8_ (temp == ans)
09:       System.out.println("Palindrome number!")_9_
10:     else
11:       System.out.println("Not palindrome number!");
12:   }
13: }
```

4) *Error Element Selection*: Error elements are randomly selected from the blank elements found by the blank element selection algorithm. Here, we change the *blank probability (BP)* at 30%, 50% and 70% to investigate the effect of BP in controlling the level of difficulty in a CAP instance.
5) *Application of Error Injection Algorithm and Blank Element Removal*: *The error injection algorithm* is applied to the selected blanks to inject errors, and the remaining blank elements are removed to generate the problem code. **code 3** shows the generated problem code for a CAP instance.

code 3

```
01: public class PalindromeExample {
02:   public void main([] args) {
03:     num = 121, temp = num, ans = 0;
04:     (num == 0) {
05:       ANS = (ans * 10) + (num % 10);
06:       num = num  10;
07:     }
08:     while (temp == ans)
09:       System.out.println("Palindrome number!"),
10:     else
11:       System.out.println("Not palindrome number!");
12:   }
13: }
```

6) *Answer Interface of CAP*: Fig. 2 specifies the answer interface of CAP where there are several erroneous input statement and students need to recognize and correct the erroneous element.

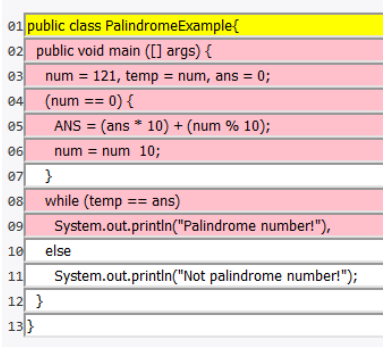

Fig. 2. Answer interface for CAP.

V. Evaluation

In this section, we evaluate the *code amendment problem (CAP)* in JPLAS through applications to 21 students in Japan and Myanmar who have studied Java programming more than one year.

A. Assigned Problems in Evaluation

For evaluations, we generated 12 CAP instances by following the procedure in Section IV using six source codes in [11] and six source codes in [12]. These 12 source codes cover the topics of *exception*, *array*, *recursive*, and *method overloading*. We used 30%, 50%, and 70% for *BP*, and generated four CAP instances with each *BP*. Here, we chose four source codes for each *BP* such that the average difficulty level becomes similar at any *BP*. Then, we asked the students to solve all the CAP instances.

B. Solution Results by Students

Table I shows the average and the standard deviation (SD) of the correct solution rates (%) for the students. When we compare the results for three different *BP*, *BP* = 50% results in the smallest average rate with the largest SD. Thus, *BP* = 50% can offer the highest difficulty level among them for a CAP instance.

TABLE I: Correct Solution Rates for CAP (%)

	30%	50%	70%	all
ave.	97.70	94.63	96.64	96.34
SD	10.35	13.18	8.72	10.60

Then, when we compare with the results for the *code fill-in-blank problem (CFP)* and the *code fixing problem (CXP)* in Table II in [19], the average rate for CAP is smallest and the SD is largest among the three problems. These results indicate that CAP is harder than CFP and CXP. Thus, CFP

and CXP should be given to beginners who just start studying Java programming, earlier than CAP.

TABLE II: CORRECT SOLUTION RATES FOR CFP AND CXP (%)

	CFP	CXP
ave.	99.89	99.66
SD	0.25	0.97

This time, we could not apply CAP instances to novice students of Java programming, due to the limited time. Therefore, the effect of *BP* in the difficulty level of CAP and the comparison of the difficulty level among CAP, CFP, and CXP should be investigated through applications to novice students in Java programming courses, which will be included in our future works.

VI. CONCLUSION

This paper proposed the *code amendment problem (CAP)* in *Java Programming Learning Assistant System (JPLAS)* to assist students in learning debugging process of Java programming. As a mixture of *code fill-in-blank problem (CFP)* and *code fixing problem (CXP)*, a CAP instance requires students to locate the missing or error elements in the code and amend them. The correctness of the answer is verified through string matching of the whole statement amended by a student and the corresponding correct one. For evaluations, 12 CAP instances were applied to 21 students in Japan and Myanmar, where the results show that *BP* = 50% offers the highest difficulty level, and CAP is harder than CFP and CXP. In future works, we will generate a variety of CAP instances, and apply them to novice students to verify the observation of this paper and examine the effectiveness in learning debugging process by them.

CONFLICT OF INTEREST

The authors declare no conflict of interest.

AUTHOR CONTRIBUTIONS

Htoo Htoo Sandi Kyaw designed and implemented the proposal, conducted the experiments, and wrote the paper as the main author. Nobuo Funabiki gave the idea of the proposal and supervised the whole activities including the experiments and the paper writing. Wen-Chung Kao advised on the experiments and improved the paper writing. All the authors had approved the final version.

ACKNOWLEDGMENT

We are very grateful to our laboratory members for fruitful discussions of advancing this research. We would also like to thank to all the students participated in the experiments.

REFERENCES

[1] S. Cass. The 2015 top ten programming language. [Online]. Available: http://spectrum.ieee.org/computing/software/the-2015-top-ten-programming-language/?utm_so
[2] Why does Java remain so popular? [Online]. Available: https://blogs.oracle.com/oracleuniversity/why-does-java-remain-so-popular
[3] N. Funabiki, Y. Matsushima, T. Nakanishi, and N. Amano, "A Java programming learning assistant system using test-driven development method," *IAENG Int. J. Comput. Sci.*, vol. 40, no. 1, pp. 38-46, Feb. 2013.
[4] K. K. Zaw, N. Funabiki, and W.-C. Kao, "A proposal of value trace problem for algorithm code reading in Java programming learning assistant system," *Inf. Eng. Express*, vol. 1, no. 3, pp. 9-18, Sep. 2015.
[5] N. Ishihara, N. Funabiki, and W.-C. Kao, "A proposal of statement fill-in-blank problem using program dependence graph in Java programming learning assistant system," *Inf. Eng. Express*, vol. 1, no. 3, pp. 19-28, Sept. 2015.
[6] Y. Korenaga, N. Funabiki, K. K. Zaw, N. Ishihara, S. Matsumoto, and W.-C. Kao, "A fill-in-blank problem workbook for Java programming learning assistant system," *Int. J Web Inform. Sys.*, vol. 13, no. 2, pp. 140-154, 2017.
[7] N. Funabiki, Y. Korenaga, K. K. Zaw, N. Ishihara, and W.-C. Kao, "A graph-based blank element selection algorithm for fill-in-blank problems in Java programming learning assistant system," *IAENG Int. J. Comput. Sci.*, vol. 44, no. 2, pp. 247-260, May 2017.
[8] H. H. S. Kyaw, S. T. Aung, H. A. Thant, and N. Funabiki, "A proposal of code completion problem for Java programming learning assistant system," *in Proc. VENOA2018*, July 2018, pp. 855-864.
[9] N. Funabiki, H. H. S. Kyaw, and K. K. Zaw, "A proposal of element/code fixing problem in Java programming learning assistant systeme," *in Proc. ICSE2019*, Dec. 2019.
[10] D. Boswell and T. Foucher, *The Art of Readable Code*, O'Reilly, 2011.
[11] P. J. Deitel and H. M. Deitel, *Java: How to Program*, 9th ed. Prentice Hall, 2011.
[12] Y. Daniel Liang, *Introduction to Java Programming*, 8th ed. 2011.
[13] M. Ahmadzadeh, D. Elliman, and C. Higgins, "An analysis of patterns of debugging among novice computer science students," *in Proc. ITiCSE*, 2005, pp. 84-88.
[14] M. Ichinco and C. Kelleher, "Exploring novice programmer example use," *in Proc. IEEE VL/HCC*, Oct 2015, pp. 63-71.
[15] J. M. Griffin, "Learning by taking apart: Deconstructing code by reading, tracing, and debugging," *in Proc. SIGITE*, Sep. 2016, pp. 148-153.
[16] T. Kakeshita and M. Murata, "Application of programming education support tool pgtracer for homework assignment," *Int. J. Learning Technologies and Learning Environments*, vol. 1, no. 1, pp. 41-60, 2018.
[17] N. Funabiki, T. Ogawa, N. Ishihara, M. Kuribayashi, and W.-C. Kao, "A proposal of coding rule learning function in Java programming learning assistant system," *in Proc. CISIS*, Sep. 2016, pp. 561-566.
[18] Camel case definition. [Online]. Available: http://searchsoa.techtarget.com/definition/CamelCase
[19] H. H. S. Kyaw, N. Funabiki, and M. Kuribayashi, "An implementation of hint function for code completion problem in Java programming learning assistant system," in *Proc. FIT. Conf.*, Sept. 2019, pp. 307-308.

Htoo Htoo Sandi Kyaw received the B. E. and M. E. degrees in information science and technology from University of Technology (Yatanarpon Cyber City), Myamar, in 2015 and 2018, respectively. She is currently a Ph.D candidate in Graduate School of Natural Science and Technology at Okayama University, Japan. Her research interests include educational technology and Web application systems. She is a student member of IEICE.

Nobuo Funabiki received the B.S. and Ph.D. degrees in mathematical engineering and information physics from the University of Tokyo, Japan, in 1984 and 1993, respectively. He received the M. S. degree in electrical engineering from Case Western Reserve University, USA, in 1991. From 1984 to 1994, he was with Sumitomo Metal Industries, Ltd., Japan. In 1994, he joined the Department of Information and Computer Sciences at Osaka University, Japan, as an assistant professor, and became an associate professor in 1995. He stayed at University of Illinois, Urbana-Champaign, in 1998, and at University of California, Santa Barbara, in 2000-2001, as a visiting researcher. In 2001, he moved to the Department of Communication Network Engineering (currently, Department of Electrical and Communication Engineering) at Okayama University as a professor. His research interests include computer networks, optimization algorithms, educational technology, and Web technology. He is a member of IEEE, IEICE, and IPSJ.

Wen-Chung Kao received the M.S. and Ph.D. degrees in electrical engineering from National Taiwan University, Taiwan, in 1992 and 1996, respectively. From 1996 to 2000, he was a Department Manager at SoC Technology Center, ERSO, ITRI, Taiwan. From 2000 to 2004, he was an assistant vice president at NuCam Corporation in Foxlink Group, Taiwan. Since 2004, he has been with National Taiwan Normal University, Taipei, Taiwan, where he is currently a research chair professor at Department of Electrical Engineering and the dean of College of Technology and Engineering. His current research interests include system-on-a-chip (SoC), flexible electrophoretic display, machine vision system, digital camera system, and color imaging science. He is a Fellow of IEEE.