International Journal of Engineering and Advanced Technology (IJEAT)
ISSN: 2249-8958 (Online), Volume-9 Issue-3, February 2020

Underlying Mental Factors Contributing to Software Complexity

Leena Jain, Satinderjit Singh

Abstract**:** ***Software complexity and program comprehension are inversely related. Higher the code complexity, poorer the comprehension. But we neither have good software complexity measure, nor do we understand how the program comprehension took place in human mind. This is because we know so little about the working of the human brain; how it processes internal and external information. In this paper we have identified 5 mental factors which adds into the code complexity. In order to explain these factors, we took 10 code snippet pairs in C language (2 each for every factor). Code snippets in pair are identical - in terms of number of variables, operators, control structure- but we believe one of the snippets in pair is carrying the higher cognitive load due to underlying mental factor identified. To the best of our knowledge these factors identified here in this paper are not used in calculating the code or software complexity. We believe these identified mental factors can be validated by various brain imaging and Eye tracking techniques like EEG and fMRI. They can also be validated by conventional software experimental methods. We believe these identified factors will increase our understanding of Program comprehension as well as it will lead better software complexity measure. This could be very useful in computer science education. The very process of understanding how the human mind decode the software can be possibly understood. In long run this could help us in better understanding of the functioning of human brain.***

Keywords **:** ***Program comprehension; Software Complexity; Cognitive metrics; Cognitive load; Code snippets; Human brain working..***

I. INTRODUCTION

Software by its very nature are inherently complex entities. But gauging the complexity of software code has turned out to be even more complex problem. Perhaps the most simple and automatic measure used for software is LOC [1]. Halstead measure is also used in many academic circles to ascertain the complexity of the software[2]. Perhaps the most widely used measures in software industry is McCabe's cyclometric measure[3]. The fitness of all these measures and most prominently McCabe's measure is lot of debated in academic and software industry circles.

Revised Manuscript Received on February 05, 2020.
* Correspondence Author
Leena Jain *, Professor and Head in Department of Computer Applications, Global Group of Institutes, Amritsar affiliated to PTU Kapurthala, India. Email: leenajain79@gmail.com
Satibderjit Singh **, Department of Computer Applications, GGNIMT, Ludhiana, affiliated to PTU Kapurthala, India Email: satty74@gmail.com

Many still regard it as valid and popular measure of software complexity[4]. But it would be safe thing to say that none these old measure captures the complexity of the software fully.
Perhaps the most fundamental reason that we still not have a satisfactory software complexity measure because we don't know how human brain handles and process complexity. At low level the problem of software complexity has a duality in terms of Program comprehension. At lower
module or program level software complexity is just the code complexity and higher it is more difficult is program comprehension. Shao and wang proposed that different software constructs require different mental effort and hence proposed cognitive weights for various Basic control structure (BCS)[5], [6]. Based upon these cognitive weights many novel complexity measures were proposed by researchers[5]–[11]. But the problem of correct measurement of code complexity (and inversely problem of understanding program comprehension) persisted.

In this paper we argue that other than factors like number of variables, input, output and cognitive weights of BCS, we must also consider mental factors which adds into the code complexity and eventually the overall software complexity. We identify five of such mental factors. In order to explain these identified mental factors, we constructed a snippet pairs such that each pair is similar snippets – in terms of number of variables, operators, input, output or basic control structure. Our contention is that code snippets in every pair will be measured with equal code complexity- both by most of existing cognitive and non-cognitive complexity measure- yet we believe that one snippet in each pair carry more cognitive load and thus more difficult to comprehend than other one -due to mental factor we have identified. We call upon researching community to put these mental factors to series of experiments to validate or dismiss them. The new brain imaging techniques- fMRI, EEG, Eye tracking – can be used in addition with conventional response time method to validate these mental factors.

The rest of paper is organized as follows: Section 2 discuss the idea of cognitive weights of BCS and resultant metrices coming out of it, along with inherent limitations. Section 3 looks at various new brain imaging techniques used to understand the program comprehension. In section 4 we explain each factor by taking two snippet pairs and explaining it. Section 5 summarize the code snippet pair examples.And in section 6 we conclude and give the future scope of the work and the direction it may take.

II. COGNITIVE WIEGHTS AND ITS LIMITATIONS

Most of the metrics touched in previous section rely on visible property of Software code - like number of variables, Lines of code (LOC), structure of the program, (McCabe's cyclometric complexity) etc. Thus, most of these measures does not consider the working of human brain in consideration. That in all probability explains why most of the software complexity measures are unable to capture the true complexity of software.
Wang and Shao did propose cognitive complexity measure of the Software in which cognitive weights (CW) was assigned to various control construct referred to as basic control structure (BCS) - as shown in Table 1 below[5], [6]. The important part of these measures is that perhaps for first time we were recognizing that brain process different Software construct differently, this giving equal weightage to all portions of software- as in LOC- is not fair. Jain and Singh raised the question of universality of these BCS weights and related concerns[12]. To the best of our knowledge, not much work has been done to validate the cognitive weights of BCS proposed by Wang and others.

Gruhn and Laue had suggested that there should be three more BCS other than 10 mentioned in Table 1[13]. The new BCS proposed by them are *lock, exception* and *internal exits*. Many researchers have multiple methods of how to calculate complexity metrics from cognitive metrics from software code. Whatever may be limitation of the Cognitive metrics, it has attracted lot of attention and some tools are being developed based upon Cognitive weight of BCS philosophy[11], [14].

Table- I: Cognitive Weights of different BCS

Category	**BCS**	**Cognitive weights (Wang 2003)**	**Cognitive weights (Wang2006)**
Sequence	*Sequence*	1	1
Branch	*If then else*	2	3
	Case	3	4
Iteration	*For-loop*	3	7
	Repeat-loop	3	7
	While-loop	3	8
Embedded Component	*Function call*	2	7
	Recursion	3	11
Concurrency	*Parallel*	4	15
	Interrupt	4	22

Nevertheless, there are issues with Cognitive metrics and tools based upon that. First and foremost is the issue of cognitive weights of BCS. Not enough experiments are carried out to validate these weights. One reason of the same could be that experiment design for such a validation is critical as variations in code are endless[15]. Jain and Singh touched upon some of the issues involved here[16]. Another issue is that since we don't know the working of human brain of even individual, how can we generalize about cognitive weights for entire human populations[12][16]. What about variations within population? So, slowly focus is shifting to understanding working of human brain – how does it process internal and external information.

III. OTHER APPROACHES TO STUDY PROGRAM COMPREHENSION

Over the years there is growing realization that understanding of the working of human brain is crucial to measure the complexity related to software code. Some researchers in a way shown that there is difference of multiples times of 10 as far as software comprehension is concerned varying from one individual to another[17]. Some researchers have shown looping structure is more difficult to understand than branching structure[18]. Also Ajami and others in year 2017 suggested that loop counting downward is tougher to interpret than otherwise[15]. The same paper also suggested that certain logic conditions (not all) involving 'not' operator is difficult to decipher than otherwise. So clearly there is difference at individual to individual basis and within an individual, different constructs (Sequence, loop, branching, function etc.) are interpreted differently.

In last half a decade researcher have started using various scanning and imaging techniques while handling the software code. In a remarkable work, Pietek and others scanned the human brain of respondents while solving code snippets using fMRI technique to identify active region of brain during the software decoding procedure[19]. The same method was also applied to compare the brain regions when solving code snippets to that when performing code debugging[19]. Another method to assess software comprehension is through Eye Tracking method[20][21]. Apart to that many researchers has applied the EEG to identify the brain regions when doing software decoding[22][23]. Researchers has shown the difference between active regions of brain of software experts and novice in software engineering using EEG technique[24].

IV. MENTAL FACTORS IN CODE COMPLEXITY

In this section we will identify some of the micro factors – other than routine factors of number of variables, number of operations, Basic software constructs etc.- which seems to enhance software complexity of the code. These are

- Constant Vs Variable
- Variables Used
- Variables Modified
- Original Vs Modified Use of variable.
- Variable Change Source

The factors play by putting an extra cognitive load on human brain in deciphering the working of software code. We have identified five mental load enhancing factors. It's possible that more such factors exist, but as of now we have found these five basic ones. In the following section we explain these factors by taking the example of pair of code snippets. Each pair has some control structure –Sequence, for, while, function, do while, switch etc. Furthermore, code snippets in each pair has same number of variables (2 in these cases), same number of mathematical operations (again 2)-although the type of mathematical operations may be different. However, each pair of code snippets contain a subtle difference which is one of the above-mentioned factors.

A. Constant Vs Variables

In Constant Vs Variable factor, we contend that in any mathematical operations – be it algebra or software code- the calculations should be slightly easier when dealing with constant than as compared to variables. The simple philosophy is that carrying a variable (and its value) is an extra cognitive load on human mind than as compared to constant mentioned in equation.

In the snippet pair in C language given below in Table 2, we have same control constructs ('while'), same number of variables (2 here), but the snippet 1 uses constant in calculations whereas snippet 2, uses variable in its calculation. So, it is reasonable to assume that snippet 2 should be trickier to human mind than snippet 1 to decode.

Table- II: Snippet pair 1 demonstrating 'Constant Vs Variable' factor- 'while' BCS

Snippet 1	Snippet 2

int a=2, b =9;
while(a<10)
{
b=b-3;
a=a*3;
}
printf("\n%d" , b);
	int a=3, b =7;
	while(a>0)
	{
	b=b/a;
	a=a-b;
	}
	printf("\n%d" , a);

Another example in sequence code snippet pair is given below-in snippets 3 & 4. Here in Table 3, we have two snippets. Both these snippets are example of sequence BCS and uses two (2) variables each. The number of non-assignment operator is also 2. Most of existing code complexity metrices – cognitive and non-cognitive would give the same value to both snippets. But if look carefully, we see that in first variable modification of variable 'b' in snippet 3 is done through declared value of 'b' and constant specified in instruction. In contrast the variable 'a' modified in snippet is done through using the declared value of two variables. In constant Vs Variable comparison our contention is that snippet 4 should be little bit more of cognitive load on human brain than snippet 3.

Table- III: Snippet pair 2 demonstrating 'Constant Vs Variable' factor - Sequence BCS

Snippet 3	Snippet 4

int a= 15, b =4;
b=b+2;
a= a/b;
printf("\n%d" , a);
	int a=7, b =4;
	a=a*b;
	b= b-a;
	printf("\n%d" , b);

B. Variables Used

Another factor which play a part in increasing the cognitive complexity of software code is the number of variables used in mathematical calculations. More the variable used more the cognitive load on human brain. Also, human can carry some finite number of variables in their mind to sort out the calculations. Beyond some point the human capacity to perform mathematical calculations deteriorates rapidly. In the example of code snippet pairs in Table 4, snippet 5 uses only one variable in mathematical calculations as compared to snippet 6, which uses 2 variables in it. Both the snippets (5 & 6) uses while construct and are also same in many other ways like number of operations performed, number of times loop runs etc.

Table- IV: Snippet pair 3 demonstrating 'Constant Vs Variable' factor- 'while' BCS

Snippet 5	Snippet 6

int a=1, b =3;
while(a<4)
{
a=a*3;
a=a-1;
}
printf("\n%d" , a);
	int a=3, b =7;
	while(a>0)
	{
	b=b/a;
	a=a-b;
	}
	printf("\n%d" , a);

Similarly, in code snippet pair of 7 & 8 of Table 5, the control structure is same function type for both the cases but the snippet 7 involves only one variable in its calculation and thus should be easier than snippet 8, which involves 2 variables.

Table- V: Snippet pair 4 demonstrating 'Variables used' factor- 'function' BCS

Snippet 7	Snippet 8

int funct (int a)
{
a= 19/a+2;
return (a)
}
int b;
b= funct(3);
printf("\n%d" , b);
	int funct (int a,int b)
	{
	b= b+13/a;
	return (b) ;
	}
	printf("\n%d", funct(4,5));

C. Variables Modified

Another factor is number of times (and numbers of variables) variables modifies in human brain in order to work out the output of software code.

The idea is lesser the times the modification is carried out, easier it is for human brain. For an obvious reason the modification (even repeated) of same variable should be preferred by human brain than multiple modifications of different variables.

In the code snippet pair example (if-else constructs) given below in Table 6, it must be noted that in snippet 9 variable 'b' is modified twice, whereas in snippet 10, two variables ('a' & 'b') are modified once each.

Thus snippet 10 should have more mental adjustment involved than snippet 9.

Table- VI: Snippet pair 5 demonstrating 'Variables modified' factor- 'if-else' BCS

Snippet 9	Snippet 10

int a=2,b=9;
if (a<b)
{
b=b/2
b=b-a;
}
else
{
a=b/2;
b=a+b;
}
printf("\n%d" , b);
	int a=6,b=11,;
	if (a>b)
	{
	a=b/2;
	b=a+b;
	}
	else
	{
	b=b/3
	a=a+b;
	}
	printf("\n%d" , a);

Similarly, in Table 7 having code snippet pair of 11 & 12, involving do-while constructs, snippet 11 have 2 variables modified twice each and snippet 12 have one variable modified 4 times. Snippet 12 should be easier to work out manually.

Table- VII: Snippet pair 6 demonstrating 'Variables used' factor- 'do-while' BCS

Snippet 11	Snippet 12

int a=9, b=4,;
do
{
a=a+b;
b=a-b
} while(b<12);
printf("\n%d" , a);
	int a=6, b=3;
	do
	{
	b=a*b;
	b=b-a;
	} while(b<12);
	printf("\n%d" , b);

D. Original Vs Modifies use of Variables

Another important factor in our views is that in the calculation-while figuring out output or new variable values-are we using original-declared at start- or modified value during the mental execution of the code. The idea being that declared values involves zero mental shifting and using modified value involves not only mental shifting to new value and remembering the new variable value. It must be made clear that this factor is different from variable modified factor. Because in variable modified factor we count only the number of variables along with number of times they are modified. But here we compare various code snippets in what values-original or modified- are used in calculating output or new variable or function values.

For example, in code snippet pair (13 & 14) of Table 8, while construct shown below loop runs two times for both. Snippet 13 have used 1 times original value of both variable 'a' and 'b', 2 times first modification of 'b' -M1(a)- 1 time the second modification of 'b'-M2(b)- and 1 time the first modification of 'a'-M1(a). All in all, 2 original values, 3 times the first modification and 1 time second modification of the variables. Contrast this with Snippet 14, here we use original variable 'a' 2 times and 'b' 1 time. Similarly, first modification of 'a'-M1(a)- is used twice and M1(b) is used once. So, we can say that snippet 13 has a higher modified variable usage value -4 out of 6 – as compared to snippet 14 which has lower usage value -3 out of 6. Also, variable modification is deeper in snippet 14 – with 1 variable used is from second modified value. Thus, on theory snippet 13 should be difficult to comprehend mentally than snippet 14.

Table- VIII: Snippet pair 7 demonstrating 'Original Vs modified' factor- 'while' BCS

Snippet 13	Snippet 14

int a=3, b =8;
while(a<10)
{
b=b-2;
a=a+b;
}
printf("\n%d" , b);
	int a=2, b =8;
	while(a<10)
	{
	b=b-a;
	a=a*3;
	}
	printf("\n%d" , b);

In Table 9, having snippet pair combination of Snippet 15 & 16, this is case of two variables modified one each vs one variable modified twice. The usage of original vs modified in calculation is same – 2 ratios to 1. We believe that snippets 16 should be easier to calculate than snippet 15.

Table-IX: Snippet pair 8 demonstrating 'Original Vs modified' factor- 'switch' BCS

Snippet 15	Snippet 16

int a=9, b=3;
switch(b)
{
case 3:
a=a+4
b=a/b;
break;
default:
b=a/b +5 ;
}
printf("\n%d" , b);
	int a=9, b=6;
	switch(a)
	{
	case 6:
	a= b+3;
	b=a-12;
	break;
	default:
	b=a/b ;
	b=b -5;
	}
	printf("\n%d" , b);

E. Variable Change Source

Here is another factor. We call it Variable change source. The essence of this factor is that when calculating the new variable value, the source is important. It should be that variable change done from same variable should be easy in terms of mental load than in the case of variable change done from another variable. Because in later case this would involve more mental shifting than the former case. Perhaps it is possible that the effect does not shows up in case of lesser number of variables, but it should show itself in case of larger number of variables. But all these conjectures need to be verified by repeated experiments done at various levels.

In the snippet pair (17 & 18) shown below in Table 10, Loop in 'for' constructs run 2 times for both the snippets. However, the snippet 17 changes the 'j' value using old variable value and in snippet 18 variable 'j' changes its value from different variable 'i'.
Thus, to us snippet 18 should be more of load to calculate than snippet 17.

Table- X: Snippet pair 9 demonstrating 'Variable change source' factor- 'for' BCS

Snippet 17	Snippet 18

int i,j=3;
for{ i=10;i>2;i=i/3)
{
j=j+4;
}
printf("\n%d" , j);
	int i,j=7;
	for{ i=2;i<4;i=9-j)
	{
	j=i*4;
	}
	printf("\n%d" , i);

In similar vein, in Table 11 we can say that snippet 19 should be less of cognitive load on human mind than snippet 20. One of the reasons for that should be that variable change from snippet 19 is from same variable and in case of snippet 20 is from different variable.

Table-XI: Snippet pair 10 demonstrating 'Original Vs modified' factor- 'function' BCS

Snippet 19	Snippet 20

int funct (int a)
{
a= 5-a;
return (a)
}
int b;
b= 4*funct(9);
printf("\n%d" , b);
	int funct (int a,int b)
	{
	b= 16/a;
	return (a-b);
	}
	printf("\n%d", funct(6,2));

V. RESULT DISCUSSION

In previous section we had identified some of the mental factors which contributes in adding to the complexity of software code.

Table-XII: Summary of Snippet pairs and mental factor

Snippet Pair	BCS	CV	VU	VM	OM	VC
1	While-loop	✓			✓	✓
2	Sequence	✓				✓
3	While-loop	✓	✓	✓		✓
4	Function call	✓	✓			✓
5	If else			✓		
6	Do-while loop			✓		
7	While-loop				✓	
8	Switch-case			✓	✓	
9	Function call				✓	✓
10	While-loop		✓		✓	✓

In order to explain the subtleties of mental factors, we have taken examples of Ten (10) snippet pairs. Table -XII summarizes the 10 code snippets pairs defined in previous section (Table I – XI). In Table XII mental factors are written is short form such that:

CV = Constant Vs Variable
VU = Variables Used
VM = Variables Modified
OM = Original Vs Modified Use of variable.
VC = Variable Change Source

The codes in snippet pairs are carefully chosen. Most of existing software complexity measure would give same complexity to both snippets in each pair -because of same external visible properties of the code. But our contention is that these snippets differs in term of cognitive load thus, making one of the snippets more difficult to comprehend than the other. The salient features of our code snippets pair study as summarized in Table XII are:

- Each pair is identical in many external ways – number of variables used, number of operators used, Basic control Structure (BCS) in the code etc.- and yet have these mental factors distinguishing between the two snippets in pair.
- More than one factor is at play while distinguishing the codes in snippet pair. As shown in Table XII, only snippet pair 5 & 6 have single distinguishing mental factor. All other snippet pairs have multiple distinguishing mental factors – varying from 2 to 4.
- Furthermore, it's quite possible – even more likely- that these five mental factors are not completely disjoint. Possibility of one factor submerging in another larger factor is quite there. From our data set in Table XII it does seems that CV factor is subset of VC factor. But more conclusive generalization is left to future researchers.
- The variation in code complexity of snippet pair exists due to peculiar nature in which human brain process information.
- The underlying mental factors can exist in any of Basic Control structure (BCS) as has been amply demonstrated in Table XII.

To the best of our understanding these factors identified in above section are not taken into consideration while measuring the software complexity by any of the existing software metrics. The idea of presenting code snippets in pair is to highlight the fact that even if on the external view of code, essential parameters are same, still the human brain is likely to comprehend with different sense of ease. It's entirely possible that more than five of such mental factors exists or that some of these mentioned mental factors overlap considerably and there are lesser number of such mental factors – same or different.

We believe that none of existing software complexity fully capture the complexity as they don't consider the underlying mental factors. We thus suggest that new measures be developed which includes these factors to better capture the software code complexity.

VI. FUTURE SCOPE AND CONCLUSION

In-spite of slew of metrics proposed there is no well accepted measure of software complexity.

Program comprehension also seems to be too tricky to be understood as of now[15], [19], [25]–[28].
One reason for that could be difficulty in identifying the mental factors which plays up while performing a code comprehension. In true sense software creation is mental activity and its various related activities like code comprehension is also deeply mental in nature. This in a way explains how a good software complexity measure is elusive to mankind despite all the persistent efforts. In this paper we identify some of the mental factors which could make the code comprehension more complex. We identified five such factors and explained it by taking pair of code snippets which are essentially same on many parameters – variable count, operator count, Basic software construct- but differ on the specific mental factor we intend to explain. Most of existing software complexity measure either do not take into consideration these mental factors at all or they only touch one or two factors and that too indirectly. One work in 2010 do indirectly refers to variable entangling factors, but it is not the same as the factors we are talking about[29].

Going further we suggest that existing program comprehension study techniques – time and correctness study, Eye tracking, EEG, fMRI of human brain- should be applied on something like any of code snippets pair mentioned in previous section. The idea is twofold. First to prove the distinct existence of such factors and second to include these factors in new proposed software complexity metrices. We further suggest that these factors can be corelated with standard psychological experiments pertaining to working of human brain. We suggest that body of code snippets of varying difficulty levels in various language are designed for all these mental factors explained above. The larger idea is that such body of work can be used by future researchers to conduct various experiments and many entangled questions -pertaining to code complexity- can be answered. This will go long way in developing and validating the truly acceptable software complexity measure. This could be helpful in education of computer programming and may help us spot very early those human talent which are likely to excel in software development.

REFERENCES

1. A. J. Albrecht and J. E. Gaffney, “Software Function, Source Lines of Code, and Development Effort Prediction: A Software Science Validation,” IEEE Trans. Softw. Eng., 1983.
2. M. H. Halstead, Elements of Software Science. North Holland, 1978.
3. T. J. McCabe, “A Complexity Measure,” IEEE Trans. Softw. Eng., vol. SE-2, no. 4, pp. 308–320, 1976.
4. C. Ebert and J. Cain, “Cyclomatic Complexity,” IEEE Softw., 2016.
5. J. Shao and Y. Wang, “A new measure of software complexity based on cognitive weights,” Can. J. Elect. Comput. Eng., vol. 28, no. 2, pp. 1–6, 2003.
6. Y. Wang, “Cognitive Complexity of Software and its Measurement,” in 5th IEEE International Conference on Cognitive Informatics, 2006, pp. 226–235.
7. Y. Wang, “On the Cognitive Complexity of Software and its Quantification and Formal Measurement,” Int. J. Softw. Sci. Comput. Intell., vol. 1, no. 2, pp. 31–53, 2009.
8. A. K. Jakhar and K. Rajnish, “A new cognitive approach to measure the complexity of software’s,” Int. J. Softw. Eng. its Appl., vol. 8, no. 7, pp. 185–198, 2014.
9. O. Esther, O. Stephen, O. Elijah, A. Rafiu, T. Dimple, and Y. Olajide, “Development of an Improved Cognitive Complexity Metrics for Object- Oriented Codes,” Br. J. Math. Comput. Sci., vol. 18, no. 2, pp. 1–11, Jan. 2016.
10. S. Misra, A. Adewumi, L. Fernandez-Sanz, and R. Damasevicius, “A Suite of Object Oriented Cognitive Complexity Metrics,” IEEE Access, vol. 6, no. January, pp. 8782–8796, 2018.
11. S. K. Dey, S. S. M. Tariq, M. S. Islam, and G. M. M. Bashir, “Cognitive complexity:A model for distributing equivalent programming problems,” in ECCE 2017 - International Conference on Electrical, Computer and Communication Engineering, 2017.
12. L. Jain and S. Singh, “A journey from cognitive metrics to cognitive computers,” IJARET, vol. 4, no. 4, pp. 60–66, 2013.
13. V. Gruhn and R. Laue, “On Experiments for Measuring Cognitive Weights for Software Control Structures,” in 6th IEEE International Conference on Cognitive Informatics, 2007, no. September 2007, pp. 116–119.
14. D. R. Wijendra and K. P. Hewagamage, “Automated tool for the calculation of cognitive complexity of a software,” in Proceeding - 2016 2nd International Conference on Science in Information Technology, ICSITech 2016: Information Science for Green Society and Environment, 2017.
15. S. Ajami, Y. Woodbridge, and D. G. Feitelson, “Syntax, Predicates, Idioms - What Really Affects Code Complexity?,” in 2017 IEEE/ACM 25th International Conference on Program Comprehension (ICPC), 2017, vol. 24, no. 1, pp. 66–76.
16. L. Jain and S. Singh, “DESIGNING THE CODE SNIPPETS TO MEASURE THE COGNITIVE WEIGHTS OF BCS,” 2019.
17. M. Klerer, “Experimental study of a two-dimensional language vs Fortran for first-course programmers,” Int. J. Man. Mach. Stud., vol. 20, pp. 445–467, 1984.
18. B. T. Mynatt, “The effect of semantic complexity on the comprehension of program modules,” Int. J. Man-Machine Stud., vol. 21, pp. 91–103, 1984.
19. B. Floyd, T. Santander, and W. Weimer, “Decoding the Representation of Code in the Brain: An fMRI Study of Code Review and Expertise,” in Proceedings - 2017 IEEE/ACM 39th International Conference on Software Engineering, ICSE 2017, 2017.
20. A. Jbara and D. G. Feitelson, “How programmers read regular code: a controlled experiment using eye tracking,” Empir. Softw. Eng., 2017.
21. T. Busjahn et al., “Eye Movements in Code Reading: Relaxing the Linear Order,” in IEEE International Conference on Program Comprehension, 2015.
22. I. Crk, T. Kluthe, and A. Stefik, “Understanding programming expertise: An empirical study of phasic brain wave changes,” ACM Trans. Comput. Interact., 2016.
23. M. V. Kosti, K. Georgiadis, D. A. Adamos, N. Laskaris, D. Spinellis, and L. Angelis, “Towards an affordable brain computer interface for the assessment of programmers’ mental workload,” Int. J. Hum. Comput. Stud., 2018.
24. S. Lee et al., “Comparing Programming Language Comprehension between Novice and Expert Programmers Using EEG Analysis,” in Proceedings - 2016 IEEE 16th International Conference on Bioinformatics and Bioengineering, BIBE 2016, 2016.
25. T. Blascheck and B. Sharif, “Visually analyzing eye movements on natural language texts and source code snippets,” in Eye Tracking Research and Applications Symposium (ETRA), 2019.
26. J. Siegmund et al., “Understanding understanding source code with functional magnetic resonance imaging,” Proc. 36th Int. Conf. Softw. Eng. - ICSE 2014, pp. 378–389, 2014.
27. J. Siegmund et al., “Measuring neural efficiency of program comprehension,” Proc. 2017 11th Jt. Meet. Found. Softw. Eng. - ESEC/FSE 2017, pp. 140–150, 2017.
28. N. Peitek et al., “A Look into Programmers’ Heads,” IEEE Trans. Softw. Eng., vol. 5589, no. c, pp. 1–20, 2018.
29. D. Beyer and A. Fararooy, “A simple and effective measure for complex low-level dependencies,” in IEEE International Conference on Program Comprehension, 2010.

AUTHORS PROFILE

Dr. Leena Jain, Professor and Head in Department of Computer Applications, Global Group of Institutes, Amritsar is a leading young researcher in the area of operation research and Artificial Intelligence. She is having 14 years of teaching experience.

Dr. Leena did her Ph.D from Punjabi University Patiala in 2011 and have dual master degree MCA and M.Sc. Mathematics. She has published more than 55 papers in various international and national Journals. She is also a reviewer in many international journals. Under her guidance 3 Ph.Ds. Scholar award their Ph.D. degree and presently two are pursuing. She has given many invited talks at various national and international conferences and chaired many scientific sessions.

Mr Satinderjit Singh is working as a Associate professor in Department of computer Applications GGNIMT, Ludhiana. He is research scholar in PTU, Kapurthala, Punjab, India. He is pursuing his Ph. D in Computer Science and Engineering. He has a 18 years of teaching experience.