Incidence of Einstellung Effect among Programming Students and its Relationship with Achievement

Jun Rangie C. Obispo
Department of Computer Science
Xavier University – Ateneo de Cagayan
Cagayan de Oro City, Philippines
jobispo@xu.edu.ph

Francis Enrique Vicente G. Castro
Department of Computer Science
Worcester Polytechnic Institute
Worcester, Massachusetts, USA
fgcastro@wpi.edu

Ma. Mercedes T. Rodrigo
Department of Information Systems and Computer Science
Ateneo de Manila University
Quezon City, Philippines
mrodrigo@ateneo.edu

ABSTRACT

The Einstellung effect (EE) refers to an individual's bias towards a familiar, working solution when solving problems even though more appropriate solutions are available. Prior studies have shown that this fixation may pose some problems when the known solution can no longer be used because this prevents one from being able to generate other solutions. In this paper, we investigate EE in the context of programming and how this phenomenon affects the performance of student programmers in a single laboratory experiment. We observed that about 33% of the students exhibited a full incidence of EE where solutions to three problems used the same category of approaches. Twenty-four percent of students exhibited partial EE, where two of three problems had similar approaches. Forty-two percent of students did not exhibit EE at all. We also observed that students with higher pre-test scores exhibited more incidences of EE. Those who exhibited more EE also performed better in solving the programming problems in terms of the number of correctly implemented plans. This study shows that EE has a positive effect on the performance of student programmers, at least in a single programming activity. This opens opportunities to further explore the effect of EE on the overall performance of students in programming.

CCS CONCEPTS

Social and professional topics → Professional topics → Computing education → Computing education programs → Computer science education

KEYWORDS

Einstellung effect; Experience bias; Computer programming.

ICE2018, October 2018, Cebu City, Philippines

https://doi.org/10.1145/1234567890

ACM Reference format:

Jun Rangie C. Obispo, Francis Enrique Vicente G. Castro and Ma. Mercedes T. Rodrigo Surname. 2018. Incidence of Einstellung Effect among Programming Students and its Relationship with Achievement. In Proceedings of Information and Computing Education Conference (ICE 2018), Cebu City, Philippines, 8 pages.

1 Introduction

The Einstellung Effect (EE) refers to an individual's bias towards a familiar, working solution to solve problems even though more appropriate solutions are available [9, 10]. EE is a fixedness on a problem solving method because of prior experience, hindering a more appropriate formulation of solution [1, 2, 11].

In the context of programming, EE poses problems because programmers tend to utilize familiar and possibly ineffective solutions. Reuse, however, is not necessarily negative. Indeed, code reuse is supported and encouraged to speed up software development and minimize both effort and the probability of error. For example, experienced programmers reuse "canned solutions" to previously encountered problems when solving new ones [12]. Programmers regularly employ known programming constructs, built-in functions, and algorithms when they encounter unfamiliar problems [6, 7].

This study aims to investigate the implications of EE among student programmers, particularly on their selection and construction of plan structures in addressing similar problems. This study focuses on the fixation on approaches they use to solve programming problems and how an intervention affects this fixedness.

Specifically, we ask the following questions:

1. To what extent do programmers exhibit EE?
2. How does the order of programming problems affect the incidences of EE?
3. What is the relationship between programmer expertise and the incidences of EE?
4. What is the relationship between the incidences of EE and student performance?
5. How does an intervention affect the incidences of EE?

2 Related Work

2.1 The Einstellung Effect

The Einstellung Effect is a psychological phenomenon wherein one forms a bias towards a familiar, working solution to solve other problems even though more appropriate solutions are available [9, 10].

The study of this phenomenon was started by Luchins [9] in his water-jug experiment. Test subjects were asked to come up with a combination of three water jugs, say A, B, and C, in order to reach a desired quantity of water. In this experiment, the first group was exposed to ten problems, where, for each problem, the desired quantity and water jug capacities are different. For the first few problems, the participants were able to discover a formula (B-2C-A) that could be used to reach the goal. In the next few problems, a simpler formula (A+C) can be used to solve the problems. However, majority of the test subjects stuck to the previous and complex formula in trying to solve the problems, failing to see a simpler solution that was available. Moreover, many of the subjects failed to answer the last problem where the complex formula can no longer be used, but the simpler one would. Notably, the solution they know that worked stuck to them and they tried to use this solution to solve all other problems, making them unable to see a simpler solution, and even leaving them unable to answer problems that required simpler solutions.

The second group of test subjects in [9] were exposed only to the last five (5) problems that were given to the first group and notably very few of them used the complex formula to solve the problems, and only few were unable to solve some of the problems. The fixation that was exhibited by the first group was less seen in the second group.

Evidences of fixedness have been noticed in the context of programming. Plans used by programmers are actually reused, based from their experience [12]. When addressing unfamiliar problems, novices bank on known programming constructs, built-in functions, and algorithms [6, 7]. These evidences, however, pertain only to incidences where programmers rely on experience in addressing problems – and these experiences tend to be helpful. The negative implications of experience bias need further investigation.

2.2 Plan Composition

Soloway investigated the problem solving strategies of programmers by looking into their plan compositions [12]. A *plan* refers to the organization and clustering of the subtasks of a problem [3] and is referred to by Soloway as a "canned solution" that a programmer knows to address a given problem [12]. Complex problems may be decomposed into several subproblems, each with their own plans. The combination of these various plans to create a single, comprehensive plan is known as *plan composition*.

Various programming problems have been used to study plan composition among novice programmers. We use some of these problems in this study. One is the Rainfall problem [12] described below:

```
Write a program that will read in integers
and output their average. Stop reading when
the value 99999 is input.
```

Another problem that appears to be different but actually has an almost isomorphic plan structure with Rainfall is the TF problem [12]:

```
Write a program that will output 'T' if all
the inputs are 'T', but output 'F' if there
is just one 'F' in the input sequence. Stop
reading when '#' is input.
```

Lastly, the Adding Machine problem [4] is described as follows:

```
Design a program called adding-machine that
consumes a list of numbers and produces a list
of the sums of each non-empty sublist
separated by zeros. Ignore input elements
that occur after the first occurrence of two
consecutive zeros.
```

This study takes interest on the possible relationship of the plan structures used by the programmers to solve the three problems above (Rainfall, TF, and Adding Machine) if they are given in succession. A potential scenario is that the programmer will use a single loop to solve the Rainfall problem, and will use the same approach to solve the TF and the Adding Machine problems. This fixation may help in being able to solve the TF problem, but may pose difficulties in addressing Adding Machine.

Though Jones cited some techniques to manage EE [8], these interventions occur at the level of course design such as the examples given throughout the course. This study looks into suggestions that will prevent or minimize this phenomenon in a particular single programming activity setting.

We are interested in investigating EE in the context of programming. To do that, we examined students' plan composition in solving a series of programming problems that could be solved with similar approaches -. We see if some extent of fixedness would be a factor in being able or unable to solve the given problems. Given the coding schemes employed by Castro and Fisler [4, 7], this study investigates the similarities of the plan structures of the programs of the students. Three programming problems were used where the plan structures show similarities: (1) the Rainfall problem as coined by Soloway [12]; (2) TF problem as still suggested by Soloway [12]; and lastly, the Adding Machine used by Castro and Fisler [4].

The choice for the Rainfall and TF problems is mainly because of the unobvious isomorphism of their plan structures. Looking at the problem statements alone, they seem to be very different. However, the plan compositions are "almost" identical. As [12] pointed out, both of the problems require a sentinel to stop reading input, the Rainfall problem requires an accumulator for the sum while the TF problem needs a flag that will set/reset depending on the input, and

both problems guard if there is no input given to give appropriate output.

The Adding Machine problem, on the other hand, may look similar to the Rainfall problem because they both need to accumulate sums and require a sentinel to terminate input, but the former has more intricate plan composition. The Adding Machine needs to separate sublists (separated by 0), and just accumulate the sum for each sublist. In addition, the solution needs to accumulate the sums of the sublists. Although, the same approach could be used to solve the Rainfall and Adding Machine problems (including the TF problems), composing the plans need subtle differences.

3 Methodology

3.1 Participants

Participants of this preliminary study were seventy-three (73) sophomore to senior computer science and information technology students from two universities in Mindanao: Xavier University – Ateneo de Cagayan in Cagayan de Oro City and Central Mindanao University in Bukidnon. Students from some programming courses from these universities were asked to be the participants. 49 (67.12%) of the participants were male and 24 (32.88%) were female.

3.2 Experiment Setup

The experiment started with a briefing about the general flow of the experiment and the participants answered pre-test questionnaires that aims to assess their knowledge in the basics of programming. Questions included the topics on basic I/O, control structures, and arrays.

Afterwards, the participants were asked to solve the three programming problems. Students were given forty minutes to solve each problem. After the allotted time, their source code should be uploaded to a submission link provided, and they will then proceed to the next problem. The specific problem statements are given in the next subsection.

Two treatments were given to the participants: (1) with and without intervention, and (2) original order of the problems and re-ordered problems.

After the programming activity, students were asked to complete an exit survey where they briefly discussed their solutions and other possible solutions to the problems given.

3.3 Programming Problems

3.3.1 The Rainfall Problem. Design a program called rainfall that takes in a series of numbers representing daily rainfall amounts as entered by the user. The input terminates once the number -999 is entered. Compute for the average of the non-negative values from the input. There may be negative numbers other than -999 in the input.

Sample input test cases are: (a) 41, 675, 72, 244, -9, 482, -1, 0, -999; (b) -4, -66, -90, -999; and (c) -999.

Output for the test cases above are: (a) 252.33; (b) cannot compute for average; and (c) cannot compute for average

3.3.2 The TF Problem. Write a program that will output 'T' if all the inputs are 'T', but output 'F' if there is just one 'F' in the input sequence. Otherwise, output 'X'. Stop reading when a '#' is input.

Sample input test cases are: (a) T, T, T, T, #; (b) T, F, T, T, #; (c) T, F, T, F, T, T, #; and (d) #.

Output for the test cases above are: (a) T; (b) F; (c) X; and (d) no input.

3.3.3 The Adding Machine Problem. from the user and produces a list of the sums of each non-empty sublist separated by zeros from the input. Stop input after the first occurrence of two consecutive zeros.

Sample input test cases are: (a) 9, 5, 7, -3, 2, 0, 3, 5, 0, 0; (b) 5, 6, 0, 0; and (c) 0, 0.

Output for the test cases above are: (a) 20, 8; (b) 11; and (c) no input.

3.4 Plan Structure Coding

To better look into the plan composition of the submitted programs, the plan structures were coded. The coded plans would help us see better the similarities of how plans were composed to solve the problems. The required plans or tasks are taken from Castro, Fisler, Ebrahimi, and [4, 5, 7, 12] while the methods for coding the plans structures are adapted from Fisler [7].

*3.4.*1 Required Plans. *Table 1 shows the required plans to solve the problems. However, students may use additional plans should they wish.*

Table 1: Plans and Codes for the Programming Problems

Plan	Purpose	Code
Rainfall Problem		
Read	Read input from user	R
Sentinel	Stop input if sentinel value	T
Negative	Ignore negative inputs	N
Sum	Total the non-negative inputs	S
Count	Count the non-negative inputs	C
DivZero	Guard against division by zero	D
Average	Average the non-negative inputs	A
Output	Print the average	O
TF Problem		
Read	Read inputs from user	R
Sentinel	Stop data input after the "#"	T
InvInput	Ignoring invalid input (not "T"s and "F"s)	I
Flag	Check how many "F" in input	F
Size	Check if T/Fs were entered	S
Output	Display proper output	O
Adding Machine Problem		

Read	Read inputs from user	R
Sentinel	Stop data input after the double-zero	T
Sublists	Identifying sublists separated by single zeros	L
Sum	Summing the elements in each sublist	S
OutputBuild	Building the output list from the sums of the sublists	B
Output	Display sums	O

3.4.2 Operators. To denote how the plans are composed to create one working solution to the problem, we use the following operators. The symbols enclosed in parenthesis for each operator are used in the coding.

- Sequential (;) – plans are executed in order; after the execution of plan A, plan B automatically follows.
- Interleaved (&) – plans are weaved together; after some code in plan A is executed, some code in plan B is executed; similarly, after some code in plan B is executed, some code in plan A is executed.
- Parallel (|) – plans can be executed in either order; the execution of plan A does not affect the execution of plan B.
- Guarded (→) – the execution of plan A affects the execution plan B; used to denote conditions or branches

3.4.3 Sample Approaches and Coding. The three problems could be solved in multiple approaches. Two of the most common approaches are the Single-Loop and the Input-First. One of the mentioned approaches, the Single-Loop approach, is presented as an illustration for the three problems. The pseudocodes are shown in Figure 1.

```
repeat until find sentinel {
        get input
        if input is non-negative
                increment count
                add it to running sum
}
if count is at least 1
        compute the average as sum/count
        output average
else report "no data"
```

(a) Rainfall Problem

```
repeat until find sentinel {
        get input
        increment size
        if input is "F"
                increment flag
}
if size greater 0
        if flag = 0
                output "T"
        else if flag = 1
                output "F"
        else
                output "X"
else
        output "no data"
```

(b) TF Problem

```
repeat until two succeeding zeros{
        get input
        if input = 0
                increment count of succ zeros
                if count of succ zero = 0
                        inc counter of sums list
        else
                reset count of succ zeros
                add input to current sum
}
```

(c) Adding Machine Problem

Figure 1: Single-Loop Approach

The sample set of Single-Loop approach pseudocodes above can then be coded as follows, as shown in Table 2.

Table 2: Sample Plan Structure Codes

Problem	Single Loop
Rainfall	(T→(R;(N→(C|S))));D→(A;O)
TF	(T→(R ; F & I)) ; S → O
Adding Machine	(T→ (R ; (L → (S | B))) ; O

The solution to the Rainfall problem presented in Figure 1 begins by asking for an input (R). Then the input is checked whether it is non-negative or not (N). If it is, the counter for how many inputs were taken is incremented, and the input is added to the running sum (N→(C|S)). Incrementing the counter and accumulating the sum can be in any order, hence C|S. This continues until the sentinel is encountered, i.e. the sentinel guards the execution of these plans (T→(R;(N→(C|S)))). Once the sentinel is encountered, the loop terminates. The program then checks if there was at least one input to guard against division by zero (D). If there is at least one non-negative input, the average is computed and printed (D→(A;O)). This gives the final plan structure code of the Rainfall solution presented as (T→(R;(N→(C|S)))); D→(A;O).

The TF problem is coded similarly. The sentinel guards the execution of the reading of input, flagging of the value for "T"/ "F"/ "X", and ignoring of invalid inputs (T→(R ; F & I)). Then the program checks if there was at least one valid input, and then appropriate output is displayed (S→O). Hence, the code for this solution is (T→(R ; F & I)) ; S → O.

Lastly, the Adding Machine, although it has a similar solution, the composition of plans is more intricate specially in the part of determining sublists separated by zeros. The whole loop is still guarder by the sentinel (T). Inside the loop, the inputs are taken (R). Then for every input, the program determines if it must separate the sublist, i.e., a single 0 is encountered (L). If the input still belongs to the current sublist, the input is added to the running sum of the current sublist. If the current input is 0, then the list of sums of

sublists is built (B). Hence, we get the coding for this block as L→ (S | B). The entire loop therefore is coded as (T→(R ; (L→(S|B))). Then, the output is simply displayed, leaving the final plan structure code as (T→(R ; (L→(S | B))) ; O.

4 Results and Discussion

4.1 Sample Incidence of Einstellung Effect

For this subsection, we present an example of an incidence of EE. We quantify an incidence of EE by counting how many of the problems are under the same category of approach. A student is said to have exhibited a full incidence of EE if he/she makes use of the same or similar approach for all three solutions. A student is said to have exhibited partial incidence if two of the three solutions are similar. Students whose solutions to the three problems are different are said to have no incidence of EE. Figure 2 below illustrates a sample of an incidence of a full-EE from a participant of the study. Some parts of the code are omitted for brevity like variable declarations, etc.

```
public static void main(String[] args){
   …
   do{
      S.o.print("Enter: ");
      input = in.nextDouble();
      if (input>=0){
        flag=1;
        count++;
        sum += input;
      }
      else continue;
   } while(input!=-999);
   if (flag == 0){
      S.o.p("Can't compute average");
   }
   else
      S.o.p(sum/count);
}
```

(A) Rainfall problem

```
public static void main(String[] args){
   …
   do{
      counter++;
      System.out.print("Enter: ");
      input = in.next().charAt(0);
      if(input == 'F'){
        flag++;}
   }while(input != '#');

   if(flag == 0 && counter != 0){
      System.out.println("T");}
   if(flag == 1){
      System.out.println("F");}
   if(flag > 1){
      System.out.println("X");}
   if(counter == 0 ){
      System.out.println("No input");}
}
```

(B) TF problem

```
public static void main(String[] args){
   …
   do{
      System.out.print("Enter: ");
      input = in.nextInt();
      if(input == 0){
        flag++;
        if(flag!=1)
                sums.add(sum);
                sum = 0;
      }
      else{
        counter ++;
        flag = -1;
        sum += input;
      }
   }while(flag!=1);
   if(counter==0){
      System.out.println("No input");
   }
   else{
      for(int a=0 ; a<sums.size(); a++){
        S.out.print(sums.get(a) + ", ");
      }
   }
}
```

(C) Adding Machine problem

Figure 2: Sample Full Incidence of Einstellung Effect

The figure above shows a full incidence of EE. All the solutions are under the category Single-Loop. The plan structure coding of the programs above is shown in Table 3.

Table 3: Coded Plan Structures of the Sample Incidence

Problem	Plan Structure Code
Rainfall	**(T→(R; N→C;S));** D→(O&A)
TF	**(T→(R; F&I));** S→O
Adding Machine	**T→(R; L→B&S);** O

The solutions presented in Figure 2 and Table 3 used the Single-Loop approach. All reading of input and necessary processing of the data were done in a single loop. After the sentinel was encountered, i.e. end of input, the corresponding output were displayed. Plans in boldface from Table 3 highlights the single loop that corresponds to the main loop presented from the programs from Figure 2.

4.2 Incidences of Einstellung Effect

We observed EE in the plan structures of the students because similar approaches were used in solving the problems given. Table 4 shows the count for the incidences of EE, broken down further per classification of the students as either novice (pre-test score ≤ 7) or intermediate (pre-test score > 7).

Table 4: Breakdown of incidences of EE

Group	Full	Partial	None	Total Students
Novice	9 (22.50%)	8 (20.00%)	23 (57.50%)	40
Intermediate	15 (45.45%)	10 (30.30%)	8 (24.25%)	33
Total Incidences	24 (32.88%)	18 (24.66%)	31 (42.46%)	73

Majority of the students who did not exhibit EE (none) committed both logical and syntactic errors, and thus failed to implement the required plans correctly. Because of the number of errors, their solutions are categorized under "Error". Further, solutions categorized under "Error" were not considered as the "same approach". This means that, for example, a student has all three solutions tagged under the "Error" category, the incidence of EE is under "None".

Comparing the incidences of the two groups (novice and intermediate programmers), the intermediate programmers have exhibited more incidences compared to the novices. Using a *t*-test, we find a significant difference between the two groups ($t\ stat = -3.03$; two-tailed $t\ critical = 1.99$; two-tailed $p = 0.003$; $\alpha = 0.05$). We discuss further the relationship between the pre-test and the incidences of EE in Section 4.4.

4.3 Order of Problems and Incidence of EE

We determined whether the order of the problems has an effect on the incidences of EE. The participants were divided into two groups where each group answered the problems in different orders. The incidences for both groups were compared using a *t*-test. We found out that the order of the problems has a significant effect on the incidences of EE (t stat=2.36; two-tailed t critical = 1.99; two-tailed p=0.02; α=0.05).

The order of the problems was Rainfall, TF, and Adding Machine for Order X; then Adding Machine, Rainfall, and TF for Order Y. The means of the incidences of EE of the two groups are $\mu_{Order\ X} = 1.8$ and $\mu_{Order\ Y} = 1.09$. This shows that Order X has exhibited more incidences of EE. This could be attributed to the nature of the problems. In Order X, the first and second problems (Rainfall and TF) have isomorphic plan structures [12], i.e. the plan composition of the two problems have very similar structures. When the programmer has used an approach to solve the Rainfall problem, and used the same approach to TF and it worked, the same approach may have been used then to solve the last problem, the Adding Machine. Some students were successful when they used an "almost same approach" in solving the Rainfall and TF problems. A successful attempt is defined as having the program with at least four (4) plan errors only for both problems already while an "almost same approach" is defined as the use of approaches under the same category, e.g. Single-Loop, but may have two to four differences in the plan composition. The success rate is 60%, i.e., 15 successful attempts out of 25 total attempts.

In the case of Order Y, the first problem (Adding Machine) and second problem (Rainfall) have slightly different plan structures. The Adding Machine requires more intricate composition of the plans such as ignoring sublist delimiting zeros and summing up all values in the sublist. This difference may not trigger the students to use a similar approach to solve the second problem (Rainfall). None of the students from this group could solve the Adding Machine problem. However, those who tried a different approach for the Rainfall problem had a success rate of 23%, i.e., 3 out of 13 could do better with the Rainfall problem using a different approach than the Adding Machine problem. A different approach for the problems is determined if the approaches do not fall under the same category.

4.4 Programmer expertise and Incidence of EE

We now consider how the expertise of the programmers, determined by their pre-test scores, affect the incidences of EE. We found out that there is a significant positive correlation between the pre-test score and the incidence of EE ($r = 0.47$; $p = 0.05$; *t*-value = 4.46 while *t*-crit=1.99), i.e., the higher the pre-test score of the student programmer, the higher incidence of EE the student exhibits.

It was assumed that those with lower pre-test scores would find it harder to solve the problems, thus sticking to similar approach used on the previous problems. However, for this case, it is the other way around. Those with higher pre-test tend to use the same approach. Better background in the concepts of programming may not translate to the use of other possible approaches in solving other problems. Higher pre-test also translated to more plans correctly implemented, hence, students might not need to change the approach in solving the problems. This suggests that better equipped programmers tend to reuse known existing solutions, and will just fashion it to solve current problems. Since the same approach would still work, they need not use other approaches. Novices, on the other hand, might have involved trial and error in using various approaches in solving the problems. We discuss further the relation of EE and how the students performed in the programming problems in the next subsection.

4.5 Incidence of EE and Student Performance

We are interested in how EE affects the performance of the students in solving the last problem. The students were grouped according to the order of the problems and analysis shows that the incidences of EE have a significant positive correlation for both orders: Order X ($r = 0.41$; $p = 0.05$; *t*-value = 3.87; *t*-crit=2.02; $n = 40$) and Order Y ($r = 0.61$; $p = 0.05$; *t*-value = 6.49; *t*-crit=2.04; $n = 33$).

For the Order X, the first two problems have isomorphic plan structures [12]. As discussed in Section 4.3, successfully (or almost) solving the Rainfall problem would help them solve the TF problem. Applying the same solution then to the last problem (Adding Machine), they could fashion the approach that could

solve the problem. With little differences in the composition of plans, the Adding Machine problem is still "solvable" using the same approach they used to solve the previous problems. Out of the 20 full-EE incidences under Order X, 5 (25%) performed good, i.e., they were able to run their programs. On the other hand, 1 of 5 (20%) performed good when using a different approach for the third problem, i.e. Rainfall and TF problems were categorized under the same approach while Adding Machine is under different a different category.

However, in the case of Order Y, the first two problems have different plan structures (Adding Machine and Rainfall). Although, similar approaches can be used, but composition of the plans would be different. For this group, only 4 had full EE and 3 (75%) did better in solving the last problem with an approach similar to the previous. Further, it should be noted that none of these students were able to perform well on the first problem (Adding Machine). They did better on the second problem (Rainfall) and the third problem (TF) with average correctly implemented plans as 4.5 (perfect score of 8) and 4 (perfect score of 6) respectively. We could look at this as an incidence where on the first problem, they still could not figure a good approach to solve the problem. Then on the second problem, they were able to do better, and have applied a similar approach to the last problem.

In sum, we could say that EE might have helped the students in solving the problems. Having experience with the approaches they used helped them fashion these approaches to solve the problem they were solving.

4.6 Intervention and Incidence of EE

Finally, we look into how the intervention, in the form of a short video presenting some approaches in solving the Rainfall problem, affects the incidence of EE. Comparing the incidences for the group with and without intervention, we see a significant difference ($t\ stat = -1.08$; two-tailed $t\ critical = 1.99$; two-tailed $p = 0.29$; $\alpha = 0.05$). Further, Set B, the group with the intervention, has exhibited more incidences of EE.

The results show that the video did not "break" the fixation of the students with a particular approach is solving the problems. The video presented two possible approaches to solve the Rainfall problem: the common approach Single-Loop, and another approach Input-First. Even though the students were exposed to another possible strategy, it might not have compelled them to indeed use a different one. They knew that the approach they used worked, and they did not need to find another one. Further, some students who had the intervention said they used a similar approach for the third problem (Adding Machine for Order X and TF for Order Y) with that of the previously given problems.

5 Conclusion and Future Work

EE is defined as a phenomenon where one is fixated to a solution that already works, and will try to use this known solution to other problems even though more appropriate solutions are available. As previous studies in EE suggests, we suspected that it would be bad for novice programmers because fixedness to a particular approach may hinder them in solving other problems.

In this study, we investigate EE in the context of computer programming. Specifically, we wanted to find out to what extent to student programmers exhibit EE, how does programmer expertise relate to EE, how does EE relate to the performance of the students, and how does an intervention affect the incidences of EE.

We found out that 32.88% of the students used the same approach in solving all the three problems, while 24.66% used the same approach to solve at least two problems. These two combined shows that a majority (57.54%) of the students exhibited this phenomenon to some extent, as to only 42.46% have not used the same approach to any of the problems. We can attribute these incidences to EE since students can use other solutions but stuck to what they used, as what they mentioned in the exit survey.

On the order of how the problems were given, those who answered in the original exhibited more incidences of EE than those who answered the re-ordered sequence. This could be attributed to how the nature of the problems affected the way students have solved the problems.

We also found out that students with higher pre-test scores tend to have higher incidences of EE as well. This suggests that better programmers use more often the same approach to solve programming problems. Further, we also found out that using the same approach to solve the problems helped students do better in solving the given problems.

Comparing the incidences of EE between those who have undergone and have not undergone the intervention in the form of a short video presentation shows a significant difference. Those who had the intervention, in fact, exhibited more incidences of EE. The video, which presented two possible approaches in solving the Rainfall problem, did not compel them to use another approach. This could also be because using the same approach helped them to perform better.

There are several factors that might have influenced the study's outcome. First, this study has gathered only 73 students from Xavier University – Ateneo de Cagayan (XU) and Central Mindanao University (CMU). Further, those from XU are BSCS students while those from CMU are BSIT students. This diversity of background of the participants may have something to do with how they have fared in the research. Students from XU have undergone already two programming courses and a course in object-oriented programming while those from CMU only had one introductory programming course.

Second, we also limited the programmer expertise only to the pre-test scores. Other factors may come into play on the expertise of the programmers especially that the participants come from various backgrounds.

Third, the experiment runs for three complete hours, not considering a possible delay in starting the experiment or some other problems. Giving of instructions, answering the pre-test, solving the programming problems, and answering the exit survey were all packed in the single experiment setup. All these activities packed within three hours may have stressed the students as some of them said in the exit survey that they needed more time.

Fourth, the exit survey asked broad questions asking the students to describe their solutions and another possible solution to the problems. Majority of the students pointed out the use of other programming constructs like arrays, loops, etc., or other possible algorithm, but were not really elaborate on their statements. This could be because students were already rushing to complete this survey just to finish the experiment.

Therefore, we recommend the following for future studies. First, we recommend having this study conducted with participants having similar background like CS students only.
Second, we suggest broadening the definition of programmer expertise by including other aspects like final grades in programming courses, how long they have been programming, number of languages known, programming habits, etc.

Third, we also recommend a deeper analysis on the plan errors and how EE could have affected them. A lot of the participants failed to give sensible solutions to the problems. These solutions were tagged under one category (Error), but they could be further investigated why some of these required plans were wrongly implemented, or missing to be exact.

Fourth, we suggest that pre-experiment proper activities, i.e. orientation, demographics and pre-test, could be done on a separate session or allotting more time for the experiment so that things would not need to be rushed.

Fifth, we recommend having exit survey questions fashioned to really validate how they could use another approach in solving the problems. If they could not, more accurate questions could be asked to know if it is really the Einstellung effect that made them unable to do so.

Lastly, this study looked at this phenomenon on a series of programming problems only. We suggest extending this study covering a longer period, and focusing more on novice programmers, i.e., those who have just started to learn how to program. Focusing on these programmers would help us see better how does EE really affect the learning of computer programmers.

ACKNOWLEDGMENTS

The authors would like to thank Engr. Gerardo Doroja, Rhea Suzette Mocorro, Jessie Lagrosas of Xavier University - Ateneo de Cagayan, and May Marie Talandron, Charles Hanz Bautista and Kent Levi Bonifacio from Central Mindanao University for allowing the conduct of this study to their students. We are also very grateful to the participants who have given their time, effort, and talent for the completion of this study. This study will not be successful as well without the financial support from the Department of Science and Technology Engineering Research and Development for Technology.

REFERENCES

[1] Bilalić, M. and McLeod, P. 2014. Why good thoughts block better ones. *Scientific American*. 310, 3 (2014), 74–79.
[2] Bilalic, M., McLeod, P. and Gobet, F. 2008. Why good thoughts block better ones: The mechanism of the pernicious Einstellung (set) effect. *Cognition*. 108, 3 (Sep. 2008), 652–661.
[3] Castro, F.E.V.G. 2016. Pedagogy and Measurement of Program Planning Skills. *Proceedings of the 2016 ACM Conference on International Computing Education Research* (2016), 273–274.
[4] Castro, F.E.V.G. and Fisler, K. 2016. On the Interplay Between Bottom-Up and Datatype-Driven Program Design. *Proceedings of the 47th ACM Technical Symposium on Computing Science Education* (New York, USA, 2016), 205–210.
[5] Ebrahimi, A. 1994. Novice programmer errors: Language constructs and plan composition. *International Journal of Human-Computer Studies*. 41, 4 (1994), 457–480.
[6] Fidge, C. and Teague, D. 2009. Losing Their Marbles: Syntax-Free Programming for Assessing Problem-Solving Skills. *Proceedings of the 11th Australasian Conference on Computing Education* (Australia, 2009), 75–82.
[7] Fisler, K. 2014. The recurring rainfall problem. *Proceedings of the 10th Annual Conference on International Computing Education Research* (New York, USA, 2014), 35–42.
[8] Jones, M. 2007. The Redesign of the Delivery of an Introductory Programming Unit. *Innovation in Teaching and Learning in Information and Computer Sciences*. 6, 4 (2007), 169–182.
[9] Luchins, A.S. 1942. Mechanization in problem solving - the effect of Einstellung. *Psychological Monographs*. 54, 6 (1942).
[10] Luchins, A.S. and Luchins, E.H. 1959. Einstellung Effect in Social Learning. *The Journal of Social Psychology*. 55, 1 (1959), 59–66.
[11] McCloy, R., Beaman, C.P., Morgan, B. and Speed, R. 2007. Training Conditional and Cumulative Risk Judgements: The Role of Frequencies, Problem-structure and Einstellung. *Applied Cognitive Psychology*. 21, 1 (2007), 325–344.
[12] Soloway, E. 1986. Learning to Program = Learning to Construct Mechanisms and Explanations. *Communications of the ACM*.