Strategies of Novice Programmers

Marjahan Begum[1], Jacob Nørbjerg[1], and Torkil Clemmensen[1]

[1]Copenhagen Business School, DK-2000 Frederiksberg, Denmark
[1]{mbe, jno, tc}.digi@cbs.dk

Abstract. In this paper we study novice programmers' strategies during different phases of programming. Programming strategies are about cognitive processes that result in the programmers' code being written from scratch, edited or deleted. The paper presents a questionnaire, which will be used to investigate how and when novice programmers use different strategies and how this affects the quality of the resulting program. We identify the cognitive processes that novice programmers utilise to complete a programming task. These cognitive processes may or may not result in immediate changes in the code. The ultimate purpose of this ongoing research is to contribute to improve the initial programmer education.

Keywords: Strategies, behaviour, programming, learning, teaching, difficulties, novices, cognitive strategies

1 Introduction

This paper investigates the programming strategies of novice programmers. Programming strategies are about cognitive processes that result in the programmers' code being written from scratch, edited or deleted. These cognitive processes may or may not result in immediate changes in the code. There is an implicit and explicit implication that as a result of using these strategies the resulting program is close to the perceived desired solution of the problem specification.

The paper identifies the programming strategies during different phases of programming: 1) understanding and design 2) coding and 3) debugging and testing. A programmer in the understanding and design phase may, for example, describe the strategy used as "I identified what programming concepts (e.g .loops, conditions) I need to solve the problem". Here the novice programmer is considering a low-level requirement for the proposed solution. There is an established assumption that these three broad phases are iterative. They are iterative in that one strategy may have influence(s) on other phases of the programming process.

There is a high drop out rate both in CS and non-CS major (with programming content) study programs due to difficulties with learning programming[1], [2]. The pass rate was estimated to be approximately 67%[3]. Several factors have been identified as contributing to the high drop out rate: lack of motivation, lack of previous similar experience and commitment to the discipline.

The purpose of identifying novice programmer strategies is that they have influence on novice programmers' program quality. In an educational context this is quantitively

measured by grades in programming courses. Our hypothesis is that strong novice programmers consistently use more of certain strategies compared to weak novice programmers. This paper outlines the programming strategies based on previous research about novice programmers, and develops a quiestionnaire-based tool for studying novice programmer strategies.

The paper is part of a larger project aiming to understand the problem solving and programming behavior of novice programmers. The premise of the research is that if we understand the interactions between programmer and the programming process we, as educators, may be better equipped to improve the education of novice programmers. More specifically if we have a better understanding of the behaviour that results in stronger novice programming, we may be able to explicitly cultivate this behaviour.

2 Related research

2.1 Programming behavior of novice programmers

A number of studies have investigated novice programmers' behavior from different perspectives. One recent study worked on the premise that if educators have an accurate understanding of the mistakes that students are likely to make, then they can address specific misunderstanding or misconceptions. The research collected compilation events of novice programmers [4], [5]. Another study found that students' problem solving ability significantly correlates with performance on programming assignments[6]. Studies have shown positive relationships between novice programmers' self-efficacy and success in their CS course[7]. Despite the significant amount of research on students' attributes and programming performance, no conclusion has been made as to which of the candidate factors influence students' performance.

2.2 Programming Learning Challenges

The difficulties with learning programming have been well-documented [8]. Learning programming is a very difficult task as it requires problem solving skills that are complex and multi-dimensional. It is complex because the skill set includes specific cognitive skills that are not used in other areas of life/work. It is multi-dimensional because it first requires an understanding of the syntax and semantics of a new language (in this case a programming language). Secondly, it requires learning and understanding logics of the fundamental programming concepts (condition, loops, methods etc.). All this takes some time to absorb and learn. Then, finally, the student has to apply all these to solving a complex problem. A recent literature review empirically identified that understanding programming structure and designing a program to solve certain task as the most difficult tasks. In terms of programming concepts, the most difficult concepts are pointers and references, and abstract data types and error handling [9].

3 Framing the Research

Research methodologies used to study novice programmers generally involve interviews, questionnaires, introducing bugs in programs and letting students identify the bug for debugging behavior[10]. Very little research has investigated the behavior of programmers from understanding the problem specification to computer program[11]. The process of programming is dynamic and interactive, so studying just debugging behavior or compilation errors may not give us the holistic view an educator needs to address the challenge of learning to program. There has been a piece of research that examined the role of self-regulated learning and performance in introductory programming modules[12]. Self-regulated learning incorporates cognitive and meta-cognitive strategies in addition to resource management strategies and motivational aspects. The results showed that there is no correlation between cognitive strategies and programming performance but that there is a positive correlation between meta-cognitive strategies and programming. The cognitive strategies used in the scale were rehearsal strategies, elaboration strategies, and organisation strategies. The nature of cognitive strategies explored in Bergin's research does not reflect (except for the organisation strategies) the cognitive strategies actually used while programming by novice programmers such as reading and understanding the problem specification, designing codes or evaluating codes. Given the nature of cognitive strategies explored it is hardly surprising that no correlation was found. Another dated, but still relevant, research that takes an holistic approach to studying novice programmers investigated the relationship between cognitive styles and personality traits that have effects on different stages of programming[13]. Her research concluded that different cognitive styles will influence at different programming stages. In particular the research investigated the influence of field-dependence and field-independence on coding and designing. Field-independence is the tendency to bring structure to an unstructured situation. The research found that the cognitive style of field-dependence / field-in-dependence co-relates positively with the performance of design and coding but there is a higher correlation for design than for coding. Distinguishing factors between cognitive styles and strategies are, first, that of the ingrained characteristics of a programmer and, second, the ways in which the programming process is completed[13].

The role of meta-cognition and self-regulation is also relevant in the programming process. Metacognition is an individual's knowledge of their own cognitive processes and their ability to control these processes by organizing, monitoring and modifying them as a function of learning [14].

Investigation into meta-cognition and program comprehension showed that the use of meta-cognition influences how well programmers understand a program[15]. The research specifically developed a questionnaire-based tool that explored two comprehension strategies: top-down and bottom-up. It required participants to complete questions concerning meta-cognition. When a deliberate use of top-down or bottom up strategy was used that is an evidence that meta-cognitive strategies were employed.

On the basis of reviewing research on novice programmers, it can be concluded that a way of studying novice programming on all the phases of programming can help educators better understand challenges and difficulties novice programmers encounter.

More specifically we can identify cognitive and meta-cognitive strategies used by novice programmers while completing a program. This has been done in mathematics education research. Though mathematics problem solving and programming problem solving are fundamentally from different domains, they do have important similarities. Two specific studies collected data on students during problem solving using a questionnaire based tool[16], [17]. The first study collected data on 48 students while the second one collected on 42 students. Both studies showed correlations between the use of cognitive and meta-cognitive strategies and their ability of the students to successfully find solutions. Thus, mathematics education research provides the starting point for developing a questionnaire-based tool for investigating the relationship between cognitive strategies and programming performance.

3.1 Cognitive and Meta-cognitive Strategies and Processeses

There are different cognitive strategies at the task level that can be associated with a programmer's individual activity. These are carried out within the different stages of programming: understanding the problem; designing the program; coding; and debugging and testing. Broadly six types of cognitive processes can be distinguished (Lang 1997). These are: Understanding the problem; Coding; Evaluating the solution concepts; Testing (associated with debugging and testing); and Researching for information.

3.2 The research tool

A questionnaire-based tool (see appendix 1) was developed to investigate novice programmers' strategies and processes which are collectively here referred to as cognitive strategies. Based: the researcher's experience, previous research and the students' reports, items in the questionnaire were developed for the four different phases of programming: understanding the problem, designing, coding, and debugging and testing. To answer one of the questions, the students were asked to describe how they progressed in order to solve the exercise. Where possible, cognitive processes were identified from these reports and categorised such as: inserting printlin, commenting out, building code prototype.

Previous research has influenced the development of the items in the questionnaire [18]–[21]. Booth's rsults [18] were used for the stages of understanding the problem, design of the problem to the coding. His research from a phenomenographic perspective shows that novice programmers approach the programming problem in four distinct ways. These approaches were categorised from interview data collected from novice programmerssolving two problems using functional language SML. These four approaches are summarised in Table 1.

The research concluded that novice programmers who adopted structural and operational approaches to writing programs adopted deep approaches to learning and those that adopted constructual and expedient approaches to writing programs adopted surface approaches to learning. Booth's evidence also shows that there is a correlation between approaches and exam results. In particular, students who adopted structural

and operational approaches performed better than those who adopted constructual and expedient approaches (Figure 1).

Table 1. The four approaches to programming [18]

Programming approach	Description
Expedient	Produce a complete program from the outset by making use of an existing program or by adopting some known program
Constructional	Recognize details of the problem in terms of features of the programming language – construct, functions and keywords – which can be used to build a program
Operational	Write a program based on an interpretation of the problem within the domain of programming; the problem is considered from the point of view of what operations the program has to perform
Structural	Write a program based on an interpretation of the problem within its own domain; the structure of the problem is considered and on that basis a program is devised

This leaves the identification of strategies for the debugging and testing stage of programming, where novice programmers spend 35% of their effort. The challenge of identifying debugging strategies is that most research on debugging used subjects to debug already developed programs rather than debugging their own programs.

The debugging process for subjects in this research begins with the analysis and understanding of defects in their own developed program leading to less dependence on their comprehension strategies.

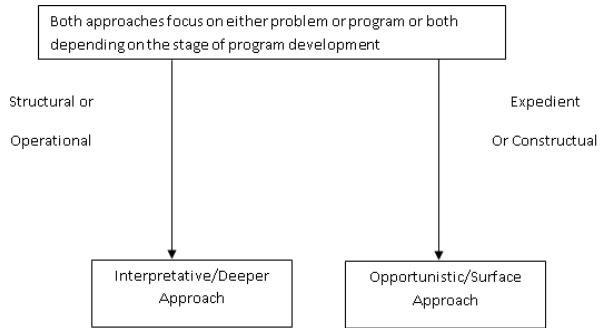

Fig. 1. Model of the behaviour adopted by novice programmers [18]

One of the first novice debugging models is illustrated in Figure 2 below [20].

Based on the observation of a typical novice programmer, the debugging process is usually initiated with the evaluation of the program which could be reading the code,

compiling the source code or testing. The program evaluation process can also begin when the programmer just finished a piece of code or a sub-routine.

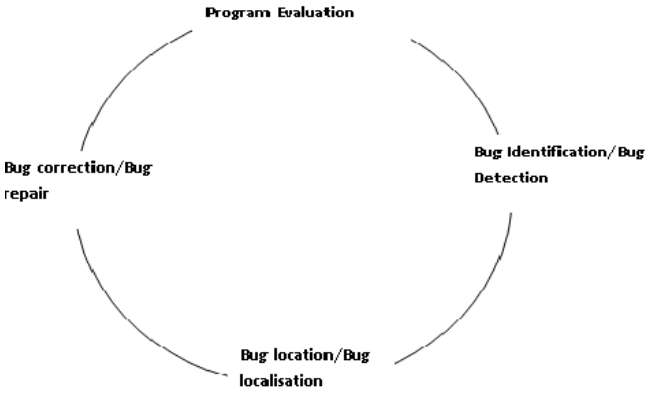

Fig. 2. The behaviour of novice programmers during the debugging stage [20]

Once the programmer is satisfied that there are no syntactical errors, the program is executed and evaluated with test data to see if it gives the correct output (as per the problem specification). If the program does not deliver the expected output, the debugging process starts from program comprehension or program evaluation as shown in the model in Figure 2.

Based on reviews of automated debugging systems and research on cognitive studies on debugging a classification system is shown below for knowledge required for debugging and successful debugging techniques [21].

Table 2. Debugging knowledge required for successful debugging

Knowledge type
Knowledge of the intended program
a. Program I/O
b. Behaviour
c. Implementation
Knowledge of the actual program
d. Program I/O
e. Behaviour
f. Implementation
Understanding of the programming language
General programming expertise
Knowledge of the application domain
Knowledge of bugs
Knowledge of debugging methods

As shown in Table 2, a wide range of knowledge is required to successfully debug a program. Novice programmers often find debugging a difficult activity because they only have limited debugging knowledge. In this section the debugging knowledge is discussed from novice programmers' point of view. Novice programmers gain knowledge of the intended and actual program through reading the specification and the process of developing the program. This contrasts with previous research on debugging where programmers debug programs written by other programmers.

In the present research the novice programmes do not need the knowledge of the intended program's input and output because it is explicitly written in the problem specifications in the form of what input data the intended program should take and what output it should show. This is associated with the data follow in the program.

Knowledge of the intended behaviour and intended implementation is gained partly from the problem specification and partly from reading, understanding and reflecting on the problem specification. Intended behaviour is associated with the control flow of the program.

It was observed that novice programmers often had difficulty in understanding the code they have written themselves when they find that the actual program does not behave (usually expected output) according to the problem specification. This initiates the process of program evaluation /comprehension where they begin the process of gaining the knowledge of actual input/output, actual behaviour and actual implementation of their own program.

Novice programmers have very limited knowledge of the programming language[1], and therefore, as a result, bug localisation and bug repair phases take longer. Novice programmers find deciphering error messages shown after the compilation process very time-consuming and difficult. Part of this difficulty can be attributed to the lack of programming knowledge and partly to understanding compiler error messages. Understanding of the compiler error messages is a knowledge domain of its own, which is part of the knowledge of the bug according to the classification.

Novice programmers often introduce logical bugs into their program, and students have logical misconceptions of programming, such as not using a temporary variable to swap two values in a variable [22]. Knowledge of logical bugs is acquired from the experience of making such mistakes and as well as from the experience of debugging when the mistakes are made.

General programming expertise is higher level knowledge which is also gained through experiences of solving different types of problems. These include programming plans, programming goals and interactions between goals and plans. Knowledge of the application domain is not relevant for this research as the problem domain is self-explanatory in the problem specification.

Debugging methods are strategies novice programmers utilise either as a result of having the debugging knowledge or to gain the debugging knowledge outlined above. One of the frequently used debugging strategies is to use `system.out.println` at various points in the program. One purpose may be to see if the control of the program

[1] In this research only 10% of the students had some, though often limited, experience of programming using Java programming language.

actually reaches a certain point in the program and therefore to gain knowledge of the actual behaviour of the program. This can occur either after executing the program and realising that control does not reach the certain point or it is simply used as a technique to understand the behaviour the program.

The purpose of the above discussion of previous research is to illustrate that statements identified in the questionnaire in the Appendix are grounded on previous work as much as it is grounded on the researcher's experience.

The cognitive strategies identified in this study include tasks associated with the direct interaction between the programming environment and the programmer which is similar to the first study (above) but there is a distinguishing factor which is best explained through examples:

- "I tried to see the problem as many small problems" is when the student is trying to simplifying the problem
- "I identified what programming concepts I needed to solve the problem" is when the student is considering the concepts requirement

4 Discussion and Conclusion

This research presents the development of a questionaire based tool to explore the strategies utilised by novice programmers in different phases of programming. Initial validation and reliability check of the questionnaire were carried out in a CS1 course.

One drawback of the tool is that strategies identified were party based on previous research that is over 12 years old and where the language that were used were not object-oriented programming. The next stage of the research would be check for further validity of the strategies based on educators who currently teach CS1 courses and on more recent research.

We will use the tool in further research to study the relationship between the programming strategies of novice programmers and the characteristics of the resulting code.

References

1. Wilusz, T., Almstrum, V., Diaz, D., Guzdial, M., Hagan, D., Utting, I.: A multi-national, multi-institutional study of assessment of programming skills of first-year CS students, *ACM SIGCSE Bull.*, 33(2), p. 125 (2001).
2. Petersen, A., Craig, M., Campbell, J., Tafliovich, A.: Revisiting why students drop CS1, In: *Proc. 16th Koli Call. Int. Conf. Comput. Educ. Res.*, pp. 71–80 (2016).
3. Watson, C.: Failure Rates in Introductory Programming Revisited, *Proc. 2014 Conf. Innov. Technol. Comput. Sci. Educ.*, pp. 0–6 (2016).
4. Brown, N. C. C., Altadmri, A.: Novice Java Programming Mistakes: Large-Scale Data vs. Educator Beliefs, *Trans. Comput. Educ.*, 17(2), 7-21 (2017).
5. Jadud, M. C., Dorn, B.: Aggregate Compilation Behavior, In: *Proc. Elev. Annu. Int. Conf. Int. Comput. Educ. Res.*, pp. 131–139 (2015).

6. Lishinski, A., Yadav, A., Enbody, R., Good, J.: The Influence of Problem Solving Abilities on Students' Performance on Different Assessment Tasks in CS1, In: *Proceedings of the 47th ACM Technical Symposium on Computing Science Education,* pp. 329–334 (2016).
7. Bhardwaj, J.: In search of self-efficacy: development of a new instrument for first year Computer Science students, *Comput. Sci. Educ.*, 27(2), 79–99, (2017).
8. Lahtinen, E., Ala-Mutka, K., & Järvinen, H.-M.. A study of the difficulties of novice programmers. *ACM SIGCSE Bulletin* 37, 14. New York, New York, USA: ACM Press. https://doi.org/10.1145/1151954.1067453 (2005)
9. Piteira, M., Costa, C.: Learning computer programming, In: *Proceedings of the 2013 International Conference on Information Systems and Design of Communication*, pp. 75 (2013).
10. Alqadi, B. S., & Maletic, J. I.: An Empirical Study of Debugging Patterns Among Novices Programmers. In: *Proceedings of the 2017 ACM SIGCSE Technical Symposium on Computer Science Education - SIGCSE '17*, pp. 15–20, New York, New York, USA: ACM Press. https://doi.org/10.1145/3017680.3017761 (2017).
11. Vihavainen, A., Helminen, J., Ihantola, P.: How novices tackle their first lines of code in an IDE, In: *Proceedings of the 14th Koli Calling International Conference on Computing Education Research* (2014).
12. Bergin S., Reilly R., Traynor D.: Examining the role of self-regulated learning on introductory programming performance, In: *First Int. Work. Comput. Educ. Res*., pp. 81–86, (2005).
13. Bishop-Clark, C.: Cognitive style, personality, and computer programming, *Comput. Human Behav.*, 11(2), 241–260 (1995).
14. A. P. Metcalfe, J., & Shimamura, Metacognition: knowing about knowing. Cambridge, MA: MIT Press., 1994.
15. Shaft, T. M., Vessey, I.: The Relevance of Application Domain Knowledge: Characterizing the Computer Program Comprehension Process, *Journal Manag. Inf. Syst*., 15(1) (1998).
16. Goos, M.: *Metacognition in Context: A Study of Collaborative Metacognitive Activity in a Classroom Community of Mathematical Inquiry,* The University of Queensland, Brisbane, (2000).
17. Biryukov, P. (2002). Metacognitive Aspects of Solving Combinatorics Problems. *International Journal in Educational Mathematics*, (74). Retrieved from http://www.cimt.org.uk/journal/biryukov.pdf
18. Booth, S.: *Learning to program: A phenomenographic perspective*, University of Gothenburg (1992).
19. Katz, I. R. Anderson, J. R.: Debugging: An Analysis of Bug-Location Strategies, *Human–Computer Interact.*, 3(4), 351-399 (1987).
20. Kessler, C. M., Anderson, J. R.: A model of novice debugging in LISP, In: Soloway, E., Sitharama I. (eds) *Empirical Studies of Programmer*, Norwood, NJ: Ablex, 1986.
21. Ducasse, M., Emde, A. M.: A review of automated debugging systems: Knowledge,strategies and techniques, In: *Proceedings of the 10th international conference on software engineering*, pp. 162–171 (1988).
22. du Boulay, J., B., H.: Some difficulties of learning to program. *Journal of Educational Computing Research*, 2(1), 57-73 (1986).

Appendix – Programming strategy questionaire

Username (e.g xyz) _____ Name : _____	**Definitely applies to me**	**Probably applies to me**	**Not sure if this applies or not**	**Probably does not apply to me**	**Definitely does not apply to me**	**Don' t know what this means**
Questions related to reading and understand the problem specification stage.						
1. I thought the problem was too difficult for me.	**A**	**B**	**C**	**D**	**E**	**F**
2. I thought the problem specification was too vague.	**A**	**B**	**C**	**D**	**E**	**F**
3. As soon as I read the problem specification, I could already see way(s) I could solve the problem.	**A**	**B**	**C**	**D**	**E**	**F**
4. I *immediately* thought about how I could structure my program.	**A**	**B**	**C**	**D**	**E**	**F**
5. My approach to the solution was by thinking about what the output will look like.	**A**	**B**	**C**	**D**	**E**	**F**
6. I thought about using specific Java syntax as soon as I read the problem specification.	**A**	**B**	**C**	**D**	**E**	**F**
7. Initially, I had problem(s) knowing where to start.	**A**	**B**	**C**	**D**	**E**	**F**
8. I *tried* to recall similar problem(s).	**A**	**B**	**C**	**D**	**E**	**F**
9. I *tried* to find information in the problem specification that would be helpful.	**A**	**B**	**C**	**D**	**E**	**F**
10. Before coding and designing I tried to see the problem as a problem consisting of *many smaller problems*.	**A**	**B**	**C**	**D**	**E**	**F**
11. Before coding and designing, I read the problem specification *more than once*.	**A**	**B**	**C**	**D**	**E**	**F**
12. It was important for me to understand how to solve the problem without thinking about the Java syntax.	**A**	**B**	**C**	**D**	**E**	**F**
13. There was information in the problem specification that I did not find helpful.	**A**	**B**	**C**	**D**	**E**	**F**
14. I *made sure* I understood the problem before starting any coding.	**A**	**B**	**C**	**D**	**E**	**F**
15. I *identified* what programming concepts(e.g loop, conditions) I needed to solve the problem.	**A**	**B**	**C**	**D**	**E**	**F**

16. I was able to *identify* specific section(s) of the problem specifications that would help me solve the problem. If you answered (A) can you describe that section(s). _____	**A**	**B**	**C**	**D**	**E**	**F**
17. I used pen and paper to help me better understand the specification. If you answered (A) what did you write/draw? _____	**A**	**B**	**C**	**D**	**E**	**F**
Questions related to program design stage.						
18. I had a concrete mental plan of how I was going to do the coding.	**A**	**B**	**C**	**D**	**E**	**F**
19. I thought about different approaches to solving this problem.	**A**	**B**	**C**	**D**	**E**	**F**
20. I approached the design of the solution in terms of what input it requires and what output it should print.	**A**	**B**	**C**	**D**	**E**	**F**
21. I broke down the problem into different parts. If you answered (A), in the space below can you write down the different components? Please also number them in the order they were developed. _____	**A**	**B**	**C**	**D**	**E**	**F**
22. I used pen and paper to plan the development of my program. If you answered (A), what did you draw/write? _____	**A**	**B**	**C**	**D**	**E**	**F**
Questions related to program coding stage.						
23. I went back to the specification because I got stuck with my program development.	**A**	**B**	**C**	**D**	**E**	**F**
24. I went back to the specification because my program would not give the correct output.	**A**	**B**	**C**	**D**	**E**	**F**
25. At some stage of coding I went go back to the specification to check what I was actually required to do.	**A**	**B**	**C**	**D**	**E**	**F**
26. I went back to the problem specification because I detected some errors during compilation.	**A**	**B**	**C**	**D**	**E**	**F**

27. While coding I went back to previously written code once or more to make changes.	**A**	**B**	**C**	**D**	**E**	**F**
28. I thought about what Java syntax/code I could use to finish the program.	**A**	**B**	**C**	**D**	**E**	**F**
29. I tried to remember if there was any code I could reuse (e.g previous exercise, course book).	**A**	**B**	**C**	**D**	**E**	**F**
30. I knew immediately I could use some code from elsewhere (previous exercise, text book etc.) to finish the program.	**A**	**B**	**C**	**D**	**E**	**F**
31. I have coded the input and output components of the program first.	**A**	**B**	**C**	**D**	**E**	**F**
32. Java syntax and semantics were major problems for me in proceeding with my program.	**A**	**B**	**C**	**D**	**E**	**F**
33. One main problem I had with my program was combining all the pieces of code together to get the required output.	**A**	**B**	**C**	**D**	**E**	**F**
34. I developed the program based on the way I thought was necessary until I was getting the right output.	**A**	**B**	**C**	**D**	**E**	**F**
35. There were times I was not sure whether my program was developing in the right direction. If you answered (A), please describe at least one specific time this happened. _____	**A**	**B**	**C**	**D**	**E**	**F**
36. I have written a piece of code even though I was not sure, but it made my program work.	**A**	**B**	**C**	**D**	**E**	**F**
37. I used Java syntax at least once without really knowing if it would work.	**A**	**B**	**C**	**D**	**E**	**F**
38. I used specific programming concepts (e.g loop, if, switch) only because it was mentioned in the specification.	**A**	**B**	**C**	**D**	**E**	**F**
39. I had to redo lot of coding. If you answered (A), please you explain what and why you had to redo. _____	**A**	**B**	**C**	**D**	**E**	**F**
40. I had to rethink the way I solved the problem at least once.	**A**	**B**	**C**	**D**	**E**	**F**
41. Knowing there was a better way of developing this program, I did not change my code.	**A**	**B**	**C**	**D**	**E**	**F**
42. There were times I got frustrated with programming. If you answered (A), please explain why you got frustrated? _____	**A**	**B**	**C**	**D**	**E**	**F**
43. There were times when I was confused about how to proceed.	**A**	**B**	**C**	**D**	**E**	**F**
44. I checked for compilation errors every time I finished a section of code.	**A**	**B**	**C**	**D**	**E**	**F**
45. I compiled the program quite regularly so that I could locate and fix errors in my code.	**A**	**B**	**C**	**D**	**E**	**F**
46. I wasted a lot of time locating errors with my program, only to realise I had simply missed a ";" or "}" or "{".	**A**	**B**	**C**	**D**	**E**	**F**

47. I submitted to the compiler when I was unsure of any syntax in Java.	**A**	**B**	**C**	**D**	**E**	**F**
48. It took me a long to fix one major syntactical error that I only got a very few times.	**A**	**B**	**C**	**D**	**E**	**F**
49. Even without compiling I knew there were some pieces of code I wrote which would not work.	**A**	**B**	**C**	**D**	**E**	**F**
Questions related to program debugging and testing stage.						
50. I knew something was wrong with my program because the value of the variable(s) needed for the calculation(s) did not have the right value.	**A**	**B**	**C**	**D**	**E**	**F**
51. I knew something was wrong with my program because the actual output is different from what the typical output should look like.	**A**	**B**	**C**	**D**	**E**	**F**
52. I traced my program line by line to understand how my program was behaving. If you answered (A), did you have difficulty with it? Yes or No	**A**	**B**	**C**	**D**	**E**	**F**
53. I traced my program to check if the control of my program goes to a particular section.	**A**	**B**	**C**	**D**	**E**	**F**
54. I manually identified change in the value of the variable(s) from one line of code to next.	**A**	**B**	**C**	**D**	**E**	**F**
55. I manually worked through my code to see if my program would give the correct result.	**A**	**B**	**C**	**D**	**E**	**F**
56. I had problems visualising what the correct program should look like.	**A**	**B**	**C**	**D**	**E**	**F**
57. At some point of the development, my program did not behave how it should behave.	**A**	**B**	**C**	**D**	**E**	**F**
58. I compared how my program works and how it should work to identify problem(s) with my program.	**A**	**B**	**C**	**D**	**E**	**F**
59. I was careful to insert comments in every major part of my program so that it would be easier for me to understand my program.	**A**	**B**	**C**	**D**	**E**	**F**
60. I commented my program so that I can better keep track of my thought process.	**A**	**B**	**C**	**D**	**E**	**F**
61. I made blocks of code inactive (e.g using "//") to narrow down source of error(s).	**A**	**B**	**C**	**D**	**E**	**F**
62. I made blocks of code inactive (e.g using "//") to see if the active code block(s) were working the way it should be.	**A**	**B**	**C**	**D**	**E**	**F**
63. I was able to solve most of the problems with my program.	**A**	**B**	**C**	**D**	**E**	**F**
64. I used "System.out.println()" at various points to locate the source of the problem in my program.	**A**	**B**	**C**	**D**	**E**	**F**
65. I used "System.out.println()" at various points in the program to see the value of the variable(s).	**A**	**B**	**C**	**D**	**E**	**F**
66. I used "System.out.println()" when there was mismatch between my program output and the typical output given in the specification.	**A**	**B**	**C**	**D**	**E**	**F**

67. The hardest part of this program was figuring out if my code was actually doing what I intended it to be doing.	**A**	**B**	**C**	**D**	**E**	**F**
68. It took me a long time to fix one major logical error that I only got a very few times.	**A**	**B**	**C**	**D**	**E**	**F**
69. I spent more time fixing my program than writing the code.	**A**	**B**	**C**	**D**	**E**	**F**
70. I ended up repeating the same types of errors and mistakes in my program.	**A**	**B**	**C**	**D**	**E**	**F**
71. I think my debugging frustration came out of bad programming style.	**A**	**B**	**C**	**D**	**E**	**F**
72. I used pen and paper to draw the structure of my program/method/class to identify problems.	**A**	**B**	**C**	**D**	**E**	**F**
73. I used erroneous and extreme data range to check that my program works.	**A**	**B**	**C**	**D**	**E**	**F**
74. At first I did not get the correct output but with trial and error I got the correct output.	**A**	**B**	**C**	**D**	**E**	**F**
75. Taking a break when struggling, helped me to solve the problem(s) in my program.	**A**	**B**	**C**	**D**	**E**	**F**
76. I tested my completed program with test data given in the problem specification.	**A**	**B**	**C**	**D**	**E**	**F**
77. I invented my own test data to see if my program was working.	**A**	**B**	**C**	**D**	**E**	**F**
78. I started one part of the program, found error, and tested for correct output and then moved onto the next part of the program.	**A**	**B**	**C**	**D**	**E**	**F**
79. I was surprised when I got incorrect output.	**A**	**B**	**C**	**D**	**E**	**F**
80. I was surprised when I got correct output.	**A**	**B**	**C**	**D**	**E**	**F**
81. After I finished a piece of code or a program, I imagined a mental picture of how the data values were changing.	**A**	**B**	**C**	**D**	**E**	**F**
82. After I finished a piece of code or a program, for a given set of input I imagined a mental picture of the control flow of my program.	**A**	**B**	**C**	**D**	**E**	**F**