Software Debugging Patterns for Novice Programmers

Woei-Kae Chen, Chien-Hung Liu, and Hong-Ming Huang

Department of Computer Science and Information Engineering

National Taipei University of Technology, Taiwan

{wkchen, cliu}@ntut.edu.tw, gcaaa31928@gmail.com

Abstract. *Debugging, the process of resolving defects that prevent correct operation of computer software (or programs), is known as one of the most challenging tasks for software developers. This is especially true for novice programmers who are frequently at a loss when a program does not behave as expected. This paper summarizes several patterns that help novice programmers understand the fundamental principles, strategies, and methods that can be used to identify and resolve bugs. By using these patterns, novice programmers can speed up debugging process and fix bugs in a more systematical way.*

Categories and Subject Descriptors

•Software engineering → Software testing

•Software engineering → Debugging process

Keywords

Coding errors, Debugging, Bugs, Pattern language, Patterns

1. Introduction

Program debugging is a very important task in software development. It has been reported that a high proportion of development time (cost) is spent in debugging [1]. To novice programmers, debugging is particularly challenging since they often do not know what to do when a bug is reported (sometimes such a bug is found by themselves). Consequently, they often spend a lot of time doing try and error without making real progress (fixing code that does not need to be fixed), and are unable to resolve problems efficiently. In this paper, we consider novice programmers (or written simply as programmers hereafter) as ones who have only limited programming experience. For example, computer science freshmen, sophomores, or juniors. We summarize common problems and solutions frequently encountered during debugging process and present them as debugging patterns.

Novice programmers can often write (develop) code without problems. In case that the code is relatively small, debugging is usually not a problem. However, as the code gets larger and larger, a systematic way of diagnosing the code and finding the root cause of the problem (or bug) is needed. We will first report a common debugging process and then present the patterns that can be applied in this process. Our patterns begin with *Bug Reproduction*, which is also the initial point of most debugging tasks. What follows are *Narrowing Down a Block of Code for Tracing* and *Speedup Debugging Cycle*. The former narrows down the scope of concern so that the programmer can focus attention on a small block of code for debugging. The latter helps speed up debugging cycle, which is often overlooked by novice programmers. Both patterns can help programmers debug more efficiently.

Two common types of mistakes (bugs) that a programmer often makes are that: (a) the programmer has the right computational thinking (or algorithm), but the logic in the code is implemented incorrectly (i.e., what you code is not what you think); (b) the programmer has the wrong computational thinking, which in turn produces incorrect results (i.e., what you think is not what you need). For novice programmers, it is easy to get confused and mixed up with the two. We will report three patterns called *What You Code Is What You Think*, *Verifying Program Logic*, and *Verifying Computational Thinking* to help programmers clarify them. Note that, in this paper, we use the term *incorrect computational thinking* to denote a conceptual mistake, i.e., the case that the programmer is using a wrong idea or wrong computational procedure to implement the code, resulting to an error. On the other hand, we use the term *incorrect program implementation* to denote a coding mistake, i.e., the case that the implemented code does not conform to its computational thinking.

Novice programmers often try to fix a bug simply by reading the code over and over again, in the hope of finding some code statements that are wrong. When a suspicious statement is discovered, they try to fix it immediately without verifying whether it is indeed problematic. Sometimes, this simple strategy works. But, most of the time, the bug can persist after changing a lot of code. The *Step by Step Trace* and *Monitor Variables* patterns help address this issue. When applying these patterns, the programmer carefully examines the runtime behavior of the code for every statement so that he/she can pinpoint the incorrect statement with confidence. After code fixing, it is important to make sure that the target bug is indeed resolved and also confirm that the fix does not create any undesired side effects. The *Bug Fixing* and *No Regression Error* patterns discuss the solutions to these problems.

2. Debugging process

For novice programmers, following a defined process is a good way to understand the essential debugging activities, to learn the associated debugging skills, and to improve the efficiency of

finding and fixing program defects. Figure 1 shows a typical software debugging process and the reported debugging patterns. Particularly, the debugging process consists of five activities including reporting bug, bug reproduction, cause identification, bug fixing, and regression testing. Each of the activities are described as follows:

- Reporting bug

The first activity of the debugging process is reporting bug. This activity is performed by a user or programmer when he/she discovers an abnormal behavior or unexpected output of the program. The user or programmer then submits a bug report or feedback about the discovered anomaly. This activity can be a simple task by merely speaking to the developer about the program anomaly or it can be a non-trivial one in which the program anomaly is described as much detail as possible using a bug tracking system [2], such as Bugzilla [3] or Trac [4]. Note that an effective bug report can help programmers to quickly identify and fix the bug. In contrast, if the bug is not reported correctly, the developer may not be able to reproduce and fix the bug.

- Bug reproduction

The bug reproduction is to reproduce bug in the development environment so that the programmer can understand what was wrong about the program. This activity is critical since the bug report may not include enough information about the symptoms of the bug and it would be difficult, if not impossible, to fix a bug if the bug cannot be reproduced. Two patterns, *Bug Reproduction* and *Speedup Debugging Cycle*, are reported and can be applied in the bug reproduction activity.

- Cause identification

The cause identification is to identify the root cause of a bug. The root cause is the erroneous lines of code that cause errors which eventually lead to subsequent failure. This can be a very challenging and time consuming activity in program debugging since there are many different ways to write programs that cause errors. Although many methods, such as fault localization and bug prediction, have been developed to locate root causes from various failure symptoms [5], the patterns including *Narrowing Down a Block of Code for Tracing*, *What You Code Is What You Think*, *Verifying Program Logic*, *Verifying Computational Thinking*, *Step by Step Trace*, and *Monitor Variables*, are reported mainly for novice programmers and can be applied in the cause identification activity.

- Bug fixing

The bug fixing is to change program in order to remove the bug based on the identified root cause. Bug fixing may change the erroneous lines of code to fix the incorrect implementation or add extra code to handle the errors introduced by the root cause. For example, an exception handler can be added to deal with the error of "division by zero." The bug fixing can be an iterative process until the bugs are resolved and *Bug Fixing* pattern described in section 3.8 can be applied in this bug fixing activity.

- Regression testing

After fixing a bug, the developer needs to perform regression testing to verify whether or not all program features still perform correctly. That is to test the program for verifying if the changes of program do not introduce any new bugs. The *No Regression Errors* pattern is reported and can be used in the regression testing activity.

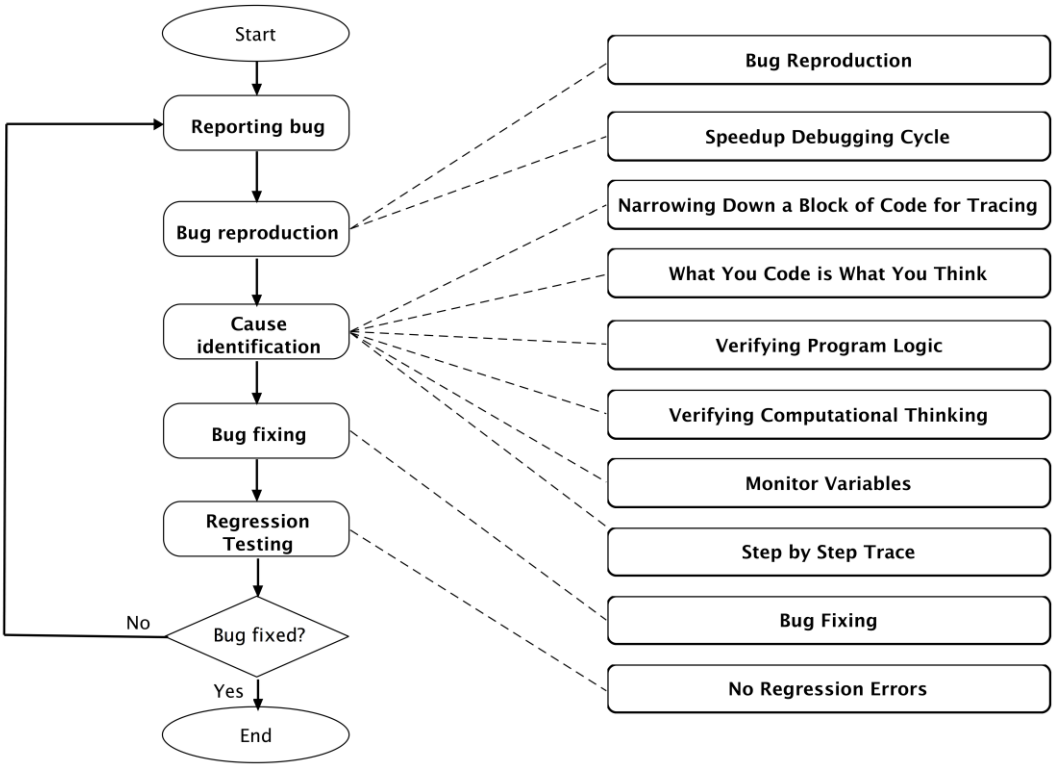

Figure 1. The process of software debugging

3. Patterns

The relationship of the patterns reported in this paper is shown in Figure 2. An arrow indicates that a pattern A uses another pattern B to complete a certain task. For example, Bug Fixing pattern uses Bug Reproduction pattern to confirm whether the target bug is fixed. In general, Bug Reproduction pattern is the starting point, which is followed by Speedup Debugging Cycle pattern and Narrowing Down a Block of Code for Tracing pattern. Once the problem has been narrowed down to a small block of code, a typically question to ask is whether the programmer is using the correct computational thinking to implement the code. What You Code is What You Think pattern addresses this question. The pattern directs the programmer to first confirm the correctness of his/her computational thinking by using Verifying Computational Thinking pattern. After verification, the question is reduced to a program implementation problem, which is addressed by Verifying Program Logic pattern.

When determining whether program implementation (or computational thinking) is correct, Step by Step Trace pattern can be used to help the programmer examine the runtime behavior of every code statement. The key is to inspect the inputs and outputs of every statement. In other words, in each program step, the value of each variable should be observed. This is addressed by the Monitor Variables pattern. It is not unusual that a novice programmer attempts to fix a bug, but ends up fixing something else, and/or creates a new bug after the fix. The last two patterns, Bug Fixing and No Regression Errors, address these issues.

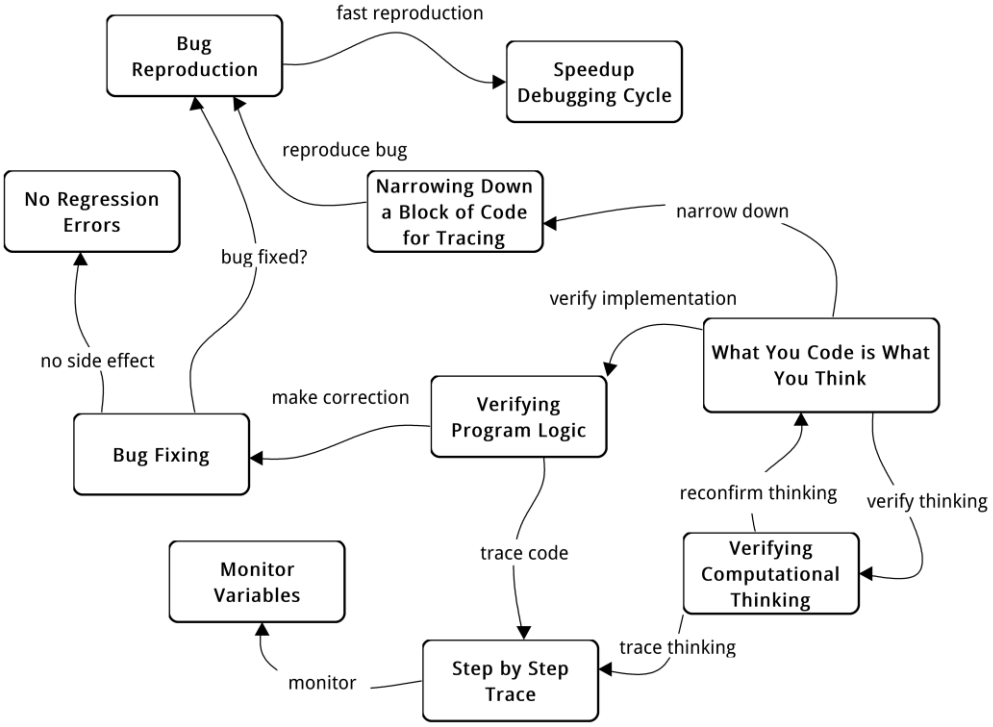

Figure 2. Pattern relationships

3.1 Bug Reproduction

Context: A bug that causes program failure has been discovered or reported and the programmer would like to reproduce the bug in order to understand and fix the bug.

Problem: How does the programmer reproduce the symptoms of a bug in the development environment so that the programmer can understand and fix the bug?

Forces:

- The programmer needs to reproduce the bug symptoms consistently by following a sequence of execution steps in order to verify if the reported bug indeed exists and use that information to analyze and identify the fault location. If the bug cannot be reproduced, the programmer may suspect the existence of the reported bug.
- The bug reporter may not be able to describe the bug symptoms clearly or the bug report may not contain detailed description about the symptoms of the bug.
- The programmer could fix the code incorrectly or waste time to fix the wrong thing due to misunderstanding of the bug.

Solution: The bug reporter (or programmer) needs to specify what it takes to reproduce the bug consistently, including detailed instructions or steps that cause the bug to happen, execution environment, triggering conditions, and input data. The relevant screen snapshots, any error message or log file, actual outputs, and expected results also need to be recorded or specified. Thus, by following the same execution steps and environment, the programmer can recreate the conditions to reproduce the bug symptoms again consistently.

For example, assume that an ATM program allows users to make a deposit. The program shall verify the deposit amount and make sure that the amount is positive and does not exceed the highest limit of integer that can be represented in the program. If not, the program shall remind the users and the deposit transaction won't be executed. If the program has a bug that allows users to deposit a negative or zero amount, the detailed deposit scenarios and corresponding conditions and data, including the account number, deposit amount, deposit date and time, the account balance before and after the deposit, the screen snapshots, and

expected results shall be specified. With such information, it would be much easier for the programmer to quickly recreate the deposit problem of the program so as to diagnose and fix the bug.
Note that bug reproduction can be used not only to recreate and understand the problem of a program, but also to demonstrate and verify if the bug is properly fixed (see details in section 3.8). Many bug tracking systems require users to provide detail information for reproducing the bug. If, however, the bug is hard to reproduce or irreproducible, such as multithreading bugs and race conditions, additional techniques or tools may be required [6].

Known Use: Heisenbug [7]; Enhancing Android application bug reporting [8].

Related Patterns:

- Narrowing Down a Block of Code for Tracing: Narrowing Down a Block of Code for Tracing pattern can use Bug Reproduction pattern to recreate the bug during the process of narrowing down the scope for code tracing.
- Speedup Debugging Cycle: Speedup Debugging Cycle pattern can use the execution steps, conditions, input data, and expected outputs of Bug Reproduction pattern to reproduce the bug quickly.
- Bug Fixing: The execution steps, conditions, input data, and expected outputs of Bug Reproduction pattern can be used by Bug Fixing pattern to verify if the bug has been removed successfully.

3.2 Narrowing Down a Block of Code for Tracing

Context: The programmer is uncertain which module/class/method (i.e., code block) of the program could be the location of the bug.

Problem: How does the programmer find the locations of error code efficiently and effectively, especially for a large program?

Forces:

- The programmer is usually unable to determine the precise location of the fault (i.e., error code) directly based on the bug symptoms. A systematic method can help programmers to identify the error code more efficiently and effectively, especially for a large program.
- For a large program, it is very inefficiently to find the location of error code either by reading or by tracing the entire program step by step.
- The program statements related to the bug symptoms or error outputs may not be the code that corresponds to the fault.
- A program is usually designed and structured hierarchically in terms of module, class, and method. Such hierarchical structure can be useful for narrowing down and identifying the fault location.
- A fault or bug symptom may correspond to a block of code (i.e., error code). The debugging process can be largely shortened if such a code block can be identified efficiently so that the scope of code tracing can be narrowed down to a small area.

Solution: Given a large program with hundreds even thousands lines of code, it takes a lot of time to pinpoint the code statement that is wrong. A common strategy is to first narrow down the scope of concern to a small block of code and then focuses attention on the code statements inside the block for debugging. The typical methods of narrowing down the scope for code tracing include (1) top-down; (2) bottom-up; and (3) divide and conquer. Assume that, as shown in Figure 3, a program takes inputs, processes the inputs through a sequence of modules (or method calls) according to its execution flow, and then generates outputs.

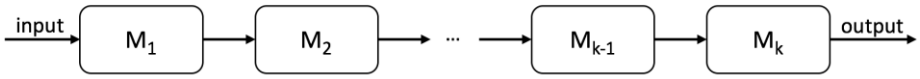

Figure 3. The execution flow of a program

The top-down strategy is to narrow down the tracing scope starting from the first module or method (i.e., M_1) of the program execution flow. The programmer provides valid inputs to the program and check if the output of M_1 is the same as with his/her expectation. If not, M_1 is the candidate block for code tracing. Otherwise, the programmer continues and checks the second module (i.e., M_2) of the execution flow to see if its output follows expectation. If not, M_2 is the candidate block. Otherwise, the programmer continues the process until the last second module (i.e., M_{k-1}) is checked. The programmer then sees if the output of M_{k-1} is consistent with his/her expectation. If not, M_{k-1} is the candidate block; otherwise, M_k is the candidate block.

On the contrary, the bottom-up strategy is to narrow down the tracing scope starting from the last module or method (i.e., M_k) of the execution flow. The programmer provides valid inputs to the program and check if the output of M_{k-1} aligns with his/her expectation. If yes, M_k is the candidate block for code tracing. Otherwise, the programmer continues and checks the output of M_{k-2} to see if it is consistent with his/her expectation. If yes, M_{k-1} is the candidate block for code tracing. Otherwise, the programmer continues the process until the output of the first module (i.e., M_1) is checked. The programmer then sees whether the output of M_1 follows expectation. If yes, M_2 is the candidate block; otherwise, M_1 is the candidate block.

The divide and conquer strategy breaks the narrowing down problem into two subproblems and solves the subproblems recursively. Based on this idea, the programmer firstly provides valid inputs to the program and checks the output of the middle module or method (denoted as M_c) of the execution flow. If the output of m_c is the same as his/her expectation, then the candidate block is in the lower-half modules of the execution flow (i.e., M_c to M_k); otherwise, the candidate block is in the-upper half modules of the execution flow (i.e., M_1 to M_c). The programmer then follows the same idea to narrow down the code block using the lower-half or upper-half modules recursively and eventually reaches the candidate block where no more division is possible.

Known Use: Solving Code-tracing Problems and its Effect on Code-writing Skills Pertaining to Program Semantics [9].

Related Patterns:

- Bug Reproduction: Bug Reproduction pattern can be used by Narrowing Down a Block of Code for Tracing pattern to reproduce the bug during the process of narrowing down the scope for code tracing.
- What You Code Is What You Think: Narrowing Down a Block of Code for Tracing pattern can be used by What You Code Is What You Think pattern to quickly identify the candidate block for code diagnosis.

3.3 What You Code Is What You Think

Context: The programmer has identified the block of code that contains errors and understood its relationship with other modules in the hierarchical structure of the program. The programmer would like to determine the root cause of the bug within the block.

Problem: How does the programmer determine whether the bug is caused by incorrect computational thinking (i.e., incorrect problem formation and solution expression) or by incorrect program implementation?

Forces:

- A software bug can be introduced either from incorrect computational thinking (i.e., misunderstanding of the problem or solution) or from incorrect program implementation. Fixing a bug caused by incorrect computational thinking is different from fixing a bug caused by incorrect program implementation.
- A programmer cannot fix a bug and can even create more bugs, if the programmer does not understand the root cause of the bug correctly.

Solution: The programmer can apply Verifying Computational Thinking pattern to the candidate code block to verify whether the root cause is indeed due to incorrect computational thinking. If there is nothing wrong with computational thinking, the programmer can then apply Verifying Program Logic pattern to the candidate code block to verify whether the program implementation is correct. Note that novice programmers often cannot make sure whether there is a mistake in computational thinking or an error in program implementation (or both). Thus, it would be necessary to recheck both of them again and again until the root cause is identified. In general, it is advised that the programmer should first verify whether their computational thinking is valid and correct mistakes if any. The programmer then verifies and fixes program implementation.

Known Use: Strategies that students use to trace code: an analysis based in grounded theory [10].

Related Patterns:

- Narrowing Down a Block of Code for Tracing: What You Code Is What You Think pattern can use Narrowing Down a Block of Code for Tracing pattern to identify the code block for problem diagnosis.
- Verifying Computational Thinking: What You Code Is What You Think pattern can use Verifying Computational Thinking pattern to verify whether the computational thinking is incorrect or not.
- Verifying Program Logic: What You Code Is What You Think pattern can use Verifying Program Logic pattern to verify whether the implementation logic is incorrect or not.

3.4 Verifying Computational Thinking

Context: The programmer would like to verify whether there is a problem with his/her computational thinking.

Problem: How does the programmer verify that the bug under investigation is the result of incorrect computational thinking?

Forces:

- The implementation of code can be correct only if the computational thinking is correct.
- If there is anything wrong with the computational thinking, the problem formation and solution design shall be revised so that the code can be properly fixed accordingly.

Solution: The computational thinking of the programmer can be transformed into programmatic thinking which can be algorithms or a sequence of execution steps to carry out the solution of the problem that the program attempts to solve. The programmer can

apply Step by Step Trace pattern to trace the code and verify if implementation logic is consistent with his/her programmatic thinking. If they are consistent, this indicates that the computational thinking of the programmer is incorrect since the code has been known to be erroneous.

If the computational thinking is incorrect, the programmer needs to rethink the problem and redesign the solution and corresponding algorithms. Based on the redesigned algorithms, the programmer refactors the program implementation and then applies What You Code Is What You Think pattern again to verify whether the new implementation logic is consistent with the new computational thinking. Note that verifying computational thinking is a repetitive process. The programmer may need to try many times to come up with the correct computational thinking.

Known Use: From computational thinking to coding and back [11].

Related Patterns:

- What You Code Is What You Think: Verifying Computational Thinking pattern can be used by What You Code Is What You Think pattern to verify if the new implementation logic is consistent with new computational thinking.
- Step by Step Trace: Verifying Computational Thinking pattern can use Step by Step Trace pattern to help the programmer find the code statement that does not follow its intended computational thinking.

3.5 Verifying Program Logic

Context: The programmer has already verified that his/her computational thinking is correct and would like to verify whether there is a problem in program implementation.

Problem: How does the programmer verify that the bug under investigation is the result of an incorrect program implementation?

Forces:

- The implementation of code is correct only if it precisely follows the verified computational thinking.
- The programmer needs to make sure that there is indeed a program implementation error so that the code can be properly fixed.
- If the programmer fixes any suspicious code without evidence of whether the code is indeed wrong, chances are that the fix does not improve the code in any way and can make the code even worse.

Solution: Two methods can be used. The first is to perform a code review for the block of code under investigation. The objective here is to verify whether the program implementation precisely follows the programmer's computational thinking. Note that the review (reading the source code) is confined within a small block of code and the purpose is not to find any suspicious code statement, but to verify whether the program implementation follows its intended computational thinking. Therefore, this code review is a step by step (or statement by statement) review and is not a random reading of the code. Once a code statement that does not follow its intended computational thinking is found, the programmer can use Bug Fixing pattern to perform bug fix and verify if the fix resolves the problem. In case that the fix does not help, the programmer needs to redo the Verifying Program Logic pattern.

Oftentimes, by simply reviewing the code, the programmer cannot easily find any code statement that does not follow its intended computational thinking. In this case, the runtime behavior of each code statement should also be investigated. Therefore, the second method is to use Step by Step Trace pattern to trace and look into the input and output of each statement. Such a step by step trace is very useful in identifying whether each code statement performs exactly as the programmer expects and can help find the problematic code statement. Once such a code statement is found, again, the programmer can use Bug Fixing pattern to perform bug fix and verify if the fix resolves the problem.

Known Use: Code Verification Techniques in Software Engineering [12].

Related Patterns:

- Step by Step Trace: Verifying Program Logic pattern uses Step by Step Trace pattern to help the programmer find the code statement that does not follow its intended computational thinking.
- Bug Fixing: Once a code statement is identified as incorrect, the programmer can use Bug Fixing pattern to resolve the bug.

3.6 Step by Step Trace

Context: The programmer has narrowed down a small block of code for debugging. The inputs to the block have been verified to be correct. However, the outputs are not as expected. The programmer would like to find the first incorrect code statement.

Problem: Given a block of code and its expected behavior, how does the programmer find the first incorrect code statement?

Forces:

- Many mistakes in computational thinking and/or errors in program implementation cannot be easily identified by simply reading the code. The runtime behavior of each statement including both its inputs and outputs offers an important clue to finding the mistake.
- The first incorrect statement (called s_f) produces incorrect outputs which become the inputs of the statements following s_f. Without correct inputs, every statement after s_f could also produce incorrect outputs. These statements are not necessarily faulty, because they are not given the correct inputs. In other words, s_f is where the bug originated. In general, the root cause of the bug can usually be identified by examining the inputs and outputs of s_f.

Solution: When the programmer has narrowed down a small block of code for debugging, it becomes possible that the programmer can trace the execution of every statement step by step. The key is to observe the inputs and outputs of each statement and find the first statement that produces incorrect outputs while given the correct inputs. When such a statement is found, the programmer can analyze the statement based on its inputs and outputs, and determine the root cause of the bug. To monitor the inputs and outputs of each statement, Monitor Variables pattern can be used. Note that Step by Step Trace can be used to find a bug that is the result of either incorrect computational thinking or incorrect program implementation. We use two examples to explain how this is done.

The first example is shown in Figure 4. The method deposit(int cash) adds a certain amount of cash to a bank account. To avoid depositing a negative amount of cash, the code is implemented to remind the user that the cash value should not be negative. Suppose the programmer knows that there is a bug when deposit(int cash) is called with the original

balance being 1000 and the parameter cash being -100. In this case, the resulting balance should not be changed because a negative deposit is not allowed. Or more precisely, the value of balance should remain to be 1000 after -100 is deposited. What the programmer can do is to follow the bug reproduction process (see Bug Reproduction pattern) to execute the program and provide appropriate inputs (i.e., run the program and try to deposit -100 into the account with a balance 1000). After the program executes line 1, the programmer immediately verifies that the cash parameter is indeed -100. Then, the programmer traces lines 2-5 step by step. First, line 2 is executed. Since cash is -100, the if condition should be true. Thus, the programmer should verify that the program enters line 3. Then, line 3 is executed. The programmer should verify that the console displays "Cash should not be negative." Then, line 4 is executed and then line 5 is executed. After the execution of line 5, the value of balance is reduced to 900. At this point the programmer discovers that balance is changed and this change is not as expected. This is the first statement that does not perform as expected and can lead to the root cause of the bug. After a bit of thinking, the programmer should be able to deduct that line 5 should not be executed when cash is -100, because a negative deposit is not allowed. Or, there is a program implementation error in line 5 and the correct program should be designed to add balance value only when cash is positive. Once the problem has been identified, the programmer can use Bug Fixing pattern to get the bug fixed. For example, a possible fix is shown in Figure 5.

```
1  void deposit(int cash){
2      if (cash < 0){
3          printf("Cash should not be negative.");
4      }
5      balance += cash;
6  }
```

Figure 4. Example code

```
1  void deposit(int cash){
2      if (cash < 0){
3          printf("Cash should not be negative.");
4      }
5      else
6          balance += cash;
7  }
```

Figure 5. Example code after bug fixing

The second example is shown in Figure 6. This example illustrates a bug resulting from an incorrect computational thinking. The method calculateBMI() calculates and reports the BMI (Body Mass Index) of the user. BMI is defined as the body mass divided by the square of the body height (note: a normal BMI value is between 18.5~25). Suppose the programmer knows that, when the weight is 150 pounds and height is 70 inches, the displayed BMI is 0.03, an incorrect value. As described in the previous example, the programmer can trace the code step by step. After executing line 3, the value of weight (150) is verified to be correct. After the execution of line 5, the value of height (70) is also verified to be correct. Then, line 6 is executed and bmi variable becomes 0.03. This is the first incorrect code statement, because the correct bmi value should be 21.5. At this point, the programmer looks at line 6 and finds that the equation of calculating BMI does not look wrong. However, the resulting bmi value is far off the mark. What is the root cause of the problem? After a bit of thinking, it becomes

clear that the BMI equation should be used with metric system (i.e., using meters and kilograms as inputs). It is the programmer who forgets to convert pounds into kilograms and inches into meters when designing the computational procedure (or having incorrect computational thinking). Once the problem has been identified, the bug can be resolved by using Bug Fixing pattern with extra code that performs weight and height conversions.

```
1  void calculateBMI(){
2      printf("Please input your weight in pound: ");
3      scanf("%f", &weight);
4      printf("Please input your height in inch: ");
5      scanf("%f", &height);
6      double bmi =  weight / (height * height);
7      print("Your BMI is %f", bmi);
8  }
```

Figure 6. Calculate BMI example code

Known Use: Rubber duck debugging [13]; A study of the influence of code-tracing problems on code-writing skills [14].

Related Patterns:

- Monitor Variables: Step by Step Trace pattern uses Monitor Variables pattern to observe the inputs and outputs of each statement.

3.7 Monitor Variables

Context: Given a particular code statement, the programmer would like to verify if the inputs of the code statement are correct and the outputs produced by the code statement are also correct.

Problem: How to examine the inputs and outputs of a code statement so as to determine whether the statement is correct?

Forces:

- Oftentimes, determining whether a code statement is correct by reading the statement is inefficient. A code statement while looks right can perform incorrectly at runtime. Observing the input and output variables can help identify whether the code statement is indeed correct.

Solution: A code statement uses some variables (inputs) to perform computations and may change one or more variables (outputs). A code statement is correct only if given the correct inputs it produces correct outputs. Therefore, it is important to monitor these input/output variables at run time. Many IDEs offer integrated debuggers that help setup break points and view variable values. Using such a debugger, the programmer can request the program to pause at a specified code statement, examines the input variables of the statement, and then executes the statement and examines the output variables. Thus, whether the code statement is indeed correct can be identified. Furthermore, code statements can also be executed line by line, making variable observations very efficient. In case that a debugger is not available or is inconvenient to use under certain cases, the programmer can add some code statements into the source code to print the variables under observation (normally in the console). The side effect of using console outputs is that it can slow down program execution significantly and the extra code must be removed sometime later. However, using console outputs can sometimes be more convenient than using debuggers. For example, if the output variables of a statement inside a loop are to be observed, using a debugger to look at the output values for every loop iteration can be very tedious. In this case, using console outputs may be more

efficient. In addition to console outputs, system logs, which store important system events and outputs, are also frequently used as a way of monitoring variable values.

Known Use: Debugger [15], Eisenstadt [16].

Related Patterns:

- Step by Step Trace: Monitor Variables pattern supports Step by Step Trace pattern for the execution and observation of each code statement.

3.8 Bug Fixing

Context: The incorrect code statement and the root cause of the bug have been identified. The programmer would like to fix the bug.

Problem: After fixing some code, how does the programmer make sure that the bug is gone?

Forces:

- Sometime, a programmer intends to fix bug. But, after some code fixing, the bug remains in the code.

Solution: A common mistake that novice programmers often make is that, after bug fixing, the bug is not completely gone. The programmer may erroneously believe that the bug has been resolved, since he/she has fixed some code statements. However, the fix does not necessarily do the job. Possible reasons are that the fix does not produce the desired effect and/or the bug is only partially fixed. The solution to this problem is to include an extra bug reproduction step both right before and right after bug fixing. That is, to make sure that a particular bug is gone, a three-step process is required: (a) use a bug reproduction procedure to reveal the bug; (b) fix code; and (c) use the same bug reproduction procedure to demonstrate that the bug is gone. The bug reproduction step right before bug fixing is used to show that the bug is present and the step after is used to demonstrate that the bug is gone. In case that the bug is not gone, the programmer may need to use Step by Step Trace pattern again and if necessary redo either Verifying Computational Thinking or Verifying Program Logic. Note that it is important to do bug reproduction *right before and after* bug fixing. This is because while fixing the target bug, a novice programmer sometimes gets distracted by some other code statements and ends up fixing some other problems that are not related to the target bug. The three-step process stresses that such distraction is not a good practice and should be avoided. That is, the programmer should focus on one bug at a time.

Known Use: The design of bug fixes [17].

Related Patterns:

- Step by Step Trace: After bug fixing, in case that the bug is not gone, Step by Step Trace pattern can be used to verify whether there are some other program implementation and/or computational thinking problems.
- No Regression Errors: When the target bug is properly fixed, the programmer may use No Regression Errors pattern to make sure that the fix does not produce any undesired side effects.
- Bug Reproduction: Before bug fixing, the programmer should use Bug Reproduction to make sure that the target bug is indeed present. After bug fixing, the programmer can use Bug Reproduction pattern to verify whether the bug is gone.

3.9 No Regression Errors

Context: The target bug has been fixed. The programmer would like to make sure the fix does produce any undesired side effects.

Problem: After bug fixing, how does the programmer ensure that the fix does not produce any undesired side effects?

Forces:

- Fixing a bug might create many more new ones.

Solution: A novice programmer often gets happy once a bug is fixed. However, in practice, whenever a code statement is changed, in addition to the target bug that the change intents to address, the change may in fact has some undesired side effects. For example, to fix the BMI bug discussed in Section 3.6, the weight and height variables are converted to using units of kilograms and meters before calculating BMI. Suppose there are some other methods (functions) that also use weight and height, but in units of pounds and inches, the bug fix can break these methods, creating some other new bugs. To avoid undesired side effects, it is important to perform regression testing right after a bug is fixed. In case that a regression error occurs, the bug fix is considered invalid and the programmer should reevaluate how the bug is fixed and find a new fix that does not produce side effects. If there is a regression error and the programmer decides to keep the fix despite of the side effect, the new bug should be reported, tracked, and fixed sometime later.

Known Use: Regression test selection by exclusion using decomposition slicing [18]; Regression test selection on system requirements [19]; Debugger [15].

Related Patterns:

- **B**ug Fixing: Bug Fixing pattern uses No Regression pattern to ensure that the fix does not produce any undesirable side effects**.**

3.10 Speedup Debugging Cycle

Context: The programmer would like to fix a bug that can be properly reproduced.

Problem: While addressing a particular bug, how to speed up the debugging cycle so as to make debugging more efficient?

Forces:

- A time-consuming bug reproduction lowers debugging efficiency.

Solution: To reproduce a bug, not only the code needs to be executed, the programmer often also needs to manipulate user interface (sometimes graphical user interface) to set proper inputs and sometimes needs to walk through a lengthy sequence of user interface operations. Such a bug reproduction process can be slow and lowers the overall debugging efficiency. The situation gets even worse when many iterations of bug reproduction are required. Novice programmers do not aware that it often takes several iterations of bug reproduction, verifying computational thinking, verifying program logic, and step by step trace just to find the first incorrect code statement. And more iterations could be required to get the bug fixed. Consequently, a lot of time is wasted in bug reproduction rather than in the most important task, bug identification. The solution to this problem is to invest some efforts in bug reproduction so that the reproduction can be made as fast as possible. Several strategies can be used: (a) modify source code to jump to the target directly, e.g., if there are several pages of user interface to manipulate, try to skip directly to the target page; (b) reduce data size, e.g., if there is a long input data file that causes a bug, try to shorten the data file as much as

possible; and (c) unit testing, e.g., use unit test code to directly execute the method in question.

Known Use: Towards detecting and solving aspect conflicts and interferences using unit tests [20].

Related Patterns:

- Bug Reproduction: Speedup Debugging Cycle tries to shorten the time needed in bug reproduction.

4. Related Work

Software debugging is critical for finding and fixing the bugs that cause a program to behave in an unexpected way. There were numerous researches addressing the approaches, techniques, strategies, tools, and education for software debugging [25][26][27]. However, there were very few studies on the pattern of software debugging, especially for novice programmers. This section briefly reviews several research studies related to our work.

Amoui et al. [21] present a pattern language for software debugging. The proposed patterns can be grouped into three categories including *software*, *program*, and *code* that aim to connect three concepts: testing, error, and debugging. The software category includes two general patterns addressing the behavioral and structural errors and corresponding debugging. The program (black-box) category contains two patterns describing functional and non-functional specification testing problems and solutions. The code (white-box) category consists of three patterns that cover the errors and corresponding testing related to the compilation, quality assurance, and coverage. Unlike [21], which addresses the patterns from the error and testing perspectives, this paper focuses on the patterns related to debugging process, strategies, and skills that can be applied for novice programmers.

Farchi et al. [22] have described and categorized a taxonomy of concurrent bug patterns. In particular, they classify the recurring bugs for concurrent programs into a set of "bug patterns." In addition, they describe timing heuristics to help programmers to uncover the concurrent bugs and present several concurrent design patterns that can be used to deduce typical concurrent bug patterns.

Ahmadzadeh et al. [23] have studied the pattern of debugging activity for computer science students for improving the teaching of programming. They collect and analyze the compilation messages, time logs, and source code from the programming exercises of students. The results indicate that there exists a pattern between producing compilation errors and misunderstanding a concept. However, there is no correlation between the number of errors and the time spending on debugging. Further, they have conducted an online programming exam to test the debugging ability of students. The results indicate that the majority of good debuggers are good programmers. However, only 39% of the good programmers are good debuggers. They suggest that the ability to read and understand program code is important for improving the debugging competence of novice programmers.

Agans [24] has proposed nine debugging rules that can be used to find the software and hardware problems for debugging. These rules can help users to find the causes of bugs and fix them more efficiently and effectively. Basically, these rules emphesize on understanding the system, recreating the system failure, seeing and analyzing the details of failure, applying the divide and conquer technique to narrow down the search of the problem, changing and testing the system incrementally, keeping an audit trial, checking the assumption of the problems, asking for fresh insights about the problems, and checking whether the problems are really fixed. Some of the rules are consistent with the patterns reported in this paper.

5. Conclusions

This paper reports ten patterns that address the problems a novice programmer frequently encounters. Teaching debugging is probably as important as teaching programming. However, from the authors' experiences, it seems that most programming courses do not cover debugging process and simply assume that students will learn debugging automatically by themselves. We hope that this paper offers novice programmers a guide of learning systematic debugging. The patterns covered in this paper are mostly problems related to debugging process. In the future, we plan to explore and collect patterns that are related to techniques of debugging and/or avoidance of bugs.

Acknowledgements

This research is supported in part by Ministry of Science and Technology under the grant 105-2221-E-027-086 in Taiwan.

References

[1] Tom Britton, Lisa Jeng, Graham Carver, Paul Cheak, and T. Katzenellenbogen. Reversible Debugging Software, 18 Jan, 2017; http://citeseerx.ist.psu.edu/viewdoc/download?doi=10.1.1.444.9094&rep=rep1&type=pdf.

[2] Bug tracking system, 10 November 2016 11:17 UTC; https://en.wikipedia.org/w/index.php?title=Bug_tracking_system&oldid=748787755.

[3] Bugzilla, 3 January 2017 21:04 UTC; https://en.wikipedia.org/w/index.php?title=Bugzilla&oldid=758163489.

[4] Trac, 27 December 2016 07:43 UTC; https://en.wikipedia.org/w/index.php?title=Trac&oldid=756862335.

[5] W. E. Wong, R. Gao, Y. Li, R. Abreu, and F. Wotawa, "A Survey on Software Fault Localization," *IEEE Trans. Softw. Eng.*, vol. 42, no. 8, pp. 707-740, 2016.

[6] P. Fonseca, C. Li, and R. Rodrigues, "Finding complex concurrency bugs in large multi-threaded applications," in *Proceedings of the Sixth Conference on Computer Systems*, Salzburg, Austria, 2011, pp. 215-228.

[7] Heisenbug, 12 October 2016 03:01 UTC; https://en.wikipedia.org/w/index.php?title=Heisenbug&oldid=743934692.

[8] K. Moran, "Enhancing Android application bug reporting," in *Proceedings of the 2015 10th Joint Meeting on Foundations of Software Engineering*, Bergamo, Italy, 2015, pp. 1045-1047.

[9] A. N. Kumar, "Solving Code-tracing Problems and its Effect on Code-writing Skills Pertaining to Program Semantics," in *Proceedings of the 2015 ACM Conference on Innovation and Technology in Computer Science Education*, Vilnius, Lithuania, 2015, pp. 314-319.

[10] S. Fitzgerald, B. Simon, and L. Thomas, "Strategies that students use to trace code: an analysis based in grounded theory," in *Proceedings of the First International Workshop on Computing Education Research*, Seattle, WA, USA, 2005, pp. 69-80.

[11] K. DePryck, “From computational thinking to coding and back,” in *Proceedings of the Fourth International Conference on Technological Ecosystems for Enhancing Multiculturality,* Salamanca, Spain, 2016, pp. 27-29.

[12] D. THAKUR, “Code Verification Techniques in Software Engineering.”

[13] Rubber duck debugging, 29 November 2016 23:29 UTC; https://en.wikipedia.org/w/index.php?title=Rubber_duck_debugging&oldid=752186741.

[14] A. N. Kumar, “A study of the influence of code-tracing problems on code-writing skills,” in *Proceedings of the 18th ACM conference on Innovation and Technology in Computer Science Education*, Canterbury, England, UK, 2013, pp. 183-188.

[15] Debugger, 29 August 2016 07:18 UTC; https://en.wikipedia.org/w/index.php?title=Debugger&oldid=736696863.

[16] M. Eisenstadt, “My hairiest bug war stories.” *Communications of the ACM,* vol. 40, no. 4, pp. 30-37, 04/01/1997, 1997.

[17] E. Murphy-Hill, T. Zimmermann, C. Bird, and N. Nagappan, “The design of bug fixes,” in *Proceedings of the 2013 International Conference on Software Engineering*, San Francisco, CA, USA, 2013, pp. 332-341.

[18] A. Ngah, and K. Gallagher, “Regression test selection by exclusion using decomposition slicing,” in *Proceedings of the doctoral symposium for ESEC/FSE on Doctoral symposium,* Amsterdam, The Netherlands, 2009, pp. 23-24.

[19] P. K. Chittimalli, and M. J. Harrold, “Regression test selection on system requirements,” in *Proceedings of the 1st India Software Engineering Conference*, Hyderabad, India, 2008, pp. 87-96.

[20] Andr, Restivo, and A. Aguiar, “Towards detecting and solving aspect conflicts and interferences using unit tests,” in *Proceedings of the 5th workshop on Software Engineering Properties of Languages and Aspect Technologies*, Vancouver, British Columbia, Canada, 2007, pp. 7.

[21] Amoui, Mehdi, Mohammad Zarafshan, and Caro Lucas. “A Pattern Language for Software Debugging.” *International Journal of Computer Science* 1 (2006): 3.

[22] E. Farchi, Y. Nir, and S. Ur, “Concurrent bug patterns and how to test them,” in *Proceedings of the 2003 International Parallel and Distributed Processing Symposium(IPDPS)*, vol. 00.

[23] M. Ahmadzadeh, D. Elliman, and C. Higgins, “An analysis of patterns of debugging among novice Computer Science students,” *Proceedings of the 10th annual Special Interest Group on Computer Science Education(SIGCSE) conference on Innovation and Technology in Computer Science Education (ITiCSE '05)*, pp. 84-88, 2005.

[24] David J. Agans, Debugging: the 9 indispensable rules for finding even the most elusive software and hardware problems, AMACOM, 2006.

[25] S. Fitzgerald, R. McCauley, B. Hanks, L. Murphy, B. Simon, and C. Zander, “Debugging from the student perspective,” *IEEE Transactions on Education*, vol. 53, no. 3, pp. 390-396, Aug. 2010.

[26] Dan Hao and Hong Mei, “Recent progress in software testing, debugging and analysis: a survey,” *International Journal of Software and Informatics*, vol. 8, no. 1, 2014, pp. 1-17.

[27] Renee McCauleya, Sue Fitzgerald, Gary Lewandowski, Laurie Murphy, Beth Simon, Lynda Thomas and Carol Zander, "Debugging: a review of the literature from an educational perspective," *Computer Science Education*, vol. 18, no. 2, 2008, pp. 67-92.