**Lab: Tracing the Code**

**Description**:

In this lab, a program written by others called UML editor is provided to you. The specs of this program will be given to you as well. However, the code has 3 bugs. The original author of the code was not in your company anymore. He has a new job in Silicon Valley and cannot help you. So, your work is to figure out what’s wrong in the program. Please trace the code and then fix the bugs as soon as possible.

**The Grading:**

1. There are three bugs in total. Once you fix one bug, you got 50 points. If you fix the second bug, you get 80 points. If you resolve 3 bugs, you get 100 points.
2. Once you locate the code of bug, typically they are very easy to fix. Don’t worry.

Trace and understand unfamiliar code is a daunting task to most people. It can be worse if the programming language is not new to you or the programming model is new to you. To make it worse, the size of the code can be large. An ordinary programmer often thinks that he must totally understand the code before fixing the bug. However, after you totally understand the code, it could be one month later. Your boss hope that you can fix these bugs as soon as possible.

So, what is the best and efficient way for program understanding?

In this handout, I describe the code tracing tricks and skills of a talented programmer. It is suggested that you follow, understand, and use these tricks to complete this lab. However, you are always welcome to use your own methods if you think you are a talented programmer as well.

1. Please read the spec document carefully first. In principle, a spec documentation describes the correct behaviors of a system. A bugs can cause abnormal system behaviors, which violate the spec.
2. Please compile the program and make them executable.
3. Play with the program as much as possible. Then compare the behaviors one by one with the use cases in spec document.
4. You should begin to find some weird system behaviors, which should be bugs. Please write down the user event sequence which can reproduce the bug. Finding a reproducible test run is the first critical step in code tracing. In this lab, the test run undoubtedly should be the one that helps you to fix the bug as soon as possible.
5. Open the source code and browse the files. If file names, method names, and class names are very clear and self-explanatory, you can have a precise guess on the functionality in these code. The source code of this lab should be of high readability. In a high readable code, the names of class, variables, and methods should have strong positive correlation to the terms in spec document. Code naming with high positive correlation implies high readability. If the correlation is not high, congratulation !! the difficulties of code reading should be increased dramatically. Because it makes code readers to guess.
6. To an ordinary programmer, the familiarity of programming language can impact the code reading but to talented programmer, this is not necessarily true. In fact if you never write a GUI program, the impact can be much more significant. So, if you never write a GUI program, I suggest you to approach TA or your classmates to understand the GUI programming model first.
7. Talented programmers use debugger to trace the code. So, please compile the program in debug mode first. If you never use a debugger before (not a good sign though), please consult the slides and video clips from Prof. Cheng.
8. Suppose you already find a test run which is A->B->C->D->E which causes some weird behaviors which violate the specs. In this sequence, ABCDE could be any input events such a mouse click, keyboard inputs, or a click to a button, etc.
9. Suppose your first input event A is a mouse click, you can adopt one of the following methods to make your program stop after event A is triggered.
   1. Set break points at all the entry of methods in the code. Then, run the program and trigger event A. In principle, your program should stop at one method after input event A is triggered. Note that this method looks stupid once you have hundreds and thousands of methods to set break points. If you do have hundreds and thousands of methods to set break points, what can you do? It is always possible to do so, such as, Such as  
      <http://stackoverflow.com/questions/11625732/can-i-set-breakpoints-to-all-methods-in-a-class-at-once-in-visual-studio>  
      However, in this lab, this stupid method actually work very well because this is not a large program. By doing so, you can easily intercept the entry point after A.
   2. Suppose you are familiar with GUI programming model and you remember something when you browse the code. You should know when a mouse click event is triggered, a method called “\*mouseClick\*” should be called. So, use the string search provided by IDE you can easily locate these methods and then set the break points. Then run your program and see if your break points do stop the program when mouse click is triggered.
   3. If your code is not GUI, you can stop the main() at the beginning and then step-by-step use debugger step/next command to reach event A. If your program is not large, sometimes, you can immediately know how to go from main() to reach event A and then you will know where to set the break points. That is, using debugger to trace the program from the starting point, sometimes, can be a useful task to do, albeit not very efficient. It can be very inefficient when the size of code is large. However, this straightforward way can be useful when your goal is to understand the whole program not to fix a bug in a short time.
   4. You are always welcome to add new method if you know one.
10. From the above description, you should now know using a debugger to help you trace and locate the entry point of a test run in an unfamiliar system.
11. You next step is to understand what is going on between A->B. The best for program understanding is to **step/next** your program step by step. Before and after the **step/next**, you can observe the change of critical variables. So, you should add the critical variables in watch window for understanding.
12. A debugger provide a window called call stack window. By call stack window, the debugger tells you how your program goes from main() to your break point. So, if you want to understand the caller of A’method, please use call stack to go up one level. Debuggers should switch the source code to caller’s source code.
13. Next, you can patiently step/next until you reach the entry point of event B. However, the size of code can be large in practice. So, if you are inpatient, please set new break points again as above, continue/resume the program to stop the program after input event B is triggered. Again, you can use call stack window to understand how your program goes through which methods from A to B.
14. Understanding what is going one between A->B->C->D can be slow and detailed or fast and rough, depending on your need. If you goal is to fix a bug, the code between A->B->C->D can be unrelated to the bug. The most important clues to resolve the bug could be between D->E。 However, if your task is to maintain or extend a system, in which thorough understanding is necessary, understanding the system architecture, logic, and control flow from A->B->C->D can be important. More time and efforts should be devoted. It is suggested to draw some architecture diagram for your understanding in this regards. Don’t be surprised if you don’t find any architecture diagram in a legacy system.
15. Suppose you finally locate the bug at E and you found the code from A->B->C->D is irrelevant to the bug, you can disable the break points from A->D. To fix the bug, it is inevitable to rerun your program several times to understand the cause of the bug. After that, how to fix the bug is up to your talent.
16. May the force be with you. Good luck.

**Summary**

1. Talent programmers utilize debuggers to trace unfamiliar code
2. Reading the code with a test run is inefficient, slow, and pointless. Always identify a test run to go through your code of interest in code tracing no matter whether you want to fix a bug or understand a program in overall.
3. Use smart thinking to set break points to make program stop at the entry point of your interest.
4. Use call stack to go up to the callers of your code of interest
5. If you are not familiar with the programming language of the code under trace, use **next/step** to go through statement step by step. In principle, each statement just changes the variable values. So, by observing the values changes before and after the statement, you can always understand the code at very details. Sometimes, by reading the method name, you can have a good guess about what is done inside the method. You can decide whether to **step in** to understand more details or skip it by **next**
6. Actually, you do not need to master a programming language before code tracing. Talented programmers can use the above methods to trace an unfamiliar code and learn the new programming language on-the-fly.
7. Code tracing requires patience and carefulness. It can be hard. Trying to stop complaining the legacy code. In practice, your code can be much worse than the code you are tracing.
8. Any poorly written code (low readability) can be understood by **step/next** the code and observing the variable values changes before and after.
9. If your task is to understand the overall program behaviors, please draw diagrams, control flow and write notes when you go through your test run by debugger. This can help you avoiding unnecessary rediscovery. If you do not take such actions, you shall find your code tracing is trapped by rediscovery.
10. You should learn how to find proper names for your class, method, and variables from this exercise. Do not write poor readability program to torture other people. You should learn code to follow KISS (Keep It Simple and Stupid) principle.