**Debugging Techniques**

Testing a program against a well-chosen set of input tests gives the programmer confidence that the program is correct. During the testing process, the programmer observes input-output relationships, that is, the output that the program produces for each input test case. If the program produces the expected output and obeys the specification for each test case, then the program is successfully tested.

But if the output for one of the test cases is not the one expected, then the program is incorrect -- it contains **errors** (or **defects**, or "**bugs**"). In such situations, testing only reveals the presence of errors, but doesn't tell us what the errors are, or how the code needs to be fixed. In other words, testing reveals the **effects** (or **symptoms**) of errors, not the **cause** of errors. The programmer must then go through a debugging process, to identify the causes and fix the errors.

**Bug Prevention and Defensive Programming**

Surprisingly, the debugging process may take significantly more time than writing the code in the first place. A large amount (if not most) of the development of a piece of software goes into debugging and maintaining the code, rather than writing it.

Therefore, the best thing to do is to **avoid the bug when you write the program in the first place!** It is important to sit and think before you code: decide exactly what needs to be achieved, how you plan to accomplish that, design the high-level algorithm cleanly, convince yourself it is correct, decide what are the concrete data structures you plan to use, and what are the invariants you plan to maintain. All the effort spent in designing and thinking about the code before you write it will pay off later. The benefits are twofold. First, having a clean design will reduce the probability of defects in your program. Second, even if a bug shows up during testing, a clean design with clear invariants will make it much easier to track down and fix the bug.

It may be very tempting to write the program as fast as possible, leaving little or no time to think about it before. The programmer will be happy to see the program done in a short amount. But it's likely he will get frustrated shortly afterwards: without good thinking, the program will be complex and unclear, so maintenance and bug fixing will become an endless process.

Once the programmer starts coding, he should use **defensive programming**. This is similar to defensive driving, which means driving under worst-case scenarios (e.g, other drivers violating traffic laws, unexpected events or obstacles, etc). Similarly, defensive programming means developing code such that it works correctly under the worst-case scenarios from its environment. For instance, when writing a function, one should assume worst-case inputs to that function, i.e., inputs that are too large, too small, or inputs that violate some property, condition, or invariant; the code should deal with these cases, even if the programmer doesn't expect them to happen under normal circumstances.

Remember, the goal is not to become an expert at fixing bugs, but rather to get better at writing robust, (mostly) error-free programs in the first place. As a matter of attitude, programmers should not feel proud when they fix bugs, but rather embarrassed that their code had bugs. If there is a bug in the program, it is only because the programmer made mistakes.

**Classes of Defects**

Even after careful thought and defensive programming, a program may still have defects. Generally speaking, there are several kinds of errors one may run into:

* **Syntax or type errors.**These are always caught by the compiler, and reported via error messages. Typically, an error message clearly indicates the cause of error; for instance, the line number, the incorrect piece of code, and an explanation. Such messages usually give enough information about where the problem is and what needs to be done. In addition, editors with syntax highlighting can give good indication about such errors even before compiling the program.
* **Typos and other simple errors** that have pass undetected by the type-checker or the other checks in the compiler. Once these are identified, they can easily be fixed. Here are a few examples: missing parentheses, for instance writing x + y \* z instead of (x + y) \* z; typos, for instance case t of ... | x::tl => contains(x,t); passing parameters in incorrect order;  or using the wrong element order in tuples.
* **Implementation errors.** It may be the case that logic in the high-level algorithm of a program is correct, but some low-level, concrete data structures are being manipulated incorrectly, breaking some internal representation invariants. For instance, a program that maintains a sorted list as the underlying data structure may break the sorting invariant. Building separate ADTs to model each data abstraction can help in such cases: it can separate the logic in the algorithm from the manipulation of concrete structures; in this way, the problem is being isolated in the ADT. Calls to repOK() can further point out what parts of the ADT cause the error.
* **Logical errors.**If the algorithm is logically flawed, the programmer must re-think the algorithm. Fixing such problems is more difficult, especially if the program fails on just a few corner cases. One has to closely examine the algorithm, and try to come up with an argument why the algorithm works. Trying to construct such an argument of correctness will probably reveal the problem. A clean design can help a lot figuring out and fixing such errors. In fact, in cases where the algorithm is too difficult to understand, it may be a good idea to redo the algorithm from scratch and aim for a cleaner formulation.

**Difficulties**

The debugging process usually consists of the following: examine the error symptoms, identify the cause, and finally fix the error. This process may be quite difficult and require a large amount of work, because of the following reasons:

* *The symptoms may not give clear indications about the cause.* In particular, the cause and the symptom may be remote, either in space (i.e., in the program code), or in time (i.e., during the execution of the program), or both. Defensive programming can help reduce the distance between the cause and the effect of an error.
* *Symptoms may be difficult to reproduce.* Replay is needed to better understand the problem. Being able to reproduce the same program execution is a standard obstacle in debugging concurrent programs. An error may show up only in one particular interleaving of statements from the parallel threads, and it may be almost impossible to reproduce that same, exact interleaving.
* *Errors may be correlated.* Therefore, symptoms may change during debugging, after fixing some of the errors. The new symptoms need to be re-examined. The good part is that the same error may have multiple symptoms; in that case, fixing the error will eliminate all of them.
* *Fixing an error may introduce new errors.*Statistics indicate that in many case fixing a bug introduces a new one! This is the result of trying to do quick hacks to fix the error, without understanding the overall design and the invariants that the program is supposed to maintain. Once again, a clean design and careful thinking can avoid many of these cases.

**Debugging strategies**

Although there is no precise procedure for fixing all bugs, there are a number of useful strategies that can reduce the debugging effort. A significant part (if not all) of this process is spent localizing the error, that is, figuring out the cause from its symptoms. Below are several useful strategies to help with this. Keep in mind that different techniques are better suited in different cases; there is no clear best method. It is good to have knowledge and experience with all of these approaches. Sometimes, a combination of one or more of these approaches will lead you to the error.

* **Incremental and bottom-up program development.** One of the most effective ways to localize errors is to develop the program incrementally, and test it often, after adding each piece of code. It is highly likely that if there is an error, it occurs in the last piece of code that you wrote. With incremental program development, the last portion of code is small; the search for bugs is therefore limited to small code fragments. An added benefit is that small code increments will likely lead to few errors, so the programmer is not overwhelmed with long lists of errors.   
     
  Bottom-up development maximizes the benefits of incremental development. With bottom-up development, once a piece of code has been successfully tested, its behavior won't change when more code is incrementally added later. Existing code doesn't rely on the new parts being added, so if an error occurs, it must be in the newly added code (unless the old parts weren't tested well enough).
* **Instrument program to log information.** Typically, print statements are inserted. Although the printed information is effective in some cases, it can also become difficult to inspect when the volume of logged information becomes huge. In those cases, automated scripts may be needed to sift through the data and report the relevant parts in a more compact format. Visualization tools can also help understanding the printed data. For instance, to debug a program that manipulates graphs, it may be useful to use a graph visualization tool (such as ATT's [**graphviz**](http://www.graphviz.org/)) and print information in the appropriate format (.dot files for graphviz).
* **Instrument program with assertions.** Assertions check if the program indeed maintains the properties or invariants that your code relies on. Because the program stops as soon as it an assertion fails, it's likely that the point where the program stops is much closer to the cause, and is a good indicator of what the problem is. An example of assertion checking is the repOK() function that verifies if the representation invariant holds at function boundaries. Note that checking invariants or conditions is the basis of defensive programming. The difference is that the number of checks is usually increased during debugging for those parts of the program that are suspected to contain errors.
* **Use debuggers.**If a debugger is available, it can replace the manual instrumentation using print statements or assertions. Setting breakpoints in the program, stepping into and over functions, watching program expressions, and inspecting the memory contents at selected points during the execution will give all  the needed run-time information without generating large, hard-to-read log files.
* **Backtracking.** One option is to start from the point where to problem occurred and go back through the code to see how that might have happened.
* **Binary search.** The backtracking approach will fail if the error is far from the symptom. A better approach is to explore the code using a divide-and-conquer approach, to quickly pin down the bug. For example, starting from a large piece of code, place a check halfway through the code. If the error doesn't show up at that point, it means the bug occurs in the second half; otherwise, it is in the first half. Thus, the code that needs inspection has been reduced to half. Repeating the process a few times will quickly lead to the actual problem.
* **Problem simplification.** A similar approach is to gradually eliminate portions of the code that are not relevant to the bug. For instance, if a function fun f() = (g();h();k()) yields an error, try eliminating the calls to g, h, and k successively (by commenting them out), to determine which is the erroneous one. Then simplify the code in the body of buggy function, and so on. Continuing this process, the code gets simpler and simpler. The bug will eventually become evident. A similar technique can be applied to simplify data rather than code. If the size of the input data is too large, repeatedly cut parts of it and check if the bug is still present. When the data set is small enough, the cause may be easier to understand.
* **A scientific method: form hypotheses.** A related approach is as follows: inspect the test case results; form a hypothesis that is consistent with the observed data; and then design and run a simple test to refute the hypothesis. If the hypothesis has been refuted, derive another hypothesis and continue the process. In some sense, this is also a simplification process: it reduces the number of possible hypotheses at each step. But unlike the above simplification techniques, which are mostly mechanical, this process is driven by active thinking about an explanation. A good approach is to try to come with the simplest hypotheses and the simplest corresponding test cases.   
     
  Consider, for example, a function palindrome(s:string):bool, and suppose that palindrome("able was I ere I saw elba") returns an incorrect value of false. Here are several possible hypotheses for this failure. Maybe palindrome fails for inputs with spaces (test " "); maybe it fails for programs with upper case letters (try "I"); maybe it fails for inputs of odd length greater than one (try "ere"), and so on.  Forming and testing these hypotheses one after another can lead the programmer to the source of the problem.
* **Bug clustering.** If a large number of errors are being reported, it is useful to group them into classes of related bugs (or similar bugs), and examine only one bug from each class. The intuition is that bugs from each class have the same cause (or a similar cause). Therefore, fixing a bug with automatically fix all the other bugs from the same class (or will make it obvious how to fix them).
* **Error-detection tools.**Such tools can help programmers quickly identify violations of certain classes of errors. For instance, tools that check safety properties can verify that file accesses in a program obey the open-read/write-close file sequence; that the code correctly manipulates locks; or that the program always accesses valid memory. Such tools are either dynamic (they instrument the program to find errors at run-time), or use static analysis (look for errors at compile-time). For instance, [**Purify**](http://www-306.ibm.com/software/awdtools/purifyplus) is a popular dynamic tool that instruments programs to identify memory errors, such as invalid accesses or memory leaks. Examples of static tools include [**ESC Java**](http://research.compaq.com/SRC/esc/) and [**Spec#**](http://research.microsoft.com/projects/specsharp/), which use theorem proving approaches to check more general user specifications (pre and post-conditions, or invariants); or tools from a recent company [**Coverity**](http://www.coverity.com/) that use dataflow analysis to detect violations of safety properties. Such tools can dramatically increase productivity, but checking  is restricted to a particular domain or class of properties. There is also an associated learning curve, although that is usually low. Currently, there are relatively few such tools and this is more an (active) area of research.

A number of other strategies can be viewed as a matter of attitude about where to expect the errors:

* *The bug may not be where you expect it.* It a large amount of time has unsuccessfully been spent inspecting a particular piece of code, the error may not be there. Keep an open mind and start questioning the other parts of the program.
* *Ask yourself where the bug is not.* Sometimes, looking at the problem upside-down gives a different perspective. Often, trying to prove that the absence of a bug in a certain place actually reveals the bug in that place.
* *Explain to yourself or to somebody else why you believe there is no bug.* Trying to articulate the problem can lead to the discovery of the bug.
* *Inspect input data, test harness.*The test case or the test harness itself may be broken. One has to check these carefully, and make sure that the bug is in the actual program.
* *Make sure you have the right source code.* One must ensure that the source code being debugged corresponds to the actual program being run, and that the correct libraries are linked. This usually requires a few simple checks; using makefiles and make programs (e.g., the Compilation Manager and .cm files) can reduce this to just typing a single command.
* *Take a break.* If too much time is spent on a bug, the programmer becomes tired and debugging may become counterproductive. Take a break, clear your mind; after some rest, try to think about the problem from a different perspective.

All of the above are techniques for localizing errors. Once they have been identified, errors need to be corrected. In some cases, this is trivial (e.g., for typos and simple errors). Some other times, it may be fairly straightforward, but the change must ensure maintaining certain invariants. The programmer must think well about how the fix is going to affect the rest of the code, and make sure no additional problems are created by fixing the error. Of course, proper documentation of these invariants is needed. Finally, bugs that represent conceptual errors in an algorithm are the most difficult to fix. The programmer must re-think and fix the logic of the algorithm.