

Successful Programming Strategies

When first learning programming, the tendency is to concentrate on the syntax: keywords, data types, the constructs for branching (IF...THEN...ELSE, CASE SELECT) and looping (FOR...TO...NEXT, WHILE...WEND), and input/output control (INPUT, etc.). But programming is more than understanding the syntax of a language like Visual Basic.

As long as programs are simple, sitting down and writing code directly, hacking away until the program works more or less correctly, will probably seem to work o.k. (This is called the 'code-first' methodology, or "lack of" methodology.) But when programs get complex, you will want to pay attention to good, orderly programming habits. Professional programmers may work as members of a team, often generating packages of 100's of 1,000's of lines of code (The space shuttle software totals more than 10 million lines of code!) Writing code without doing some initial planning seems risky in such a situation.

Software engineering is based on a disciplined approach to the design, production and maintenance of computer programs, utilizing tools to help manage the size and complexity of the resulting software.

Identifiable software engineering steps (many of which can be handled concurrently) are:

1. Analyzing the problem: understanding the technical nature of the problem to be solved.
2. Defining the requirements: specifying just what the program must do.
3. High- and low-level design: recording how the program will meet user requirements, from the big picture to the program's final, detailed design.
4. Implementing the design: coding the program in a specific computer language like Visual Basic.
5. Testing and verifying: detecting and fixing errors, and demonstrating correct operation.
6. Executing the program.
7. In maintaining a program: having a professionally produced software release, the software author(s) can expect to make changes to fix errors as detected and reported by users "in the field", and to gradually add or modify functions of the program in response to perceived needs by the users.

The tools a programmer will use to build a program include hardware (computers and peripherals), software tools (operating systems, text editors for inputting program, compilers, interactive debuggers, etc.); and, "ideaware", the shared body of knowledge programmers have collected over time. Programmers build their own "libraries" (both physically real, books on their shelf, and virtual - their retained memories of what works) of algorithms used to solve common programming problems, programming methodologies, and various tools for measuring, evaluating and proving the correctness of their programs. Is programming an art or a science? There's plenty room for creativity, cleverness, innovation, but a programmer must still follow some basic principles of the craft, like any scientist or engineer.

The goal in programming, of course, is to generate a piece of quality software, a 'good' program. What makes for a good program? Quality software has these attributes: (1) it works, (2) its source code can be read and understood by others; (3) the code can be modified, if necessary, without excruciating time and effort, and is (4) the software project is completed on time (and, in the real world, within budget).

Goal 1 The program must work correctly and completely. So, the first step in writing good software is knowing exactly what it's supposed to do. You must have a definition of the program's requirements. For this you need a clear specification of data input format, and the expected output. How fast must the program execute? How big will the program be? How accurate will the results generated by the program be? How will the program handle user errors? The program must be

usable by a nonprofessional: if the program requires keyboard input from the operator sitting at the computer, the program should prompt the user for appropriate input. Are the program's outputs readable and understandable? Creating a good user interface is important. Of course, the form design features of Visual Basic are intended to facilitate creation of an effective graphical user interface (GUI). The program must be as efficient as it needs to be. The program shouldn't waste user time. The importance of execution time is especially critical in developing real-time control software, such as might be written for satellite launch control, or for laboratory experiment control.

Goal 2 The program must be readable and understandable. Clearly written programs are easier to understand. If it isn't understandable, it probably isn't correct. Note that this goal is quite applicable to your work on class exercises: if the instructor can't understand what you've done (or what you were attempting to do) in your exercise, then you have a problem.

Goal 3 The software must be modifiable without excruciating time and effort. Why? Software is always getting changed as the perceived needs for the software change. Software gets changed in the design phase (as you figure out how to solve the problem), in the coding phase (e.g. in response to interpreter and compiler error messages), in the testing phase (if the program crashes or gives erroneous results), and during the maintenance phase (for errors not discovered during testing, or when users specify new functions to be added, or when a user decides the output format should be different). Often the changes are made by someone other than the original author. What's required for easy modifiability? The program should be easily read (goal 2) and be able to survive little changes without making a mess of the entire program.

Goal 4 The must be completed on time, within budget. (This should apply to your course exercises; by the way, your allocated *dollar* budget for these projects, unfortunately, is zero. Your budget will likely be non-zero if and when you do this as part of your career activities.) The consequences of tardiness in timely completion of a software project can be profound in the real world; delays can hang up many others counting on getting functional software. There are usually monetary penalties to software houses when they miss delivery deadlines.

Understanding the problem. Here are some statistics regarding last semester's Bio 595 class and how they reacted to the first problem set in Basic: 38% panicked, 23% picked up a pencil and started writing code; 4% dropped the course; 8% copied the code from a smart classmate; and 27% stopped to think about how to design the program. You should instantly discard the first three of these solutions. Regarding the other two solutions, try to resist the temptation to grab a pencil and start writing code. First, read and then reread the problem. Ask questions (of yourself, and of the instructor, if necessary). The difficulty with writing code right off the bat is that it tends to lock you into the first solution you think of, which might not be the best approach. It's a natural tendency to feel that once you've put something in writing, you've invested too much time in the idea to toss it out and start over.

Writing specifications Start by writing a complete definition of the problem, including expected inputs, outputs, all the assumptions about the problem, and the necessary processing. The process of writing this out will bring to light any holes in the requirements (like, errors in the input). If the assigned problem is ambiguous (and the instructor's writing usually is), here's your chance to resolve the ambiguities. Here are some examples of questions to ask yourself. How is the input formatted? When do you know all the files have been read? How should output be formatted? Should you display a whole record, or just selected results? How about labels on the GUI? How should the data be sorted? It is important to note that details not explicitly stated in problem can be handled according to the programmer's preferences (that's you). Details about unspecified or ambiguous specifications should be written out explicitly in program's specifications as assumptions. A good way to cover program specifications is to provide four sections: inputs, outputs, processing requirements, and assumptions.

Solving the problem There is more than one way to solve most problems. Several functionally correct answers, and assorted algorithms to get those answers, may exist. (Remember that the word "algorithm" refers to a logical sequence of discrete steps used to solve a problem.) Knowing the purpose of the software often suggests appropriate choice of the algorithm. (For example, the size of the data set may suggest which sorting algorithm to use.)

Top-down design: How will your program solve the problem? You can develop and record a strategy to meet program goals by sketching out your algorithm on paper. The best approach is to use a "top-down" design, with step-wise refinement and a 'divide-and-conquer' approach. Try to break the problem into several large tasks, then divide each task in turn into sections, then subdivide the sections. Defer the details until you have a specific solution strategy, starting from a general one. Next, write a program outline, identifying the main modules; or, draw a diagram (a flow chart). Follow an outline format, in which specific details are hidden from higher levels of the outline. Once your outline is complete down to finest details, the program pretty much writes itself. Top-down programming starts with a 'big-picture' solution, then a general strategy for dividing the problem into modules. The problem at this stage can be written in pseudo-code, or in English, or in a flowchart form. Eventually, the main strategy becomes the main program, the modules become subprograms to be called. This approach encourages logical programming habits, and modular design.

Here's an example of an entire main program:

```
CALL GETDATA
CALL SORTDATA
CALL SHOWRESULTS
```

A top-down design to a program can be thought of as a tree-like structure, with the general problem at the top, and all the details branching out beneath it. The results of applying a top-down design to a software project is that the details that are specified in lower levels of the program design are hidden from the higher levels: this is called information hiding. The value of this 'hiding' is that it prevents high levels of the program's design from becoming dependent on low-level design details that are more likely to be changed. This strategy reduces the risk of errors. Information hiding separates a program's function into clear-cut levels of abstraction. Once we've stated the function of a low-level procedure and how input and output to and from the procedure is handled, we can forget about the precise details of how this is accomplished in the procedure. These are two fundamental concepts in software engineering: information hiding, and abstraction.

A program consists of an algorithm and data. Data structures can also be designed in a top-down fashion.. Proceed from an initial abstract view of the data to a more detailed picture, deferring details as long as possible. Just as there are multiple functionally correct solutions to a programming problem, so are there multiple possibilities for storing the data in a program. The first level of description would involve identifying a data file as lists or tables or records. Multiple lists or one table?

Implementing the solution: when the design is complete, write the code for the program. And, use liberal amounts of documentation. This involves more than just adding comments after you've written the program code: it includes the written description, specifications, design and actual source code of the program. Documentation is external (written material outside the body of the source code) and internal (comments, self-documenting code [see below], and program formatting for readability, e.g., tabs). Comments should help the reader understand what the programmer is doing at various points through the program; it isn't necessary to provide obvious comments, like

```
sngSum=0    'Set sum equal to zero.
```

Comments can be effectively used in your program in a number of key places, often next to the code lines described.

Declarations should include a comment on the intended use of each variable, such as

```
intSysPres(i) 'Systolic Blood Pressure of each patient
```

Program structures: add a comment next to each branch or loop in the program to explain the branch's or loop's function. Also, provide a comment after each WEND if it isn't in close proximity to the corresponding WHILE.

Subprogram or subroutine calls: explain the call's effect on the data. For example,

```
CALL DUMPIT      'Subroutine DUMPIT prints out current patient records
```

Also, comment on the parameter list of a subprogram call, especially if there are literal, non-obvious values or lots of parameters passed. In general, for any non-obvious code, or any confusing or particularly complicated programming, add explanations (but if possible, really tricky stuff should be avoided completely).

Self-documenting code: use meaningful identifier names to convey the function of variables, constants, and subroutine or subprogram names. For example, variable names X, Y and T don't reveal anything about the significance of these variables, but names like intPatientNumber and sngTensionInGrams do.

Program formatting (or 'pretty printing'): use blank lines, tabs or indentations to make readability of results better. Use constants with appropriate names instead of fixed numeric values; this makes it easier to change values later in the program. Example: FOR intpatno=1 TO 12 could be written as FOR intpatno=1 TO intTotalPatients, where the constant intTotalPatients was previously declared. If your program contains 6 loops where intpatno is indexed from 1 to 12, and you later increase the number of patients to 20, only one line of code instead of six lines need be changed.