Software Implementation – Q& A

1. Of the various languages available to you, list their benefits and their costs.

    Ada
    Benefits: truly portable; strongly typed; supports modularity; excellent file handling capabilities; supports abstraction; facilitates implementation of object-oriented design.
    Costs: unfamiliar to most DP programmers.

    BASIC
    Benefits: quick and easy to use; highly interactive; good string handling; good for DP purposes.
    Costs: no standard; not modular; slow to execute; not very portable.

    C/C++
    Benefits: powerful constructs; widely available; modular; highly portable; C++ facilitates implementation of object-oriented design and hence reuse. There are C compilers for most microprocessors, and many database APIs are available.
    Costs: easy to write dangerous code; low-level file handling; hard to maintain unless carefully written in a style that does not use "fancy" features.

    COBOL
    Benefits: business oriented; largely standardized; widely used and hence many programmers available; widely available; highly portable.
    Costs: verbose; **perform** statement changes global variables, not parameters; no typing.

    FORTRAN
    Benefits: efficient compilation; efficient to execute; widely implemented.
    Costs: not business oriented; implicit typing; poor file handling; incompatible language extensions.

    Java
    Benefits: widely available; strongly typed; modular; totally portable; strictly object-oriented; facilitates reuse; good exception handling. There are many APIs to facilitate database access as well as programming for embedded microcontrollers. A Java Virtual Machine can be loaded onto most microcontrollers.
    Costs:

    Lisp
    Benefits: interactive.
    Costs: slow; self-modifying code; not business oriented; hardly known by DP programmers.

    Pascal

Benefits: easy to use; widely available; block structured.
Costs: no separate compilation; incompatible language extensions.

Smalltalk
Benefits: strongly object-oriented; highly portable.
Costs: not widely used; hardly known by DP programmers.

2. How do coding standards for a one-person software production company differ from those in organizations with 300 software professionals?
There should be no difference at all because of the possibility that the organization will expand or be sold to someone else. Also, there is no guarantee that at maintenance time the one-person software developer will recall every detail of every artifact he or she has written. Finally, lack of an independent SQA function in a one-person environment means that particular care has to be taken to reduce faults, and adhering to standards is one such way.

3. How do coding standards for a software company that develops and maintains software for intensive-care units differ from those in an organization that develops and maintains accounting products?

   In theory there should be no difference whatsoever; software should always be built to the highest possible standards. In practice, however, a company that builds intensive care units will usually take greater care to ensure that their software is correct because human lives are at stake. (It is sad but true that, for many companies, the real reason they will take greater care is because of the risk of being sued should the intensive care unit software malfunction.) In order to achieve higher quality software, coding standards are likely to be more detailed and more rigorously enforced.

4. Consider the statement
   < *condition* 1> **and** < *condition* 2>
   In what programming languages is < *condition* 2> evaluated even if < *condition* 1> is false?

   Ada (the short-circuit operators are **and then** and **or else**)
   Pascal
   Visual Basic (the short-circuit operators are **AndAlso** and **OrElse**)

5. Why does deep nesting of **if** -statements frequently lead to code that can be diffi cult to read?

   An **if** statement has two possible successor statements, one if the condition is true, the other if the condition is false. That is, each **if** statement leads to two paths through the code. When **if** statements are nested too deeply, there are multiple paths, and the code becomes extremely complex.
6. Why has it been suggested that modules ideally should consist of between 35 and 50 statements?

Before there were monitors, the primary output device was the line printer. A module with 35 to 50 executable statements fitted comfortably on two sheets of line printer paper, making it easy for a programmer to debug a module while seated at a desk. With today's monitors, a programmer can debug a module with 35 to 50 executable statements without too much scrolling up and down. However, multiple windowing has made this "rule" less important.

7. Why should backward **goto** statements be avoided, whereas a forward **goto** may be used for error handling?

   All **goto** statements violate the tenets of structured programming. A backward **goto** is a loop, and should be programmed as such; there should never be a need for a backward **goto**. However, in the case of an error condition being detected, it is sometimes necessary to jump to a different part of the code, for example, to prevent damaging a database or causing a stack to overflow. In such case, a forward **goto** may be the most effective way to recover from the error condition or to minimize the resulting damage.

8. What are the similarities between product testing and acceptance testing? What are the major differences?
   Product testing is essentially a dress rehearsal for the acceptance test. As a consequence, the aim of product testing is to be as close as possible to acceptance testing in every way. The major difference is that product testing is performed by the developers, acceptance testing by the client. Also, acceptance testing should be performed as far as possible on actual data, not test data, whereas this may not be possible for product testing.