**SOLUTION NOTES**

**Software Engineering II 2002 Paper 2 Question 8 (LCP)**

(*a*) Top-down refinement is covered in the first lecture. It works by dividing the programming task into several subtasks, which are 'coded' by dummy procedures or *stubs*. Each subtask is then implemented by recursive application of top-down refinement.

Advantages are, for example, that the program is always 'executable' (even if it does nothing), that the final product's structure will have a natural hierarchical structure, and that the lower levels can be parcelled out to different team members. Drawbacks are, for example, that this style tends to produce programs that have lots of trivial units. Also, the first decompositions give the impression that the top levels are already finished, when further refinements may indicate that they need modification. Finally (though this is always the case) poor problem decompositions made initially can be hard to fix later, and if they are not undone then they can continue to cause problems throughout the development.

(*b*) Efficiency is covered in lecture three. Obviously, choose good data structures and algorithms, and try to use efficient libraries. But for general code, it is best to code in a straightforward style, avoiding gross inefficiecy but not particularly coding for efficiency either. Once the product or unit is finished, and if it is not efficient enough, use profiling tools to discover sources of inefficiency. These can be improved by the obvious methods, including recoding parts in assembly language. But 'write the whole program in assembly language' is wrong, as it sacrifices relability and productivity.

(*c*) This again is from lecture three. Possible answers include using a language with built-in garbage collection, such as Java, or using tools such as Third Degree or Electric Fence, which check for common memory-reference errors. (Students do not need to remember the names of these tools.) You can include debugging code that initializes all allocated data with a carefully-chosen bit-pattern: one that would lead to an exception being raised if it is accessed later. Memory management is one of the leading causes of bugs.

(*d*) Z is described in lecture five. A *schema* enumerates state variables with their types and an associated predicate. A $\Delta$ schema refers to an existing schema, but has two copies of each state variable: one unprimed and one primed, as in $x$ and $x'$. The primed variables describe the state after execution of the action in relation to the state beforehand; for example $x' = x + 1$ describes an action that adds 1 to $x$. A $\Xi$ schema is simply a $\Delta$ schema with the additional constraint that each primed variable equals the unprimed one, e.g. $x' = x$; that looks redundant, but it ensures that both types of schema have the same signature (set of variables) and can therefore be combined using Z connectives.

(*e*) Lectures two and five are relevant. Both types of invariant describe a relationship that must hold among the state variables. A loop invariant holds just before the loop body is executed, and (in the case of a **while** loop) immediately before the loop is exited. Therefore, from the combination of the invariant and the loop's termination condition, we can deduce something about the resulting state.

A Z invariant describes a relationship that must hold among a schema's state variables. Each $\Xi$ or $\Delta$ schema inherits the invariant for both the primed and umprimed variables, in effect adding the invariant to the specification of the corresponding action.