Chapter 7: Writing the programs

1. A computed case statement interferes with the top-down flow of a program by jumping from the case statement to various other parts. It might enhance maintainability, however, by making the decision structure very explicit and clear. The result of each value of the variable will be obvious in a computed case statement. New cases are easy to add and can be assumed not to conflict with any of the other cases. Reusability may be hampered because the conditions for branching are constrained to be the value of one variable. The entire case statement would have to be rewritten to accommodate more complicated decisions.

2. All parties involved have some area of responsibility in ensuring that the component does not fail. The original developer as well as those who modified the component are all responsible for ensuring its reliability. All developers are also responsible for documenting their work clearly enough to avoid misunderstandings on the part of those who might modify it later. In theory, the developer who made the original error which resulted in a fault in the code that then caused the software to fail is responsible, but this is usually not possible to ascertain. Anyone evaluating a component for possible reuse should be sufficiently convinced of its reliability to be willing to take responsibility for any failures.

3. Recursive definition: A list is null, a single element, or an element followed by a list.

   Items are added to a list by using the third part of the recursive definition. The new element followed by the original list is, by definition, a list.

   Items are also deleted using the third part of the definition. If the first element of a list is deleted, the result, by definition, is still a list. If the list is null, it remains unchanged (or deleting causes an error).

4. Following are two Pascal code fragments, both of which add an item to a list. The first is recursive (using links), the second is not. The first should be easier to understand.

```
/***************/
type
       list = ^node
       node = record
              data: integer;
              next: list;
       end;

var    L : list;

procedure add_data (var L: list; datum: integer)
begin
       new_node := new (node);
       new_node^.data := datum;
       new_node^.next := L;
       L:= new_node
end;

/**************/
type
       list = record
              data: array[1..MAX] of integer;
              first: 1..MAX;
              last:  1..MAX;
       end;

procedure add_data (var L: list; datum: integer);
var i: integer;
begin
       if L.first = 1 then begin
       /* move all elements down 1 spot */
              if L.last=MAX then
              /* so we don't run over the end of the array */
                     L.last := L.last-1;
              for i := last downto 1 do
                     L.data[i+1] := L.data[i];
              L.data[1] := datum;
              L.last := L.last + 1
```

```
                end
                else begin
                        L.first := L.first - 1;
                        L.data[L.first] := datum
                end
        end;
```

5. One possibility is to store, for each possible year that the user could specify, an array element containing the day of the week on which the year starts, and whether or not it is a leap year. The program to print the calendar would index into the array and use the two pieces of information to print out the entire year. This would require that the rules about the numbers of days in each month would have to be coded into the computational part of the program.

   Another strategy is to store the rules about leap years and the number of days in each month in a single data structure. This structure could have an array of 12 numbers, each corresponding to the number of days in a month. It could also contain the constants needed in the algorithm to determine whether or not the year entered is a leap year (e.g. 4, 100, etc.). In this case, the computational part of the program would be much more complex, but the data structure much more compact.

6. Comments:

```
/* First, calculate the discriminant, b² - 4ac */
:
/* Case 1: The discriminant is less than 0, so there are no real roots - output
a message */
:
/* Case 2: The discriminant is equal to 0, so there is just one real root -
calculate and output the root */
:
/* Case 3: The discriminant is greater than 0, so there are two real roots -
calculate and output the roots */
:
```

   External documentation:

   The algorithm for calculating the roots of a quadratic equation has two steps. The first step is to calculate the discriminant, which is defined by a formula involving the coefficients of the equation. The second step depends on the value of the discriminant calculated in the first step. If the discriminant is less than zero, this indicates that the equation has no real roots, and a message to that effect is output. If the discriminant is equal to zero, then there is one real root. This root is calculated using the quadratic formula and output. If the discriminant is greater than zero, then there are two real roots. In this case, both roots are calculated using the quadratic formula and output.

7. This OS implements the LRU (least recently used) page replacement policy. Each process running is allocated a particular number of page frames, depending on the current load on the system. As a process requires different pages from secondary memory, they are loaded into the frames allocated to that process. Each time a page in memory is read or written, it is timestamped (i.e. it is marked with the time that it is accessed). Each time a page in memory is written to, that page is also marked "dirty," signifying that it needs to be copied back to secondary memory at some point. If a process requires a page that is not in main memory, and all the process's frames are allocated, then a page must be replaced. The page that has the oldest timestamp is replaced. If it is dirty, then it is written to secondary memory before being replaced.

8. Some improvements to consider:

   - standard header comments for each component
   - comments containing pseudocode
   - meaningful variable names
   - top-down control structure
   - greater modularity
   - greater generality
   - localizing input and output procedures
   - effective use of formatting and white space

9. The advantages are the economies of scale in terms of training, documentation, and tool licenses. As well, improvement efforts will be more efficient as improvements that are discovered in one application

area would more likely be applicable to other areas. The disadvantage is that it might constrain some projects to use a particular language or tool when it is not the best choice for that application, thus resulting in inefficiencies for that project.

10. Many standards can be coded into the tool itself, such as standards for header comments and variable names. Design and documentation standards might be facilitated by modifying them so that they are at least partially satisfied by the specifications written for the code generator. In the case of reuse from a repository, the standards must be enforced for the higher-level components that call or include the reused components. Enforcing them in the reused components may negate any cost savings from reuse.

11. In fact, it can't easily, which is a major problem for testing and documentation, especially when OO is used for real-time systems. It is sometimes possible for the software designer to map out the **intended** control flow, but in an OO program, one can never tell in advance the order in which objects will be invoked. So there is more uncertainty, and in a sense less control, with an OO program or design than there is with a procedural (hence the name) design. But you can capture most of the relevant information by using a state transition diagram/chart or petri net to describe each object's states and possible actions. Then the charts or nets can be used to help trace the likely control flow.