A Survey of Programming Techniques

This chapter is a short survey of programming techniques. We use a simple example to illustrate the particular properties and to point out their main ideas and problems.

Roughly speaking, we can distinguish the following learning curve of someone who learns to program:

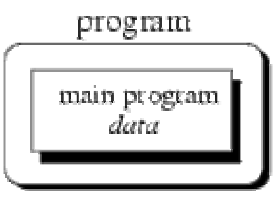
- Unstructured programming,
- procedural programming,
- modular programming and
- object-oriented programming.

This chapter is organized as follows. Sections 2.1 to 2.3 briefly describe the first three programming techniques. Subsequently, we present a simple example of how modular programming can be used to implement a singly linked list module (section 2.4). Using this we state a few problems with this kind of technique in section 2.5. Finally, section 2.6 describes the fourth programming technique.

2.1 Unstructured Programming

Usually, people start learning programming by writing small and simple programs consisting only of one main program. Here ``main program" stands for a sequence of commands or *statements* which modify data which is *global* throughout the whole program. We can illustrate this as shown in Fig. 2.1.

Figure 2.1: Unstructured programming. The main program directly operates on global data.

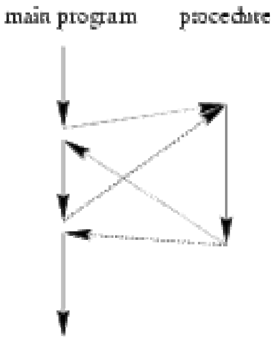


As you should all know, this programming techniques provide tremendous disadvantages once the program gets sufficiently large. For example, if the same statement sequence is needed at different locations within the program, the sequence must be copied. This has lead to the idea to *extract* these sequences, name them and offering a technique to call and return from these *procedures*.

2.2 Procedural Programming

With procedural programming you are able to combine returning sequences of statements into one single place. A *procedure call* is used to invoke the procedure. After the sequence is processed, flow of control proceeds right after the position where the call was made (Fig. 2.2).

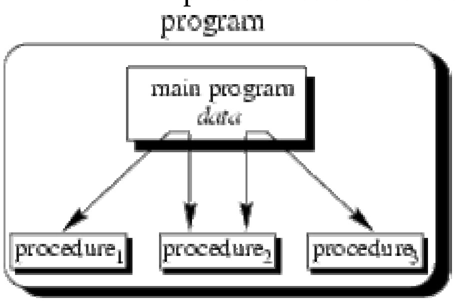
Figure 2.2: Execution of procedures. After processing flow of controls proceed where the call was made.



With introducing *parameters* as well as procedures of procedures (*subprocedures*) programs can now be written more structured and error free. For example, if a procedure is correct, every time it is used it produces correct results. Consequently, in cases of errors you can narrow your search to those places which are not proven to be correct.

Now a program can be viewed as a sequence of procedure calls. The main program is responsible to pass data to the individual calls, the data is processed by the procedures and, once the program has finished, the resulting data is presented. Thus, the *flow of data* can be illustrated as a hierarchical graph, a *tree*, as shown in Fig. 2.3 for a program with no subprocedures.

Figure 2.3: Procedural programming. The main program coordinates calls to procedures and hands over appropriate data as parameters.

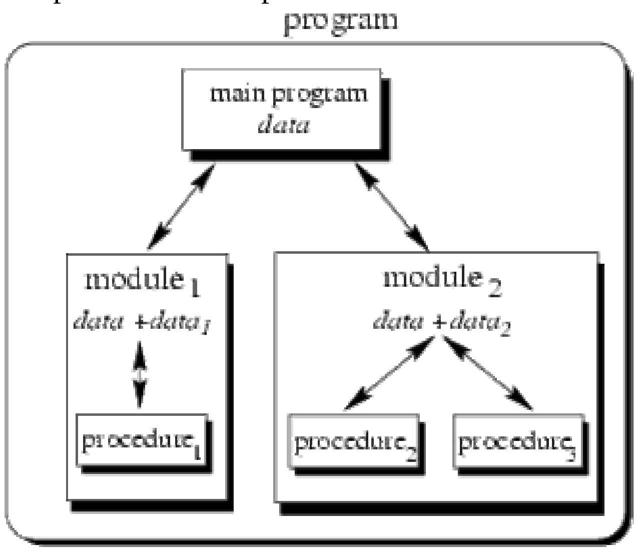


To sum up: Now we have a single program which is devided into small pieces called procedures. To enable usage of general procedures or groups of procedures also in other programs, they must be separately available. For that reason, modular programming allows grouping of procedures into modules.

2.3 Modular Programming

With modular programming procedures of a common functionality are grouped together into separate *modules*. A program therefore no longer consists of only one single part. It is now divided into several smaller parts which interact through procedure calls and which form the whole program (Fig. 2.4).

Figure 2.4: Modular programming. The main program coordinates calls to procedures in separate modules and hands over appropriate data as parameters.



Each module can have its own data. This allows each module to manage an internal *state* which is modified by calls to procedures of this module. However, there is only one state per module and each module exists at most once in the whole program.

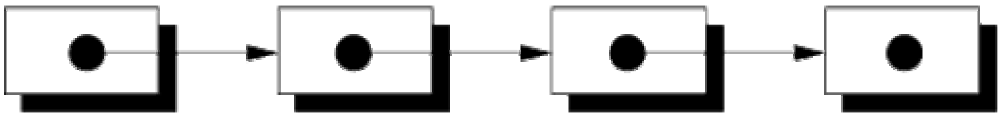
2.4 An Example with Data Structures

Programs use data structures to store data. Several data structures exist, for example lists, trees, arrays, sets, bags or queues to name a few. Each of these data structures can be characterized by their *structure* and their *access methods*.

2.4.1 Handling Single Lists

You all know singly linked lists which use a very simple structure, consisting of elements which are strung together, as shown in Fig. 2.5).

Figure 2.5: Structure of a singly linked list.



Singly linked lists just provides access methods to *append* a new element to their end and to *delete* the element at the front. Complex data structures might use already existing ones. For example a *queue* can be structured like a singly linked list. However, queues provide access methods to *put* a data element at the end and to *get* the first data element (*first-in first-out* (FIFO) behaviour).

We will now present an example which we use to present some design concepts. Since this example is just used to illustrate these concepts and problems it is neither complete nor optimal

Suppose you want to program a list in a modular programming language such as C or Modula-2. As you believe that lists are a common data structure, you decide to implement it in a separate *module*. Typically, this requires you to write two files: the *interface definition* and the *implementation file*. Within this chapter we will use a very simple pseudo code which you should understand immediately. Let's assume, that comments are enclosed in ``/* ... */''. Our interface definition might then look similar to that below:

```
/*
 * Interface definition for a module which implements
 * a singly linked list for storing data of any type.
 */

MODULE Singly-Linked-List-1

BOOL list_initialize();
BOOL list_append(ANY data);
BOOL list_delete();
     list_end();

ANY list_getFirst();
ANY list_getNext();
BOOL list_isEmpty();

END Singly-Linked-List-1
```

Interface definitions just describe *what* is available and not *how* it is made available. You *hide* the information of the implementation in the implementation file. This is a fundamental principle in software engineering, so let's repeat it: You *hide* information of the actual implementation (*information hiding*). This enables you to change the implementation, for example to use a faster but more memory consuming algorithm for storing elements without the need to change other modules of your program: The calls to provided procedures remain the same.

The idea of this interface is as follows: Before using the list one has to call *list_initialize()* to initialize variables local to the module. The following two procedures implement the mentioned access methods *append* and *delete*. The *append* procedure needs a more detailed discussion. Function *list_append()* takes one argument *data* of arbitrary type. This is necessary since you wish to use your list in several different environments, hence, the type of the data elements to be stored in the list is not known beforehand. Consequently, you have to use a special type *ANY* which allows assigning data of any type to it. The third procedure *list_end()* needs to be called when the program terminates to enable the module to clean up its internally used variables. For example you might want to release allocated memory.

With the next two procedures *list_getFirst()* and *list_getNext()* a simple mechanism to traverse through the list is offered. Traversing can be done using the following loop:

```
ANY data;

data <- list_getFirst();
WHILE data IS VALID DO
    doSomething(data);
    data <- list_getNext();
END
```

Now you have a list module which allows you to use a list with any type of data elements. But what, if you need more than one list in one of your programs?

2.4.2 Handling Multiple Lists

You decide to redesign your list module to be able to manage more than one list. You therefore create a new interface description which now includes a definition for a *list handle*. This handle is used in every provided procedure to uniquely identify the list in question. Your interface definition file of your new list module looks like this:

```
/*
 * A list module for more than one list.
 */

MODULE Singly-Linked-List-2

DECLARE TYPE list_handle_t;

list_handle_t list_create();
              list_destroy(list_handle_t this);
BOOL          list_append(list_handle_t this, ANY data);
ANY           list_getFirst(list_handle_t this);
ANY           list_getNext(list_handle_t this);
BOOL          list_isEmpty(list_handle_t this);

END Singly-Linked-List-2;
```

You use *DECLARE TYPE* to introduce a new type *list_handle_t* which represents your list handle. We do not specify how this handle is actually represented or even implemented. You also *hide* the implementation details of this type in your implementation file. Note the difference to the previous version where you just hide functions or procedures, respectively. Now you also hide information for an user defined data type called *list_handle_t*.

You use *list_create()* to obtain a handle to a new thus empty list. Every other procedure now contains the special parameter *this* which just identifies the list in question. All procedures now operate on this handle rather than a module global list.

Now you might say, that you can create *list objects*. Each such object can be uniquely identified by its handle and only those *methods* are applicable which are defined to operate on this handle.

2.5 Modular Programming Problems

The previous section shows, that you already program with some object-oriented concepts in mind. However, the example implies some problems which we will outline now.

2.5.1 Explicit Creation and Destruction

In the example every time you want to use a list, you explicitly have to declare a handle and perform a call to *list_create()* to obtain a valid one. After the use of the list you must explicitly call *list_destroy()* with the handle of the list you want to be destroyed. If you want to use a list within a procedure, say, *foo()* you use the following code frame:

```
    PROCEDURE foo() BEGIN
        list_handle_t myList;
        myList <- list_create();

        /* Do something with myList */
        ...

        list_destroy(myList);
    END
```

Let's compare the list with other data types, for example an integer. Integers are declared within a particular scope (for example within a procedure). Once you've defined them, you can use them. Once you leave the scope (for example the procedure where the integer was defined) the integer is lost. It is automatically created and destroyed. Some compilers even initialize newly created integers to a specific value, typically 0 (zero).

Where is the difference to list ``objects''? The lifetime of a list is also defined by its scope, hence, it must be created once the scope is entered and destroyed once it is left. On creation time a list should be initialized to be empty. Therefore we would like to be able to define a list similar to the definition of an integer. A code frame for this would look like this:

```
    PROCEDURE foo() BEGIN
```

```
        list_handle_t myList; /* List is created and initialized */

        /* Do something with the myList */
        ...
     END /* myList is destroyed */
```

The advantage is, that now the compiler takes care of calling initialization and termination procedures as appropriate. For example, this ensures that the list is correctly deleted, returning resources to the program.

2.5.2 Decoupled Data and Operations

Decoupling of data and operations leads usually to a structure based on the operations rather than the data: Modules group common operations (such as those *list_...()* operations) together. You then use these operations by providing explicitly the data to them on which they should operate. The resulting module structure is therefore oriented on the operations rather than the actual data. One could say that the defined operations specify the data to be used.

In object-orientation, structure is organized by the data. You choose the data representations which best fit your requirements. Consequently, your programs get structured by the data rather than operations. Thus, it is exactly the other way around: Data specifies valid operations. Now modules group data representations together.

2.5.3 Missing Type Safety

In our list example we have to use the special type *ANY* to allow the list to carry any data we like. This implies, that the compiler cannot guarantee for type safety. Consider the following example which the compiler cannot check for correctness:

```
    PROCEDURE foo() BEGIN
        SomeDataType data1;
        SomeOtherType data2;
        list_handle_t myList;

        myList <- list_create();
        list_append(myList, data1);
        list_append(myList, data2); /* Oops */

        ...

        list_destroy(myList);
    END
```

It is in your responsibility to ensure that your list is used consistently. A possible solution is to additionally add information about the type to each list element. However, this implies more overhead and does not prevent you from knowing what you are doing.

What we would like to have is a mechanism which allows us to specify on which data type the list should be defined. The overall function of the list is always the same, whether we store apples, numbers, cars or even lists. Therefore it would be nice to declare a new list with something like:

```
    list_handle_t<Apple> list1; /* a list of apples */
    list_handle_t<Car> list2; /* a list of cars */
```

The corresponding list routines should then automatically return the correct data types. The compiler should be able to check for type consistency.

2.5.4 Strategies and Representation

The list example implies operations to traverse through the list. Typically a *cursor* is used for that purpose which points to the *current element*. This implies a *traversing strategy* which defines the order in which the elements of the data structure are to be visited.

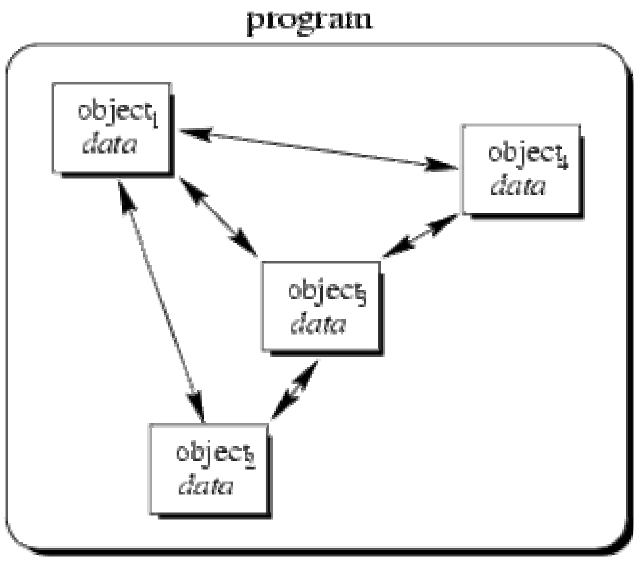
For a simple data structure like the singly linked list one can think of only one traversing strategy. Starting with the leftmost element one successively visits the right neighbours until one reaches the last element. However, more complex data structures such as trees can be traversed using different strategies. Even worse, sometimes traversing strategies depend on the particular context in which a data structure is used. Consequently, it makes sense to separate the actual representation or *shape* of the data structure from its traversing strategy.

What we have shown with the traversing strategy applies to other strategies as well. For example insertion might be done such that an order over the elements is achieved or not.

2.6 Object-Oriented Programming

Object-oriented programming solves some of the problems just mentioned. In contrast to the other techniques, we now have a web of interacting *objects*, each house-keeping its own state (Fig. 2.6).

Figure 2.6: Object-oriented programming. Objects of the program interact by sending messages to each other.



Consider the multiple lists example again. The problem here with modular programming is, that you must explicitly create and destroy your list handles. Then you use the procedures of the module to modify each of your handles.

In contrast to that, in object-oriented programming we would have as many list objects as needed. Instead of calling a procedure which we must provide with the correct list handle, we would directly send a message to the list object in question. Roughly speaking, each object implements its own module allowing for example many lists to coexist.

Each object is responsible to initialize and destroy itself correctly. Consequently, there is no longer the need to explicitly call a creation or termination procedure.

You might ask: *So what? Isn't this just a more fancier modular programming technique?* You were right, if this would be all about object-orientation. Fortunately, it is not. Beginning with the next chapters additional features of object-orientation are introduced which makes object-oriented programming to a new programming technique.