



Ahoo Engineering Group

Software Engineering

by Nguyen Xuan Huy

Institute of Information Technology,

National Center for Natural Science and Technology

Email: nxhuy@ioit.ncst.ac.vn

Duration: 45 hours



Chapter 4 Implementation

Objectives

This chapter gives a general discussion of software implementation. First we give an overview of the attributes of programming languages which impact on quality; then we discuss certain elements of good programming style such as structuredness, expressiveness, outward form and efficiency. Finally we discuss the main properties of software such as portability and reusability.

Contents

- 4.1 Programming environments
 - 4.2 Programming style
 - 4.3 Portability and reuse
 - 4.3.1 Software portability
 - 4.3.2 Machine architecture dependencies
 - 4.3.3 Operating system dependencies
 - 4.3.4 Software reuse
 - 4.4 Computer-aided software engineering
 - 4.4.1 CASE workbenches
 - 4.4.2 Text editing systems
 - 4.4.3 Language processing systems
 - 4.5 Incremental implementation
 - References and selected reading
-

- Implementation of a software system = the transformation (coding) of the design results into programs that are executable on certain target machine.
- A good implementation reflects the design decisions.
- The implementation should ensure the following:
 - The decomposition structures, data structures and identifiers selected and established during the design procedure are easily recognized in the implementation.
 - The abstraction levels of the design (the classes, modules, algorithms, data structures and data types) must also be feasible in the implementation.



- The interfaces between the components of a software system should be explicitly described in the implementation.
- It must be possible to check the consistency of objects and operations immediately with the compiler (before the actual testing phase).
- The degree to which the above characteristics are fulfilled depends on the choice of the implementation language and on the programming style.

4.1 Programming environments

- ❖ The question of which is the “right” programming language has always been a favorite topic of discussion in programming circles.
- ❖ The choice of a programming language for the implementation of a project frequently plays an important role.
- ❖ In the ideal case, the design should be carried out without any knowledge of the later implementation language so that it can be implemented in any language.

Quality criteria for programming languages:

- Modularity
 - Documentation value
 - Data structures
 - Control flow
 - Efficiency
 - Integrity
 - Portability
 - Dialog support
 - Specialized language elements
- ***Modularity of a programming language***
- ❖ Degree to which it support modularization of programs.
 - ❖ The decomposition of a large program into multiple modules is a prerequisite for carrying out large software projects.
 - ❖ Without modularization, division of labor in the implementation becomes impossible. Monolithic programs become unmanageable: they are difficult to maintain and document and they impede implementation with long compile times.
 - ❖ Languages such as standard Pascal (which does not support modules, but compare with Turbo Pascal and Modula-2) prove unsuitable for large projects.
 - ❖ If a language supports the decomposition of a program into smaller units, there must also be assurance that the components work together. If a procedure is invoked from another module, there must be a check of whether the procedure actually exists and whether it is used correctly (i.e. whether the number of parameters and their data types are correct).



- ❖ Languages may have independent compilation (e.g. C and FORTRAN), where this check takes place only upon invocation at run time (if at all)
- ❖ Alternatively, languages may have separate compilation (e.g. Ada and Modula-2), where each module has an interface description that provides the basis for checking its proper use already at compile time.
- ***Documentation value of a programming language***
 - ❖ Affects the readability and thus the maintainability of programs.
 - ❖ The importance of the documentation value rises for large programs and for software that the client continues to develop.
 - ❖ High documentation value results, among other things, from explicit interface specifications with separate compilation (e.g. in Ada and Modula-2). Likewise the use of keywords instead of special characters (e.g. **begin . . . end** in Pascal rather than { . . . } in C) has a positive effect on readability because the greater redundancy gives less cause for careless errors in reading. Since programs are generally written only once but read repeatedly, the minimum additional effort in writing pays off no more so than in the maintenance phase. Likewise the language's scoping rules influence the readability of programs.
 - ❖ Extensive languages with numerous specialized functions (e.g. Ada) are difficult to grasp in all their details, thus encouraging misinterpretations. Languages of medium size and complexity (e.g. Pascal and Modula-2) harbor significantly less such danger.
- ***Data structures in the programming language***
 - ❖ Primarily when complex data must be processed, the availability of data structures in the programming language plays an important role.
 - ❖ Older languages such as FORTRAN, BASIC, and COBOL offer solely the possibility to combine multiple homogeneous elements in array or heterogeneous elements in structures.
 - ❖ Recursive data structures are difficult to implement in these languages.
 - ❖ Languages like C permit the declaration of pointers to data structures. This enables data structures of any complexity, and their scope and structure can change at run time. However, the drawback of these data structures is that they are open and permit unrestricted access (but compare with Java [Heller 1997]).
 - ❖ Primarily in large projects with multiple project teams, abstract data takes on particular meaning. Although abstract data structures can be emulated in any modular language, due to better readability, preference should be given to a language with its own elements supporting this concept.
 - ❖ Object-oriented languages offer the feature of extensible abstract data types that permit the realization of complex software systems with elegance and little effort. For a flexible and extensible solution, object-oriented languages provide a particularly good option.
- ***Structuring control flow in the programming language***
 - ❖ Languages like BASIC and FORTRAN include variations of a GOTO statement, which programmers can employ to create unlimited and incomprehensible control



flow structures. In Pascal and C the use of the GOTO statement is encumbered because the target must be declared explicitly. In Eiffel, Modula-2 and Smalltalk there is no GOTO statement at all, which forces better structuring of the control flow.

- ❖ In technical applications additional possibilities for control flow can play an important role. These include the handling of exceptions and interrupts as well as parallel processes and their synchronization mechanisms. Many programming languages (e.g., Ada and Eiffel) support several of these concepts and thus permit simpler program structures in certain cases.

Notes: *Every deviation from sequential flow is difficult to understand and thus has a negative effect on readability.*

➤ **Efficiency of a programming language**

- ❖ The *efficiency* of a programming language is often overrated as a criterion. For example, the programming language C bears the reputation that it supports the writing of very efficient programs, while object-oriented languages are accused of inefficiency. However, there are few cases where a language is in principle particularly efficient or especially inefficient.
- ❖ Optimizing compilers often generate excellent code that an experienced Assembler programmer could hardly improve upon. For time-critical operations, it pays to use a faster machine or a better compiler rather than a “more efficient” programming language.

➤ **Integrity of a programming language**

- ❖ The integrity of a programming language emanates primarily from its readability and its mechanisms for type checking (even across module boundaries).
- ❖ Independent of the application domain, therefore, a language with static typing should be preferred. Static typing means that for each expression the compiler can determine which type it will have at run time. Additional integrity risks include type conversions (type casts) and pointer arithmetic operations, which are routine for programming in C, for example.
- ❖ Run-time checks are also important for integrity, especially during the development phase. These normally belong in the domain of the compiler.
- ❖ Mechanisms for formulating assertions (e.g. in Eiffel) and features for exception handling (e.g. in Eiffel and Ada) also contribute to the integrity of a programming language.

➤ **Portability**

- ❖ *Portability* can be a significant criterion if a software product is destined for various hardware platforms. In such a situation it makes sense to select a standardized language such as Ada or C. However, this alone does not suffice to ensure portability. For any external modules belonging to the language also need to be standardized. This is a problem in the language Modula-2 because various compiler producers offers different module libraries.
- ❖ Beyond standardization, another criterion is the availability of compilers for the language on different computers. For example, developers of software for mainframes will find a FORTRAN compiler on practically every machine.



➤ *Dialog support*

- ❖ For interactive programs the programming language must also provide *dialog support*. For example, FORTRAN and COBOL offer only line-oriented input and output;
- ❖ Highly interactive programs (that react to every key pressed) can thus be developed only with the help of specialized libraries.
- ❖ Some languages like BASIC and Logo are particularly designed provide dialog support for the user (and with the programmer), making these languages better suited for such applications.
- ❖ Object-oriented programming languages also prove well suited to the development of interactive programs, especially with the availability of a corresponding class library or an application framework.
- ❖ For specialized tasks, *specialized language elements* can be decisive for the selection of a programming language. For technical applications, for example, the availability of complex number arithmetic (e.g. in COBOL) can be important. For mathematical problems, matrix operations (e.g. in APL) can simplify the task, and translation and character string operations are elegantly solved in Snobol. The lack of such specialized language elements can be compensated for with library modules in modular languages.
- ❖ Object-oriented languages prove particular suited to extending the language scope.

Additional characteristics of programming languages:

- Quality of the compiler
- Availability of libraries
- Availability of development tools
- Company policy
- External requirements

➤ *Quality of the compiler*

The *quality of the compiler* is decisive for the actual implementation phase. A good compiler should not only generate efficient code, but also provide support for debugging (e.g. with clear error messages and run-time checks). Particularly in the area of microcomputers, many compilers have been integrated in development systems. Here the user-friendliness of such systems must also be considered.

➤ *Availability of libraries*

With modular programming languages, the *availability of libraries* for various application domains represents a significant selection criterion. For example, for practically all FORTRAN compilers, libraries are available with numerous mathematical functions, and Smalltalk class libraries contain a multitude of general classes for constructing interactive programs. The availability of libraries can also be used in C or Modular-2 if the compiler supports linking routines from different languages. On the other hand, there are libraries that are available only in compiled form and usable only in connection with a certain compiler.



➤ *Availability of development tools*

Today's trend is toward the use of *tools* to support software development. Many tools intended for the implementation phase are bound to a particular programming language or even a certain compiler. Examples of such tools include structure-oriented editors, debugger and program generators. For example, if tools like Lex and YACC (tools for the application of attribute grammars on UNIX machines) are to be employed, then an implementation in C becomes unavoidable; choosing another language would only be justified if it promises gains greater than the cost of forfeiting the tool.

➤ *Company policy*

Often a particular *company policy* influences the choice of a programming language. Frequently the language decision is made not by the implementors, but by managers that want to use only a single language company-wide for reasons of uniformity. Such a decision was made by the U.S. Department of Defense, which mandates the use of Ada for all programming in the military sector in the U.S (and thus also in most other NATO countries). Such global decisions have also been made in the area of telecommunications, where many programmers at distributed locations work over decades on the same product.

Even in-house situations, such as the education of the employees or a module library built up over years, can force the choice of a certain language. A company might resist switching to a more modern programming language to avoid training costs, the purchase of new tools, and the re-implementation of existing software.

➤ *External requirements*

Sometimes *external requirements* force the use of a given programming language. Contracts for the European Union increasingly prescribe Ada, and the field of automation tends to require programs in FORTRAN or C. Such requirements arise when the client's interests extend beyond the finished product in compiled form and the client plans to do maintenance work and further development in an in-house software department. Then the education level of the client's programming team determines the implementation language.

4.2 Programming style

- After they have been implemented and tested, software systems can very seldom be used over a longer time without modifications. In fact, usually the opposite is true: as the requirements are updated or extended after completion of the product and during its operation, undetected errors or shortcomings arise. The implementation must constantly be modified or extended, necessitating repeated reading and understanding of the source code. In the ideal case the function of a program component should be comprehensible without knowledge of the design documents, only from its source code. The source code is the only document that always reflects the current state of the implementation.
- The readability of a program depends on the programming language used and on the programming style of the implementor. Writing readable programs is a creative process. The programming style of the implementor influences the readability of a program much more than the programming language used. A stylistically well-written



FORTRAN or COBOL program can be more readable than a poorly written Modula-2 or Smalltalk program.

Most important elements of good programming style:

- Structuredness
- Expressiveness
- Outward form
- Efficiency

This refers to both the *design* and the *implementation*.

Although efficiency is an important quality attribute, we do not deal with questions of efficiency here. Only when we understood a problem and its solution correctly does it make sense to examine efficiency.

4.2.1 Structuredness

- Decomposing a software system with the goal of mastering its complexity through abstraction and striving for comprehensibility (*Structuring in the large*.)
- Selecting appropriate program components in the algorithmic formulation of subsolutions (*Structuring in the small*.)

Structuring in the large

- ❖ Classes and methods for object-oriented decomposition
- ❖ Modules and procedures assigned to the modules.
- ❖ During implementation the components defined in the design must be realized in such a way that they are executable on a computer. The medium for this translation process is the programming language.
- ❖ For the implementation of a software system, it is important that the decomposition defined in the design be expressible in the programming language; i.e. that all components can be represented in the chosen language.

Structuring in the small

- ❖ The understanding and testing of algorithms require that the algorithms be easy to read.
- ❖ Many complexity problems ensue from the freedom taken in the use of GOTO statements, i.e., from the design of unlimited control flow structures.
- ❖ The fundamental ideal behind structured programming is to use only control flow structures with one input and one output in the formulation of algorithms. This leads to a correspondence between the static written form of an algorithm and its dynamic behavior. What the source code lists sequentially tends to be executed chronologically. This makes algorithms comprehensible and easier to verify, modify or extend.
- ❖ Every algorithm can be represented by a combination of the control elements sequence, branch and loop (all of which have one input and one output) ([Böhm 1996]).



- ❖ D-diagrams (named after Dijkstra): Diagrams for algorithm consisting of only elements sequence, branch and loop.

Note: *If we describe algorithms by a combination of these elements, no GOTO statement is necessary. However, programming without GOTO statements alone cannot guarantee structuredness. Choosing inappropriate program components produces poorly structured programs even if they contain no GOTO statements.*

4.2.2 Expressive power

- The implementation of software systems encompasses the naming of objects and the description of actions that manipulate these objects.
- The choice of names becomes particularly important in writing an algorithm.

Recommendation:

- Choose expressive names, even at the price of long identifiers. The writing effort pays off each time the program must be read, particular when it is to be corrected and extended long after its implementation. For local identifiers (where their declaration and use adjoin) shorted names suffice.
- If you use abbreviations, then use only ones that the reader of the program can understand without any explanation. Use abbreviations only in such a way as to be consistent with the context.
- Within a software system assign names in only one language (e.g. do not mix English and Vietnamese).
- Use upper and lower case to distinguish different kinds of identifiers (e.g., upper case first letter for data types, classes and modules; lower case for variables) and to make long names more readable (e.g. CheckInputValue).
- Use nouns for values, verbs for activities, and adjectives for conditions to make the meaning of identifiers clear (e.g., width, ReadKey and valid, respectively).
- Establish your own rules and follow them consistently.
- Good programming style also finds expression in the use of comments: they contribute to the readability of a program and are thus important program components. Correct commenting of programs is not easy and requires experience, creativity and the ability to express the message concisely and precisely.

The rules for writing comments:

- Every system component (every module and every class) should begin with a detailed comment that gives the reader information about several questions regarding the system component:
 - What does the component do?
 - How (in what context) is the component used?
 - Which specialized methods, etc. are use?
 - Who is the author?



- When was the component written?
- Which modifications has have been make?

Example:

```
/* FUZZY SET CLASS: FSET
```

```
   FSET.HPP    2.0,    5 Sept. 1996
```

```
   Lists operations on fuzzy sets
```

```
   Written by Nguyen Xuan Huy
```

```
*/
```

- Every procedure and method should be provided with a comment that describes its task (and possibly how it works). This applies particularly for the interface specifications.
- Explain the meaning of variables with a comment.
- Program components that are responsible for distinct subtasks should be labeled with comments.
- Statements that are difficult to understand (e.g. in tricky procedures or program components that exploit features of a specific computer) should be described in comments so that the reader can easily understand them.
- A software system should contain comments that are as few and as concise as possible, but as many adequately detailed comments as necessary.
- Ensure that program modifications not only affect declarations and statements but also are reflected in updated comments. Incorrect comments are worse than none at all.

Note: *These rules have deliberately been kept general because there are no rules that apply uniformly to all software systems and every application domain. Commenting software systems is an art, just like the design and implementation of software systems.*

4.2.3 Outward form

➤ *Beyond name selection and commenting, the readability of a software systems also depends on its outward form.*

Recommended rules for the outward form of programs:

- For every program component, the declarations (of data types, constants, variables, etc.) should be distinctly separated form the statement section.
- The declaration sections should have a uniform structure when possible, e.g. using the following sequence: constant, data types, classes and modules, methods and procedures.
- The interface description (parameter lists for method and procedures) should separate input, output and input/output parameters.
- Keep comments and source code distinctly separate.



- The program structure should be emphasized with indentation.

4.3 Portability and reuse

The objective of this section is to describe the problems which can arise in writing portable high-level language programs and suggest how non-portable parts of a program may be isolated. The section is also concerned with software reuse. The advantages and disadvantages of reuse are discussed and guidelines given as to how reusable abstract data types can be designed.

4.3.1 Software portability

([Brown 1977], [Tanenbaum *et al.* 1978], [Wallis 1982], [Nissen 1985]).

- Can be achieved by one machine on another using microcode, compiling a program into some abstract machine language then implementing that abstract machine on a variety of computers, and using preprocessors to translate from one dialect of a programming language to another.
- A characteristic of a portable program is that it is self-contained. The program should not rely on the existence of external agents to supply required functions.
- In practice, complete self-containment is almost impossible to achieve and the programmer intending to produce a portable program must compromise by isolating necessary references to the external environment. When that external environment is changed those dependent parts of the program can be identified and modified.
- Even when a standard, widely implemented, high-level language is used for programming, it is difficult to construct a program of any size without some machine dependencies. These dependencies arise because feature of the machine and its operating system. Even the character set available on different machines may not be identical, with the result that programs written using one character set must be edited to reflect the alternative character set.
- Portability problems that arise when a standard high-level language is used can be classified under two headings:
 - problems caused by language features influenced by the machine architecture, and
 - problems caused by operating system dependencies.

These problems can be minimized, however, by making use of abstract data types and by ensuring that real-world entities are always modelled using such types rather than inbuilt program types.

- The general approach which should be adopted in building a portable system is to isolate those parts of the system which depend on the machine architecture and the operating system in a portability interface. All operations which make use of non-portable characteristics should go through this interface.
- The portability interface should be a set of abstract data types or objects which encapsulate the non-portable features and which hide any representation characteristics from the client software. When the system is moved to some other hardware or operating system, only the portability interface need be rewritten to reimplement the software.



- The rate of change of computer hardware technology is so fast that computers become obsolete long before the programs that execute on these machines. It is therefore important that programs should be written so that they may be implemented under more than one computer operating system configuration. This is doubly important if a programming system is widely marketed as a product. The more machines on which a system is implemented, the greater the potential market for it.

4.3.2 Machine architecture dependencies

The principal machine architecture dependencies arise in programs because the programming language must rely on the conventions of information representation adopted by the host machine. Different machines have different word lengths, different character sets and different techniques for representing integer and real numbers.

The length of a computer word directly affects the range of integers available on that machine, the precision of real numbers and the number of characters which may be packed into a single word. It is extremely difficult to program in such a way that implicit dependencies on machine word lengths are avoided.

For example, say a program is intended to count instances of some occurrence where the maximum number of instances that might arise is 500 000. Assume instance counts are to be compared. If this program is implemented on a machine with a 32-bit word size, instance counts can be represented as integers and comparisons made directly. However, if the program is subsequently moved to a 16-bit machine, it will fail because the maximum possible positive integer which can be held in a 16-bit word is 32 767.

Such a situation poses a difficult problem for the programmer. If instance counts are not represented as integers but as real numbers or as character strings this introduces an unnecessary overhead in counting and comparisons on the 32-bit machine. The cost of this overhead must be traded off against the cost of the reprogramming required if the system is subsequently implemented on a 16-bit machine.

If portability considerations are paramount in such a situation and if a programming language which permits the use of user defined types is available, an alternative solution to this problem is possible. An abstract data type, say *CountType*, whose range encompasses the possible values of the instance counter should be defined. In Pascal on a 32-bit machine this might be represented as follows:

```
type CountType = 0..500000;
```

If the system is subsequently ported to a 16-bit machine, then *CountType* may be redefined:

```
type CountType = array [1..6] of char;
```

Instead of using an integer to hold the counter value, it may be held as an array of digits. Associated with this type must be the operations permitted on instance counters. These can be programmed in Pascal as functions.



The procedure letter must be rewritten when the program is transferred to a machine with an incompatible character set.

4.3.3 Operating system dependencies

- As well as machine architecture dependencies, the other major portability problem which arises in high-level language is dependencies on operating system facilities. Different machines support different operating systems. Although common facilities are provided, this is rarely in a compatible way.
- *Some de facto* operating system standards have emerged (MS-DOS for personal computers, UNIX for workstations, for example), but these are not supported by some major manufacturers. At the time of writing, their future as general standards is uncertain. Until standards are agreed, the problem of dependencies on operating system facilities will remain.
- Most operating systems provide some kind of library mechanism and, often, a set of routines which can be incorporated into user programs.

Standardized and installation libraries:

- (1) Standardized libraries of routines associated with a particular application or operating system. An example of such routines are the NAG library routines for numerical applications. These routines have a standard interface and exactly the same routines are available to all installations which subscribe to the library.
 - (2) Installation libraries which consist of routines submitted by users at a particular site. These routines rarely have a standard interface and they are not written in such a way that they may easily be ported from one installation to another.
- The re-use of existing software should be encouraged whenever possible as it reduces the amount of code which must be written, tested and documented. However, the use of subroutine libraries reduces the self-containedness of a program and hence may increase the difficulty of transferring that program from one installation to another.
 - If use is made of standard subroutine libraries such as the NAG library, this will not cause any portability problems if the program is moved to another installation where the library is available. On the other hand, if the library is not available, transportation of the program is likely to be almost impossible.
 - If use is made of local installation libraries, transporting the program either involves transporting the library with the program or supplementing the target system library to make it compatible with the host library. The user must trade off the productivity advantages of using libraries against the dependence on the external environment which this entails.
 - One of the principal functions of an operating system is to provide a file system. Program access to this is via primitive operations which allow the user to name, create, access, delete, protect and share files. There are no standards governing how these operations should be provided. Each operating system supports them in different ways.



- As high-level language systems must provide file facilities, they interface with the file system. Normally, the file system operations provided in the high-level language are synonymous with the operating system primitives. Therefore, the least portable parts of a program are often those operations which involve access to files.

File system incompatibilities:

1. The convention for naming files may differ from system to system. Some systems restrict the number of characters in a file name, other systems impose restrictions on exactly which characters can make up a file name, and yet others impose no restrictions whatsoever.
2. The file system structure may differ from system to system. Some file systems are hierarchically structured. Users may create their own directories and sub-directories. Other systems are restricted to a two-level structure where all files belonging to a particular user must reside in the same directory.
3. Different systems utilize different schemes for protecting files. Some systems involve passwords, other systems use explicit lists of who may access what, and yet others grant permission according to the attributes of the user.
4. Some systems attempt to classify files as data files, program files, binary files or as files associated with the application that created them. Other systems consider all files to be untyped files of characters.
5. Most systems restrict the user to a maximum number of files which may be in use at any one time. If this number is different on the host machine from that on the target machine, there may be problems in porting programs which have many files open at the same time.
6. There are a number of different file structuring mechanisms enforced by different systems. Systems such as UNIX support only character files whereas other systems consider files to be made up of logical records with many logical records packed into each physical block.
7. The random access primitives supported by different systems vary from one system to another. Some systems may not support random access, others allow random access to individual characters, and yet others only permit random access at the block level.

There is little that programmers can do to make file access over different systems compatible. They are stuck with a set of file system primitives and those parts of the system must be modified if the program is moved to another installation. To reduce the amount of work required, file access primitives should be isolated, whenever possible, in user defined procedures. For example, in UNIX, the mechanism to create a file involves calling a system function called `create` passing the file name and access permissions as parameters:

```
create ("myfile",0755)
```



creates a file called myfile with universal read and execute access and owner write access
In order to isolate this call, a synonymous user function which calls create can be included:

```
access_permissions := "rwxr_x_x"  
create_file ("myfile", access_permissions)
```

In UNIX, create_file would simply consist of a single call to the system routine create. On other systems, create_file could be rewritten to reflect the conventions of the system. The parameters to create_file could be translated into the appropriate form for that system.

- It might be imagined that the input/output facilities in a programming language would conceal the details of the operating system input/output routines. Input/output should therefore cause few problems when porting a system from one installation to another.

This is true to some extent. In some programming languages (FORTRAN and COBOL) input/output facilities are defined and each implementation of the language provides these facilities. In other languages, such as Pascal, the input/output facilities are poorly defined or inadequate. As a result, the implementors of a compiler 'extend' the I/O facilities to reflect the facilities provided by the operating system and this leads to portability problems because of incompatible extensions. In particular, the provision of interactive terminal support sometimes causes problems with Pascal as it was developed before interactive terminals were widely used.

- Different systems consider interactive terminals in different ways. In UNIX, a terminal is considered as a special file and file access primitives are used to access it. In other systems, terminals are considered to be devices distinct from files and special terminal access primitives are provided.
 - ❖ **Advantages in considering a terminal as a file:** Input and output to and from a program can come from either a terminal or a file on backing store.
 - ❖ **The disadvantage in considering a terminal as a file:** The characteristics of a terminal are not exactly those of a file. In fact, a terminal is really like two distinct files, an input file and an output file. If this is not taken into account, portability problems are likely to arise.
- Many systems are made up of a number of separate programs with job control language statements (in UNIX, these are called shell commands) used to coordinate the activities of these programs. Although it might be imagined that these would be relatively small programs, systems like UNIX encourage the use of job control languages as a programming language. Moderately large systems are sometimes written using this notation.
- There is no standardization of job control languages across different systems with the consequence that all job control programs must be rewritten when transferring a system from one installation to another. The difficulties involved in this are exacerbated by the vastly different facilities provided in different languages and may be further compounded by WIMP interfaces such as are used in the Apple Macintosh. Only a move towards a standard operating system will help resolve these problems.



4.3.4 Software reuse

([Horowitz 1984], [Prieto-Diaz 1987]).

- Costs should be reduced as the number of components that must be specified, designed and implemented in a software system is reduced.
- It is possible to reuse specifications and designs.
- Reusing the component might require modification to that component and this, conceivably, could cost as much as component development.
- Advantages of systematic software reuse:
 1. *System reliability is increased.* It can be argued that only actual operational use adequately tests components and reused components, which have been exercised in working systems, should be more reliable than components developed anew.
 2. *Overall risk is reduced.* If a component exists, there is less uncertainty in the costs of using that component than in the costs of developing it. This is an important factor for project management as it reduces the overall uncertainty in the project cost estimation.
 3. *Effective use can be made of specialists.* Instead of application specialists joining a project for a short time and often doing the same work with different projects, as proposed in [Brooks 1975], these specialists can develop reusable components which encapsulate their knowledge.
 4. *Organizational standards can be embodied in reusable components.* For example, say a number of applications present users with menus. Reusable components providing these menus mean that all applications present the same menu formats to users.
 5. *Software development time can be reduced.* It is often the case that bringing a system to market as early as possible is more important than overall development costs. Reusing components speeds up system production because both development and validation time should be reduced.
- While standard subroutine libraries have been very successful in a number of application domains in promoting reuse, systematic software reuse as a general software development technique is not commonly practiced.

Technical and managerial impediments to generalization of systematic software reuse:

1. We do not really know what are the critical component attributes which make a component reusable. We can guess that attributes such as environmental independence are essential but assessing the reusability of a component in different application domains is difficult.
2. Developing generalized components is more expensive than developing a component for a specific purpose. Thus, developing reusable components increases project costs and manager's main preoccupation is keeping project costs down. To develop reusable components requires an organizational policy



decision to increase short-term costs for possible, unquantifiable, long-term gain, and senior management is often reluctant to make such decisions.

3. Some software engineers are reluctant to accept the advantages of software reuse and prefer to write components afresh as they believe that they can improve on the reusable component. In most cases, this is probably true, but only at the expense of greater risk and higher costs.
 4. We do not have a means of classifying, cataloguing and retrieving software components. The critical attribute of a retrieval system is that component retrieval costs should be less than component development costs.
- The importance of systematic software reuse is now widely recognized and there is a great deal of research effort being expended on solving some of the problems of reuse.

In one application domain, however, reuse is now commonplace. This is in data processing systems where fourth-generation languages (Martin, 1985) are widely used. Broadly, many data processing applications involve abstracting information from a database, performing some relatively simply information processing and then producing reports using that information. This stereotypical structure was recognized and components to carry out these operations devised.

On top of these was produced some control language (there is a close analogy here with the UNIX shell) which allowed programs to be specified and a complete application to be generated.

This form of reuse is very effective but it has not spread beyond the data processing systems domain. The reason for this seems to be that similar stereotypical situations are less obvious in other domains and there seems to be less scope for an approach to reuse based on applications generators.

The initial notion of reuse centred around the reuse of subroutines. The problem with reusing subroutine components is that it is not practicable to embed environmental information in the component so that only functions whose values are not dependent on environmental factors may be reused.

Subroutines are not an adequate abstraction for many useful components such as a set of routines to manipulate queues and it is likely that software reuse will only become widespread when languages such as Ada with improved abstraction facilities become widely used. Indeed, it is probable that the most effectively reusable component is the abstract data type which provides operations on a particular class of object.

Given this assumption, what are useful guidelines to adopt when developing abstract data types for reuse? It can be assumed that abstract data types are independent entities which behave in the same way irrespective of the environment in which they are used (except for timing considerations, perhaps). Thus, we have to be concerned about providing a complete and consistent set of operations for each abstract type. Unfortunately, there is a large number of possible operations on each abstract type and there is no way in which the component developer can anticipate every possible use. Some broad guidelines as to what facilities should be provided are:



1. An operation should be included to create and initialize instances of the abstract type. Creation is sometimes accomplished simply by language declarations but the programmer should not rely on default initializations provided by the implementation language.
2. For each attribute of the abstract type, access and constructor functions should be provided. Access functions return attribute values; constructor functions allow attribute values to be changed.
3. Operations to print instances of the type and to read and write type instances to and from filestore should be provided.
4. Assignment and equality operations should be provided. This may involve defining equality in some arbitrary way which should be specified in the abstract type documentation.
5. For every possible exception condition which might occur in type operations, a test function should be provided which allows that condition to be checked for before initiating the operation. For example, if an operation on lists might fail if the list is empty, a function should be provided which allows the user to check if the list has any members.
6. If the abstract type is a composite type (that is, is made up of a collection of other objects), operations to add objects to and to delete objects from the collection should be provided. If the collection is ordered, multiple add and delete functions should be provided. For example, for a list type, operations should be available to add and delete an element to and from the front and the end of the list.
7. If the abstract type is a composite type, functions to provide information about attributes of the composition (such as size) should be provided.
8. If the abstract type is a composite type, an iterator should be provided which allows each element of the type to be visited. Iterators allow each component of a composite to be visited and evaluated without destroying the structure of the entity.
9. Wherever possible, abstract types should be parameterized using generic types. However, this is only possible when parameterization is supported by the implementation language.

4.4 Computer-aided software engineering

4.4.1 CASE workbenches

- CASE workbench systems are designed to support the analysis and design stages of the software process.
- These systems are oriented towards the support of graphical notations such as used in the various design methods. They are either intended for the support of a specific method, such as Structured Design, or support a range of diagram types which encompasses those used in the most common methods.

Typical components of a CASE workbench:



1. A *diagram editing system* that is used to create data-flow diagrams, structure charts, entity-relationship diagrams, etc. The editor is not just a simple drafting tool but is aware of the types of entities in the diagram. It captures information about these entities and saves this information in a central repository (sometimes called an encyclopaedia). The design editing system discussed in previous chapters is an example of such a tool.
2. *Design analysis and checking facilities* that process the design and report on errors and anomalies. As far as possible these are integrated with the editing system so that the user may be informed of errors during diagram creation.
3. *Query language facilities* that allow the user to browse the stored information and examine completed designs.
4. *Data dictionary facilities* that maintain information about named entities used in a system design.
5. *Report generation facilities* that take information from the central store and automatically generate system documentation.
6. *Form generation tools* that allow screen and document formats to be specified.
7. *Import/export facilities* that allow the interchange of information from the central repository with other development tools.
8. Some systems support *skeleton code generators* which generate code or code segments automatically from the design captured in the central store.

CASE workbench systems, like structured methods, have been mostly used in the development of data processing systems, but there is no reason why they cannot be used in the development of other classes of system. Chikofsky and Rubenstein ([Chikofsky 1988]) suggest that productivity improvements of up to 40% may be achieved with the use of such systems. They also suggest that, as well as these improvements, the quality of the developed systems is higher, with fewer errors and inconsistencies, and the developed systems are more appropriate to the user's needs.

Deficiencies in current CASE workbench products tools ([Martin 1988]):

1. The workbenches are not integrated with other document preparation tools such as word processors and desktop-publishing systems. Import/export facilities are usually confined to ASCII text.
2. There is a lack of standardization which makes information interchange across different workbenches difficult or impossible.
3. They lack facilities which allow a method to be tailored to a particular application or class of application. For example, it is not usually possible for users to override a built-in rule and replace it with their own.
4. The quality of the hard-copy documentation which is produced is often low. Martin observes that simply producing copies of screens is not good enough and that the requirements for paper documentation are distinct from those for screen documentation.



5. The diagramming facilities are slow to use so that even a moderately complex diagram can take several hours to input and arrange. He suggests that there is a need for automated diagramming and diagram arrangement given a textual input.

A more serious omission, from the point of view of large-scale software engineering, is the lack of support for configuration management provided by these systems. Large software systems are in use for many years and, in that time, are subject to many changes and exist in many versions.

Configuration management is probably the most critical management activity in the software engineering process. Configuration management tools allow individuals versions of a system to be retrieved, support system building from components, and maintain relationships between components, and their documentation.

The user of a CASE workbench is provided with support for aspects of the design process, but these are not automatically linked to other products such as source code, test data suites and user documentation. It is not usually possible to create multiple versions of a design and to track these, nor is there a straightforward way to interface the workbench systems with other configuration management tools.

The lack of data standards makes it difficult or impossible to transfer information in the central repository to other workbench systems. This means that users may be faced with the prospect of maintaining obsolete CASE workbenches and their supporting computers for many years in order to maintain systems developed using these workbenches. This problem has not yet manifested itself because of the relative newness of these systems but is likely to arise as new, second-generation CASE products come onto the market.

4.4.2 Text editing systems

- The function of a text editor is to enable the user to create and modify files kept on-line in the system, and most environments used for software development offer a number of different editors.
- Because this is such a common activity, the power of the editor contributes significantly to the productivity of the software engineer.

Although some editors, such as UNIX's *vi* editor, have facilities which are designed to support program preparation, most editors are general-purpose text preparation systems. This naturally means that it is possible for the user to prepare syntactically incorrect programs with these editors but it has the advantage that the same editing system may be used to produce any type of document. There is no need for the user to learn more than one editing system.

To support program preparation and editing, some work has been done in developing language-oriented editors (sometimes called structure editors) designed to prepare and modify programs in a specific programming language. An example of such a system is the Cornell Program Synthesizer, described by Teitelbaum and Reps ([Teitelbaum 1981]), which is intended to help beginners prepare programs written in a subset of PL/1 called PL/C. Such a system must include a PL/C syntax analyser as well as editing features. Rather than manipulating unstructured text, the system actually manipulates a tree structure representing the program. In fact, the Cornell system is more than just a



syntax-directed editor. It is a complete language-oriented environment with integrated editing, translation and program execution facilities.

Such systems are valuable for beginners wrestling with the idiosyncrasies of current programming languages. However, they do have limitations. Passing information about context-sensitive language constructs is difficult, particularly in large programs where the whole program is not visible on the screen. More fundamentally, perhaps, these systems do not recognize the fact that many experienced programmers work by laying out an (incorrect) program skeleton, then filling in that skeleton correcting inaccuracies and inconsistencies. Structure editors force a mode of working on the user which does not always conform to professional practice.

Some of the limitations of existing editing systems (particularly for program editing) are a result of display inadequacies where simple 80 by 25 character terminals are used. Such hardware forces documents or programs to be viewed sequentially. A sequential representation is hardly ever the most appropriate for programs where the reader wishes to look at and modify program parts which are logically rather than textually related. Furthermore, it may be appropriate to view and edit programs as diagrams rather than text. As bit-mapped workstations become the normal tool of the software engineer, new editing systems such as that described by Shneiderman *et al.* ([Shneiderman 1986]) will become commonly used.

Such systems use multiple windows to present different parts of the program and allow interaction in terms of program concepts. For example, one window might show the code of a Pascal procedure, with the program declarations displayed beside this in another window. Changing one part of the program (for example, modifying a procedure parameter list) results in related parts being immediately identified and presented for modification.

Such systems are impractical in development environments such as UNIX where the files are untyped. However, in an integrated development environment, stored entities are normally typed so that the editing system can examine what is being edited and present appropriate facilities to the user. Thus, as a user moves through a document containing text, programs, tables and diagrams, the editor identifies the working context and gives the user editing facilities geared to the type of object being edited. There is no need to prepare diagrams separately and paste them into documents or to move explicitly between program and text editors.

4.4.3 Language processing systems

- Language processing systems are used to convert programs to machine code.
- The provision of a helpful compilation system reduces the costs of program development by making program errors easier to find and by producing program listings which include information about program structure as seen by the compiler.
- The error diagnostic facilities of a compiler are partially dependent on the language being compiled. A Pascal compiler, for example, can detect many more errors than a FORTRAN compiler. Not only are the rules that govern the validity of a program more strict for Pascal than for FORTRAN, but the Pascal programmer must also supply more information to the compiler about the entities to be manipulated by the program. This information allows the compiler to detect forbidden operations on these entities.



As well as providing information to the programmer, a compilation system must also generate efficient machine code. This latter task involves a good deal of program analysis and can be very time consuming. This has the consequence that it is generally uneconomic to carry out this operation for anything apart from completely developed programs.

- A software engineering environment, might contain two compatible compilers for each language - a development compiler and an optimizing compiler.

Development compilers should be written to compile code as quickly as possible and to provide the maximum amount of diagnostic information to the programmer. Optimizing compilers, on the other hand, should be tailored to generate efficient machine code without considering compilation speed and diagnostic facilities. Programs are developed using the development system and, when complete, the optimizing system is used to produce the final version of the program for production use.

- Within the confines of the language being processed, development compilers should provide as much information as possible about the program being compiled:
 1. The compiler listing of the program should associate a line number with each program line.
 2. When a program syntax or semantic error is discovered, the compiler should indicate where it found the error and what the error appears to be. It may also be appropriate to indicate the possible cause of the error.
 3. The compiler should include directives which allow the programmer some control over the program listing generated by the compiler. These directives should allow the suppression of parts of the listing, control over the pagination of the listing, and the enhancement of program keywords by **bold** printing or underlining.
 4. When a program in a block-structured language is compiled, the compiler should indicate the lexical level at the beginning and the end of each block. This allows misplaced **begin/end** brackets to be easily identified.
 5. The compiler should separate source text provided by the user from information provided by the compiler. This can be accomplished by delimiting the input source using special characters such as '|', and prefacing compiler messages by some string of special characters such as '%%'.
 6. The compiler should identify where each procedure in a program starts and finishes. When a program listing is searched for a particular procedure, the location of that procedure is often not immediately obvious because the name of the procedure is not distinguished from the remainder of the source text. When compiling a procedure heading, the procedure name should be abstracted and, as well as listing the procedure heading normally, the procedure name should be reprinted so that it stands out from the rest of the program text.

These facilities were supported in the XPL compiler ([McKeeman 1970]), which was part of a compiler construction system. It is unfortunate that many of the display facilities provided by XPL have not been taken up by later compilers.

- If an environment supports both development and optimizing compilers, there is no need for the optimizing compiler to provide comprehensive diagnostic facilities. Rather, given that code optimization is a time-consuming business, the compiler may



allow the user to specify the degree of optimization to be carried out by the compiler or whether time or space considerations are most important. Ada has a specific construct called a pragma which, amongst other things, provides some facilities for the user to control compiler optimization.

4.4.4 Separate compilation

- Separate compilation of program units: Units may be compiled separately and subsequently integrated to form a complete program.
- The integration process is carried out by another software tool known as a linker or link editor. Without the facility of separate compilation, a programming language should not be considered as a viable language for software engineering.
- A large program may consist of several hundred thousand lines of source code and it may take hours or even days to compile the complete program. If every program unit needed to be recompiled each time any one of the units was changed, this would impose significant overhead and increase the costs of program development, debugging and maintenance. If separate compilation is available, compiling the whole system is unnecessary. Only modified units need be recompiled and relinked.

4.4.5 Compiler support tools

- The compilation process involves an analysis of the support text and, given this analysis, it is possible to provide additional information for the programmer and to lay out the program code, automatically, in a standard way. Commonly, these facilities are embedded in a compilation system. Other analysis tools, called static program analysers, are intended to detect anomalies in the source code.

Example of compiler support tool:

Program crossreferencer.

Such a tool provides a collated listing of the names used in the program, the types of the named objects, the line in the program where each name is declared, and the line numbers where a reference is made to that object. More sophisticated cross-referencers can also provide, for each procedure in the program, a list of the procedure parameters and their types, the procedure local variables and the global variables referenced in the procedure.

This latter facility is particularly useful to the programmer who must modify the value of some global variable. By examining the cross-reference listing, either manually or with an automatic tool, those procedures which reference that variable can be identified. They can be checked to ensure that global variable modification will not adversely affect their actions.

In addition to cross-reference systems, other source code analysis tools include layout programs (prettyprinters) which set out programs in some standard way. Prettyprinters are tools which incorporate standard layout conventions, and it is sometimes suggested that the availability of such tools makes disciplined layout on the part of the programmer redundant. Furthermore, if all listings are produced using a prettyprinter, the maintenance programmer will not be presented with the problem of getting used to different layout conventions.

The problem with most prettyprinting systems is that they incorporate a set of conventions which have been invented by the tool designer and which are rarely



explicitly specified. This means that, if an organization has existing standards, it is not possible to tailor prettyprinters to these standards. This may preclude the use of a prettyprinter and the programmers may revert to manual code layout.

All tools for static program analysis are language-oriented and must include a language syntax analyser. This has led to suggestions that the process of analysis normally carried out by the compiler should be factored out. Program analysis would be distinct from compiling and the analyser output would be processable by translation tools, editors, analysis tools, etc.

4.5 Incremental implementation

- The basic idea of incremental implementation is to blur the distinction between the design and implementation phases rather than to uphold the strict separation of phases that the classical sequential software life-cycle model requires.
- The recommendation of this approach is founded on the experience-based assumption that design and implementation decisions strongly affect one another, and thus a rigorous separation of design and implementation does not accomplish its goal. In many cases only the implementation can determine whether the decomposition structure of the design proves adequate for the problem.
- Incremental implementation means that after each design step that is relevant to the architecture, the current system architecture is verified on the basis of real cases. This means that the interplay of the system components specified in the design (in the form of interface specifications) is verified. To make this possible, the system components (i.e., their input/output behaviour) is simulated or realised as a prototype. Naturally the developer must devote thought to the implementation of individual components. The knowledge gained in the process, as far as possible, is translated directly into the implementation of the components, or documented for the subsequent implementation process. If there is any doubt concerning the feasibility of a component, then the design process is interrupted and these components are implemented. Only when their implementation and their embedding in the previous system architecture have been checked, is the design process continued, or the architecture is adapted according to the knowledge gained in the implementation of the component.

The efficiency of this approach depends on the extent to which it is possible to integrate the system components, which can be written in different formalisms and completed to varying degrees, to an overall system in order to carry out experiments close to reality. Some system components, e.g., the user interface or the data model, might be present in the form of prototypes; other components, which might stem from an existing component library or already exist as a complete implementation, are present in the form of executable code; still other system components might only be available as interface specifications. For the validation of the current system design, whenever the user interface is employed, the corresponding prototypes need to be activated. If a system component is available in compiled form, it needs to be executed directly. A system component for which only the interface specifications are available must be simulated. This is only possible with a software development environment that supports the integration and execution process for such *hybrid systems*. Describing such a development environment and its application scenarios would exceed the scope of this book. The interested reader can find both in [Bischofberger 1992].



References and selected reading

- [Bischofberger 1992] Bischofberger W., Pomberger G., *Prototyping-Oriented Software development*, Springer, 1992
- [Böhm 1996] Böhm D., Jacopini G., Flow Diagrams, Turing Machines and Languages with only two Formation Rules, *CACM*, Vol. 9, 1996
- [Brooks 1975] Brooks F.P., *The Mythical Man Month*, Reading Mass., Addison-Wesley, 1975
- [Brown 1977] Brown P.J. (ed.), *Software Portability*, Cambridge Univ. Press., 1977
- [Heller 1997] Heller P. and Roberts S., *Java 1.1 Developer's Handbook*, SYBEX, 1997
- [Horowitz 1984] Horowitz E. and Munsen J.B., An expansive view of Reusable software, *IEEE Trans. Software Eng.*, SE-10 (5), 1984
- [McKeeman 1970] McKeeman W.M., Horning J.J. and Wortman D., *A Compiler Generator*, Englewood Cliffs, NJ, Prentice Hall, 1970
- [Nissen 1985] Nissen J. and Wallis P.J.L. (eds.), *Portability and Style in Ada*, Cambridge Univ. Press, 1985
- [Prieto-Diaz 1987] Prieto-Diaz R. And Preeman P., Classifying software for reusability, *IEEE Software*, 4 (1), 1987
- [Shneiderman 1986] Shneiderman B., *Designing the User Interface*, Reading, Addison-Wesley, 1986
- [Tanenbaum 1978] Tanenbaum A. Specification, Klint P. and Bohm W., Guidelines for software portability, *Software – Practice and Experience*, 8, 1978
- [Teitelbaum 1981] Teitelbaum T. and Reps T., The Cornell Program Synthesiser: a syntax-directed programming environment, *CACM*, 24 (9), 1981
- [Wallis 1982] Wallis P.J.L., *Portable Programming*, London: Macmillan, 1982