International Institute for Applied Systems Analysis
IIASA www.iiasa.ac.at

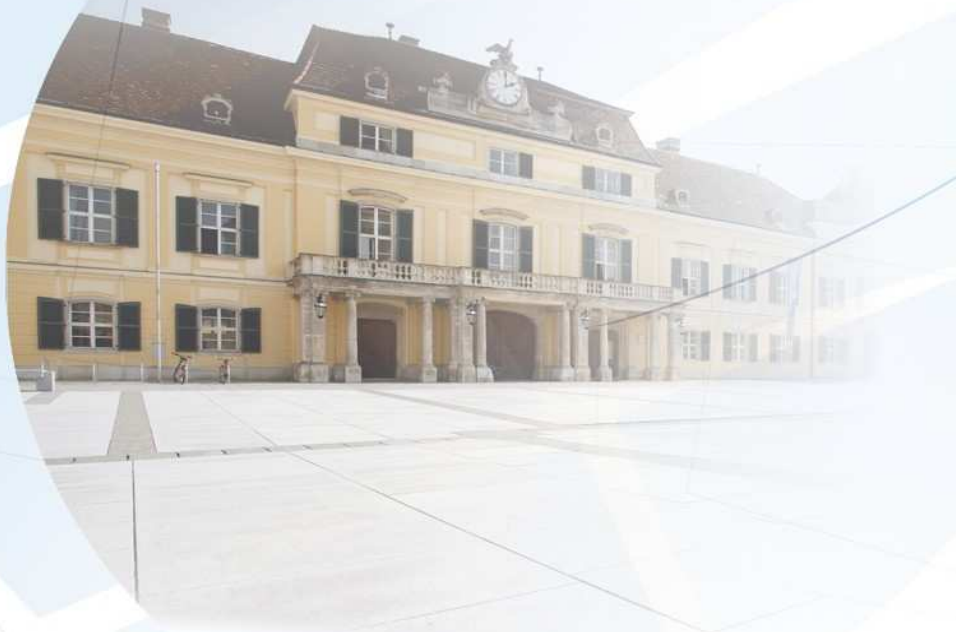
Guidelines for Software Development for Decision Support Systems

Makowski, M.

IIASA Working Paper

WP-91-015

May 1991



Working Paper

Guidelines for Software Development for Decision Support Systems

Marek Makowski

WP-91-15
May 1991

IIASA International Institute for Applied Systems Analysis □ A-2361 Laxenburg □ Austria
Telephone: +43 2236 715210 □ Telex: 079 137 iiasa a □ Telefax: +43 2236 71313

Guidelines for Software Development for Decision Support Systems

Marek Makowski

WP-91-15
May 1991

International Institute for Applied Systems Analysis □ A-2361 Laxenburg □ Austria
Telephone: +43 2236 715210 □ Telex: 079 137 iiasa a □ Telefax: +43 2236 71313

Foreword

One of the activities of the System and Decision Sciences Program is the collaboration with scientists and institutes from different countries and with other programs and projects at IIASA in the field of Methodology of Decision Analysis. The main results of these activities are the development of theory, methodology and software for Decision Support Systems. A considerable amount of experience and software has been accumulated for over ten years of this type of cooperation. However, the analysis of the results of software development activities indicates the need for setting and observing some specified *Guidelines for Software Development.*

This Working Paper contains a first version of such Guidelines which is the result of discussions with software developers. The draft of this Working Paper has been distributed for comments to about hundred scientists involved in development of software for Decision Support Systems. Since no reservations for the proposed Guidelines have been communicated to the author, one can assume that there is a general agreement for setting these Guidelines as a working scheme for the development of software for Decision Support Systems within the cooperation with the System and Decision Sciences Program. Although the Guidelines are oriented for a specific type of software, most of them can and should be observed in the development of any kind of software.

Alexander B. Kurzhanski
Chairman
System and Decision Sciences Program

Contents

Guidelines for Software Development for Decision Support Systems

Marek Makowski

1 Introduction

The Methodology of Decision Analysis Project has stimulated and organized the activities on development of the theory, methodology and software for the Decision Support Systems (DSS). Many researchers from many different institutes have been participating for several years in these activities. A considerable amount of both software packages and experience has been accumulated. However, it has been observed that a lot of these activities have been duplicated (because there is practically no way to use the developed software for an application which is being developed by another author) and the software is not robust enough and is difficult to modify (for example, for a new application or to provide an interface in languages other than English) and to be reused. Such a situation might have been justified in the early stages of DSS development in which software was used either for specific applications or for illustrating and experimenting with the developed theory and methodology of DSS. However, now it is necessary to agree upon the Guidelines for the Development of Software for DSS within the cooperation with the MDA Project.

This Working Paper is based on the Draft for such Guidelines distributed in March 1991 (further on referred to as the Draft) among about 100 software developers who work both with the Institutes collaborating with the MDA Project and at IIASA. This proposal was based on the discussion held in Warsaw on February 16, 1991 with the participants of the Contracted Study Agreement between the Polish Academy of Sciences and the SDS Program. It also contained the author's point of view on the development of software for DSS. The latter was obviously very subjective despite the fact that most of the proposed Guidelines are commonly considered as good practice in software enginering.

There was practically no controversy on the basic assumptions during the meeting in Warsaw. Comments about the Draft sent and communicated to the author have not suggested any major changes to the more detailed Guidelines formulated in the Draft. Therefore this Working Paper does not differ remarkably from the Draft distributed in March, 1991. However, a few items have been modified, mainly to provide clearer formulation and/or arguments.

The Guidelines should be treated as an integral part of those contracted study agreements with the MDA Project which include software development. The Guidelines will be updated when a considerable amount of experience prove it to be desired.

2 Purpose

A sufficient amount of experience has been gathered to consider measures which could result in a more efficient utilization of the programming effort. This Working Paper is not devoted to discussing the organization of possible cooperation between researchers from different countries, but is aimed at defining the necessary conditions to make such cooperation possible. Obviously an agreement on the Guidelines for writing software (aimed to be used in DSS) is a necessary condition for such cooperation. The author strongly believes that most of the following

Guidelines are commonly observed anyway because they result from a good practice of software development even in independent projects. Any additional restrictions could, however, be accepted only if the additional effort to conform to them will be "paid back". The side effect of discussing and observing the Guidelines should result in the dissemination of a better software engineering practice.

The purpose of setting the Guidelines are the following:

Software robustness and reuse : Software robustness is understood as its ability for functioning according to the specification and to the needs of the user. This also includes proper behaviour for possibly any data supplied for the program, any action of the user and for typical hardware problems. Software for DSS should obviously have a high level of robustness. The software robustness is also the necessary condition for the software reuse and justifies the careful design and testing of all software components. A large part of the software should be reusable both in different applications and by different teams without the necessity of any modification of the available software. This requires some additional programming effort, but it is enough to consider how much time is necessary to debug and test new software to justify this effort. This reusable software should be carefully designed, tested and converted into libraries which are well documented. The reuse of some specific software (e.g. solvers) requires an additional agreement on the common structure of data for a specific mathematical programming problems.

Software portability : The portability is an important issue, especially for the software developed within the DOS environment. In the past, DOS may have been considered as a good operating system[1] for prototyping and small scale applications, however, for many real-life applications more powerful operating systems (mainly Unix) are more appropriate. Unfortunately too many DSS software packages developed under DOS are hardly portable to any other operating system. Although DOS may be still useful for prototyping and small scale application non-portability should be limited as much as practical in future software development.

Software modification : Well-designed and implemented software should be easily modifiable for specific purposes. This includes e.g. modification of the user interface for a specific decision making problem or translation to another spoken language or extension for supplying additional options.

There are many historical reasons which make it difficult to apply the Guidelines to the existing software. The main problem is a result of two groups of issues. First, many software packages were initially designed for a limited number of functions and have "grown" along with the acquisition of additional features. Some of these packages which are considered useful for new applications should be most probably written again from scratch. However, it does not seem reasonable to apply the Guidelines to other software already existing which does not require any substantial modification or which is not aimed at further development. The second group of problems is caused by a vast variety of languages and tools used for software development. This applies especially to the usage of specific DOS tools that are applied in such a way that a port of a software to a different compiler or operating system often requires the rewriting of the entire software package.

Rare situations may exist which justify the violation of selected elements of the Guidelines. This is particularly related to the use of another programming language to a specific application. Therefore we shall foresee some exceptions. However, in general, the new software which is developed within the framework of cooperation with the MDA Project, should conform to the Guidelines and any deviations from the standards should be well justified. This also includes

[1] Mainly because, in the last few years, it has been one of the very few widely available operating systems. However, because of the recent change in the price structure for computer hardware, the Unix-based workstations are becoming more readily available and are likely to replace PC's for many DSS applications.

the existing software that is subject to any major revision. To avoid any *ex post* justifications it is assumed that any deviation from the Guidelines has to agreed upon with the MDA Leader before starting the new software or any major revision development.

It is strongly recommended that the Guidelines also be observed as much as possible for the software being developed independently from any formal agreement with the MDA Project. This seems to be not only a rational approach for development of a DSS software, but it is also the necessary condition for cooperation with a large research community that develops software useful for DSS.

3 Guidelines

3.1 Implementation

1. The general rule for software design and implementation gives the following order of priorities: first, the program should perform exactly the task it was designed for, second, it should be robust, third, it should be easily portable and modifiable, and fourth, it should be efficient. The first two criteria seem to be hardly questionable. Some specific applications may imply a more demanding specification of the task the software is designed for (e.g. speed or memory requirements).
2. Proper modularization of the software is critical for both efficient development and modification. Therefore the software should have a modular structure which consists of at least two levels. Top modularization should follow the general structure of a DSS (cf Section 4) with an additional module (hereafter referred to as the library) which is composed of functions and of header files used in more than one DSS module. The second level of modularization corresponds to separate programs and may be subdivided further, if practical. Each module[2] should be placed in a separate directory.
3. The *make* utility should be used for compilation and linkage (or creating libraries). A *"parent" Makefile* for the generation of *Makefile* files for each module (and for other functions like back-up, restore) should be placed in a parent (for directories that contain separate modules and a library) directory. Individual *Makefile* files should be designed in such a way that dependencies on global *include* files are accounted for.
4. It is recommended that two versions of all software being developed be created, namely *a development version* and *a production version.* However, for practical reasons, both versions should be in one set of sources files (cf the next item). The development version usually contains additional checking and diagnostics which are helpful for debugging and testing. Those additional functions should be suppressed (e.g. via conditional compilation of different macro's definitions or of parts of the function's body) for the production version of the software. The same mechanism can be used to differentiate the production version for users with different levels of experiences or needs (e.g. by enabling or disabling selected options or features).
5. Conditional compilation should be used for different environments, tools and compilers as well as for development and production versions of the program. Since the standard Fortran does not supply conditional compilation, the C preprocessor should be used for generating the respective version of the Fortran source file.
6. Each function should be placed in a separate file, whose root of the name[3] is the same as the function name. The corresponding header file should also have the same root of the name. There are two exceptions to this rule. First, small functions can be grouped together in one file. Second, if a short function is used only by one calling function, it may be placed together with the source of the calling function.
7. The code should be sufficiently commented upon. The comment at the beginning of each

[2]By a module we understand here a set of files that contains all sources for one executable program or for a library.

[3]The extension of the file name corresponds to a compiler requirements.

source file should contain the information required for each function from the Reference Manual (cf Section 3.5). Comments are the primary source of help for the management of software, therefore all information necessary to understand the operation of a function should be contained in comments with the source code. Unfortunately very few sources are properly commented upon. The following rules are recommended for comments:

- be as terse as possible while keeping in mind that a year later you will not remember any details,
- do not write obvious things,
- remember that what is obvious today may not be obvious a few months later.

8. The source code should be written using consistent indentation. The source code for C can be "cleaned" by the standard *indent* utility.

3.2 Programming language and tools

Most of the DSS software developed so far within the cooperation with the MDA Project was written under the DOS operating system and using three programming languages: C, Fortran and Turbo-Pascal. Very few programs conform to a standard definition of the respective language since most programs intensively use a compiler specific extensions and tools. Such an approach has allowed for rather fast software development but it has also one crucial drawback, namely that the software is hardly portable for another operating system and it can hardly be used with another compiler for the same programming language.

The Guidelines for the Programming Language and tools are proposed, taking the following arguments into consideration:

Environment and portability: Unix and DOS will most probably be the two main environments for development and applications of DSS with a rapidly growing share of Unix implementations. A DOS environment will be still used for both development and selected applications within the next few years, but most of the software should be easily portable to the Unix-based computer workstations. Most of the software should be ported to Unix workstations within the next one or two years. Additionally, software developed under Unix should also be portable to DOS.

Object Oriented Programming (OOP): The advantages of using the OOP are hardly questionable, therefore the arguments will be restricted just to few lines. The OOP allows for increasing both robustness, reuse and maintainability of software by the two new powerful programming approaches. The first is the extension to the traditional programming languages by providing mechanism for easy creation and usage the Abstract Data Types (ADT). An ADT consists of a set of data and the necessary methods to manipulate it. Implementation of the methods is encapsulated with the relevant data and a part of methods (the one that need not be used by functions that do not belong to a particular ADT) is hidden which makes it possible to modify them without changing another part of the code that uses instances of a particular ADT. The second is the mechanism of inheritance (which facilitate deriving new ADT from another ADT defined by a user), which promotes reusability.

Therefore, the OOP should be used as much as possible. Even if an OOP language is not yet being used for a project, the OOP techniques should be applied as far as practical within a procedural programming language. It should, however, be made clear that OOP requires a careful design of classes and therefore OOP is not recommended for a *quick and dirty* programming style.

Re-inventing the wheel : The existing programming tools, especially libraries which ease implementations of solvers and user interfaces should be used. However, it should be kept

in mind that the used tools might not exist in another environment or it may become reasonable to replace a tool by another one. Therefore it is advised to observe the rules specified below for the usage of programming tools and language extensions.

The following Guidelines regarding programming languages and tools for development of DSS software are proposed:

1. The main target programming language for DSS is C++. The chief reasons for this choice are twofold[4]. First, C++ is practically the only OOP language widely available for both a Unix and DOS environment. Second, it allows for a smooth transition from C, which is one of main languages used so far (therefore both a lot of software has been developed in C and many programmers have good experience in using this programming language) and is the best supported language for Unix-based computers. The other two OOP languages available for various environments are Smalltalk and Objective-C. However none of them is as popular as C++ and porting applications from C to either Smalltalk or Objective-C would involve major reworking of the code and for most programmers would require learning a new language.
2. In the transition period (i.e. the period needed for C++ to become more popular), C language is the second choice. Programming in C should, however, be done in such a way that it allows for the reuse of a possibly large part of the code in future C++ applications.
3. It may be reasonable to keep some parts of the solvers coded in Fortran, mainly because a lot of good and verified numerical codes exist in Fortran. Since writing a user interface in Fortran is the approach no one would recommend, the only way to use a Fortran code is to apply mixed-language programming with C (cf below).
4. For any of the recommended languages the corresponding standard of the language, i.e.:
   - AT&T's C++ version 2.0,
   - ANSI - Programming Language C, X3.159-1989,
   - ANSI - Programming Language Fortran X3.9-1978,

   should be applied. Compilers that do not support the above standards should be avoided.
5. Mixed-Language Programming may be a rational approach[5] in one of the following two situations. First, when a good and well-tested numerical code exists for a specific problem. Second, when deeply nested functions (performing floating point operations) are identified[6] to be faster when coded in Fortran. The following issues should be considered for mixed-languages programming:
   1. *Naming Convention,* i.e. the way a compiler alters the name of a function before placing it into an object file.
   2. *Calling Convention,* i.e. the language implementation realizes the function call.
   3. *Passing Parameters,* i.e. the communication protocol between two functions used for passing parameters and for returning a function value.
   4. *Consistency of data types,* i.e. the problem of the representation of respective data types and the way of accessing the global data.

   At least one easy way exists to cope with the above problems. Namely, it is possible to specify a set of header files (which makes the mixed-languages, at least for C and Fortran, easy) and

[4]The Turbo-Pascal is no longer recommended despite the fact that it used to be one of the programming languages for DSS software. The reasons for this are the following: there is no Turbo-Pascal for non-DOS environment; mixed language programming with Turbo-Pascal is complicated and inefficient (because Turbo-Pascal uses non-standard representation of floating point numbers); Turbo-Pascal has no substantial advantages in comparison with C++ or even with an ANSI C (some developers used to prefer Turbo-Pascal over old C compilers for features like prototyping or save linking; most of these features are now supplied by ANSI C).

[5]The author has had positive experiences with mixed-language programming with C and Fortran, however, he has no experience with mixing C++ and Fortran.

[6]For this situation an initial coding in C and running a profiler is recommended for the identification of such a part of the code. This case, however, will be probably less important when C compilers become more efficient for floating point operations.

to include those files in both C and Fortran code. Since the standard Fortran does not supply file inclusions (the various language extensions adopts different syntax for this purpose) it is recommended to run a C Preprocessor (either a stand alone preprocessor or running a C compiler as a preprocessor) to generate a Fortran code which conforms to the standard Fortran-77.

6. Use of both tools[7] and compiler (or environment) specific language extension could save a great amount of the software development time and contribute remarkably to the software robustness. Therefore the use of reliable tools is highly recommended. However, it should be done in a way that balances the obvious advantages with the inevitable disadvantages which are mainly due to the portability problems. The following Guidelines refer to the use of tools or of any extensions of a programming language:
   1. no extensions nor tools should be used if a standard implementation supplies a corresponding function,
   2. the number of different tools should be as small as possible and only well-tested tools should be used,
   3. language extensions and tools should be applied in such a way that their replacement (i.e. by tools more appropriate for another application or environment) is as easy as possible; this guideline is often contradictory to the efficiency requirements therefore it is assumed that the portability and ease of modification should be balanced with the efficiency of both the development and the running of the software; usage of macros or of intermediate functions is recommended instead of direct calls to functions not supplied by a standard compiler.
7. Compiler and installation dependent extensions of the programming language should be used only when really practical. Should any extension (of the respective language standard) be used, the approach recommended for tools (cf above) should be applied. However, neither compiler (or hardware) specific side effects (this also includes representations of integer and floating point numbers), nor so called "smart techniques" (e.g. usage of LOGICAL*1 instead of a CHARACTER variable), should be used.

3.3 Programming style

There is no uniform programming style that can be recommended for all applications and for all developers. One can simply suggest that the practice of good software engineering is to be observed. However, the analysis of different source codes and discussions with the software developers show that a proposal of the Programming Style Guidelines may be useful.

The following Guidelines are mainly based on the author's limited experience and knowledge, therefore comments and additions are still encouraged. The *Local Conventions* adopted by the ACA Project at IIASA were very helpful in writing the Guidelines and a few of them are just copies of rules from the *Local Conventions*.

3.3.1 General

1. Write and modify software by adding or modifying only one function at a time. Test each function separately and carefully, for all representative values, if possible. Take care of boundary values. Always check the return value of a function (or cast the function call to void, if you think you really do not need to check).
2. Each function should perform just one, clearly defined task. An encapsulation of data and methods (functions) that process the related data is one of the basic requirements and should be fulfilled even if no OOP language is used.

[7] This name *tools* covers various libraries that facilitate the development of the implementation of otherwise time-consuming parts of the program (like functions facilitating user interface, numerical or graphical libraries).

3. Program in the simplest possible way. State clearly what you mean. Use mnemonics as variable names and define constants whenever possible (i.e. for switches, loops).
4. Make your code read from top to bottom. Whenever possible, place the `return` at the end of a function. Use the `goto` statement sparingly (i.e only if necessary to make the code simpler).
5. Follow each decision as closely as possible with its associated action. A general rule: after you make a decision perform the related action, do not just go somewhere. Then you can see at a glance what each decision implies.
6. Do not re-invent the wheel: use the appropriate functions from the standard library and make use of the provided controls and operators (do not forget the bitwise operators) as well as the available and efficient programming techniques like singly and doubly-linked lists, stacks, queues, collections.
7. Do not patch a bad code, write it again from scratch. Make the program correct before you make it faster, keep it correct when you make it faster, and make it clear before you make it faster. Do not optimize the program at the expense of transparent programming (e.g. by introduction of temporary variables). Let your compiler do the simple optimization.
8. Replace any repetitive expressions by calls of a common function. Do not sacrifice clarity for small gains in efficiency. Use a profiler to determine the critical parts of your software (typically most of the execution time is spent in just a fraction of a code.) Usually, finding a better algorithm is the best way to improve a critical part of the code.
9. Program defensively, allow for every possible user action or data error. Take care of all problems that a casual user would not like to know about (like time limits, limits in storage space, making files back-up, preparation for and performing of recovery action in the case of an unexpected shutdown of the system, etc). Provide status information of any critical activity related to the program execution.
10. Apply strong function prototyping. Prototypes should be placed in a separate (for each module) header file and should be included in all functions. The file with prototypes should `#include` the library prototypes.
11. Implement customized functions for memory management. Memory allocation and relocation should always be done with checking the storage availability. Also check for a possible `NULL` pointer before calling the `free()` function and assign `NULL` for a pointer which points to a memory location that is freed.

3.3.2 Data

1. Choose a data representation that makes the program as simple as possible and let the data structure your program. Use structures for related data. Try to avoid multidimensional arrays. Minimize coupling between modules. Encapsulate data and functions that process those data.
2. As a general rule, avoid using global data. All global data, if any, should be declared in the header files and be included only in those functions that really need them. A separate module should define and initialize the global data. The recommended way to declare global variables is to use the `EXTERN` attribute which is defined as `extern` in all modules except the one in which the variables are defined (in the latter module the name `EXTERN` is defined as empty string).
3. Initialize all variables, preferably when they are defined. Do remember that static local variables are initialized only when the program is loaded.
4. Avoid hard-coding numbers. Use the `sizeof` operator to determine a number of elements – e.g. by defining the macro: `num_of(A) (sizeof(A) / sizeof(A[0]))` – of data defined in your code (very often you will modify the number of elements). Allocate memory for the number of data elements actually needed (instead of defining arrays with fixed "maximal" dimensions). Use mnemonic constants (via `#define` or `const` attribute) whenever you do

not find another way to avoid putting a number as a dimension or control parameter. The same applies to filenames and other names used in the program.
5. Constant strings (used in messages, menus, help, etc.) should be gathered together to facilitate (possibly by conditional compilation) modification of the user interface for different languages. Customized functions for generating menu and data forms and context sensitive help are recommended to ease modification of the user interface.
6. Make sure that data do not violate limits (both respective range and storage that is allocated for). Check the size of each data type and the range of the processed data. Utilize information in `limits.h` and `float.h`.
7. Set and check tolerances for floating point operations and different "zero" tolerances. Do not compare floating point numbers for equality.
8. Use casts if you mix different types of data to make your intentions for conversions explicit.
9. Use `const` attribute for any variable (also pointers) that is not supposed to be modified.

3.3.3 Input-output

1. Use your library functions to output all types of messages, which should preferably be of one of the following types: informative, error, fatal error, internal[8] error. This allows for the development of a consistent user interface and for easy modification for different environments. The parameters of the above functions (with the exception of internal error function) should correspond to parameters of the standard C function `printf()`.
2. Never use absolute pathnames. Define and use an environment variable if there is a reason to use a specified directory.
3. Check and properly set a file mode to minimize the probability of losing data. Determine the existence of a file to avoid overwriting data without confirmation.
4. Use binary files for communication between executable modules, if the respective modules run on the same computer (or on different computers with the same binary data representation). Processing of ASCII files is much slower, therefore ASCII data representation should be used only when necessary.
5. Reading and writing of communication files should be done by the same function. A pointer to the function performing either input or output should be set to the appropriate library function (`read()` or `write()`). The rest of the code for input and output data can remain the same. Such a function in the reading mode can also take care of allocation of appropriate amount of storage for actual size of data.
6. Sizes of data items (arrays, structures) should not be coded explicitly but should be computed or read from a communication file.
7. Always validate the input data. The assumption that only valid input is provided is the guaranteed source of many problems.
8. Terminate input by EOF or by a marker, use a counter, if possible, to double check that correct amount of data is processed. Identify bad input and provide information that eases the correction of input.

3.3.4 Hints

1. Remember that storage for the automatic variables is allocated from stack, therefore variables that require more memory should not be declared automatic. For such variables dynamic memory allocation should be implemented. To avoid too much overhead by a frequently called function use a static pointer and your customized function for memory allocation (usually the same function can also take care of memory reallocation).
2. Parenthesize to avoid ambiguity, to clarify the code and to make modifications less error prone. Use parenthesis also to avoid confusion due to the precedence of the various operators and possible errors while modifying indented (but not parenthesized) code.

[8] This corresponds to the situation that you think would "never" occur.

3. Be extremely careful with string operations, use the library functions for string operations, check dimensions, etc.
4. Consider the reassignment of pointers instead of copying vectors. Use the standard library functions (e.g. `memset`, `memcpy`, `memmove`) for copying instead of assignment loops.
5. Watch out for off-by-one errors[9].
6. Avoid arithmetic `if` (e.g. instead of `if (x-y < 0)` use `if (x < y)`).
7. Use `if-else` to emphasize that only one of two actions is to be performed.
8. Implement a macro, e.g. `#define IERR ierror(__FILE__, __LINE__)`,[10] and place it in all places of your code that should "never" be reached. Use the `assert` library function, if you need more information.
9. Remember that there is probably no bug-free complex software. Therefore test, test, and test again. This is difficult to accomplish because most of software development projects are often behind schedule. Therefore the importance of the proper scheduling of software development should not be underevaluated.

3.4 Miscellaneous

1. The program should compile without warnings for the highest level of warnings available for a compiler. Should there be a reason for accepting a warning, a relevant comment should be placed in the code. Running the *lint* utility is usually also helpful.
2. Each module should have a version `ID`, which should be checked in critical points (i.e. while reading communication binary files). Version `ID` should be changed at least whenever data structure is modified. Version `ID` should be stored also in the data communication files and it should be compared with the actual `ID` to avoid data and program inconsistency problems.
3. Implement interrupt handlers to deal properly with "unexpected" problems. As the minimum the floating point errors and user interrupts should be handled properly (i.e. by asking for confirmation of a user interrupt, providing information about a floating point operation errors and by clean exit). Apply the library function `setjmp()` and `longjump()` to retain control on the program. If you want to make sure of the values of variables after `longjump()` you should declare them to be `volatile`.
4. Consider the use of the hypertext techniques for implementing context sensitive help.
5. Command line arguments are recommended for more experienced users to execute a driver. They can also be used by drivers for executing modules in different modes via call of `exec` family library functions and also for passing back critical information from those modules to driver (i.e. in case of problem with writing communication file).

3.5 Documentation

All software should be adequately documented. The documentation is of two types: the first type is the properly commented code (cf Section 3.3), and the second kind is the hard copy type documentation which should be composed of[11]:

Methodological guide : The purpose of this document is to provide a sufficient understanding of the mathematical, methodological and theoretical foundations of the software. It should define the problem being solved by the program, describe the corresponding mathematical programming model, and document the implemented numerical methods and computational algorithms. All necessary references should be mentioned here.

[9] This type of error is illustrated by a question: how many fenceposters placed 5 feets apart does it take to support a fence 50 feets long ?

[10] where `ierror()` is your function that handles internal errors.

[11] The documentation structure has been proposed by A. Lewandowski in the unpublished draft. The following guidelines are based on this proposal but some of them have been modified.

User's manual : This documentation is directed towards a potential user of the software who is not interested in the methodological background. The user's manual should be composed of:

1. *An Executive Summary* which contains a general description of the purpose of the software, a functional and brief technical description (including hardware and software environment) and provide information on the available documentation and support. It should also clearly state to which software category (cf Section 5) this particular software belongs.
2. *A User Reference Manual* which describes all of the functions provided by the software. It should serve as a reference document for preparation of data, interpreting the results and performing all necessary interaction with the software.
3. *A User Training Manual* aimed at helping a new user to become familiar with the software. One or several tutorial examples are recommended (one is necessary) to at least illustrate the usage of the basic software options. A self-running demonstration version of the software is also recommended for providing a guided tour through the software options and features.

Short program description : This short (2-5 pages) documentation is aimed at providing basic information about the software for a potential user who is looking for a software package that fits his requirements. The short software description should be composed of:

1. *An Executive Summary* which contains a general description of the purpose of the software and an outline of the methodology. It should clearly state to which software category (cf Section 5) this particular software belongs.
2. *References to Applications* which provide an overview of applications of this particular software.
3. *Hardware and Software Requirements* which specify (minimal and recommended) hardware requirements and software environment.
4. *Availability and Maintenance* provides information of the conditions of the software availability and the details of software support and maintenance.

Installation and maintenance manual : This is directed to the individuals responsible for the installation and maintenance of the software. It also should provide information necessary to install the software (this should include possible modifications of the *Makefile* and information regarding required compilers and libraries, if the source code is provided) and to run the software. It also should provide information about diagnostics and error messages which can be dealt with by a computer services staff.

Reference manual : This documentation is provided for those who can modify the source code (or a specific part of it[12]). It should describe the overall program structure and logic and provide all necessary technical information of program control and data flows. For each function the following information should be provided:

1. The purpose of the function.
2. Functions that use this particular function.
3. A list of arguments and their usage. The arguments should be divided into three groups: input (those should be passed by value), output and input-output (the latter are those which values are both used and possibly modified by the function).
4. Specification of return value(s).
5. Usage and/or modifications of global variables by the function.
6. The relationship between the program mnemonics and the program variables.

The Reference manual is optional (but highly recommended) for transferable software, but is required for libraries offered for general use. Its preparation is greatly simplified, if comments contained in the code are sufficiently adequate.

[12] For example, to translate user interface into another language.

All of the hard copy documentation should be prepared using LaTeX[13], which is the widely used[14] text processing system on both DOS and Unix based installations. The modular structure for the documentation is also highly recommended and cross references should be made by using `\label` and `\ref` commands since this eases remarkably any corrections and updates.

4 DSS structure

The top level modularization (cf Section 3.1) of the software should preferably correspond to a DSS general structure. Each software module should correspond to a DSS component. For larger projects each such module can be divided further into smaller modules. There should be an additional module (library) which is composed of functions and headers used in different modules. The use of the library is strongly recommended since it improves remarkably the robustness of the software as well as eases the development of larger projects.

There are many approaches for the definition of a DSS structure, but the classical structure of a DSS proposed by Minch[15] probably serves as the best one for structuring the design and implementation of a DSS. Therefore, for the sake of setting the Guidelines, we assume that a DSS is composed of:

USER : The user is obviously not a part of a DSS, but without clear identification of the requirement analysis, even the best DSS could, at the most, satisfy the needs of the designers, but not the intended users and therefore will never actually be used in any real-life application. We will not discuss this component despite the fact that there is no more important, yet more neglected, element of DSS design than the proper identification of the user.

DMS - Dialog Management System : This is the module which is responsible for all communication between the user and a DSS. The complexity involved in the design and implementation of a DSS is often overwhelming for a typical user who is not trained in modelling and computer techniques. This complexity should be hidden from the user who should only understand those details of the DSS operation which are necessary to: first, identify which part of the decision making process is covered and supported by the DSS and second, what the capabilities and limitations of the DSS are. Both elements are necessary to give the decision maker confidence in results supplied by the DSS, and to make him comfortable with how well the DSS reflects reality and provides a useful analysis. Therefore the design and implementation of a user interface is usually the most critical part of the design of a DSS. The functions serving directly for the user interface should be placed in the library and should be used in all modules.

PPS - Problem Processing System : A PPS is the software module that is responsible for the proper coordination of cooperation of all the other software components. This is traditionally called a driver. The driver performs three main tasks:

- Processes requests generated by the DMS. This is self-explanatory, although efficient coordination of cooperation between often complicated software components is not easily designed and implemented.
- Provides status information of any activity within the DSS. This includes answering questions generated by the DMS and generating informative and error messages according to the current status of any related process. This is an easy task for a

[13] Cf. e.g. L. Lamport, A Document Preparation System LaTeX User's Guide & Reference Manual, Addison-Wesley.

[14] LaTeX is also a standard system for publishing IIASA's Working and Collaborative Papers.

[15] Cf Minch R. and J.R. Burn, Conceptual Design of Decision Support Systems Utilizing Management Science Models, IEEE Transactions on Systems, Man and Cybernetics, Vol. SMC-13, no. 4, pp. 549-557, 1983.

single task operating system[16], but its proper implementation is quite complicated on systems with server-client architectures.

- Works quitely and efficiently as a trouble shooter for all problems that a casual user would not like to know about (like time limits, limits in storage space, making file back-up, preparation for and performing of recovery action in the case of an unexpected shutdown of the system, etc).

MMS - Model Management System : The MMS is involved in all activities associated with the use of mathematical models in DSS, including the building, implementation, testing, validation, execution, maintenance and interfacing of models. For the model based, aspiration-led DSS, models are actually algorithms for solving mathematical programming models, but for other types of DSS models may do nothing more than translate or reformat data.

The separate part of the MMS (called solver) is dedicated to solving the related numerical problems. The solver is problem specific and its implementation requires the involvement of good specialists in numerical methods since the solver has to be both robust and efficient. For the aspiration-led DSS the conversion of the multiple criteria optimization problem to an equivalent single objective problem is made by a specialized preprocessor. The resulting single objective problem can be solved using one of the standard or a specialized mathematical programming techniques. However, it should be stressed that the solver must be robust and accept input as well as provide information in a way acceptable for the particular implementation of the DSS. This very often excludes usage of stand-alone commercial packages. The solution obtained is converted by a specialized postprocessor to its original, multiple objective formulation, providing the DMS with information about objectives, attainability of aspirations, feasibility, etc.

DBMS - Data Base Management System : The design and implementation of data bases is one of the best developed areas of software enginering. There is a large collection of commercial implementations of stand-alone data base systems that provide interfaces to other applications. There are also many software tools that make it easy for the implementation of a customized data base. Therefore the only problem is to properly choose a DBMS and link it with the DSS. One should stress however, that only original data should be handled by a DBMS, scaling data (in order to avoid numerical problems and for report purposes) should be performed by the MMS and the DMS modules, respectively.

5 DSS software categories

The following[17] are basic categories of software which are being developed within the CSA:

Experimental software : This is the software used to compute examples illustrating theoretical or methodological research. Experimental software cannot be treated as commonly distributed software, but it can be made available for testing. This software is distributed directly by the authors. IIASA will distribute only short program descriptions, if provided by the authors.

Pilot software : This is the software which is still under development and it is intented that this be upgraded to *transferable software*. The current versions are released for testing and comments. Usually these are programs designed to be general-purpose tools (e.g. solvers, modules for user interface). The software should be tested and documented well enough

[16] Although one should also take care of problems such as the clean handling of user interrupts or numerical exceptions.

[17] This classification is a slight modification of the taxonomy proposed by A. Lewandowski in the unpublished draft.

to make it reasonable to use. However, the user interface can be poor and some bugs and inconsistencies are admissible. The documentation should clearly mention all of the limitations so the user is not surprised if he detects problems during experimentation with this type of software. This software can also be distributed by IIASA.

Transferable software : This is the software that is on a sufficient level of development and is of enough general interest to be made publicly available for research and educational purposes. The software should be as bug-free as possible, sufficiently tested and verified. User interface should be carefully designed and implemented (this includes the context sensitive help). A high level of robustness and sufficient error detection must be implemented. The documentation should be composed of the methodological guide, user manual (with at least one tutorial example) and a short program description. The self-running demonstration version of the program is also strongly recommended.

Customized software : This is software having the development and documentation level that is required for the transferable software but which is customized for a specific application. The customization usually involves the user interface, tunning the solver and handling the data. It is worthwhile making such a software publicly available, if a good demonstration can illustrate its features that are of general interest and if such a software is easily modifiable for another application.

The other two categories of software (professional and commercial) are not discussed here since their development requires testing and verification procedures as well as documentation and support which cannot be provided in the near future by IIASA or by any of the cooperating institutions.

6 Instead of conclusion

The following text that was left on the blackboard by students in a Real-Time Systems course[18] illustrates another approach for software development:

6.1 How to program in C language

1. Use lots of global variables.
2. Give them cryptic names such as: X27, a_gcl, or Horace.
3. Put everything in one large .h file.
4. Implement the entire project at once.
5. Use macros and `#define`'s to emulate Pascal.
6. Assume that the compiler takes care of all the little details you didn't quite understand.

"It's 5:50 a.m., Do you know where your stack pointer is?"

6.2 How to debug a C program

1. If at all possible, don't. Let someone else do it.
2. Change majors.
3. Insert/remove blank lines at random spots, re-compile, and execute.
4. Throw holy water on the terminal.
5. Dial 911 and scream.
6. There is rumour that "printf" is useful, but this is probably unfounded.
7. Port everything to CP/M.
8. If it still doesn't work, re-write it in assembler. This won't fix the bug, but it will make sure no one else finds it and makes you look bad.

[18] Thanks are due to Lothar Winkelbauer who has provided this text.