A systematic approach to the development of system programs

by F. M. TRAPNELL

Qandac Associates Limited
Zug, Switzerland

INTRODUCTION

The brief history of the development of large programming systems shows a persistent inability to predict cost and time associated with it. For this reason I want to discuss some of the principles and practice which experience (either good or bad) has shown can yield a higher level of predictability. I do not pretend that these principles are always easy to apply or that they do not have to be interpreted to particular situations. However, I do believe that the principles are fundamental in that predictability can be guaranteed to increase if they are followed.

Rapidly changing system technology is causing rapid increases in the scale of functional complexity and sheer size of these systems which in turn has caused rapid increases in the cost of and in the time required to develop them. Thus, in any given undertaking there will continue to be very little precedent for the leaders in the technology to follow in developing successive systems. Given that there is little precedent to follow, it should come as no surprise that the main theme of this paper is to call for carefully detailed and explicit planning of all phases of the development project.

Specification

Of all the problems facing the developer the lack of an adequate systems specification is both the hardest to overcome and at the same time the one which, if not overcome, will cause the most grief. This specification is a statement of what the system will do under all conditions, how it will do it, how fast it will operate, and how large it will be. The choice of language in which the specification is written is important in that it must be understandable to a wide range of people: the programmers, the system engineers, the managers, the users, and so forth. This, combined with the need to be both explicit and detailed regarding all situations in which the program might find itself, makes it an extremely difficult document to produce. From a practical point of view, however, one can regard it as axiomatic that what is not contained in the specification will not be in the system when it is finally built.

Structure

An important factor in achieving an adequate specification is deciding on the structure of the specification, itself. Only a few people can have a thorough knowledge of a large system; yet it may be necessary to have a large number of people working on programming and testing various parts of the system in order to complete it on schedule. To accomplish this, the structure of an overall specification must be such that it can easily be broken into as many sub-specifications as are required. The process is similar to constructing a bill of materials explosion in planning a manufacturing process. The cleavage into sub-programs will depend upon considerations of modularity, how maintenance of the system is to be carried out, how important is it to have the ability to substitute alternate program modules, and the fact that there may be natural boundaries between functions of the system which make natural interfaces. In any event, and however this is done, the size of each specified module should be no bigger than that which one man can program and test during the allotted development cycle. Thus a module becomes the property of one individual, who must be held responsible for achieving its specification, including proper interfaces, proper use of system facilities, function, speed and size.

Cross system communications

With such a breakdown of specification, it becomes a major task to control the communication between these modules. Ideally a developer would like to ensure that these communications operate through system defined mechanisms. That is to say, every communication between modules would take place through macro instructions with symbolic parameters which are expanded into machine code and data either at assembly time or execute time or some of both. These expansions, and system tables, lists, rolls, or control blocks which they use, should be specified and controlled by the central specification control agency discussed earlier. Thus, they are part of the systems specification itself and they must not be changed by individual programmers.

One of the problems in achieving this ideal is that many of these high level communications facilities are, or seem to be, cumbersome and slow. Thus a strong case may be made for private communications between modules of the system where the two programmers in question understand in detail how their code works and therefore can use this internal knowledge to locally speed things up. I, myself, have succumbed to this kind of rationale when in the heat of development there did not seem to be any satisfactory alternatives. (This, incidentally, I attribute to not having done the kind of specification job that I described earlier.) On the other hand, I am satisfied that this internal communications approach is unsound in large systems and can in fact be disastrous if it is arrived at by private agreement without the overall knowledge of the specification control authority. The right way to deal with this performance problem is to define the communications problem accurately as part of the specification and then design the cross-system communication mechanisms so that they meet the performance required. If necessary the system developer should obtain additions or modifications to the hardware which help to solve these problems.

(Incidentally, one of the worst character defects of system programmers is that they will go beyond the point of reason to avoid demanding changes to hardware. Frequently this is due to their inability to discuss these problems intelligently with cost conscious engineers; but some part of it is due to the fact that they often regard it as their job to program around all hardware problems.)

One reason to control communications at the system level is to maintain knowledge of how communications ought to take place so that when system bugs occur they can be analysed more easily and systematically. A second reason is to provide the flexibility to change the communications paths, the communication media (e.g., Vector tables), as well as the location and identity of the source and destination modules without having to redesign and recode the system. This can pay dividends, not only as the system is changed and expanded but also during the testing phase when one can more easily put in 'scaffolding' programs to simulate the action of routines that have not yet been written.

Development of the specification

The development of an adequate specification is, of course, a continuing process. No one is smart enough to sit down cold to produce a completely adequate specification for a large, complex system any more than one could sit down cold to produce a completely adequate constitution to govern a state or nation. As the program develops, new knowledge and insight will be gained which will lead to the need to change the specification. On the other hand, the nearer in detail the initial specification is to the final one, the quicker the development and specification changing processes will converge to an adequate design.

I have rarely seen enough time and effort spent in achieving an adequate initial specification for a large system to optimise either the overall development time or the overall cost. Time can be wasted in over-designing certain phases of the system, and there is a strong temptation to feel that unresolved specification problems will somehow disappear in the course of implementation. Thus, an aggressive development manager may prematurely decide to stop designing and start programming. As the experienced high altitude aviator disciplines himself to recognize the insidious symptoms of the lack of sufficient oxygen, so the experienced developer should discipline himself to recognize the insidious symptoms of not having sufficient specification. In practice I would suggest the following:

The main inadequacy of specifications is that they are usually too narrow in scope. Typically, they cover in depth those parts of the system which the designers feel are elegant, sophisticated and new; often they either skim or avoid problems which the designers feel are not technically interesting. Hartman and Owens in their paper[1] in the proceedings of the 1967 FJCC indicate the sort of breadth required in a compiler specification. As they say, a special and frequently troublesome class of problems arises from a lack of consideration for user problems.

It is essential that the developer avoids becoming so engrossed in the detail of given aspects of the system that he does not achieve completeness in the breadth of conditions and situations for which the system is designed to operate. One way to achieve this is to arrive at a formal specification through an iterative process of successive specifications. At each stage:

a. some problems are solved and some design choices made; these in turn raise new problems
b. the designers attempt to be exhaustive in identifying and describing these outstanding problems to be solved in order to achieve an adequate design
c. they limit the extent to which they try to solve each problem.

Throughout this process the developers must concentrate on the question "Have we thought of all the situations and conditions which must be taken care of?" Until that quest is ended they must not be distracted by the fact that they may not have solutions in hand for all the problems encountered.

Figure 1 illustrates this process, using some suggested stages. The first stage is called External Requirements, which tells how the system appears to the outside world: what the users, operators, manager, etc., see. The second is a functional cleavage of the system: what functional block there will be, what each will do. The third is interfunction communication; this describes the linkages to programs, system control blocks, system tables, etc., in terms of their symbolic names, linkage parameters required, data formats of parameter lists, etc.; at this point internal conventions and standards will be established and a control function for these will be appointed. In the fourth stage, called program module specification, the system is carved into modules to be programmed: input/output/function specifications are written for each module. These I/O/F specifications list every input (entry point), every output with its destination (exit point, interrupt, external reference, etc.), the linkage parameters and formats associated with each, the function which the module performs and its normal place of residence (in core, or disc, etc.). The fifth and final stage will provide detailed flow charts and final estimates of size and performance for each module.

To get the system design right these stages will have to be gone through more than once! For this reason the path on the left of Figure 1 shows a return to stage one at the completion of each stage. This is meant to imply that before proceeding to the next stage in sequence, the designers should go back through all previous stages to formally make the corrections found necessary as a result of the last stage completed.

These changes should be restricted to what is necessary to proceed; but on the other hand they give the designers the opportunity to correct mistakes made early in design. The programming manager who tries to short circuit the process will either end up with a poor system or will come face to face with the old system design adage which says: "There never seems to be time to do it right the first time, but there always turns out to be time to do it over again."

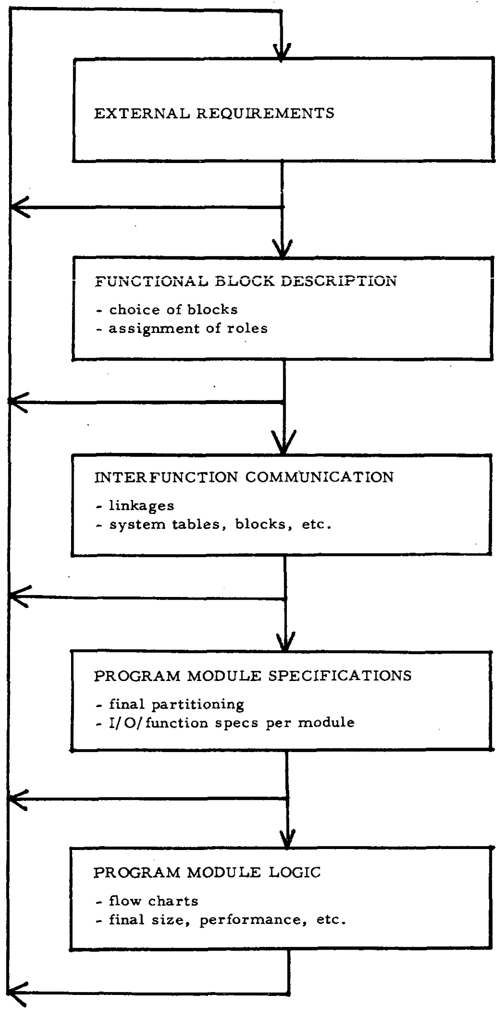
EXTERNAL REQUIREMENTS
FUNCTIONAL BLOCK DESCRIPTION
- choice of blocks
- assignment of roles
INTERFUNCTION COMMUNICATION
- linkages
- system tables, blocks, etc.
PROGRAM MODULE SPECIFICATIONS
- final partitioning
- I/O/function specs per module
PROGRAM MODULE LOGIC
- flow charts
- final size, performance, etc.

Figure 1—Specification sequence

The overall objective is to achieve a specification on which to base confident extimates of feasibility and development cost and time. The minimum requirements for such a specification are:

For the overall system:

1. A summary description of system facilities, how the system works, and the intent behind system standards as well as linkage and access requirements for system owned facilities.
2. A specification of all system owned facilities including services, tables, lists, etc., and the linkage or access requirements for each.
3. A specification of the logical flow between system modules.
4. System standards to be followed.

5. The file and directory structure for externally stored system information.
6. Overall size estimates for total system and core resident portion.
7. Execution time estimates for important paths, including I/O time.

For each module:

1. A brief summary functional description, size target, execution time estimate for principal paths.
2. Detailed specification of entry point linkage requirements, including all parameters required and the general format of these.
3. Detailed specification of exit linkages including destination, the parameters provided, and their general format. (Note: the latter can simply cross-reference a specified entry point in the destination).
4. Detailed specification of all system owned facilities used, including services, tables, list, etc., and the linkage thereto (as above the latter can simply cross-reference linkage requirements of these).

Machine processable specifications

A most useful tool in planning and controlling the changes to a large and complex system is a comprehensive where-used file for every module and communication facility in a system: that is to say, a file which for every such facility tells where to find every possible reference to it. Thus, when the designer plans a change to the system, he can tell immediately where the impact of the change will be felt. It is virtually impossible, however, to produce such a document unless plans are laid early in the design stage to generate the information required to make up this file.

This information should be contained in the specification for every module that is written. It should include every entry and exit point, as well as all the parameters that go to make up those linkages. It includes any references made to system facilities, blocks, tables, rolls, lists, etc.

These should be written on a formatted specification sheet which can then be put into machine processable form, and if desired, stored on disc or tape. A program can be written to scan this information to generate all cross-references and to test that these are consistent between the calling or referencing program and the facility which is called or referenced. Thus the system designers can have an early opportunity to spot cross-system communication faults in the specification.

This process can be carried a step further when actually coding such references if the programmer is required to use system controlled macros which, themselves, can be cross-referenced to the specifications. When the programs are coded and before they have actually been tested in the system, they can be scanned to compare macro cross-references with the specifications for the module to ensure that they are consistent. It is worth noting that these inter-module and cross-system errors are particularly difficult to debug; hence, the value of doing this sort of checking ahead of time should not be underestimated.

Implementation

Starting from a suitable specification, one should try to implement it by way of a systematic build and test process. This begins with programmers coding and testing modules with tests they write to satisfy themselves that the programs are working properly. These may be combined with other modules to form sub-functions which are in turn tested by tests written by the group responsible for the sub-function. However, because of the complexity of debugging a large system, it is essential at the early stages in system assembly to hand over these sub-functions to a central group, called *integration*; more about them later.

At this time these sub-functions will be tested by tests written, not by the programmers, but by a test production group. They are intended to verify that the sub-functions work at the level which is expected for incorporation into a system or sub-system which in turn is to be tested by system or sub-system tests written by this group. Once these sub-functions have been accepted they are combined with other functions and scaffolding, where appropriate, so that overall tests can be run.

Detected bugs are reported to the programmers who are given a copy of the system along with the test case or cases that failed, so that they can reproduce the problem.

The system against which overall tests are run also may be used as a master system by programmers working on functions that are not included (but perhaps scaffolded) in the master. When fixes have been found for the bugs in the master system at the current level (unfixed bugs may be converted to restrictions) and when the new function to be applied to the master system is available, then a new master system will be built using the old master, plus the fixes for its problems, plus the new function, see Figure 2. When this system has been built, it will be tested using a pre-specified newer and higher level of tests. The bugs thus found will be fixed or restricted, and it will become the new master system, available to programmers for the de-

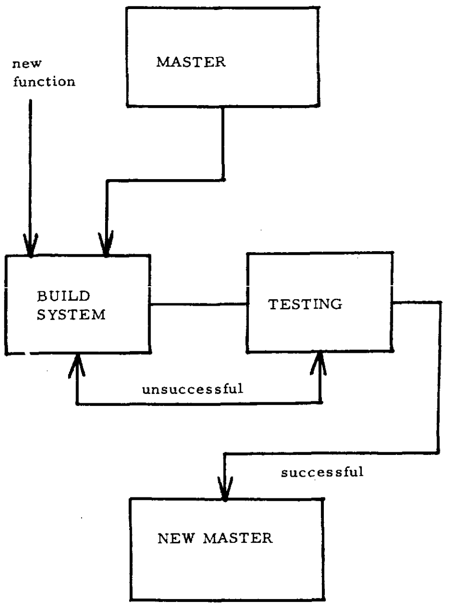

Figure 2—Build process

bugging of new functions. This process is repeated until all specified functions are incorporated and working in the system. Thus the systematic built process starts with a system at a given level, develops new function and fixes for the system at that level, applies these in order to create a new level of system which after test becomes the new master system.

Build plan

Given such an incremental build process one must be able to plan the points in time when given functions will be incorporated into the master system. For in most cases every new function has functional pre-requisites which must work properly. Thus, the build process requires a build plan which serves to synchronise all of the effort in the build process. This plan will specify the functional capability of the various system prototypes at every stage so that the programming groups will know what will be available and can thereby make their plans. Further, it will give the schedule on which these prototypes are to be achieved; it will specify when sets of test cases must be available to facilitate further debugging; and it will show a schedule of required hardware configurations to support the various levels of testing. Thus, the build plan describes the overall logistics and schedule of the build process. It is the central control document against which system build progress will be measured and in accordance with which all other system plans must be derived. As with drawing up the system specifications it will pay dividends for the system developer to take the care and time to lay out the build plan and then to make sure that everyone who is concerned with the build process understands it thoroughly.

Following the D-day landing at Normandy, General Eisenhower was asked whether all the planning that went into the invasion preparation was worth it, in view of the fact that practically nothing went according to the plans laid. His response is reported to have been that the plans themselves really did not prove very worthwhile, but that the planning process had been of infinite value. The same is true of building a large system. It is almost a certainty that there will be some unplanned chaos during the implementation period, but proper planning will prove invaluable in recovering from that chaos. It achieves the following:

1. Everybody associated with the process knows what are the critical factors and understands the sequence in which steps must take place. Thus, everyone is more able to assess immediately the effect of a change in plan on them and to re-align his plans quickly.
2. A widespread general understanding of the plan requires that all adopt common terminology and that meanings and implications of words will be thrashed out so as to facilitate much better communciations between everyone. Thus when changes in plan are discussed one can be more certain that everyone is talking about the same thing.
3. It permits those concerned to more readily identify the sensitive steps in a sequence so that contingency plans can be laid to back up the main thrust of planned operations.

Integration

Integration is the central organisation which carries out the build function in accordance with the build plans. They maintain programs in controlled libraries; from these they build systems in accordance with the build plan and cause these to be tested. They apply approved changes to the programs contained in the controlled libraries, and they provide systems at certified change levels to programming groups which require them in order to test new functions.

The integration libraries are the central store for the system; they service different groups of programmers who are testing and debugging various parts of the system in parallel. Throughout this period the attempt is made to maintain the rate of fixing bugs at the highest possible level, hence the change activities to these libraries is very high. Yet the changes applied to them must be stringently controlled to avoid regression, as discussed later.

Structure of libraries

One convenient way to structure integration libraries is to divide them into three sub-libraries, with different controls on each; one of them is the master library for the system at its current level; changes to it must be stringently controlled to avoid the application of changes which would either put the master copy at an unknown change level or would cause regression. The second library will contain a system or systems taken from a master library against which programmers may apply changes freely in order to determine proper fixes for bugs. The third, called the build library, is where integration is building the system at the next planned change level. That is to say, a copy of the current master library is being updated by carefully vetted and certified changes; when complete and tested it will become the new master system, and will be moved to the master library.

Regression

The scourge of this process is called regression, wherein: a fix is applied to a program, which may or may not fix the intended problem, but which produces problems or bugs elsewhere that did not exist before. Unless very careful control is kept over the changes applied and over the change level contained in a given library, regression is a virtual certainty. In a large system where change control procedures are not adequate it can hang like a malaise over the whole project. I have seen situations where on average for every bug fixed another one was created, so that over a period of time the net bug fixing progress was zero. The cure for regression is like the cure for rabies: it is painful, but it is guaranteed successful. Application of the following principles is required:

1. A single individual (in integration) must be designated as responsible for control of the libraries. He is responsible for setting up and operating all procedures for updating libraries and for authorising all changes to them. He should be made directly accountable for any regressions that occur. Practical problems such as working multiple shifts may require that he delegate responsibility for authorising changes to the library, but this should only be given to a few carefully chosen managers reporting directly to him.
2. Each bug to be fixed is described on a master list along with the manager responsible for providing the associated fix. The only fixes accepted as library updates are those which are on this list, when authorised by the appointed manager.
3. Any changes made to the build library should be the smallest modification which solves the problem which the change is intended to fix. Under no circumstances, short of absolute necessity, should changes be made to the library by replacement of whole modules or programs. Changes should instead be made on an add/delete basis; the libraries themselves, the programs contained in them and the programs which perform the updating must be designed so as to facilitate this. The integration manager authorising a change should have certification from the programming manager submitting it that it fixes a given identified problem and that it does nothing else.
4. Where possible, the integration manager authorising the change should review the code himself and be personally satisfied as to the extent of the modification. If he is in any doubt he must ask that the programming manager submitting the change review it with him before he agrees to accept it.
5. No other avenues for changes to the build library may be permitted.

The control procedures I have outlined are an attempt to prevent the application to the master libraries of unknown and often unwanted changes. These can arise through failure to keep an accurate accounting of the nature and scope of each applied change; through misunderstanding between the programmers and the integration people as to the functional level of the system in the library (and hence what changes are appropriate); and last, but not least, from purely altruistic motives of programmers who want to make improvements to the system that are not required to meet specification. Throughout the build process one must constantly strive to identify those problems which must be fixed, to fix them, and to avoid making other unnecessary changes.

The problem of controlling information in a system library is an order of magnitude worse than the problem of controlling accounts in a large bank. In the bank one has to keep track of how much money there is in each of a relatively few numbers of accounts and cash drawers. In controlling a system library one has to keep track of the value in a very large number of bit positions. This is rather like asking the U.S. Treasury to keep track of where every dollar bill is by serial number.

While the banks long ago adopted a very rigid accounting system to solve their relatively minor problem, I have seen installations where the whole control process was carried out on the back of an envelope. It is not normally essential for large system programmers to adopt the rigidity of banking procedures because the cost of error is not so high; I am suggesting, however, that the prudent application of a few strict controls can save time, toil, tears and money in the development of systems programs.

Testing

It is axiomatic that the quality of a program is no better than the quality of the tests that have been successfully run against it. Experience shows that those functions which have been tested and are known to work, will work; and those which have not been tested most frequently do not. It will be illustrative for the reader to recall how seldom he has taken what he thought to be a well established and fully exercised program and was successful the first time in running it in a new environment. The design of the tests and the laying out of the sequence in which they are to be applied to the various functional levels of the system is a most important part of the build process.

In designing a test procedure it is important to set aside the notion that the objective of testing is to achieve success. The real problem for the systems tester is to identify and isolate as many as possible of the most important bugs in the shortest time. It is most useful to regard debugging as a process in which one is trying by experiment to identify the bugs in a system. Just as in scientific experiments, the aim must be to derive the maximum amount of information for the least cost and time.

Design of tests

In testing a newly coded program, one would like to have tests that are function-specific; that is to say, they exercise only one funtion at a time in a way that does not call on other facilities and programs. On the other hand, the successful running of such a set of cases is not indicative of the true state of the program in its operating environment; since in normal operation of a program one function calls another. The most difficult bugs to find are usually those which result from interactions between modules or segments of programs when the individual functions seem to work but the combination of them does not perform as specified.

Therefore, the set of function-specific cases is not enough. Tests must be written which exercise cross-system capability as well. Ideally, one would like to have such a set of tests for every level of interaction in the system, but the economics of this are normally overwhelming. Thus a compromise is required to limit the size of the tests. Test planning and design is a most complex subject and one worthy of discussion separately. In this paper I only want to mention some straightforward but nevertheless important principles in systematic testing.

The test designer will work out the design of tests and the sequence in which they are to be applied with two criteria in mind:

1. In the event of failure of one or more tests then he wants to gain the maximum information from the test failure pattern
2. In the final stages of testing, it is necessary to apply tests which exercise the system in a comprehensive way. Prior to that, however, the test designer will want tests which are as independent of common functions as possible consistent with the need to test increasing functional interaction as testing progresses.

Thus, the first tests applied to a system each will separately exercise perhaps only one critical functional element of the system. These tests are used to check that these functions are actually in the system and working properly in a limited environment. When these are successful, then multi-function tests can be applied which aim to test both functional elements and their interaction in a restricted environment. Finally, full functional testing can be undertaken. When planning and designing these tests and their application sequence, however, the test designer should proceed in reverse order:

1. He should begin by deciding what will constitute the ultimate system test.
2. He will then work backwards to establish the multi-function tests required.
3. Finally, he can plan the functional element tests required to begin with.

Just as reproduceability is an essential characteristic of a valid scientific experiment, so an essential characteristic of a successful testing and debugging is the ability to reproduce reliably the occurrence of bugs in the programs under test. This can pose problems in interactive systems where the sequence of events depends upon the unpredictable sequence or occurrence of events in the outside world; thus careful thought is required to achieve reproduceability in these test situations.

Regarding the amount of effort one should put into test design and development, I would suggest that the proper level of effort in independent test development should be something between 15 percent and 20 percent of the system development effort, not counting early test writing by programmers to get their programs off the ground. Even in a system developed with a relatively small effort by a few people, it is important to have independently developed tests in order to ensure that the written specifications for the operation of the program .can be properly understood by someone besides the people directly involved in writing it.

System tests must be considered an integral part of the system; the experienced system developer in laying out plans for getting the system into operation will not distinguish between the need to get the system programs working and the need to get the tests working properly. The system will be no better than the tests applied to it; hence functional failure of the test should be treated in the same way as functional failure in the system.

The test cases must be held in libraries under the same controls as the system itself, so that once tests are running properly, their integrity can be guaranteed.

CONCLUSION

In the foregoing I have outlined some of the basic principles that I feel are important in the designing amd implementing of large systems. The principles I have discussed are only the beginning; I believe there is broad scope for innovation in management and control techniques which aid large system development. But I believe that use of these techniques can free designers and programmers from the severe strain of control by pure discipline. This will not only make the development process more predictable, but will make it faster, cheaper and more rewarding for those involved.

REFERENCE

1 P H HARTMAN D H OWENS
How to write software specifications
Proc F J C C 1967
Hartman and Owens in this paper discuss many facets of compiler specifications:
* User orientation
* Performance, size
* Optimization
* Debugging
* Statement of intent
* Acceptance tests
* Specification maintenance