

**Good Practice Guidelines**

for

**Model Building &  
Programming**

in the

**Design of Process Plant**

Program development is not a trivial job and to do it well requires special skills and experience. Engineering decisions will be based upon the results generated by these programs. The program must therefore work correctly and proper records must always be kept.

The adoption of good practice from the beginning (and the beginning is especially important) will not only be consistent with your duty of professional care but will also save you time, effort and problems.

*These Guidelines will continue to evolve and develop as our understanding of the issues and our experience of using existing and new tools and techniques develops. The Working Party therefore welcomes and encourages feedback from readers, both in general and on specific items (as noted in the text), regarding ways to make these Guidelines both better and more widely applicable.*

# Contents

Preface	3
Scope	4
Copying	5
Legal & Professional Implications	5
Feedback & Comments	5
<b>Guidelines for Model-Building and Programming</b>	<b>6</b>
Responsibilities	6
A Methodical Approach	7
Quality & Project Management	
Objectives & Detailed Requirements	
Detailed Design	
Build & Test	
Validation	
Documentation	
Training, Support & Maintenance	
Some Critical Issues	9
<b>Appendices:</b>	
Delivering a Better Program More Quickly	13
Legal & Professional Aspects	35

# Preface

Computers are now indispensable in the design and operation of process plant. The great benefits provided by today's computers to undertake extensive calculations bring with them the need to recognise that they are also capable of delivering wrong answers to high degrees of precision if care is not exercised.

The detection of such errors becomes correspondingly more difficult as the extent of computerized activity grows and the complexity of programs increases. Nevertheless, Chemical Engineers are subject to the provisions of the law, such as the Health and Safety at Work Act 1974, and must pay due attention to the implications of the decisions they make, whether or not they are based on the results of computer calculations.

The CAPE Subject Group of the IChemE therefore decided that the time had come to bring this guide up to date. The Working Party which produced the revised guide has attempted to condense more than 150 manyears of their own collective experience of computing in process design along with that of the numerous other contributors.

We hope that it will receive the widest possible dissemination and, moreover, by making this experience available, that at least some of the disasters that might have occurred may be averted.

Readers should note that we do not regard this as in any way a final edition and we welcome and look forward to feedback for use in the next edition.

**Dr Rob Best, South Bank University**  
**Chairman, Computer-Aided Process Engineering Subject Group**

## Scope

These guidelines contain suggestions for good practice to those engineers who become engaged either in model development or the preparation of computer software for use in the design of process plant.

The principal feature of such tools is that they are used in a decision support environment; computer tools can be used to provide information or even advice but, in all cases, a qualified engineer makes and is ultimately responsible for all design decisions.

These guidelines **do not address** models or software for such areas as:

- computer-aided draughting and the three-dimensional visualisation of plant and pipework layout, for which the reader is referred, for example, to the British Standards Institute, The Institution of Mechanical Engineers, etc.
- online or embedded systems, for which the reader is referred, for example, to “Safety Related Systems - Guidance for Engineers”, The Hazards Forum, London, 1995, ISBN 0-9525-1030-8 or to a number of publications of the Institution of Electrical Engineers.

*Readers should note that these guidelines are in no way intended to modify or replace engineers' responsibility under the appropriate legislation* (see below): these guidelines must be treated as suggestions and in the spirit of “necessary but not necessarily sufficient”. *The working party accepts no liability whatsoever for the use which may be made of them.*

## Note

This document is an extract from:

### **Good Practice Guidelines**

### **The Use of Computers by Chemical Engineers**

Guidelines for practising engineers, engineering management, software developers and teachers of chemical engineering in the use of computer software in the design of process plant

A copy of the complete document may be downloaded free of charge from either of:

**<http://CAPE.icheme.org>  
<http://CAPENET.chemeng.ucl.ac.uk>**

# Copying

The Working Party intends that these Guidelines should have the widest possible circulation amongst practising engineers and we hope that the style of presentation will allow sections to be copied for use in documents used in training and for display above the desks of engineers and managers; we ask only that the source is acknowledged, the copyright notice is not removed and that, unless by prior consultation, they are reproduced without alteration. Companies and/or HEIs are welcome to incorporate these guidelines into their own procedures, again, subject to acknowledgement, etc.

## Note, however:

- The materials in these Guidelines are copyright and reproduction in any form for the purposes of commercial gain is expressly forbidden
- These Guidelines will be updated from time to time and it is the *sole responsibility* of anyone making a copy to ensure that their copy is kept up to date by reference to the most recent public version.

# Legal & Professional Implications

Within the UK the work of the chemical engineer is subject to the provisions of various Acts of Parliament, including the Health and Safety at Work Act 1974.

This Act has important consequences for the way we work, laying down a number of duties for employers and employees and making it a criminal offence to fail to discharge those duties. For an overview of the way in which the Act and other aspects of the law may impact upon the work of the chemical engineer, see Appendix A6.

Similar or equivalent legislation operates in other Countries and the implications are the same: you, the professional engineer, are responsible for all decisions which you make, whether or not a computer is involved.

Attention is also drawn to the Rules of Professional Conduct of the UK Institution of Chemical Engineers, an extract from which is included within Appendix A6.

# Feedback & Comments

The Working Party welcomes and encourages feedback from readers, both in general and on specific items, regarding ways to make these Guidelines both better and more widely applicable.

Such feedback should be sent in the first instance to either:

CAPE Subject Group Officer  
IChemE  
165-189 Railway Terrace  
Rugby  
Warwickshire, CV21 3HQ

or:

The document editor: Tony Perris (Tony.Perris@btinternet.com or t.perris@ucl.ac.uk)

# Guidelines

## for

# Model-Building & Programming

Sooner or later, a case will be encountered where the existing/standard tools (ie. those available either within the organisation or from vendors) are not adequate and where some model-building and/or programming will be required. Common examples include modelling a novel reactor or tray design, adding a model to a flowsheet simulator, developing a special-purpose thermophysical property model and so on.

The purpose of this section is to provide guidance in the preparation of programs to represent such models. The special cases of using spreadsheet programs, neural networks or knowledge-based systems are covered in the section “Guidelines for Engineers Using Software” and the use of so-called equation-oriented simulators will require the careful interpretation of both that section and the following.

*The essential point to be born in mind is that engineering decisions will be based upon the results generated by these programs. The program must work correctly and proper records must always be kept (which will become a part of the audit trail).*

## Responsibilities

In setting out to prepare such programs, you are taking on additional responsibilities. In some ways these are more onerous than those involved in design activities: it may not be “intuitively obvious” when something is wrong (such as the apocryphal column two feet high by three hundred feet wide!) and, of course, your program may be widely used by other engineers engaged in other projects, so the consequences of error may cause widespread damage to the organisation. In short, **you've got to get it right and you should consider such a development only as a last resort!**

*Program development is not a trivial job and to do it well requires special skills and experience.* Thus, before you even begin, you must ask yourself a number of questions:

- is there really **nothing** available to do the job (or at least something adequately close to it)?
- do **you** have the necessary skills & training? (If not, do not even start until you have acquired them - learning on the job from scratch is dangerously unprofessional.)
- is it cost-effective for **you** to develop it, or should you define it and then get someone else (who has more specialist skills) to do it?
- do you have your manager's support and approval for the planned development? (**It is not a “spare time job” - if you treat it this way, then you will get a “spare time result”.**)

You will also need your support services' help, encouragement and support sooner or later, and they may need to take it over as a company standard for longer term and more widespread use.

These Guidelines present some suggestions for a methodical and professional approach to engineering program development and draw attention to key aspects of what is involved. The objective is to help you get it right and avoid too many of the common problem areas.

The adoption of good practice from the beginning (and, as you will see, the beginning is especially important) will not only be consistent with your duty of professional care but will also save you time, effort and problems.

***These Guidelines are not intended to replace a proper grounding in the subject. They are not a programming textbook or manual: they are necessary but by no means sufficient.***

Such activities are Software or Systems Engineering: there is a growing literature in these technologies and it is a part of your professional approach and responsibilities to go and read at least some of it. A short bibliography is provided in Appendix A5 but this is a fast-moving field and you should be prepared to seek more up-to-date material - the basic principles will not change but the details will during the expected life of these Guidelines.

The remainder of this section presents a methodical approach to the program development process and lays out some general principles. Detailed material with specific guidance, checklists, etc, is presented in Appendix A5.

## **A Methodical Approach**

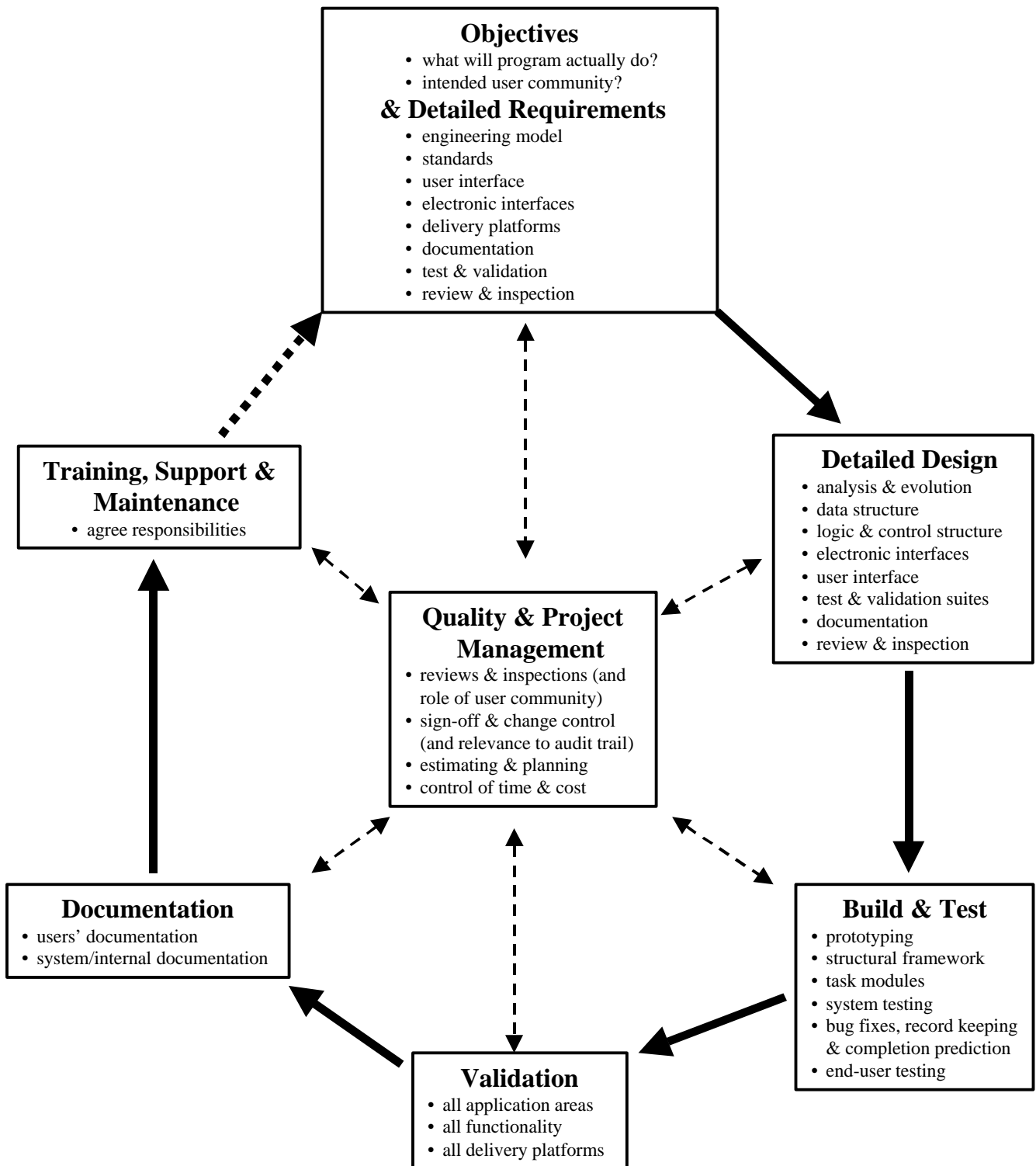
Writing a program is a project, just like an engineering project, and should be handled in broadly the same way. The key stages are illustrated in the diagram below and are described in more detail in Appendix A5:

- define objectives and end-user requirements
- design program to satisfy requirements
- build program and test that it works correctly
- validate program to ensure that it correctly represents the physical system
- deliver program with full supporting documentation
- provide necessary training and support

***Each of these stages should be thoroughly documented.***

There is a clear analogy to the approach typically used for an engineering project, including the need for proper records to be kept.

(... continued)



*It cannot be over-emphasised that, just like an engineering project, the earliest stages (ie. the specification and the design) are the most important: you wouldn't dream of building plant without them, would you? If you take care to get these right, then your life will be appreciably easier but, if you get them wrong, then a great deal of time and effort will almost certainly be wasted and serious engineering errors may result in subsequent project work.*

A variety of methods is available for assisting with the specification, development and design of software and with the management of software projects (see Appendix A5) and the use of such software engineering techniques is recommended for larger projects. It is important to remember, however, that sophisticated methods are **not** any sort of substitute for **thinking and taking care**, just as you would if you were designing a process. (Your chemical engineering training will help you quite a lot. Many of the concepts of flowsheets and modular unit operations can readily be transferred to software development and this analogy will be used extensively in these Guidelines.)

## Some Critical Issues

It is your duty to **exercise professional care**, just as it is when you take on any engineering task.

Above all, the end product (ie. the program) must **enable end-users to do their job professionally**, as described in their Guidelines. If it doesn't, then you have failed to discharge the responsibilities which you assumed when you took on the program development task

### User Community

If your program is at all useful, someone else will quite reasonably want to use it, so the user community will **not** just be you. Try to identify your potential end-users and then **involve this user community in specifying and evaluating the proposed program** - their feedback will be invaluable, as will their support and encouragement when difficulties are encountered. Your support services will also be able to contribute valuable experience.

### Quality Management

The program will effectively become a part of the design process and thus of the **audit trail**: it is a part of your responsibility to ensure that traceability and completeness of this trail is not compromised.

The key to achieving this will be Quality Management.

Above everything else, **the program must work correctly**. Any "approximations" should be made (and documented) deliberately and explicitly in the choice of engineering models and data, not a lottery on whether the program works or not!

**Thorough testing and validation is vital and should never be short-cut**, even under pressure. To do so is to take serious risks with your design (or that of your end-user) and, therefore, your responsibilities.

*Note that validation is not the same thing as testing! (See Appendix A5)*

If (when!) pressures do build up:

- give some early warning to the user community, to help them plan around it - last-minute shocks will seriously damage your most important relationship. (They will almost certainly agree that it is better to be right and a bit late than to deliver errors on time!)
- in consultation with your user community, consider reducing the functionality to be delivered "at day one". It may be possible to deliver only the key functions and add others later. Remember, however, that what you do deliver **must work** and that any additional limitations must be reflected in the documentation.

- **do not cut testing and validation** - you will end up delivering a program which does not work reliably. Not only will your credibility suffer but you will have failed to exercise proper professional care.

## User Interface

A clear, consistent and unambiguous user interface should be provided that helps to minimise the potential opportunities for errors or misunderstandings on the part of the user. A program should not only be "*easy to use*" but should also be "*difficult to misuse*"!

It is also good practice to place a prominent reminder to the user on the program output that he/she carries the responsibility regarding the use of the program and/or its results. You might refer them to the appropriate sections of these Guidelines.

## Project Management

*Developing a program is a project - treat it like one.*

We are all painfully familiar with the phases through which a project typically progresses:

Enthusiasm  
Doubt  
Panic  
Search for the guilty  
Punishment of the innocent  
Promotion of the uninvolved

The following notes may help prevent this happening to **you!**

### Estimating and Planning

- plans (and thus delivery dates) are only as good as the estimates that go into them: developing a program **always** takes far longer than you think, especially if you are a "casual" or "spare time" programmer
- any cost estimates or plans prepared before the users' requirements specification is complete and frozen are pure fiction (and are almost invariably a **wild underestimate** - anything up to a factor of 5-10 or possibly even worse)
- once a frozen specification is available, you can have a **guess** (a factor of 2-5?)
- the first point at which it should be possible to generate any sort of reliable estimate or plan is when the design is complete and frozen (typically a factor of up to 3 - see Appendix A5)

### Cost Control

- keep an eye on the potential costs and benefits: **re-evaluate them at major milestones** (eg. when the formal specification is finished and again when design is complete)
- rigorously avoid the temptation to over-elaborate and add lots of "nice to have" features **at any stage**: stick to what you **need** and adopt a "minimalist" approach

## Change Control

- *Changing objectives is the single most damaging cause of the serious problems which IT developments often get into!* See Appendix A5 for a more detailed discussion.
- The Users' Requirements is the most critical document of all, followed by the Design. They are the foundation upon which everything else is built - *get them right and freeze them!*

## “Fast Track” Short-Cuts

- **any deviations** from standard procedures must be specifically justified and authorised or the audit trail may be broken, with the attendant potential consequences. Like any engineering project, it may be legitimate to vary the levels of adherence, depending on circumstances, but it must be a conscious decision, taken collectively and **documented for audit purposes**.
- **never short-cut testing or validation** (see Quality Management, above)

## Support and Maintenance

To justify its development costs, a program must achieve widespread use. It must therefore be professionally supported and maintained. (If is **not** going to be properly supported and maintained, then it is a part of your professional duty to prevent access to it - throw it away or lock it up, to avoid confusion and potential catastrophe!)

A particular issue, therefore, is: **who will become responsible for ongoing support and maintenance**, if/when this program becomes a part of the standard toolset?

Program support and maintenance is a **critical role** whose function is to increase the effectiveness of the “mainline” engineers (and thus the overall effectiveness of the organisation) by providing a range of services, such as:

- train & support end users
- deal with more esoteric areas, such as convergence!
- inculcate the right culture in the user community (such as proper checking & sensitivity analyses)
- last line of defence from a project blunder!
- develop and maintain programs for standard applications
- deal with potential or actual problems (eg. maintain anti-viral environment) which may arise as a result of importing software of uncertain origin (eg. shareware, via networks)
- advise & support “casual” developers

*Support staff thus have special training and experience needs and “savings” in this area amount to a serious lack of professionalism on the part of the organisation.*

## External Software Suppliers

Support and maintenance staff must also be able to deal with suppliers of software from outside the company (so-called “third party software”) and impose good practice/standards on them too

- you are their customer, so they must keep you satisfied
- indirectly (ie. via you) they must satisfy your users' needs (eg. proper documentation and training, to enable them to follow their Guidelines)

- they must satisfy your needs (for example, proper system documentation, test suites [with verified answers], guaranteed support/response times [and penalties], training courses)
- their products must be written to proper standards (get a copy of their standard procedures, etc, and examine them carefully)

In short, **validate your suppliers** and place clear, written, responsibilities on them. Make it clear what your requirements are and then insist that they deliver. For example, you might require that they adhere to proper QA standards (such as BS5750 or ISO 9000) and you might request names of other (presumably satisfied?) clients - often referred to as “reference clients”, with whom you might discuss the supplier’s performance and capabilities.

You must bear in mind, however, that providing sufficient information to you may conflict with suppliers' reasonable commercial secrecy and special agreements may be required.

## **Liability/Legal Issues**

These are discussed in Appendix A6.

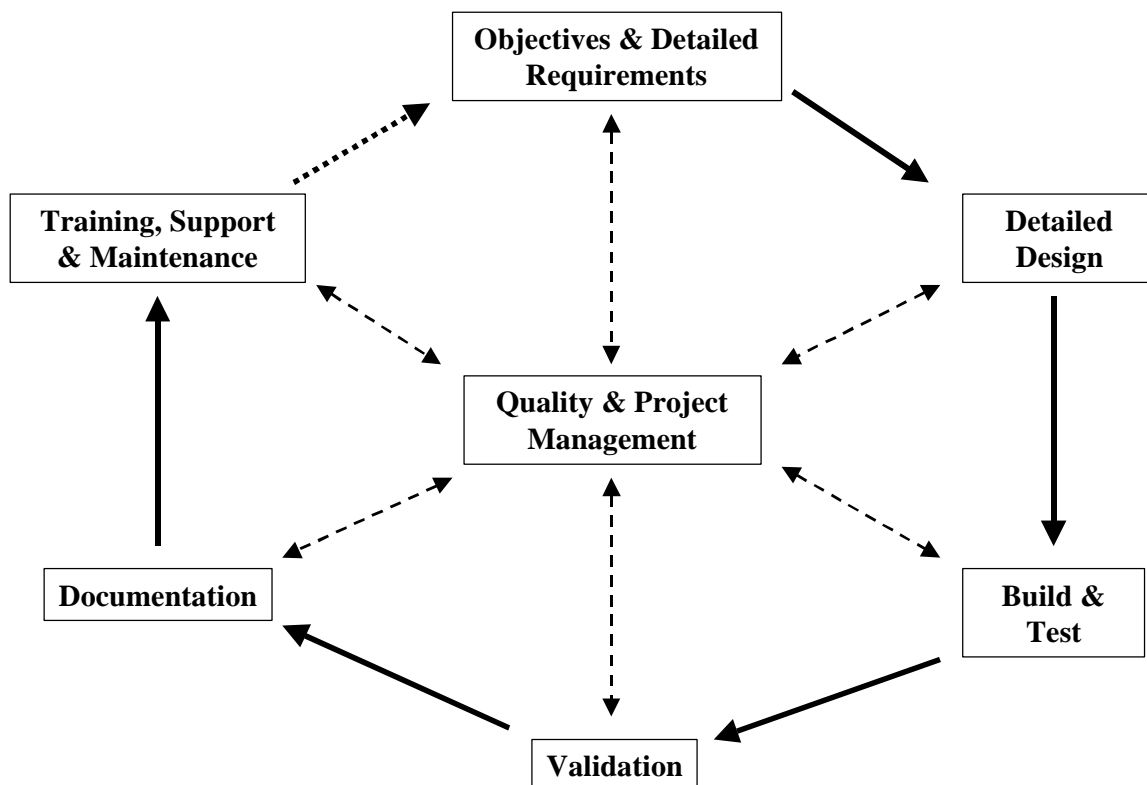
# Appendix A5

## Deliver a better program, more quickly!

This Appendix supplies additional and more detailed suggestions and checklists for the various activities involved in the development of an effective design tool. It is based on many years of experience accumulated by the members of the Working Party and by those who helpfully contributed further ideas. Much of this material is available elsewhere (see, for example, the Bibliography) but has been drawn together into this Appendix to provide a working overview, coloured by our own experience of applying such techniques to process engineering software, and thus to avoid the need for the reader to collect together too many independent references, unless a greater level of detail is required.

*These suggestions are “necessary but not necessarily sufficient” - it is your duty to take professional care in discharging the additional responsibilities involved in program development.*

The following sections contain suggestions and checklists, corresponding to each of the boxes in the diagram below.



It is appropriate to re-emphasise once again the critical importance of the earliest stages: specification, design and quality and project management are the **key** items - get these right and your project will go well. There is a clear similarity in this respect (and, indeed, in many other respects too) with a typical engineering project.

# Quality and Project Management

Ensuring proper quality and project management is an important part of discharging your responsibility to demonstrate “professional care” and thus of limiting your potential liability if things do go wrong.

## Reviews & Inspections

It is important to monitor the progress of the project, through **all** its stages, not just as some form of *post-mortem*.

In any but the most trivial cases, organise “walk-throughs” of the specification and the design: it is always helpful to gather comments and suggestions from your prospective user community, from your colleagues and from your support services. There is a clear analogy to project and/or design reviews in an engineering project.

In important or complex cases, you should consider making this a formal “**Inspection**”, as recommended by IBM (see article by Fagan in the Bibliography). The additional skills, training and experience required are not too demanding at this early stage and Inspections can be very effective at removing potential problems before they have a chance to become serious. ANSI/IEEE (see bibliography) recommend the use of key check-words when reviewing specifications and designs:

- concise
- complete
- unambiguous
- consistent
- traceable
- verifiable
- modifiable

to which might sensibly be added

- minimal - is this the **minimum** that is truly **necessary**, with no “nice to have” extras?

Experience indicates that this Inspection approach can detect and thus prevent some 60 - 80% of errors and inadequacies at the specification or design stage and can be moderately effective even in the absence of specialised skills or training (though such skills, training and experience do, of course, make it significantly **more** effective).

The Fagan article also recommends such Inspections for reviewing coding (ie. programs themselves). In common practice, however, this requires appreciably higher levels of skills, training and experience and seems to be rather less effective (and, of course, approaches to programming have changed markedly since that article was written).

## Sign-Off and Change Control

*Experience shows that the single most damaging cause of serious problems which IT projects often get into is that of changing objectives and specifications.* The causes are various but the effects are almost always serious. So:

- reviews and Inspections are important

- beware creeping (and undocumented) growth of function - freeze the specification and keep to what you really **need** and no more
- **always** proceed in a step-by-step manner, completing each step before starting the next
- once a specification or design has been reviewed, formally sign it off and place it under strict **change control** - these documents will form an important part of the audit trail

“Solid and stable” specifications and designs are absolutely **vital**: they are the foundation upon which everything else will be built, so **get them right and freeze them**. Ask yourself: would you build a plant any other way?

Some changes will, of course, be **necessary**, when errors are discovered or as engineering requirements change.

When such changes **must** be made, then it is important that they are subjected to the same rigorous process: specification, design, Inspection, cost/benefit, etc. They must **never** be added in an uncontrolled manner into the later stages (for example, into the build stage, without a proper design change) - this is a primary (and almost certain) source of errors and inconsistencies and, of course, the audit trail is effectively destroyed.

*Changes which are not strictly necessary, fully cost-justified and approved should be rigorously excluded.*

## Prototyping

Be very careful about prototyping. The basic message is: *“don’t, unless you really have to and, if it really is necessary, then a prototype must be for a specific and clearly defined purpose and it must be specified and designed in the normal way”*. (See also “Build & Test”, below.)

## Estimating and Planning

Plans and completion dates are only as good as the estimates which go into them. Experience suggests that these are almost always seriously inaccurate and are always under-estimated - you may have heard the joke: *“get an estimate from the three best experts you can find and then **add them up**”* (in reality, this is often still an under-estimate)!

Many factors seem to be involved, such as:

- program developments are often done by “enthusiasts” and enthusiasts are always optimistic
- information technology is still an immature and fast-moving field, with very little case-lore and relatively few thoroughly-trained and experienced staff.
- all program developments contain at least an element of innovation and, thus, of R&D - you never build “two-of-a-kind”. Major engineering projects containing significant novelty typically run late and over-budget too.
- estimates are almost always made **far** too early (typically before there is a proper specification or design) and they are seldom formally reviewed and re-estimated as and when these vital documents do become available - it is hardly surprising that problems arise?
- such work is perhaps subject to an extreme form of the 80/20 rule - the calculation part, at least, of the program must be **perfect** and must contain **no** errors: 99.9% is not good enough and you cannot “add a design margin”!

- the above-mentioned factor of three can be explained by recognising that an (enthusiastic) expert will typically estimate how long it might take him/her to do just the programming, whereas a professional development consists of three major phases:

specify & design  
build and test  
validate

Experience suggests that each of these takes approximately the same length of time to do properly!

There are no magic formulae but some very basic advice can be given:

- **do not guess** - estimates should not be made until at least the specification (and preferably at least an initial design) are complete, approved and frozen
- if you **have** to guess, then make it absolutely clear that it **is** a guess (or a “class one estimate”!) and that it is subject to revision, almost certainly both substantial and upwards. **Incorporate this revision process (and any appropriate re-approvals) into your plans.**
- if you do a thorough job of the specification and design, then you have some chance of estimating to within a factor of three!
- **any changes whatsoever will have a dramatic and disproportionate impact on both costs and timescales** - make certain that such changes are

necessary  
properly specified and designed  
estimated separately and formally approved  
subject to a supplementary budget

*Experience shows that you can never “work it in”.*

- double the estimates if the work is to be done on a “casual” or part-time basis or by inexperienced staff (and double them again if both apply). *Ask yourself: would you run an engineering project this way, without massive supervision from senior, experienced staff? This level of rigour is seldom applied to software projects!*
- when estimating timescales, be ruthlessly pessimistic regarding the true available manhours per week, especially if working on a part-time basis - it will be a **lot** less than you think, by the time you’ve allowed for interruptions, other work, illness, holidays, ..., especially if several people are involved over an extended period (and if several people are involved, add a significant allowance for communications, integration and rationalisation - see “The Mythical Manmonth”, in the bibliography)

## Control of Time and Cost

Cost and time over-runs can have a number of causes:

- the estimates were wrong in the first place
- the truly available manhours per week are less than you expected
- a high level of disruption is causing serious productivity problems
- the specification is changing or hidden extras are being added
- inadequate training or experience

(... continued)

At each major milestone, examine these factors and:

- never assume you can catch up - you can't (we now know that we have under-estimated the work to date: what reason is there to suppose that we have over-estimated the remainder?) - things will always get worse and action must be taken
- make appropriate changes to the estimates and plans, based on your newly-acquired information and experience - this new information is the best and most relevant that you have. (Note that a plan which is out of date and clearly impractical is a powerful demotivator, quite apart from the damage which it can do in the wrong hands - a formal revision should be issued).
- keep your potential user community informed, so that they can plan around any problems - nasty shocks late in the day will seriously damage your most important relationship.
- organise any necessary training or resources

## Objectives and Requirements

As with an engineering project, you must establish accurately what it is that is to be delivered **before** trying to build it.

### Objectives

- What are you trying to do, in basic engineering terms?
  - Set out precisely what it is that this program will do.
  - Adopt a “minimalist” approach - keep the required functions and capabilities to the absolute minimum that are necessary to achieve your (engineering) objectives.
  - Be ruthless in excluding non-vital extras, which will consume time and effort and which must all be tested, validated and documented.
  - Keep it as short and sharp as possible - this is the “ball to keep your eye on” - and beware uncontrolled growth of the specification. ***Keep it simple wherever you can: problems and costs escalate rapidly with program complexity*** and, if a number of different capabilities are required, it will probably be best to have several small/simple programs, rather than one big one with lots of options. Each will then be easier and simpler to prepare and test, since there will be far fewer “paths” through the system. If these capabilities share a lot of common ground, then they should also be able to share some of the code but there may be no compelling reason why they have to be integrated into a single complex system.
- What is the expected user community? (Remember that it **won't** just be you!)
  - Do they all have the same (engineering) objectives?
  - If not, is it actually appropriate to try to satisfy them all within a single program, or will the above comments regarding program complexity make it more appropriate to develop several simpler systems?

### Detailed User Requirements Specification

Having established and agreed your engineering objectives, you must now establish exactly how the program is to satisfy them. At this stage, you must consider this from the **users'** perspective - the internal workings of the program (ie. how it will deliver these capabilities) will be developed under “Design”, below.

- **What engineering model is to be used?** What are the
  - engineering assumptions/simplifications
  - limitations/exclusions/range of validity
  - error conditions and appropriate diagnostics
- **What Standards are to be adhered to?** For example, there may be Company Standards for
  - representation of thermophysical properties (ie. a standard thermopack)
  - programming languages
  - documentation
  - user interfaces
- **What will the User Interface look like?** For example
  - input information, including
    - general layout and units of dimension
    - checks/validation of the input data and appropriate error/warning messages
    - defaults to be supplied
  - output information, including, for example
    - a reprint of the input information. This must include diagnostics and defaults, ie. the current and complete state of the problem definition. This will enable the user to be **certain** that the program has understood (and will therefore try to solve) the problem as he/she intended. (See additional remarks under “Design”, below.)
    - error and warning messages during calculations (eg. out of range of correlations). (See remarks on diagnostics under “Design”, below.)
    - intermediate/monitoring information (eg. to enable the user to monitor the progress of convergence)
    - final results, including diagnostics, clear cross-referencing to current problem definition, comprehensive labelling (including units of dimension), special information to enable the user to check the validity of the results, ...

It is sensible to sketch mock-ups of both input and output, to avoid misunderstandings between users and developers - modern tools make this much more cost-effective than it used to be

The program needs a good effective user interface or it will not be used but, remember, it has all got to be tested and documented, so keep it simple/minimalist! Your objective should be not only to make the program easy to use but also to make it difficult to misuse.

- **What electronic interfaces are required?**
  - Define any need for electronic transfer of information to or from other systems or databases: electronic transfer is a good way of avoiding (eg.) transcription errors but, of course, you must get it right, or it will simply automate the problems!
  - The precise form of such interfaces may be defined by the other system: if so, put it in here as a specification; if not, it is a matter for design (see later)

- **What are the intended delivery platforms?**

- Define very carefully any requirements for portability (ie. the ability to run on more than one computer or operating system). Such portability requires special attention to testing - see below - and may also have implications for design.

- **What documentation is required?**

Documentation also benefits greatly from being specified and designed

- **users' documentation:** this must be sufficient to enable users to comply with their own Guidelines (see "Guidelines for Engineers Using Software")
- **programmers'/system documentation:** wherever feasible, system documentation should ultimately be included in the program itself, as comments of some kind, so that it is impossible for it to get lost. During your development activities, it will also help you to remember where you've got to, especially if you are a "casual" or part-time programmer and, therefore, subject to extended interruptions.

*Draft documentation will be needed for your testing and validation activities and its early preparation and review often reveals many shortcomings and extra requirements.*

## Test and Validation Requirements

It is important to appreciate that testing and validation are not the same activity and are directed at quite different purposes:

- **testing** is to prove that the program correctly implements and solves the engineering model
- **validation** is to prove that the engineering model, as implemented in the program, adequately represents the physical system concerned

Both of these activities are important if the end-users' needs are to be satisfied: **testing should be completed before validation is begun and the general approach must be "assume that it is wrong and prove that it is right"**.

- you will need to assemble a "test suite" and a "validation suite" of example problems which are properly representative of the uses to which the program will be put. *These problems must all be accompanied with verified solutions.*
- these problems should cover **all** the claimed functionality of the program - use lots of small tests to "poke in all the corners" but some at least of the problems in the two suites must be "serious" enough to explore the limits of the program's expected capabilities.
- the verified solutions should be calculated (or at least comprehensively checked) independently of the program under development, either using other (validated) programs or, if necessary, by hand
- **all** these tests and validations should be performed on **all** the intended delivery platforms
- do not forget to "test" the documentation too

It is important to appreciate that thorough testing and validation **must** be carried out and that they take time, care and effort. Make sure that your estimates and plans properly reflect this and **never** short-cut them.

## Quality and Project Management of Specifications

As outlined earlier in this Appendix, more software projects probably go wrong because of shifting/growing specifications than for any other single reason.

These requirements specifications are **the** critical reference documents, upon which all else will be built: it is **vital** that you get them right and then place them under **formal change control**.

Review especially:

- does this capability (or something reasonably close to it) already exist? Consult your support services and **only** go ahead if it is really necessary and cost-justified to do so.
- do these documents/requirements satisfy the ANSI keywords and so form a sound basis for subsequent steps?
- do the estimates and plans still make sense? Be realistic - they will almost certainly require significant revision. Re-assess the cost-justification and obtain re-approvals where required.
- is it appropriate for **you** to go any further or are more specialist skills and experience required? Again, take a realistic and professional view.

Effort/care expended here is repaid **many** times over as the project progresses: for example, Inspection can catch around 60-80% of the problems at this stage and is (relatively) straightforward.

Conversely, get it wrong at this stage and .... !

## Design Specifications

Once the specification has been **frozen**, you can start on the design.

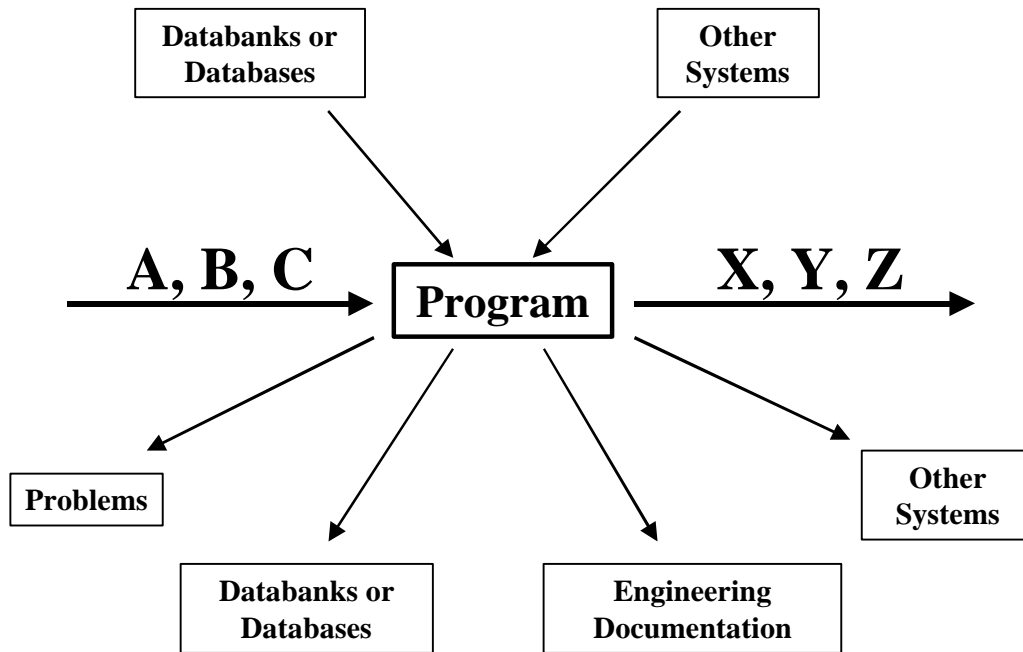
*The basic objective is to design a program which will implement the specified users' requirements - no more and no less.*

The following are some suggested steps (very much like putting together a process design). As before, these should be regarded as necessary but not necessarily sufficient. In particular, for larger projects, the use of formal software engineering techniques should be considered (see Bibliography).

The basic approach is to use flowcharts - very much the equivalent of process flowsheets. It is recommended that, at least initially, these should be at a "high/structural level", annotated in something similar to English, rather than, as are sometimes used, annotated in either pseudo-program code or even in the programming language itself. Much as a process flowsheet is helpful in visualising the process and its structure and inter-relationships, the program flowchart will help you visualise, develop and structure your program.

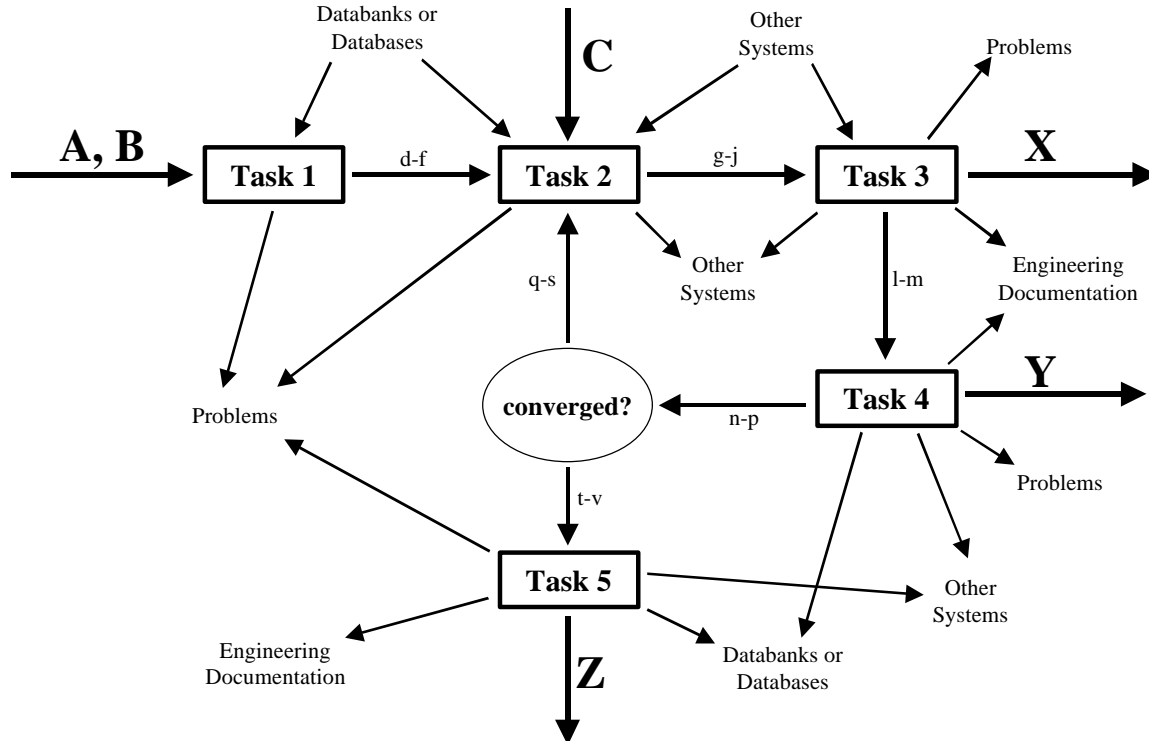
### Step 1: Basic Specification

Re-state the users' specification in the form of a simple flowchart, showing the program, its inputs and its outputs:



## Step 2: Segmentation

Segment the program into lower-level task modules (for example, the initial breakdown might be into input, calculation and output) and develop your flowchart to show these modules and the information flows between them, to and from the user and other systems, etc



Each task module should:

- be testable, with a clearly defined and verifiable function
- be as self-contained as possible, with minimum cross-linkage

You should:

- make **all** communications between task modules explicit, visible and testable
- include “problems” - exceptions, range checking, diagnostics, etc
- include primary “recycles” - such as iteration loops, and “control logic” - decision points, such as selecting options, checking for convergence, valid ranges, etc.

Just like a process flowsheet, this flowchart will describe the specific tasks, how they are organised and the information which flows between them.

### Step 3: Evolution of Design

Repeat steps 1 & 2 for each task module, gradually increasing the detail of the breakdown, until you can clearly identify:

- **standard tasks** (eg. calculate a bubble point) to be adopted as building blocks (go and talk to your support services - they will be able to tell you what is available). In some cases (eg. the thermopack?) these standard tasks may actually be a part of the specification.

**Do** make use of existing program blocks - **do not** write your own unless there is a **definite need** to do so:

- they avoid duplication of effort
- they provide consistency between programs that use them
- they will have been written by the relevant expert(s) and will have been fully tested and validated: this reduces opportunities for introduction of errors;

**But:**

- watch out for “special versions” - make certain that you really are using a **standard** module, not someone's unofficial "improvement" of it!
  - if you “customise” it in any way whatever, then it must be tested as if it is new code
  - they must be subjected to validation *in situ*, once the system has been assembled and tested
- **where numerical methods are required** to solve the engineering model equations (ie. where a group of equations requires solution by trial-and-error or iteration)

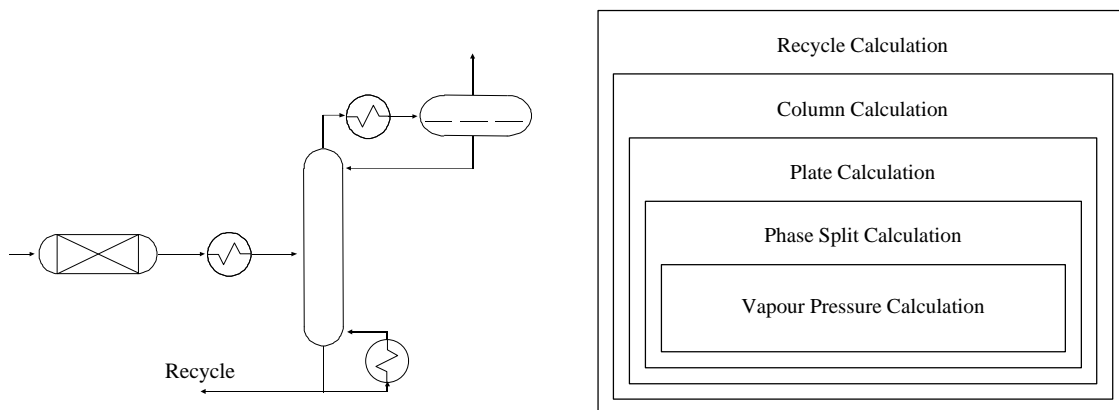
Again, **do** make use of standard program blocks (such as those listed in the Bibliography). This is a highly-specialised area - **do not** write your own unless there is a **definite need** to do so

Define convergence criteria, intermediate output, diagnostics, failure and warning conditions, etc (see also "User Interface - diagnostics", below) and be especially careful where "nested" calculations may be involved (see box).

- **small/simple/testable task modules** which you are **certain** you can write, test and make work properly (for example, a block of equations representing the engineering model itself, which can be solved in order and without iteration)
  - each of these modules must have a clear and specific, limited function which can be exhaustively tested

## Some Remarks on "Nested" Calculations

A very common situation is where several layers of iterative calculation are "nested" within each other. For example, when solving the flowsheet shown below, five layers of iteration may be involved:



At each layer, any method other than basic resubstitution will attempt to estimate the slope of the convergence "surface" represented by the next layer down and will be "confused" if that surface is not smooth/consistent but contains bumps and hollows. This "orange-peel effect" occurs when the inner calculation is not completely converged (the same effect can occur at any or all of the layers) and may then create problems in converging the outer layers. This will apply particularly to acceleration methods (both the simpler ones, such as Wegstein, and the more sophisticated methods such as Newton-Raphson, etc) and will typically manifest itself in the form of the "problem" curve shown in Appendix A3 - convergence will proceed moderately well so long as the current estimate is well-removed from the solution and large steps are being taken (by analogy, so long as the method is examining only the global shape of the orange) but will get into trouble once the solution is approached and smaller steps are required (when it becomes aware of the bumps and hollows and the surface no longer has a simple slope). (Simple resubstitution makes no attempt to estimate the shape of the surface and is often unaffected by this phenomenon but, of course, inadequate convergence brings other problems - see Appendix A3.)

**As a general rule, in order to facilitate robust and reliable convergence, each layer should be converged to at least one decimal digit (preferably two) more than the layer above.**

As illustrated in the "nest" diagram, this is potentially a serious problem for thermophysical property calculations, which are typically at the innermost layer. In order to enable reliable overall convergence, especially using gradient methods, it is important to converge all such "low level" calculations (including phase separations, bubble points, etc) very tightly indeed. Very roughly, **for the outermost layers to be able to converge reliably to, say, 0.01%, the innermost layer (typically the fluid density) should be converged to something like 8-10 decimal digits. (Remember, however, that this is the nominal precision of the calculation, the primary purpose of which is to ensure robust and reliable convergence. As noted elsewhere, this is not the same as the accuracy of the results, which will depend on the accuracy of the raw data and the suitability of the model.)**

### Diagnostic "tracebacks" for Nested Calculations

In cases of nested iterations, the end-user must be able to trace the source of a problem. For example, a message from a failed vapour pressure calculation will be largely meaningless unless it is possible for the user to trace that it occurred during a phase separation calculation on tray 14 of column C104 on the third flowsheet iteration.

**Make careful use of diagnostic codes and flags to control diagnostic printouts from the various layers and to signify the severity of the error (ie. a near miss or a catastrophe).** By this means, it is possible to provide a "traceback" from the actual occurrence of the problem (for example, a failure in the calculation of a phase split) up through the layers to its root cause (for example, conditions in a column too near a critical point or giving rise to reactive equilibrium) and its probable effects on the validity of the results.

- the flows of information into and out of these modules should be explicit, visible and testable and each module should be free of hidden interactions or “crosstalk”
- task modules should rigorously check both inputs and outputs for potential problems, such as unexpected flow regimes (eg. condition which imply a vapour stream, not the expected liquid) or correlations being used outside appropriate ranges. (See also “User Interface - diagnostics”, below.)
- under no circumstances whatever should it be possible for a task module to generate physically unrealistic conditions and transmit them to the rest of the program.
- **critical unknowns** - areas where you are not **certain** that it will all work correctly or where special checking and testing may be required
  - break these down further if you can, to give a better focus on where the problems really are
  - these may need some targetted prototyping (see later)
  - seek appropriate help and advice from your support services

## Step 4: Data Structure

From your flowcharts, identify the flows of information and develop an appropriate underlying data structure.

It is very easy to create a complex and tangled mess, which eventually has to be completely redesigned or which will cause serious problems at later stages and in ongoing maintenance:

- take care
- keep it simple
- try to make it so that it is easy to modify and extend (when you remember all the things you forgot!)

In complex cases, you may find it helpful to use standard data modelling approaches, such as STEP/EXPRESS (see Bibliography), especially for external information flows (ie. those to/from other systems or databases). (See also “Electronic Interfaces”, below.)

## Step 5: Logic and Control Structure

From your flowcharts, identify overall logic flows and decision points (eg. selection of key options, convergence issues, checking and diagnostics, etc), to form the overall control structure of the program.

Many of the remarks at Step 4 above apply.

## Step 6: Electronic Interfaces

Design the format of files which are to be received from or sent to other systems or databanks and the subsystems which will generate or read them. (In many cases, these formats will be pre-defined as a part of the specification.)

In defining these formats, the use of appropriate Standards, such as STEP (see Bibliography) is strongly recommended and, conversely, the use of proprietary file formats is strongly discouraged. Even if you do decide to use a file structure and format of your own, make sure that it is simple and consistent and that it can be extended in the future if required - remember that, if your program is a success and generates

useful information, then integration may well be required at some time in the future, so adoption of the appropriate standard at this stage may save a lot of problems later.

*These data flows must also be rigorously checked, or errors will be widely propagated before they are discovered and serious and widespread damage may occur.*

## Step 7: The User Interface

The quality of the user interface will have a dramatic effect on the level of use of the program, on user satisfaction and on the number of errors which users make in using it.

*Effective user interfaces do not happen by accident - they are the result of careful specification and design.*

On the other hand, it is easy to become obsessed with trivial improvements, so a balance must be struck - remember that all of the facilities must be tested and documented, so keep it simple and “minimalist”.

It is strongly recommended practice to sketch some mock-ups and agree them with the user community - modern tools make this much less of a chore than it used to be.

- **a good user interface must:**
  - implement the requirements set out in the specification (see above)
  - be as intuitive to its **user** community as possible (not necessarily to its developers!)
  - be consistent. For example:
    - use the same units of measurement for the same or similar items and avoid the use of different scale factors for values without warning (eg. m/mm, absolute/gauge pressures, mass/mole fractions)
    - clearly label all values being requested from or presented to the user with their units of measurement
    - use a similar layout for similar information or functions
    - avoid the use of obscure (especially numeric) codes to select options - if their use **cannot** be avoided, codes must be intuitively obvious and the same code should mean the same thing wherever it appears
- **reprinting the input information**

In implementing the users' requirements, a particular issue will arise with interactive/online systems in ensuring that there is a proper correspondence between the filed results and the exact input (ie. problem definition) which produced them (see “Guidelines for Engineers Using Software”).

The problem definition will change many times during an interactive session, as the design evolves, and so facilities should be provided to print out the current problem definition:

- at any time, on demand from the user
- automatically, at the commencement of the attempt to solve the problem
- automatically attached to any set of results

This reprint may form a part of the audit trail and so must be clearly understandable, properly labelled and complete with any warning or error messages, units conversions, clearly-labelled defaults, etc

- **presentation of final results**

Visualisation tools should be used where practical, as this gives good intuitive error checking - for example, even a small error in a trend or plot is normally very visible, which is seldom the case with tables of figures.

- **checking information**

Appropriate information must be provided to enable the user to thoroughly cross-check the results (see “Guidelines for Engineers Using Software”).

- **intermediate/monitoring information**

Calculations do not always progress smoothly and clear, labelled information should be presented to the user regarding, for example, the progress of convergence

- **diagnostics**

*Diagnostics is a vital area which requires careful design but which is all-too-often ignored.*

Program operation will **not** go perfectly, problems **will** arise and you **must** provide sufficient checking and feedback to enable users to track them down, understand them and assess the potential impact on the results.

- As a general rule, you must assume that everything is suspect and check **everything** very carefully, at all stages - during input from the user and during calculation.
- Diagnostic messages should always be expressed in engineering terms - ie. in terms which are meaningful to the user.
- Diagnostic messages must contain sufficient information for the user to be able to identify precisely where they have come from, what has caused them and their severity (ie. their likely impact on the validity of the results). This will require particular care in the case of "nested" calculations (see box above).
- Potential problems may fall into several categories and you need to give thought to what action is appropriate when such violations are detected, so that the engineer is guided to a workable solution. It is best to be guided by what makes physical sense:

**fatal errors:** a problem which is so severe that further progress is simply impossible and no “sensible and safe” action can be taken (see below).

Typically, these might represent physical nonsense, such as a negative flowrate. In these cases, **do not** fall into the trap of simply assuming something sensible (for example, resetting negative flows to zero is likely to create mass-balance and other problems within iterative calculations). Something has gone seriously wrong and the user must be left in no doubt about it!

Check especially anything which may result in program malfunction. Good examples might be calculating the composition of a stream with zero total flow (a zero flow is physically realistic but, of course, it has no composition and dividing by zero will cause nonsense!) or anything which affects array subscripts (this was claimed to be one of the causes of the serious and much-publicised problems with the London Ambulance System). (See also comments under Testing, below.)

**non-fatal errors:** problems which are less severe and where it is possible to take sensible and safe action.

Such problems often arise in the early stages of trial-and-error calculations - early estimates may be far removed from the true solution and may, for example, lie outside the valid range of a correlation. A typical example might be when all input flows are zero - physical sense can be preserved by simply setting all the outputs to zero (and the iteration will often "struggle on" successfully).

Depending on severity, it may be appropriate to continue the calculation and to maintain a "flag" of some kind to indicate the presence of a problem, in the hope that the iteration will eventually find its way into the feasible region. Another frequent source of this type of error is an iterative calculation which does not completely converge but where a physically sensible value is available to enable calculations to continue.

*Note, however, that these situations must be properly flagged and diagnostics provided to the end-user.*

- diagnostic messages and internal flags should be classified according to the severity of the possible problem and the user must be able to select the level and quantity of information displayed.
- **it must not be possible for error and warning messages to be suppressed when they are associated with the final results and the calculation or final iteration should always be repeated with full message display, over-riding the user's selection.**

*Remember that it is your professional responsibility to ensure that all results are properly labelled with information regarding their validity and diagnostic messages are a crucial part of that information.*

## Step 8: User and System Documentation

It is good practice to begin to assemble a skeleton of the documentation at this stage. This will consist largely of the above information and the earlier you do this the better, especially if you are a "casual" or part-time programmer - it will help you a great deal in remembering where you've got to after the inevitable interruptions and you will need it for "walk throughs" and for testing (see below). Experience also shows that attempting to explain a program to its users almost always exposes weaknesses - it is, in effect, another form of Inspection.

## Quality and Project Management of Designs

As indicated earlier in this Appendix, inadequate design is second only to shifting/growing specifications in causing software projects to go wrong.

These design specifications form a part of the audit trail and are critical reference documents, upon which all subsequent stages will depend: *it is vital that you get them right and then place them under formal change control.*

(... continued)

Review especially:

- does this design make maximum use of whatever already exists? Consult your support services and **only** go ahead if it is really necessary and cost-justified to do so.

and repeat the checks made under "Quality & Project Management of Specifications", above.

Once again, thorough Inspection can catch around 60-80% of the problems at this stage.

*Problems which escape detection at this stage will be very much more difficult to find later.*

## Build and Test

Only at this stage (ie. when the design has been formally approved) should you start to create code.

*The basic objective is to implement the design - no more and no less.*

In principle, if you've done the specification and design thoroughly, building the system will be straightforward and, of course, there won't be any errors, so testing will be no problem!

In practice, of course, things don't work out quite so neatly. If (when?) things start to get complicated, then go back and review the preceding steps:

how did these complications arise?

are you adding/changing bits without respecifying and redesigning them? (This is especially easily forgotten during testing and debugging.)

have you given adequate attention to designing the information and logic flows? (This, too, is especially easily forgotten during testing and debugging.)

Adding extra features will, almost certainly, be a significant temptation but must be firmly resisted - new features must be added in a professional manner and **anything which is not essential and fully cost-justified should be delayed for addition in a later version, it must not be allowed to disrupt the initial development.**

## Prototyping

You may need to investigate some key areas of uncertainty by preparing prototypes:

- such prototypes must have a clear, specific and verifiable purpose and must be specified and designed to deliver it (usually this will take the form of "will this work?") and they must be minimalist
- such prototyping activities should be done **first**, not last, since, if it fails, the entire development must be re-evaluated and effort invested in other parts of the system may be wasted

Prototyping is often used as an excuse to go ahead without proper specifications and/or designs. The **worst** way to build a system is by evolutionary prototyping - would you build a plant without a clear specification and design or by "prototyping"? This is a certain route to serious problems.

## Structural Framework

- build the structural framework first:
  - the data structure
  - the segmentation and control structures
- then test rigorously at this structural level:
  - is the right information arriving in the right place?
  - are the checks and decision points working correctly?

Make full use of whatever monitoring and testing tools are available, such as array-bound checking, uninitialised variables, mismatched argument lists, execution trace, etc.

To enable thorough testing, you may need to develop skeleton versions of some of the task modules, in order to provide these checks. The alternative approach is to prepare “test harnesses”, but these must be properly specified and designed (they are a sort of prototype, really, and should be treated in the same way) and can therefore be just as costly to develop as the task modules themselves (and, of course, the harnesses are of no further value, unlike the modules). **Do not**, however, fall into the temptation of developing any more than an irreducible minimum of such modules before the structural framework has been completed and thoroughly tested.

- try to be very systematic, keep proper records of the tests and the errors found and fixed and file them (these will provide important information regarding the progress of the testing phase - see "Bug Fixes..." below)

## Task Modules

- starting at the data input, add sections of detailed code between control points
- test them individually, as thoroughly as you can. (If you have been careful and thorough in your design, then each module or section of code should have a specific and testable function.)
- keep re-checking the above structural aspects too, as you add each new piece (see system testing, below)
- be systematic and keep careful records of the testing process - these will be important, see "Bug Fixes...", below

## System Testing

Having developed and tested the structural framework and each module, you are ready to test the whole system.

**If** your specification and design were good and thorough, **and** your system is truly modular, with explicit interlinking and minimal “hidden crosstalk”, **and** you’ve been thorough in your testing along the way (see above) **then**, in principle: this should be easy!

In practice, however, it usually isn’t that simple - *this is usually the most difficult and frustrating part of the entire project!* If serious problems begin to occur, it probably means that there is more “crosstalk” than you thought and/or you’ve been “enhancing” the specification again! If (when?) problems are encountered:

- be **very** systematic: tests should be planned to explore a specific issue/area and remember that it either works or it doesn't - do not fall into the trap of "it almost works - near enough for the moment"! **Get it right** before moving on to the next test.
- go back a stage or two and repeat your testing, using different test cases
- add the task modules in a different order and see where it first starts to go wrong (it doesn't necessarily mean that the error is in the latest addition, so you might need to try various orders and combinations until a pattern emerges)
- use lots of small tests to "poke in all the corners" and a few large ones to really "stretch" the program's limitations
- test the system thoroughly on **all** your planned delivery platforms - they may have different characteristics (such as the handling of uninitialised variables, or array bound checking). A different computer system may also help you in tracing some of the more obscure failures during normal testing.
- as a **last** resort, when **all** other options have been tried and have demonstrably failed, and serious confusion reigns, remember that "standard" routines have been written and tested by mortal human beings and may, therefore, contain bugs, be badly documented or have been "improved":

*Remember that this really is a last resort - it is much more likely to be something you've done or not done yourself.*

- make absolutely **certain** that you really are using the **standard** version, not someone's private improvement of it - all specific problems have their own characteristics and any special improvement may work for them but not for you
- involve your support services to see if they've got any suggestions or any previous history of problems with this routine

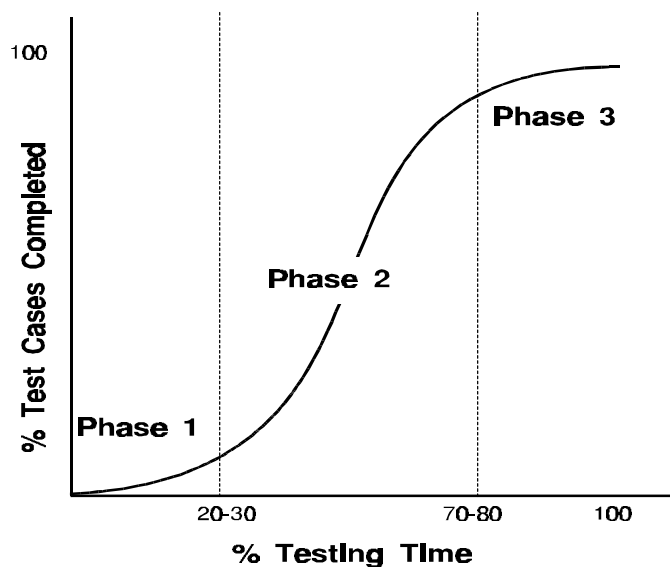
## Bug Fixes, Record Keeping and Predicting Completion

All but the most trivial "bug-fixes" should be subject to careful specification and design too - remember that you've got it wrong once already and experience shows that "fixes" applied in a hurry will probably lead to (possibly several) more problems, especially if there is any significant "crosstalk" between modules.

Your records of testing and fixes can be put to good use in reviewing the progress and effectiveness of testing. A method has been developed by Hitachi (see Bibliography) which, *provided proper records have been kept from the very early stages* (this is important - this technique does not work properly otherwise), will give a good indication of the probable "condition" of the program and when it may be worth exposing it to further testing by the user community (see below).

The method is based on a plot of the number of test cases completed against testing time. It was found that this generates a surprisingly consistent and well-defined "S-curve", with three distinct Phases (see figure).

(... continued)



This curve can be used to predict the "condition" of the system and approximately how long it will be before the testing is "completed". Plot the curve, from your records, and try to detect the first breakpoint. Then do the arithmetic! The accuracy is not great (and there is little statistical information on **engineering** software) but it is a useful indicator and well worth doing, especially since you must keep proper records anyway. It also serves as a useful morale booster, as it provides a faint glimmer of light at the end of the tunnel!

## Testing by the End-User Community

Once you are confident that your program is basically in a good condition and contains very few bugs, get your colleagues and/or the expected user community to help test it, working from the draft users' documentation.

**Do not** do this while there is still a significant number of errors in the program - you will simply waste everyone's time and patience and they may not be too pleased to be asked again later! Make use of the "Hitachi Curve" to assess when user-testing is appropriate and likely to be productive (probably not before Phase 3?).

## Program Validation

Once testing has been completed and you are satisfied that the system is bug-free, then it must be validated on a range of problems, to establish and demonstrate "fitness for purpose" on different classes of uses to which it may be put.

As pointed out previously, this is **not** the same thing as testing:

**testing** establishes whether the program operates correctly and correctly implements the engineering model

**validation** establishes whether the program (and thus the engineering model itself) correctly represents the physical system being modelled and establishes the range of potential applications and the corresponding accuracy which may be expected

(... continued)

Your validation suite must:

- be properly representative of **all** the intended application areas
- cover **all** claimed functionality
- contain at least some large problems, to “challenge” the program’s capabilities and limitations and indicate potential levels of accuracy under different circumstances
- be comprehensively documented, including “standard” results (ie. results which have been validated by other means and which are considered to be correct) and the results from your program. Users may also find them useful as the basis for preparing data for their own problems in “live” use, so they should be listed in the users’ documentation.
- be completely re-evaluated on **all** delivery platforms
- be completely re-evaluated at **all** new releases
- be continually updated and expanded to reflect experience gained in applications by users

Such activities are very “resource hungry” but **must not be short-cut** - this is one of the key aspects of demonstrating your professionalism and thus of limiting any potential liability: once again, be systematic, keep proper records and file them!

## Documentation

Complete the documentation, according to the plans and requirements.

- **users’ documentation:** remember that you will need a good draft much earlier than this:
  - to assist with testing - see above
  - to expose weaknesses - it is surprising what gets thrown up when you try to tell someone how to use the system
- **system/internal documentation:** do this as you go along, too, and **complete it thoroughly** - it forms an important part of your audit trail and ongoing maintenance will be extremely difficult without it

Formal Inspections are quite useful, to make sure that your end-users and your support services can understand it all, not just you!

## Training, Support and Maintenance

*Unless formal arrangements are made to the contrary, you will be implicitly responsible for ongoing support and maintenance of the program* (and, remember, there will be other users, who will require at least some training in its use). It is, therefore, important to consider, at an early stage:

- does this program remain a “one off” (ie. for **your use only**, in which case you must **prevent** other use - throw it away or lock it up!) or does it become part of the standard toolset?
- if the latter, who is responsible for ongoing training, support and maintenance, as the inevitable errors and inadequacies reveal themselves and/or new capabilities are required?

These questions must be discussed with your support services from the outset and, at the very least, they must know enough about your program to be able to prevent duplication of your work by others in the future.

# Review and Feedback

What have we learned from all this which may help us next time?

This learning and continuous improvement process is important and must not be neglected. Some things will almost inevitably have gone wrong or be capable of improvement and the questions must be asked and properly answered. Involve your support services - they are the central repository of such information, so that everyone benefits from your experiences.

**This, too, is an important aspect of demonstrating your professionalism.**

## Bibliography

The following suggested reading must be treated in the spirit of “necessary but not necessarily sufficient” - software engineering is a rapidly-moving field and books and papers rapidly get out of date. These items, however, contain much good sense and useful information and so will provide a basis for further reading.

*Further suggestions for suitable reading material would be welcome.*

### 1. Standards and Codes of Practice:

There are several relevant Standards and Codes of Practice, such as:

- BS 5515, 5887, 6488: documentation, testing and configuration management (respectively) of computer-based systems
- BS 5750, ISO 9000, EN29000 (ie. the “quality standards”), particularly:

BS EN ISO 9000-3: 1997, Quality Management & Quality Assurance Standards, Part 3:

"Guidelines for the application of ISO 9001: 1994 to the development, supply, installation & maintenance of computer software"

- American National Standard 830 - 1984 “Guide to Software Requirements Specifications”
- American National Standard 1016 - 1987 “Recommended Practice for Software Design Descriptions”

### 2. Books and Articles:

- “The Mythical Man-Month”, F P Brooks, Addison-Wesley, ISBN 0-201-00650-2
- “Software Testing and Quality Assurance”, Boris Beizer, Van Nostrand-Reinhold, ISBN 0-442-21306-0
- “Design and Code Inspections to Reduce Errors in Program Development”, M E Fagan, IBM Systems Journal, No. 3 (1976) 182-211
- (Hitachi) “An Analysis of Software Project Failures”, Abe et al, Proceedings of Fourth International Conference on Software Engineering, Munich, Sept 1979.

### **3. The TickIT Scheme: “making a better job of software”**

Details from

DISC TickIT Office  
2 Park Street  
London, W1A 2BS  
UK

### **4. Numerical Methods**

- "Applied Mathematics and modelling for Chemical Engineers", R G Rice & D D Do, Wiley 1995. ISBN 0-471-11156-2.

# Appendix A6

## Legal & Professional Aspects

*It should be noted that there is, as yet, very little established precedent regarding the legal implications and liabilities associated with the use of software in engineering design activities. What follows, therefore, is necessarily a matter of opinion/judgement/interpretation on the part of the Working Party and input and suggestions from readers on potential enhancements to this Appendix would be welcomed.*

### 1. Legal Aspects

The work of the chemical engineer in the UK is subject to the provisions of various Acts of Parliament, including the Health and Safety at Work Act 1974. This Act has important consequences for the way we work, laying down a number of duties for employers and employees and making it a criminal offence to fail to discharge those duties.

#### Some Points from the Act

- The duty is imposed on the individual, unless the individual can demonstrate that training or guidance from the employer is inadequate.
- If an employer's practice is faulty, or the individual is not adequately trained in good practice, then the employer would be held liable.
- If an individual is negligent, then it can result in criminal prosecution and/or being sued for damages through the civil court. Negligence implies a deliberate action done with knowledge, but ignorance would not be a defence if the individual was in a position of responsibility. A corporate body can equally be held to be criminally liable.
- Software which directly affects the operation of plant (eg. process control software or online optimiser) must be designed and constructed so as to be safe, adequately tested and supplied with adequate information to ensure that it is properly used (ie. to "safety critical standards").
- Penalties for breach of the Act are principally criminal (fines and custodial sentences).
- If an individual is injured following a breach of duty under the Act, liability will be deemed proven also.

Similar or equivalent legislation operates in other Countries and the implications are the same: you, the professional engineer, are responsible for all decisions which you make, whether or not a computer is involved.

(... continued)

## Licensed Software

Note that software licences usually contain such broad-ranging exclusions and/or disclaimers as to be almost meaningless. It should also be noted, however, that the legal validity of such disclaimers is often unclear and may be subject to a judgement of what is and what is not considered to be "reasonable". Many licences almost certainly contain clauses which would be deemed "unreasonable exclusions" if challenged in court.

However, validation before use is often a critical issue, whether it is your own software or is licensed from a vendor. (Validation of your own developed software is covered in Appendix A5.) You would therefore be expected to take "reasonable steps" to validate the software for each of your intended applications, in order to establish "fitness for purpose" and the vendor would be expected to cooperate in a "reasonable" manner to facilitate this validation.

## 2. Professional Responsibilities

Most organisations will have established corporate standards and guidelines and, in general, it is your professional responsibility to take reasonable care to follow accepted good practice and your company's procedures. If you do not, you may be increasing your liability. It is therefore important to maintain records which show that you have: this is one reason for the emphasis placed on keeping records and the audit trail in the various chapters of these Guidelines.

The following is an extract from the UK IChemE's "Rules of Professional Conduct", Issue 2: October 1991:

*"3. A member, when discharging his professional duties:*

*(a) shall satisfy himself as to the extent of those duties, and, if in doubt, obtain such clarification or confirmation as is necessary to satisfy himself as to their extent before entering upon them, and shall not accept professional obligations which he believes he has not sufficient competence or authority to perform;*

*(b) shall accept due responsibility for all work done by him or under his direct supervision, and shall take all reasonable steps to ensure that persons working under his authority are competent to carry out the tasks assigned to them, and that they accept personal responsibility for work done under the authority delegated to them;*

*(c) shall, when called upon to give an opinion in his professional capacity and based on the facts disclosed to him, give an opinion that is objective and reliable to the best of his ability; and*

*(d) shall, if his professional advice is not accepted, take all reasonable steps to ensure that the person over-ruling or neglecting his advice is aware of the possible danger which he believes may result from such over-ruling or neglect.*

*4. A member shall take all reasonable care in his work to minimise the risk of death, injury, or ill-health to any person, or of damage to property. In his work, a member shall respect all laws and statutory regulations applicable to the design, operation and maintenance of chemical and processing plant. In addition, a member shall have due regard for the need to protect working and living environments, and the need to ensure efficient use of natural raw materials and resources."*

