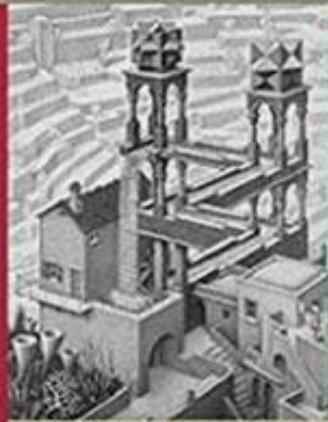


**SOLUTIONS MANUAL**

**SOFTWARE  
ENGINEERING**

Principles and Practice

HANS VAN VLIET



THIRD EDITION

Answers to Exercises  
SOFTWARE ENGINEERING: Principles and Practice  
Third Edition

Hans van Vliet  
Department of Computer Science  
VU University  
De Boelelaan 1081a, 1081 HV Amsterdam  
Email: hans@cs.vu.nl

July 31, 2008

## Preface

This Guide contains answers to a number of exercises from the textbook. Exercises for which a straightforward answer can be found in the text, like ‘Define the term software engineering’ or ‘What is the difference between verification and validation’ (exercises 1 and 4 of Chapter 1) are not included in this guide. Answers to open-ended questions, like ‘Study both the technical and user documentation of a system at your disposal. Are you satisfied with them? Discuss their possible shortcomings and give remedies to improve their quality’ (exercise 17 of Chapter 1) obviously are not included either.

## 1 Introduction

1.9 No, the linear model is not really appropriate. The linear model assumes that we do things right the first time, know everything up front, are able to elicit the true requirements early on, etc. This is usually not the case. On hindsight, we may *document* the development process as if the sequence of steps from the linear model were followed. It is a rational reconstruction rather than a model of how things are done. The linear model confuses project control issues (progress control) with the actual development of the system.

Chapter 3 in particular discusses the drawbacks of the linear model.

1.10 Major differences are: software is not continuous, progress is hardly visible, software is logical rather than physical (maintenance is not caused by wear and tear; reliability is determined by different factors), and the costs are incurred during design rather than production.

Parnas (1999) contains an eloquent discussion of the ‘engineering’ component of software engineering.

- 1.12 Several professional societies for computer professionals have a code on professional conduct. The Association for Computing Machinery (ACM) and the Institute for Electrical and Electronic Engineers (IEEE) have jointly developed a code of ethics. See Section 1.5.

The UK have a means to certify software engineers. The British Computer Society can accredit engineers to the qualification of Chartered Engineer (C.Eng). It is the same qualification as is awarded to other professional engineers. So it is not a software engineering specific qualification. It involves graduation at an accredited institute as well as practical experience of at least 4 years.

Voluntary certification of software professionals in the US through the Institute for Certification of Computer Professionals (ICCP) is supported by the ACM, IEEE, and several other professional organizations. The certification involves an education requirement, an experiences requirement, and passing an exam. In 1998, the Texas Board of Professional Engineering established software engineering as a recognized discipline. Since there is, as yet, no recognized software engineering exam, only highly experienced software engineers are eligible. The state of affairs with respect to professionalizing software engineering is discussed in the November/December 1999 issue of IEEE Software.

The April 1988 issue of Communications of the ACM (vol 31, no 4, pp 372-375) contains a somewhat polemic discussion, entitled "Why I never met a programmer I could trust". It refers to the famous code of Hammurabi, which includes some well-known eye-for-an-eye, tooth-for-a-tooth constructs. One of the messages is that software discipline requires enforcement. The Self-Assessment procedure on the Ethics of Computing (Communications of the ACM, vol 33, no 11 – november 1990 –, pp 110-132) gives further points to ponder.

- 1.13 The important point to note here is that there are opposing requirements for this project. As a software engineer, you have essentially two possibilities: you may look for a compromise, or you may opt for either party. In both cases, you play an *active* role. Many software engineers have a more naive view and expect that the true requirements will show up in some magical way, without their active intervention.

If you look for a compromise, this may take quite some extra time. There is a danger that the compromise is not wholeheartedly accepted by one or both parties. A compromise may leave both parties dissatisfied. Choosing for either party will make the other one unhappy. Usually, there is some power-relationship between the parties involved, and the boss wins. However, that system may well turn out to be unsuccessful, since the end users have not been listened to. (See the LAS system discussed in Section 1.4.3).

One of the main objectives of the software architecture phase is to make conflicts between stakeholders explicit, and engage in a discussion of the tradeoffs involved. See Chapter 11, and especially Section 11.5.

- 1.18 The issues raised in this question can also be illustrated through classroom projects.
- 1.20 These principles are dealt with at various places in this book. Some very brief answers:
- A. If you do not know the current situation (for example, how productive your team is, how many errors your team removes per month), you can not make sound predictions. To measure is to know; see also Chapter 6 and 7.

- B. Reuse means less work, and the quality of the pieces reused will in many cases be of a higher quality. See Chapter 17.
- C. Complexity has many facets; see Chapter 6 and Section 12.1.4.
- D. Sloppy descriptions of artifacts lead to misunderstandings between developers, between developers and the client, etc. This results in errors and rework. See Chapter 9.
- E. Things will change, whether you like it or not. A rigid software process can and will not be followed; see Section 3.7.
- F. Software engineering projects are team projects. This requires discipline: changes must be dealt with in an orderly way, decisions must be documented, etc. Once a disciplined approach is followed (CMM level 2, more or less) there is room for further improvements. See Section 6.6.
- G. If you do not understand the problem, you can hardly be expected to solve it. The requirements then will not address the real issues, and much rework will result. See Chapter 9.
- H. Formal quality management means that there are formal procedures to decide on quality issues. If these procedures are informal, things slip through, there is an excuse to postpone further testing, etc. But quality can not be built in at a later stage. See Chapter 6.
- I. If components have little interaction, changes will more often be local (see Section 12.1), and their reuse opportunity becomes larger (Chapter 17).
- J. Software grows; Big Bang projects are dangerous. See Chapter 3.
- K. Not only the software product is important. User documentation, training material, etc. has to pass the quality tests. See Chapter 6.
- L. If change is not planned, both the product and the process are rigid. Changes then become more difficult to handle and more difficult to implement. See Chapter 3.
- M. If tradeoffs are not made explicit, there is a chance that the rationale for decisions will be forgotten, and someone will make the wrong decision, e.g. during maintenance. See Chapters 12 and 14, and Section 11.2.
- N. Like in any engineering branch, a lot can be learned from successful (and unsuccessful) solutions that others have found. It prevents mistakes, improves insight, and helps to build a catalog of useful building blocks. See Chapters 11 and 12, and specifically Sections 11.4 and 12.5.
- O. There will always be risks. See Section 8.3.

## 2 Introduction to Software Engineering Management

- 2.6 Quantitative data on past projects is a valuable source of information when planning a new project. The characteristics of the new project can be compared with those of earlier projects, resulting in estimates of time and budget that are well underpinned. "To measure is to know". See also Chapter 7.

2.8 Note that neglecting environmental issues is a common cause of problems in many projects. All too often, it is thought that the software is the only thing that matters, and that the project is finished as soon as the software is delivered to the customer.

2.9 Brooks' arguments for this increase in cost run as follows ((Brooks, 1995, p. 6)): A programming product requires that the program is written in a generalized fashion. In particular, the range and form of inputs must be generalized. Also, a programming product must be thoroughly tested and documented.

### 3 The Software Life Cycle Revisited

3.13 Arguments pro a thorough requirements analysis phase include:

- The scope of the project is known at an early stage, which allows management to properly plan and budget the project;
- The effects of the new system on the organization are known at an early stage. Non-technical issues, such as changing working procedures, can thus be planned well in advance;
- People involved know early on what is expected from them. This allows for clear testing procedures and acceptance criteria. Having a well-delineated requirements specification allows that change requests can be identified as such and properly dealt with.
- The resulting system is likely to be more robust and better maintainable.
- Conflicting views between interested parties are resolved at an early stage.

Arguments in favor of a prototyping strategy include:

- The resulting system is more likely to fit real user needs. Bells and whistles can be identified as such. Real user requirements can only be identified when users have had the opportunity to work with the system;
- The occurrence of the Big Bang effect is precluded. Requirements evolve as the system evolves;
- The organization may gently accustom itself to changing working procedures and the like;
- People feel more closely involved with the project and the resulting system. This increases the chances of acceptance of the system.

3.15 The merits of evolutionary prototyping are listed in the suggested answer to exercise 3.13. A major disadvantage of the evolutionary approach to prototyping is that long-term quality aspects (maintainability) tend to be neglected.

A major advantage of throwaway prototyping is that, once the real requirements are known, a thorough (architectural) design and implementation path can be followed, without distractions caused by on-the-fly change requests. A disadvantage of throwaway prototyping is that users get used to the prototype and may get disappointed when that prototype is discarded.

3.17 Software maintenance cannot be completely circumvented and, consequently, neither can the deterioration of system structure. There are two major ways to counteract this phenomenon. Firstly, maintenance should be done in a structured way, not in quick-fix mode. Changes should be properly designed, implemented, and documented. Changes should not be realized by code patches. Secondly, system structure should be monitored, and timely actions should be taken to regain a declining system structure. This is known as perfective maintenance. See also Chapter 14 for an elaborate discussion of maintenance issues.

3.18 At a global level, documents like a requirements specification or design description provide an inadequate measure of progress. This measure is too gross. Projects fall behind schedule one day at a time, and schedule slippage may thus show itself far too late. Also, real development does not occur in a strict linear order. Developers tend to jump from, say, requirements analysis to testing, and back.

A project can be broken into a number of subtasks. The granularity of the smallest subtasks should be fairly small (say at most one person-month). Examples of such subtasks could be: code module A, test integration of modules for subsystem X, review module B. The completion of some set of such subtasks then corresponds to a formal project milestone, but day-to-day control is executed at a much finer level.

Next to this type of control, experienced project managers use various other means to track progress, major ones being:

- Periodic status meetings with project members;
- Informal meetings with project members to get a subjective assessment of progress;
- Previous experience. Based on his experience with similar projects and/or members of the team, a project manager may for instance foresee that a certain subsystem is likely to pose problems, or know that Bill is generally overoptimistic in his estimates.

3.19 Main differences between RAD and PD are (see also Carmel et al. (1993)):

- The goals of RAD and PD are different. The main goal of RAD is to accelerate system development, whereas PD aims to accentuate the social context in which the system is to be developed and deployed.
- User participation is different. In RAD it is possible to have a few representatives of the (end) users on the design team. PD aims at consensus; the responsibility for the software development process lies with the users, so a few representatives won't suffice.
- RAD focuses on structure. It employs a number of well-defined techniques, such as workshops and timeboxing. PD does not employ a fixed set of techniques; it focuses on creativity, learning by doing.
- RAD concentrates on team building (the SWAT team). PD is focused on the mutual learning process of IT staff and users.
- RAD is concerned with speed (viz. the timebox). PD tries to reach its goals stepwise, irrespective of the timeframes.

## 4 Configuration Management

4.7 The major tasks of software configuration management (SCM) are the same during development and maintenance. In both cases, SCM is concerned with identifying and controlling changes. In both cases, it must ensure that changes are properly implemented and reported to interested parties.

Major differences between SCM during development and maintenance are:

- Most of the identification and definition of configuration items takes place during development;
- During development, change requests are issued by both developers and users. During maintenance, most change requests will come from users.
- During development, the assessment and handling of change requests is impacted by the necessary orderly progress of development. During maintenance, continuation of the system's operation is a major criterion.
- During maintenance, the operational baseline must be thoroughly separated from the version that is being changed because of bugs reported or changes that need to be incorporated. Thus, version control plays an even more important role during maintenance.

4.8 Software Configuration Management (SCM) is concerned with: identification, control, status accounting, and auditing. For these activities, the main differences between a traditional development model and an evolutionary model are (see also Bersoff and Davis (1991)):

- (identification) In an evolutionary model, there are many variants of components in use at the same time; in a traditional model, usually one or only a few such variants are in use. SCM must be able to distinguish between all those variants.
- (control) In evolutionary development, multiple versions of baselines are deployed at the same time, and multiple versions of components and baselines are under development simultaneously. In traditional development, there is one baseline and, usually, one version of each component.
- (status accounting) Is challenged as well in evolutionary development, with its multivariate product in various simultaneous stages of development and deployment.
- (auditing) Has to be done frequently and quickly in evolutionary development. Baselines change rapidly, and the results of audits must be promulgated timely to baselines in use and in development.

4.9 Artifacts like design documents and test reports can be subjected to the same configuration management procedures as source or object code modules. This can be supported by similar tools as well. In fact, integrated project support environments do so (see Chapter 15).

4.10 In a small development project, configuration management could be limited to SCCS-like support for handling the project's information. Formalized change procedures and a Configuration Control Board need not be established.

In a large project, formalized procedures are a *sine qua non*. There are too many people involved, and the number of items is too large, to be able to do without formalized schemes.

In Perry and Kaiser (1991), this difference in user scale is discussed in sociological terms. A small development project is compared with a family, where informal rules suffice, in general. A large project is likewise compared with a city, where the rules have to be more strict. See also Chapter 15.

4.11 Configuration management tools may keep track of the following quantitative data:

- Start and end dates of activities relating to configuration items (such as start and finish of implementation of a module). These data directly relate to project control.
- Number and type of bug reports and change requests. These data impact project control as well. They also relate to quality management; they may indicate poor quality components, and trigger subsequent maintenance activities.

## 5 People Management and Team Organization

5.9 A major advantage of having a true wizard in the team is that he may boost team productivity. Other team members may profit from his knowledge and increase their own knowledge and skills. Potential disadvantages (which should be counteracted by proper management attention!) are:

- Technical issues may get too much emphasis. The team may easily loose itself in beautiful technical solutions to problems that are hardly relevant to the users.
- A less disciplined mode of operation may result. Proper procedures regarding documentation and configuration control may be discarded, since the team guru has all the necessary knowledge in his head.
- The project may get into serious trouble if this person leaves the team. This holds especially if the previous point is not adequately dealt with.

5.11 Pros of a vertical organization are:

- By specializing in vertical areas like databases or human-computer interaction, team members may acquire greater expertise in these fields, which positively impacts the productivity and quality of their work.
- The similarity between and reuse across products is potentially enlarged. For instance, user interface components may become much more uniform.
- Career development can potentially be better monitored and stimulated.

Pros of a horizontal departmentalization include:

- Team members may acquire a better knowledge of the system. System knowledge is likely to be less thinly spread, and communication overhead may be smaller.
- Management has better accountability of people effort.



- Team spirit may be greater, since team members pursue the same (project) goals. In a vertical organization, team members have a partly interest in the success of any particular project they are involved in.

5.13 Advantages of letting people rotate between projects from different application domains are:

- Their "stock" of useful knowledge chunks increases.
- They are more easily led to properly document things, since their work has to be handed over to other people.
- They become less easily dissatisfied with their position, since they are regularly confronted with new challenges.

The major advantage of letting people become true experts in a given application domain is their increased expertise within that domain. The productivity and quality of their work increases as their knowledge of that domain grows.

## 6 Software Quality

6.12 The primary task of the SQA organization is to check whether work gets done the way it should be done. Suppose the SQA organization is not independent from the developing organization, and some serious problem crops up. From an SQA point of view, some remedial action is required, which may delay delivery of the system. Such is not very attractive to the developing organization. The SQA people may then get crunched between these opposing interests. The situation is like that of an accounting department who is responsible for its own auditing.

6.13 For any type of data collected, be it quality data, data on effort spent on design activities, data on the number of change requests received, etc., feedback to the suppliers of those data is important. The data supplier must be one of the main users of those data. Such forces the data supplier to provide accurate and complete input. He will harm himself if he does not do so. Secondly, it prevents these users from asking irrelevant data. Thirdly, if the data suppliers do not see the benefits of data collection (which is likely to occur if they do not get feedback), chances are that they do not appreciate the importance of accurate data collection either. Thus, the data collection process will deteriorate, and so does the value of the data collected.

6.17 Suppose one of the quality requirements is 'The system should be fast', a typical example of a non-quantified quality requirement. Such a requirement can not be tested; there is no way to tell whether the test 'succeeds' or not. Such a requirement also easily gives rise to debates later on, when the user complains that the system is not fast (enough). Such a requirement does not give the developer guidance either. Such a requirement is useless.

6.19 The easiest measurable property of a software product that may be assumed to relate to Modularity is module size. Larger modules tend to adversely affect system structure.

We may then impose some desirable upper bound on the size of a module. A measure  $S$ , defined as

$$S = 1 - (\text{nr of modules that exceed size} / \text{total number of modules})$$

then gives an indication of the extent to which this rule is obeyed. The value of  $S$  lies between 0 and 1. Larger values of  $S$  are assumed to reflect a better structure.

Other measurable properties that relate to Modularity are discussed in Sections 12.1.2 and 12.1.5. Section 12.1.2 mentions two structural design criteria: *cohesion* and *coupling*. Cohesion is the glue that keeps a module together. Grouping of components into modules in a haphazard way obviously leads to a bad system structure. Modules that encapsulate abstract data types are much better in this respect. Coupling is a measure for the strength of connections between modules. Modules that know much about each other's internal details are more tightly coupled than modules that can be viewed as independent entities. High cohesion and low coupling are considered to be desirable properties of system decomposition. We may therefore count the number of modules that fall into the various levels of cohesion and coupling as given in Section 12.1.2, and use this as a measure of system modularity.

Section 12.1.5 takes a look at the call graph, the graph that depicts the uses-relation of a set of modules. On the one hand, we may consider the form of this graph, and postulate that a tree-like pattern is better than a more general graph structure. A measure for this is the *tree impurity*, which expresses the degree to which the call graph deviates from a proper tree structure. On the other hand, we may look at the amount of information that flows through the edges of this graph. More information leads to tighter dependencies between modules. The *information flow* measure expresses these dependencies in numbers.

Operability (the ease of operation of the software) may be related to measurable properties such as (cf (Vincent et al., 1988, p 44)):

- All steps of the operation are described (normal and alternative flows);
- All error conditions and responses are appropriately described;
- There is a provision for the operator to interrupt, obtain status, save, modify, and continue operation;
- Number of operator actions reasonable (1 - time for actions/total time for job);
- Job set-up and tear-down procedures are described;
- Hard-copy log of interaction is maintained;
- Operator messages are consistent and responses standard.

Though these properties are highly desirable, most of them certainly do not guarantee easy operation. As discussed in Chapter 16, it is very important that user operations match concepts from the task domain. Well-established measures to express this do not really exist. What we can do though is to measure learning time of a system, and the time needed for typical tasks within the domain for which the system is needed. Tests with real users can express these in numbers, numbers that relate to the operability of the system. These numbers relate to all of the issues from the above list (if steps

of an operation are not fully described, it is likely that use of that operation becomes more difficult), but these numbers also reflect something of the ‘semantic’ dimension of operability.

Neither of these measures constitutes an objective criterion. The relations between properties measured and the corresponding quality criteria are still subjective. For example, a system in which a few modules have a size quite above what is considered a good standard may still be well modularized; see for example Redmond and Ah-Chuen (1990), cited in Section 12.1.4.

- 6.20 A small development organization most likely cannot afford full-time staffing of a separate SQA group. Global activities, like the drawing up of a Software Quality Assurance Plan, may be the collective responsibility of a small number of senior staff members. Thereafter, SQA activities may be assigned to individuals on a part-time basis. For instance, a team member from project A may be assigned the auditing task for project B, and vice versa.

In a larger organization, it is worthwhile to consider the establishment of a separate SQA group of, say, 3-5 people. The advantage of having a separate group is that specific SQA knowledge can be built up, that company-wide trends can be observed and global policies can be established, and that the SQA group may be a trigger in fostering quality consciousness in the organization.

For a proper understanding of the suggested answer, you should note that we assume the task of the SQA group to be an auditing one. I.e., the normal testing activities are not considered the responsibility of the SQA group.

- 6.22 The developer’s view of user-friendliness is likely to be determined by technical properties of the interface: use of windows, buttons, scrollbars, pop-up menus, etc. The developer is inclined to look at user-friendliness from the inside. On the other hand, the user’s main concern is to get his job done in the most effective way. User interfaces should not be friendly; they should effectively support the user at work.

Important ways to measure the usability of a system are:

- The time needed to learn to use the system (learnability),
- The time needed to perform typical user tasks (efficiency),
- The retention of system knowledge over time (memorability),
- The rate with which errors are made (safety), and
- A subjective assessment by real users.

Requirements for these characteristics can be expressed in measurable terms. Also, tests can be devised to determine whether these requirements are met. So-called ‘user-friendly’ systems may turn out to score well on these scales, but such is not a priori clear. See also Chapter 16.

- 6.26 The data show that the increase in the number of parties involved is almost completely caused by an increase in the involvement of *indirect* customers of the software to be developed: management and staff departments. There is no increase in the categories of people *directly* involved: the customers and the developers. So it may well be true

that the political coloring of cost estimates (see also Section 7.2) is even stronger in 1998 than it was in 1988. The situation then has become worse.

## 7 Cost Estimation

7.9 An early cost estimate gives a target to aim at. As such, it will influence the project. If we know that the project is estimated to cost 10 person months, we may be inclined to sacrifice quality in order to meet the corresponding deadline. Maintenance will then suffer. If a more realistic cost estimation were given, a higher-quality product could have been delivered, with corresponding savings during maintenance. Conversely, a tight effort estimate may force developers to ignore implementation of bells and whistles, resulting in a leaner system, a system which better fits real user needs.

7.13 Using the schedule equation of organic COCOMO ( $T = 2.5E^{0.38}$ ), the nominal development time of this project is approximately 14 months. A schedule compression of more than 100% (to 6 months) must be deemed very unrealistic, in view of the arguments given in Section 7.3.

7.14 Cost estimation models predict the cost of future projects, based on data from past projects. These predictions are only valid if future projects resemble past projects.

Development environments change. The types of systems developed change, people change (because of personnel turnover, learning effects and the like), tool usage changes over time, etc. Therefore, the cost estimation model should be recalibrated too, in order to make sure that the most accurate model of the present situation is used when estimating effort and time for future projects.

7.16 Both adding people to the project and softening quality requirements may shorten development time. Adding people to the project should be done with care (re Brooks' Law, discussed in Section 7.3). The impact of softening quality requirements on the time schedule could be estimated (for instance, several of these turn up as cost drivers in models like COCOMO).

Other ways to finish the project in time can be discerned by considering the various factors that influence cost (and, therefore, schedule):

- Write less code (reuse, use of high-level languages, concentration on essential features and ignoring bells and whistles), since size is the major determining cost factor;
- Employing better people;
- Better working environments and other incentives for employees. For example, DeMarco and Lister (1985) shows that programmers with adequate secretarial support and sufficient floorspace are significantly more productive than their colleagues that are worse off. See also DeMarco and Lister (1987);
- Employing (more powerful) tools;
- Avoiding rework, by a conscious attention to user requirements right from the start, and regular feedback to customers (validation).

7.17 Most likely, a cost estimation model based on COBOL as the implementation language cannot be used to estimate the cost of a project that uses Pascal. For one thing, systems written in Pascal often have different characteristics than systems written in COBOL; COBOL programs point at business applications, Pascal programs generally do not. Secondly, COBOL programs are more verbose than Pascal programs, so that a pure count of LOC provides a misleading estimate of the "intrinsic size" of the system. For instance, one function point (in the FPA context) is likely to incur more COBOL lines of code than its equivalent realized in Pascal. This effect has also been noted in COCOMO ((Boehm, 1981, p. 479)), where a significant pattern of overestimation was identified on COBOL programs.

This pattern is less likely to occur when the implementation language is C rather than Pascal. However, one has to proceed cautiously. A proper calibration of model parameters to the environment should underpin or falsify this hypothesis.

7.18 COCOMO 2 mentions three cost drivers that relate to project attributes:

- Use of software tools,
- Multi-site development, and
- Required development schedule.

The use of software tools allows the developer to concentrate on his real, intellectual tasks, while all kinds of bookkeeping duties are taken care of by the tools at his disposal. This would thus incur productivity gains.

If development takes place at more than one site, this incurs extra costs: for traveling, for written documentation instead of face to face communication, and the like. The chance for miscommunication, and thus for rework, increases as well.

The effort multipliers for the development schedule are somewhat more surprising: both acceleration and stretchout of the nominal schedule incur higher cost. That acceleration of the schedule incurs higher costs is quite plausible: it requires more people, with associated communication and learning cost. According to (Boehm, 1981, p. 470), stretchout of the schedule primarily implies spending a longer time with a smaller front-end team to thoroughly develop and validate requirements and specifications, test plans and draft users' manuals. This would then translate into higher-quality products and/or less maintenance.

It is interesting to note the differences in cost drivers between COCOMO and COCOMO 2. The multi-site cost driver was not present in COCOMO; apparently, multi-site development projects were not very common at that time. On the other hand, COCOMO had a cost driver 'use of modern programming practices' (in particular information hiding); this has become common practice, and its role as a cost driver has consequently disappeared.

Further corroboration of the values of the original effort multipliers of COCOMO, most of which are retained in COCOMO 2, together with pointers to supporting literature can be found in (Boehm, 1981, Chapters 24-27).

## 8 Project Planning And Control

8.12 The members of hospital staff may have a clear idea of the requirements of the planning system. If that is the case, a reasonably certain situation occurs, where the only problem could be the translation between the two domains involved (the hospital world and the planning world, respectively). Thereafter, the problem becomes one of realizing the functionality agreed upon, and a direct supervision style of management seems viable.

On the other hand, if hospital staff has no idea of what a planning system may do for them, a much more uncertain situation arises. A commitment style of management, in which the parties involved (hospital staff and analysts) search for the appropriate requirements (possibly involving the development of prototypes) then becomes the right choice. After such an initial stage, one may as yet switch to a direct supervision style of management for the later stages of the project.

8.13 The patient planning system by itself is not a very large system. As such, any team organization will not involve that many layers. In a hierarchical organization, we may for example distinguish small subteams concentrating on aspects like: the design of databases from which information on patients, operating rooms, clinical staff, and the like is obtained, the design of the user-interface part of the system, and the design of the actual planning part of the system. If this type of work division amongst subteams is chosen, the difference with a matrix-type organization is not that large. In both cases, the subteams are characterized by the specific expertise of their members. Points to be considered when the choice between the two options is still open, are:

- Is the project large enough to merit a separate subteam for, say, the user interface part;
- Is the extra dimension of a matrix-type organization warranted in a project of this size.

Most likely, the patient planning system is part of a larger system, and certain subsystems will be shared, such as those regarding the databases and user interfaces.

8.15 I would look for incentives to keep this employee motivated. Examples are:

- an increase in salary (though it is questionable whether this will be enough in the long run);
- assignment of more challenging tasks for part of his time (if this time is available);
- educational opportunities;
- assignment of an apprentice who may, after some time, take over some of his tasks.

Probably, some combination of the above works best.

8.17 If this situation arises, you are in real trouble. Neglecting the issue (delivering a system of inferior quality) is likely to be detrimental in the long run, and is therefore not recommended. Discussion with the client has to bring a solution (another delivery date after all, or delivery of the system without the faulty component, for example).

If I were a member and the manager is ignoring the signals, a nasty situation arises. A serious discussion with the manager might solve the issue. If it does not, my professional

ethics tell me to look for other ways, for example contact my next higher superior; see also Section 1.5.

## 9 Requirements Analysis

9.17 The environment of an elevator control system is likely to be fixed and stable, and there will be no conflicts as regards the requirements of the system. For an office information system, there is a much higher chance that the environment expresses contradictory requirements. The people in the environment may have different views of the system, their requirements may conflict, and their requirements are likely to change over time.

As a result, the major task during the requirements engineering phase of the elevator control system is to find out what the requirements are. A functionalist approach can be followed when doing so. For an office information system, a non-functionalist paradigm is more appropriate. Some prototyping or incremental development process may help to sort out the requirements. A flexible approach is needed, since the requirements of this type of system *will* change, whether we like it or not.

9.18 Stakeholders for an office information system could be:

1. the office workers themselves,
2. their manager, and
3. the client who pays the bill.

The client's main concern is likely to be his return on investments (ROI): how much the system is going to cost, and how much he will get back in return. This might translate into things like: a higher productivity of the office workers, less people to be employed, and other types of cost savings.

The office manager may want to use the system to ease his job by collecting administrative data on the tasks of his employees, he may want to use the system to assess the productivity of his workers (using the same set of data), and he may want the system to make sure that tasks get done more effectively, more uniformly, etc.

The office workers themselves are likely to emphasize the system as a tool in getting their work done. So the system should ease their job, take over the boring part of their work, but leave the challenging and rewarding aspects to the workers.

(Obviously, the positions of the various stakeholders need not be as black and white as sketched above. In most cases, they aren't.)

9.19 This mode of working is extensively discussed in Chapter 16; its essence is captured in figure 16.6.

9.20 In order to assess the pros and cons of various descriptive means for describing requirements, we have to consider the two major audiences for such descriptions: the user/client, who has to determine whether the requirements reflect his true needs, and the developer, who has to design and implement the system according to the requirements specification.

The user/client is best served by a description he is most familiar with: natural language and/or pictures. Such a description is more likely to fit his domain language. Obvious disadvantages are that such a description has a high chance of being ambiguous and incomplete. Regular interaction with the client organization after the requirements have been fixed then is often needed to resolve ambiguities or conflicts. It is a priori unclear to what extent these necessitate rework.

- 9.24 A hypertext-like browsing system for a technical library potentially offers features that are very unlike those offered by keyword-based retrieval systems. Therefore, a functionalist approach to requirements specification is likely to result in a system which does not utilize the full potential of hypertext or hypermedia systems. For example, users of a technical library in the field of aeronautics may search the library by selecting from a set of pictures of different types of planes or engines, or simulations of characteristics of different types of engines. A prototyping approach, in which applications of hypertext in other fields is used to assess its potential, is more likely to result in a system utilizing the extra capabilities of hypertext.
- 9.25 In many a technical field, there is quite some consensus as to the major concepts and issues involved. Therefore, we may assume that there are no major conflicts involved (though this need not always be the case). Given the unfamiliarity with hypertext, there will not be an immediate consensus as regards the features to be offered. The requirements analyst then has to facilitate the learning process within the client organization and a central question becomes one of how to exploit the unique capabilities of hypertext within this particular field.
- 9.27 Meyer (1985) contains a very elaborate and insightful discussion of this example.
- 9.28 See the answer to exercise 20 of this Chapter.

## 10 Modeling

- 10.9 The common role of a contract between a client and a contractor is described in Meyer (1992):
- It protects the client by specifying *how much* should be done. The client is entitled to receive a certain result.
  - It protects the contractor by specifying *how little* is acceptable. The contractor must not be liable for failing to carry out tasks outside the specified scope.<sup>1</sup>

In software, the relations between clients and contractors are specified in assertions, such as pre- and postconditions of routines, and class invariants for a whole class. For a routine, the precondition expresses requirements that should be satisfied upon calling. The postcondition expresses properties that will be true upon return. Together, they constitute a contract for the implementor of the routine.

---

<sup>1</sup>There is a potential caveat, though. Just like the buyer of a new car expects the roof of the car to be waterproof, even if this is not explicitly stated in the contract he signed, so will the software client expect certain ‘obvious’ qualities, even if not explicitly stated in the contract.