

Answers to Exercises  
SOFTWARE ENGINEERING: Principles and Practice  
Second Edition

Hans van Vliet  
Division of Mathematics and Computer Science  
Vrije Universiteit  
De Boelelaan 1081a, 1081 HV Amsterdam  
Email: hans@cs.vu.nl

October 26, 2000

## 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 14 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 current 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).

- 1.15 The issues raised in this question can also be illustrated through classroom projects.
- 1.17 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 11.1.4.
- D. Sloppy descriptions of artifacts lead to misunderstandings between developers, between developers and the client, etc. This results in errors and rework. See chapters 9 and 15.
- E. Things will change, whether you like it or not. A rigid software process can and will not be followed; see section 3.8.
- 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 11.1), 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 11 and 14.
- 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 chapter 10.
- 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.
- 2.7 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.

2.8 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

3.10 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.

3.11 The merits of evolutionary prototyping are listed in the suggested answer to exercise 3.10. 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.

3.12 If the notion of software factory is used to emphasize reusability across products, the software development process is impacted in two major ways. Firstly, the scope of the project is extended to include maintenance as well. This means that quality aspects that pertain to the operational life of a product (various constituents of maintainability) have

to be taken into account from the very beginning. In particular, total life cycle costs count, rather than the mere cost of initial development. Secondly, reusability across products is taken into account during the development process. During development, the reusability of parts of the product, as well as the possible reuse of existing parts, is a major guiding criterion (the so-called software-development-for-reuse model; see section 17.3).

If the notion of software factory is used in a more general sense, viz., that of an environment which allows software manufacturing organizations to design, program, test and maintain software products in a unified manner, the main emphasis is on standardization of procedures, policies, methods and techniques. As a consequence, the development process will have a rather structured, standardized form across projects, necessitating a very disciplined mode of operation. An integrated set of tools is often considered an essential ingredient of such a factory. These tools may enforce, or at least promote, the use of prescribed standards.

3.13 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.

3.14 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.

3.15 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 19).

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 19.

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

5.8 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.

5.10 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.

5.11 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.

- 6.16 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.
- 6.18 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 11.1.2 and 11.1.5. Section 11.1.2 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 11.1.2, and use this as a measure of system modularity.

Section 11.1.5 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 11.1.4.

- 6.19 According to figure 6.7, high Reusability is expected to result in high Maintainability, Testability, Flexibility and Portability. On the other hand, high Reusability is expected to result in low Reliability, Efficiency, and Integrity.

A highly reusable component will in general be tested quite thoroughly, and therefore require less maintenance and less testing when reused. Highly reusable components will in general embody machine-independent concepts, and are therefore portable; non-portable components are not very reusable either. Similarly, if components are to be reusable, they must by necessity be flexible to the extent such is needed for these components to be reusable at all.

Reliability (defined in this context as ‘the extent to which a program can be expected to perform its intended function with required precision’) necessitates a careful handling of exceptional conditions, and means to recover from incorrect input data and computational failures, which tends to incur a tight coupling between a component and the environment in which it is to be used. This in turn tends to impact its reusability in negative ways (but note that this need not always be the case; a careful handling of exceptions may also have a positive influence on the component’s reusability). Reusable components are required to be general, which incurs extra costs to cater for all possible cases; thus, efficiency is sacrificed. Similarly, the generality of reusable components tends to cause protection problems.

- 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.

- 6.25 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 7.1) is even stronger in 1998 than it was in 1988. The situation then has become worse.

## 7 Cost Estimation

- 7.10 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.
- 7.12 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 7.4.

7.13 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.

7.15 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 7.4). 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).

7.16 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.

7.17 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

8.11 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.

8.12 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.

8.14 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.

8.15 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

9.13 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.

9.14 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.)

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

9.16 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. Formal languages like VDM result in more precise specifications. However, these specifications are difficult to comprehend for non-experts. Subtle decisions made in the specification may then easily go unnoticed. Though the description may be very precise, it need not reflect the true user needs, which may again necessitate rework later on.

9.17 For both systems, I would favor a formal requirements specification. An alternative description in (constrained) natural language could be offered to the client. For an office information system, such an alternative description probably is a necessity.

9.20 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.

- 9.21 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.
- 9.23 [Meyer, 1985] contains a very elaborate and insightful discussion of this example.
- 9.24 See the answer to exercise 16 of this chapter.

## 10 Software architecture

10.2 The top-level design has to guide the developers in their work. The architecture has a wider scope and purpose:

- it it supports the communication with **all** stakeholders, not only the developers
- it captures early design decisions, for example on behalf of early evaluation
- it is a transferable abstraction, and as such its role surpasses the present project or system.

10.10 First, the results of design have to be communicated to various parties: the developers, clients, testers, maintainers, and so on. For this communication to be effective, its language must have a clear semantics. Second, design elements correspond to concepts in the application or solution domain. If these concepts are known to the parties involved by simple labels, these labels will in due time serve as representations of these knowledge chunks, and thus improve effective communication about the design.

Like sine and cosine are well-known concepts from the language of mathematics, so are quicksort and B-tree in the language of computer science. At the design level, the factory pattern, MVC and the implicit-invocation architectural style represent such knowledge chunks. From an application domain, say finance, concepts like ledger result.

10.15 The development organization of the World Wide Web (the original version) is CERN, the European Laboratory for Particle Physics. The (single) developer was Tim Berners-Lee, a researcher with a background in internet and hypertext. His aim was a system to support the informal communication between physics researchers. He anticipated a weak notion of central control, as in the then existing internet, and not uncommon in a research environment. The main requirements were: remote access (the researchers should be able to communicate from their own research labs), interoperability (they used a variety of hardware and software), extensibility and scalability. Data display was assumed to be ASCII; graphics was considered optional.

These requirements were met through libWWW, a library that masks hardware, operating system, and protocol dependencies. libWWW is a compact, portable library; it

is used to create clients, servers, databases, etc. It is organized into five layers, from ‘generic utilities’ to ‘application module’. The libWWW-based client and server communicate with each other using HTTP, and with the consumer and producer through HTML. For further details, see [Bass *et al.*, 1998, chapter7].

10.16 Patterns and architectural styles may guide us during software comprehension, more or less like programming plans do in programming and debugging. If the reader knows that the proxy pattern is used, such guides him in building a model of what the software does. Usually, such information will not be obvious from the code, but be indicated in the documentation. See also the answer to exercise 10, and section 14.2.

10.19 First, design patterns embody best practices. These best practices have stood the test of time; they constitute proven solutions to recurring problems. Second, many design patterns explicitly address specific quality issues. For example, separation of concerns (flexibility) is addressed in the proxy pattern, while expandability is addressed in the factory pattern.

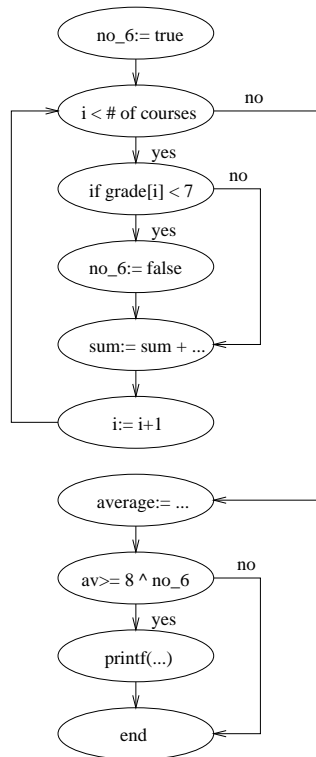


Figure 1: Control flow graph of the ‘grading program’

## 11 Software Design

11.7 The control flow graph of this program is given in figure 1. The graph contains 10 nodes and 12 edges, so  $n = 10$ ,  $e = 12$ , and  $p = 1$ . The cyclomatic complexity then

is 4 (which concurs with the intuitive notion of cyclomatic complexity: the number of decisions +1).

- 11.8 The bottom part of the control flow graph for this version is given in figure 2. The full control flow graph now has 11 nodes, and 14 edges, so the cyclomatic complexity now is 5. The difference with the answer to the previous exercise is caused by the fact that the compound boolean expression is treated as one decision in exercise 7, and as two decisions in this exercise. Of course, this does not obey the representation condition.

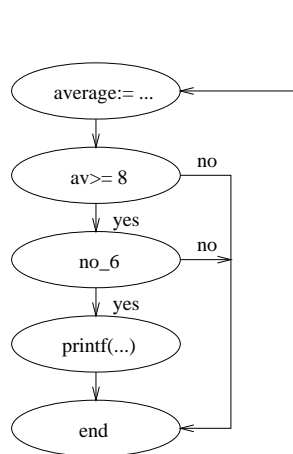


Figure 2: Control flow graph of ‘simple’ if statements

- 11.10 Cyclomatic complexity is not really a good indicator of system complexity. For one thing, it concentrates on measuring intra-modular complexity, by counting the number of decisions made. For a system as a whole, the information flow between modules (the amount and kind of data passed to and from modules) is a major determinant of system complexity as well.
- 11.11 The call graphs for the first two decompositions from section 10.1 are given in figures 3 and 4. These call graphs closely resemble figure 10.1 and 10.2, respectively. For the call graph of figure 3, the number of edges and nodes equals 4 and 5. The tree impurity metric then equals 0. This is what we would expect, since the call graph is a tree. For the graph in figure 4, the number of edges and nodes equals 8 and 6. The corresponding tree impurity metric therefore equals 0.3.

These numbers do not really fit our intuitive idea of the quality of the decomposition. If we just rely on the tree impurity metric, the main-program-with-subroutines decomposition should be considered perfect. Yet, the information flow between the modules from this decomposition is rather complex, resulting in a tight coupling between those modules. In the abstract-data-type decomposition, various abstract data types (**Store** and **Shift**) are used by more than one other module. This increases the tree impurity metric, yet may be considered good design practice.

- 11.12 Using Shepperd’s definitions, and *ignoring duplicate flows*, the fan-in and fan-out of the first two modularizations are given in figures 5 and 6 below. Note that in both the

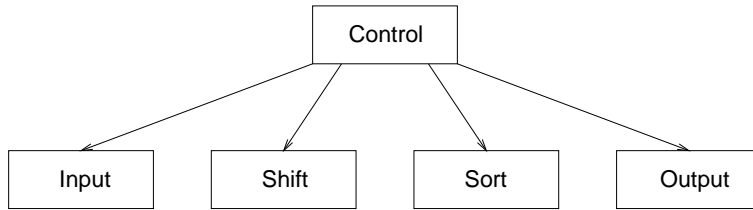


Figure 3: Call graph for the first modularization

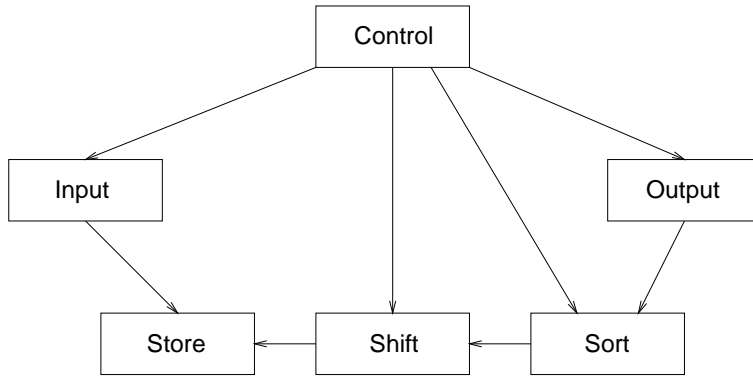


Figure 4: Call graph for the second modularization

Shepperd and Henri&Kafura article the term ‘module’ really means ‘procedure’. In the decompositions dealt with here, modules denote collections of procedures. This might make a difference in how we count flows.

Since the main-program-with-subroutines modularization only uses communication via global datastructures, the fan-in and fan-out of its modules is solely determined by the number of global variables read and updated. Likewise, the abstract-data-type modularization only uses procedure calls, so the fan-in and fan-out of its modules is solely determined by the parameters and return values used. For example, the number of (direct) local flows from module `Sort` to module `Shift` in the abstract-data-type modularization equals 3, because of the following types of information passed from `Sort` to `Shift`:

- the index (number) of a given shift,
- the index (number) of a given word in a given shift,
- the index (number) of a given character position in a given word in a given shift,

Likewise, the number of (direct) local flows from `Shift` to `Sort` equals 4, because of the following types of information passed from `Shift` to `Sort`:

- the total number of shifts,
- the number of words in a given shift,
- the number of characters of a given word in a given shift,

	fan-in (sink)	fan-out (source)
Control	0	0
Input	0	3
Shift	1	1
Sort	2	1
Output	2	0

Figure 5: Fan-in and fan-out of modules from the main-program-with-subroutines decomposition

	Control	Store	Input	Shift	Sort	Output	fan-out (source)
Control	–	0	0	0	0	0	0
Store	0	–	0	4	0	0	4
Input	0	4	–	0	0	0	4
Shift	0	3	0	–	4	0	7
Sort	0	0	0	3	–	1	4
Output	0	0	0	0	1	–	1
fan-in (sink)	0	7	0	7	5	1	

Figure 6: Fan-in and fan-out of modules from the abstract-data-type decomposition

– the character stored at a given position.

Using the numbers from figures 5 and 6, the complexity of the corresponding modules is as given in figure 7 (ignoring the length component, as suggested by Shepperd). These numbers do not correspond to our intuitive understanding of the complexity of the two modularizations. This is due to the fact that the flows in the main-program-with-subroutines modularization concern complicated global variables, while the flows in the abstract-data-type modularization concern simple data structures (mostly integers).

	Complexity	
	Modularization 1	Modularization 2
Control	0	0
Store	–	$28^2 = 784$
Input	0	0
Shift	1	$49^2 = 2401$
Sort	2	$20^2 = 400$
Output	0	1
Total:	3	3586

Figure 7: Information-flow metrics for the two modularizations

11.17 Abstraction: the main-program-with-subroutines decomposition uses procedural abstraction, the abstract-data-type decomposition uses data abstraction (mostly).

Modularity: when defined as coupling and cohesion levels: see exercise 18.

Information hiding: information hiding is not used in the main-program-with-subroutines decomposition. The abstract data types in the adt decomposition do have a secret, viz. the data they implement.

Complexity: of individual modules is difficult to assess without having access to the code, but should be OK for both modularizations.

System structure: the tree impurity metric is given in exercise 11, the information flow metric in exercise 12.

11.18 The cohesion and coupling levels for the main-program-with-subroutines decomposition and the abstract data type decomposition are reflected in figures 8 and 9..

Module	Cohesion type	Coupling type
Input	functional	common
Shift	functional	common
Sort	functional	common
Output	functional	common
Control	sequential	common

Figure 8: Cohesion and coupling levels main-program-with-subroutines decomposition

Module	Cohesion type	Coupling type
Store	data	data
Input	functional	control
Shift	data	data
Sort	functional	control
Output	functional	control
Control	sequential	control

Figure 9: Cohesion and coupling levels abstract-data-type decomposition

11.19 The flow graph of this program is given in figure 10. From this flow graph, it follows that  $n = 11, e = 11, p = 2$ , so  $e - n + p + 1 = 3$  and  $e - n - 2p = 4$ .

The flow graph of the same program, with procedure P drawn inline, is given in figure 11. Now,  $n = 8, e = 9, p = 1$ , so  $e - n + p + 1 = 3$ , and  $e - n + 2p = 3$ .

We would expect the outcome for the two versions of this program to be the same (and, since the program contains two decisions, the answer should be 3). Most text book discussions of cyclomatic complexity give the wrong formula, and use examples consisting of one component only. In that case the outcome of both formulas is the same.

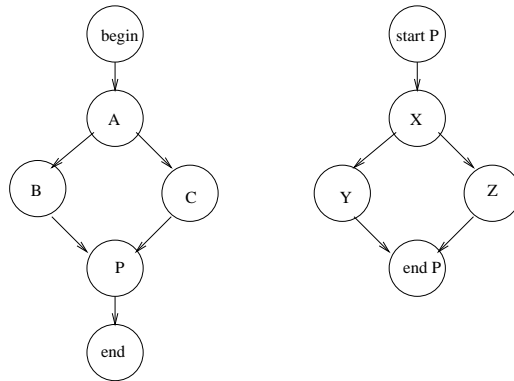


Figure 10: Flow graph with separate procedure P

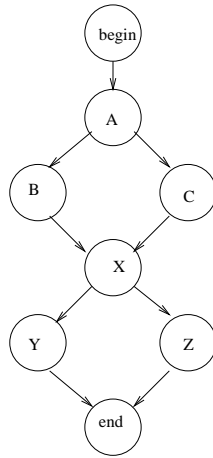


Figure 11: Flow graph with inline procedure P

11.24 A tree impurity metric  $m$  measures the extent to which a call graph  $G$  deviates from a proper tree. It is then only natural to expect that  $m(G)$  equals 0 if and only if  $G$  is a tree (property 1).

If two call graphs have the same number of nodes, the one with the larger number of edges is to be considered worse, since it deviates more from a proper tree structure (more edges have to be removed in order to get a proper tree) (property 2).

If the number of edges that has to be removed in order to get a proper tree structure is the same for two graphs  $G_1$  and  $G_2$ , but the number of nodes in  $G_1$  is larger than that of  $G_2$  then, relatively speaking,  $G_1$  is better than  $G_2$ . The penalty of having a few extra edges should be relative to the number of nodes in the graph (property 3).

Finally, the worst possible situation occurs if each pair of nodes in the graph is connected through an edge. In that case, the graph is called complete (property 4). The upperbound 1 is somewhat arbitrary; it could have been any constant.

## 12 Object-Oriented Analysis and Design

12.12 The central idea behind information hiding is that a module hides a secret. This secret could be the representation of some abstract data type, but it could be something else as well. So, the result of information hiding need not be the encapsulation of an object and its operations. Furthermore, the classes resulting from an object-oriented design are related through a subclass/superclass hierarchy (inheritance). Inheritance does not result from the application of the information hiding principle.

12.13 The ‘extra’ part is given in figure 12. The objects client and library are the same as the corresponding objects in figure 12.15.

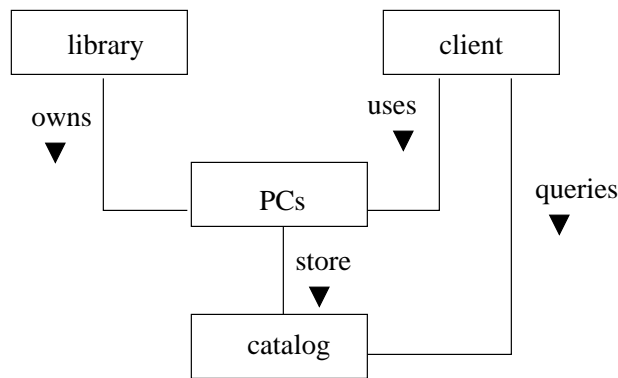


Figure 12: Part of the object model: handling of user queries

12.14 The main objects from this model are: Client, Employee and BookCopy. We then assume that the ownership of computers, bar code readers and the like is not modeled in the system. The identification card is assumed not to play an active role either; it is simply used to identify the client. We assume there is only one library.

The client may have attributes Name, Address, BooksOnLoan, and Fine. BooksOnLoan is a simple count of the number of books this client has loaned. Fine is account of the outstanding fines for this client. Client has the following services: BecomeMember, ChangeAddress, AddToFine, SettleFine, LoanBook, ReturnBook, CeaseToBeMember. If a book is returned whose due-back date has passed, AddToFine updates the account of his fine. If (part of) the fine is settled, SettleFine takes care of that.

An employee has an EmployeeName and Password. Its main services are BecomeEmployee, ChangePassword, and CeaseToBeEmployee.

A book copy is identified by its Number. It has an attribute RefBook which refers to the book this one is a copy of. Furthermore, each book copy has attributes OnLoanSince and OnLoanTo. These attributes are updated when the copy is loaned and returned; they are also used to update the fine administration. Services of a book copy are: GetAcquired, GetLoaned, GetReturned and GetDiscarded.

12.16 The common role of a contract between a client and a contractor is as follows [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.

When P is a subclass of Q, certain features of Q may be redefined in P. Following the contract metaphor, the effect of such a redefinition must stay within the realm of the original definition. This means that the precondition may be replaced only by a weaker one (i.e. fewer requirements to the client), while the postcondition may only be replaced by a stronger one (i.e. more properties hold upon return). If P redefines certain features of Q, we request that the subcontractor is able to do the job better or cheaper, i.e. with a weaker precondition. And the subcontractor should at least do the job requested, i.e. have a stronger postcondition.

Note that most object-oriented languages do not enforce these restrictions in their inheritance mechanism.

12.17 The KWIC index example suggests various obvious object classes: `Word` (a sequence of characters), `Line` (a sequence of words), and `Text` (a sequence of lines). These classes essentially are all lists. There are quite a few ways to realize the interplay between such a class, say `Line`, and a general class `List`, which probably already exists in some library. Some of the possibilities are:

- `Line` could be made a subclass of `List`. However, `Line` is not a proper subclass of `List` in a generalization/specialization type hierarchy. Semantically, `Line` and `List` are unrelated.
- To circumvent this improper subtype relation, `Line` could be made a *private* subclass of `List` (in C++ terminology). The inheritance relation then would not show up in the public inheritance structure.
- `List` could be used to merely implement `Line`, i.e. the implementation of `Line` is delegated to `List`. `Line` then shares code with `List`.

The latter possibility is preferred. If class `List` is generic, `Line` could declare an instance of `List`, as in

```
class Line: {
private: List <Word> l;
...
}
```

`Word`, `Line` and `Text` would have a number of similar attributes (such as their number of elements) and operations (such as an operation to add an element at the end of the

---

<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.

list). These operations can be inherited from `List`, and additional operations can be added where needed.

After the input has been read and stored, shifts have to be determined. One way to do so is to include an operation `Shift(n)` in `Line`. `Shift(n)` shifts the object over `n` words. A subclass `Shifts` of `Text` would then obtain these shifts by successive calls of `Shift(n)` for each of the lines contained in the text.

Next, the shifts have to be sorted. This requires a sorting routine and an operation to compare two lines. In order to be able to compare lines, lines must be elements from some ordered set. We may then view `Line` as a subclass of an (abstract) class `Sortable` (providing abstract operations to compare elements). The abstract sorting routine, probably available from some library, would then use the abstract operations from the class `Sortable`. Finally, a redefinition of an operator `LessThan` is required within `Line`. The construction of shifts and their subsequent sorting can be combined into one operation `Process` within `Shifts`. Possible refinements of this processing, such as the removal of shifts starting with some irrelevant word, is then easily realized by redefining `Process`.

Finally, a "main" program is needed. This main program invokes a read routine from `Text`, the routine `Process`, and an output routine from `Shifts`.

The system thus obtained contains three classes, each of which rely on an underlying list-class for their internal representation, a specialization of `Text` to realize shifts, and a simple main program. On behalf of the sorting operation, `Line` is viewed as a subclass of `Sortable`, thus allowing reuse of a general sorting operation.

- 12.18 Each of the four main classes from this design applies data abstraction. Each of them in turn relies on a more general list structure for their implementation. The sorting operation is a very general one, whereby only the comparison operation is defined for the objects to be sorted.

The modules distinguished exhibit both data cohesion and data coupling, and thus provide for very simple interfaces.

As a consequence, each module hides a secret (data representation) to its users.

Complexity metrics, both at the module and system level, cannot easily be determined. The metrics discussed in sections 11.1.4 and 11.1.5 do not really apply to object-oriented designs. Given the information contained in the answer to exercise 17, we can make a reasonable estimate of the object-oriented metrics as defined in section 12.4. The value of WMC will be fairly small. DIT has value 1, if we do not count inheritance from library classes. In that case, only `Text` has a subtype `Shifts`. For the same reason, NOC has value 1. CBO is largest for class `Text`. This class is coupled with all other classes of the system, so its value is 3. RFC will have a small value, and so will LCOM.

- 12.19 A central issue in object-oriented design is that concepts from the Universe of Discourse have a direct counterpart in the design. Object-oriented design results in a model of the problem, rather than any particular solution to that problem. Conversely, data-flow design is much more concerned with modeling a solution, in terms of functions to be performed in some given order. These functions need not map well onto problem domain

concepts, which may result in a less natural design, a design which poses comprehension problems when studying the rationale for certain design decisions.

This difference is not all that apparent if we look at the decompositions arrived at in section 10.1 and exercise 17 above, although the main-program-with-subroutines decomposition is strongly geared towards the steps to be taken in a particular solution (first read the input, then compute shifts, etc.).

## 13 Software Testing

13.9 With a slightly different lay-out, and with linenumbers added to lines containing executable statements, the body of the routine reads as given below. Note that we have not labeled the lines containing exit statements. We could have done so, but it does not really make a difference as far as the control flow graph is concerned. Execution of such a statement incurs execution of the statement at line 8 as well.

```
1  parent:= k; child:= k + k; Ak:= A[k]; insert:= Ak;
   loop
2  if child > n
       then exit
       end
3  if child < n then
4       if A[child] > A[child+1]
5       then child:= child + 1
       end
       end;
6  if insert <= A[child] then
       exit else
7       A[parent]:= A[child]; parent:= child; child:= child + child
       end
       end;
8  A[parent]:= Ak
```

The corresponding control flow graph is depicted in figure 13. The numbers inside the bubbles refer to the linenumbers given in the routine text. On behalf of exercise 10, the edges have been labelled with capital letters. Outgoing edges of decision statements are labelled "yes" and "no".

When the routine is executed with the given input, all lines labeled with a number will be executed. A 100% statement coverage is therefore obtained.

13.10 The single test case given in exercise 9 does not yield a 100% branch coverage. In particular, the branches labelled E, G, and I will not be executed by this test. The following additional test case will do for a 100% branch coverage:

```
n = 4, k = 1.
A[1] = 60, A[2] = 20, A[3] = 30, A[4] = 80
```

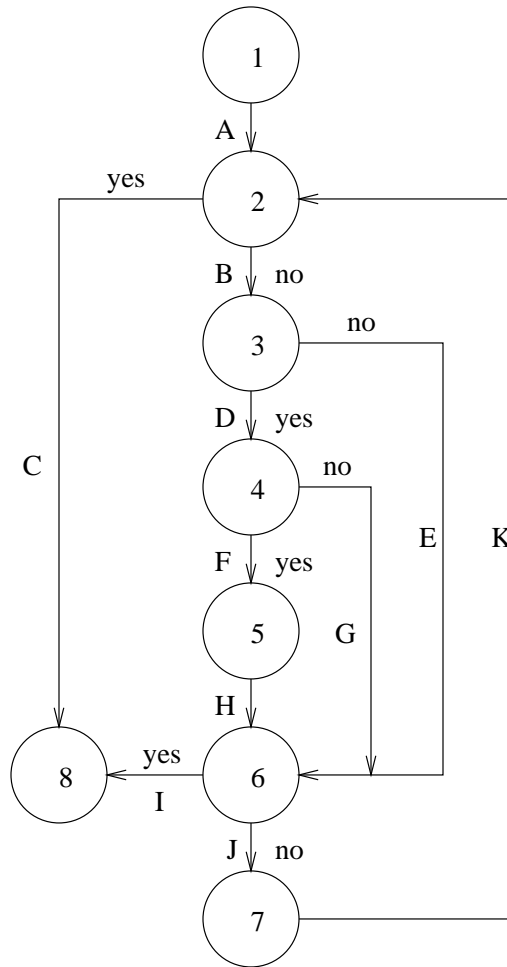


Figure 13: Control flow graph of the SiftDown routine

13.11 For the All-Uses coverage, a path from each definition of a variable to each P- or C-use and each successor of that use must be traversed. The program has four variables: `parent`, `child`, `Ak`, and `insert`. By carefully looking at the text, it can be observed that we only have to look at definitions and uses of `child`, since the definition-use paths of the other variables are "subsumed" by those of `child`.

This leads to the following paths:

1.  $1 \rightarrow 2 \rightarrow 8$   
A test  $n = 1, k = 1, A[1] = \text{any value}$  will do.
2.  $1 \rightarrow 2 \rightarrow 3 \rightarrow 6$   
A test  $n = 2, k = 1, A[1] = \text{any value}, A[2] = \text{any value}$  will do.
3.  $1 \rightarrow 2 \rightarrow 3 \rightarrow 4 \rightarrow 5 \rightarrow 6 \rightarrow 8$   
A test  $n = 3, k = 1, A[1] = 1, A[2] = 3, A[3] = 2$  will do.
4.  $1 \rightarrow 2 \rightarrow 3 \rightarrow 4 \rightarrow 5 \rightarrow 6 \rightarrow 7$   
A test  $n = 3, k = 1, A[1] = 4, A[2] = 3, A[3] = 2$  will do.

5.  $1 \rightarrow 2 \rightarrow 3 \rightarrow 4 \rightarrow 6 \rightarrow 7$   
A test  $n = 3, k = 1, A[1] = 1, A[2] = 2, A[3] = 3$  will do.
6.  $1 \rightarrow 2 \rightarrow 3 \rightarrow 4 \rightarrow 6 \rightarrow 8$   
A test  $n = 3, k = 1, A[1] = 4, A[2] = 2, A[3] = 3$  will do.
7.  $7 \rightarrow 2 \rightarrow 3$   
A test  $n = 4, k = 1, A[1] - A[4]$  having any values will do.

13.12 The only difference between the two fragments is that the first one uses an if-statement to see whether the element sought has been found, while the second one does so through an assignment. If we have a test set with a 100% branch coverage for the first fragment (i.e. both possible outcomes of the test `table[counter] = element` are tried at least once), a 100% branch coverage is also achieved for fragment 2. Note that the reverse need not be true.

13.20 The inner loop (plus the initialization `small := k`), results in an index `small` such that `r[small]` is smallest amongst `r[k] .. r[n]`. If we next swap `r[k]` and `r[small]`, the result will be:

$$\forall j \in \{k+1, \dots, n\} : r[k] \leq r[j]$$

Successive executions of the body of the outer loop will then yield:

$$\begin{aligned} &\forall j \in \{2, \dots, n\} : r[1] \leq r[j] \text{ (i.e. } r[1] \text{ is the smallest element),} \\ &\forall j \in \{3, \dots, n\} : r[2] \leq r[j] \text{ (i.e. } r[2] \text{ is the next smallest element),} \\ &\text{etcetera, i.e. the whole array gets sorted in increasing order} \end{aligned}$$

Finally, the if-statement at the outermost level simply tests whether there is anything to sort.

- 13.21
1. Change the first if-statement into **if  $n = 1$  then**. Only the array with random elements will (probably) give the wrong result.
  2. Change the first if-statement into **if  $n < 1$  then**. Again, the array with random elements will be the only one to give the wrong result.
  3. Change the first if-statement into **if  $n > 2$  then**. All tests will yield the right answer.
  4. Change the assignment `small := k` into `small := n`. The array with random elements will probably give the wrong result.
  5. Change the test in the second if-statement into `r[j] > r[small]`. Now the array will be sorted in reverse order, so the sorted array and the array with random numbers will give the wrong results.
  6. Change the test in the second if-statement into `r[j] = r[small]`. The array with random elements will probably give the wrong result.
  7. Change the test in the second if-statement into `r[k] < r[small]`. No swapping will take place, so the random array again gives the wrong result.
  8. Change the second for-statement into **for  $j := k + 0$  to  $n$  do**. All tests will yield the right answer.

9. As the final mutant, we change the swap-statement into `swap(r[k], r[n])`. The sorted array and the array with random elements will probably give the wrong answer.

So this test set leaves us with 2 live mutants. This means that the quality of the test set is quite high. Note that the situation becomes really worse if we drop the array with random numbers. The quality of this test set is really determined by the latter test. Most of the other tests only exercise some boundary value.

13.22 The antidecomposition axiom says that if some component is adequately tested in one environment, this does not imply that it is adequately tested for some other environment. Consider a sorting algorithm that has been tested in some environment, resulting in a 100% statement/branch coverage. It may well be that the input to the sorting algorithm always happens to be somewhat peculiar (say, only positive integers), and that nevertheless every statement/branch of the algorithm is tested. Statement or branch coverage for the sorting part is then adequate too, while it may well contain errors that are revealed if the algorithm is included in some other environment.

The anticomposition axiom reflects the opposite: if components have been tested adequately, their composition need not be tested adequately. Suppose we have a component P which is statement/branch tested using some test set T, and the output of P for T is some set T'. Suppose furthermore that Q is statement/branch tested using T'. Then, T is always 100% statement/branch tested for P;Q as well.

- 13.24
- Functional/structural testing: a good method for fault finding, because of their structured approach to covering the input domain and program domain, respectively. Experiments suggest that these methods tend to find different types of faults (see section 13.8.3). Both methods are less suited for confidence building, for two reasons. For both methods, the testing quality hinges on the quality of the equivalence partitioning, which is often not perfect. Secondly, they treat every fault equally hazardous, which need not be the case during operation of the system.
  - Correctness proofs: the major problem with correctness proofs is that the program is proved correct with respect to some other formal description. Under the assumption that this other description is correct, correctness proofs are both good at finding faults and increasing our confidence. Additional testing is needed to cater for the possible imperfectness of the formal description against which the proof is given (and possible errors made in the proof itself).
  - Random testing: may be somewhat less strong in finding faults, since the method concentrates on the operational profile of the system, thereby paying little attention to those parts that are hardly ever used. Experiments suggest that it is quite good at building confidence.
  - Inspections: the strengths and weaknesses of inspections are similar to those of functional and structural test methods. Again, experiments suggest that inspections reveal different types of errors.

13.25 The major difference with other types of reviews is the direct user involvement, which may strengthen their commitment with the project. Discussion of possible usage scenarios also has a stronger validation character than other types of review.

## 14 Software Maintenance

- 14.7 See section 14.2.1. The key idea is that design recovery requires *outside* information, information that has gone lost during the transition from requirements to design, or design to code.
- 14.8 During maintenance, changes are made to existing artifacts (design descriptions, code, etc). This results in different versions of those documents. Software configuration management helps keep track of revision histories and versions. Older versions remain available, so that changes can be undone, and the revision history itself can be of help during maintenance. Additional support for building executables both optimizes this process (unchanged components need not be compiled anew) and helps to get the right executables (those that contain the most recent version).
- 14.9 The major role of an acceptance test by the maintenance organization prior to the release of a system would be to assess the maintainability of the system that is about to become operational. Such a test will then pay particular attention to aspects that are relevant during maintenance: the quality of the technical documentation, the structure and complexity of individual components as well as the system as a whole, the reliability of the system. The structure of such an organization could be similar to that of other test groups. In particular, the future maintainers should be represented.
- 14.10 Maintenance concerns all work on a system after it has become operational. We may differentiate between maintenance and development by taking the end-user point of view: work is considered development insofar as it concerns the offering of new possibilities to them. Everything else is maintenance.
- Since development from scratch is the exception rather than the rule, the distinction between development and perfective maintenance easily gets blurred. We might say that ‘real’ maintenance concerns all activities that do not change the functionality of the system, while development concerns the addition of functionality.
- The classification of development and maintenance activities as given in the exercise does make this careful distinction between adding functionality, and everything else.
- 14.11 Software maintenance requires understanding of the software to be maintained. Explicit codification of this knowledge, and subsequent help in browsing through the resulting network of knowledge chunks, offers additional support over other means to acquire that knowledge (documentation, design information, and the like are essentially linear organizations of this knowledge, from which subtle interactions and mutual dependencies are hard to distill).
- 14.15 If components are reused, they will generally be of a higher quality than newly developed parts. They have simply stood the test of time. This in itself should have a positive impact on (corrective) maintenance effort. Secondly, reused components are likely to be more stable; they reflect the growing understanding of domain concepts. This in turn should positively impact perfective maintenance effort.
- 14.16 Object-oriented software development emphasizes the modeling of a problem, rather than any particular solution to that problem. The structure of the resulting system

should then better reflect the structure of the problem domain. Stable entities are the focus of attention, and volatile functionality is implemented through operations on objects. This should help to reduce maintenance effort. At the code level, systems written in OO languages tend to be shorter because of the code sharing that results from inheritance. Smaller programs require less maintenance. Finally, changes to programs can be realized through subclasses, rather than through tinkering with existing code.

On the negative side: OO programs may be more difficult to comprehend, because the thread of control is more difficult to discern. The inheritance mechanism may make it more difficult to decide which parts of the system ‘apply’ at a given point. Excessive use of inheritance is likely to make the system really difficult to comprehend and maintain, hence the slogan ‘Inheritance is the goto of the 1990’s’.

14.17 Changing a program involves three subtasks:

1. comprehending the program,
2. making the change, and
3. retesting the program.

Subtasks 1 and 3 require an effort proportional to the length of the program; this effort is hardly, if at all, affected by the size of the change. Subtask 2 may be expected to incur a cost proportional to the size of the change. If the cost of activity  $i$  is  $C_i$  per line of code, then a 10% change in a 200 LOC program is the more costly one if:

$$200C_1 + 20C_2 + 200C_3 > 100C_1 + 20C_2 + 100C_3$$

This is true for any nonnegative value of  $C_1$  to  $C_3$ .

## 15 Formal Specification

15.9 Since `Add` only adds new entries to the list under certain constraints, we need a hidden function, say `Add1`, to do the primitive addition of an element to the list. We therefore add this function to the signature:

`Add1: List × Course × Points × Score → List`

The axioms may then read as follows (we assume an operator `max` which yields the maximum of two values of type `Score`, as well as some obvious relational and arithmetic operators):

```
Add(Create, C, P, S) =
  if S <= 50 then Create else Add1(Create, C, P, S)
Add(Add1(L, C1, P1, S1), C2, P2, S2) =
  if S2 <= 50 then Add1(L, C1, P1, S1)
  else if C1 = C2 then Add1(L, C1, P1, max(S1, S2))
  else Add1(Add(L, C2, P2, S2), C1, P1, S1)
GradePoints(Create) = 0
GradePoints(Add1(L, C, P, S)) = P + GradePoints(L)
PartyTime(L) = GradePoints(L) > 126
```

15.10 No, for this example final semantics would necessitate the same axioms.

15.11 In the following, the axioms have been numbered. This numbering is referred to in the solution to exercise 12. We have to introduce one extra function, called `Substring1`, in order to be able to give the axioms for `Substring`. `Substring1(S1, S2)` yields true iff `S2` is the tail part of `S1`.

- 1 `Delete(Create, n) = Create`
- 2 `Delete(Append(S, c), n) =`  
     if `n = 1` then `S` else `Append>Delete(S, n-1), c)`
- 3 `Substring(S, Create) = true`
- 4 `Substring(Create, Append(S, c)) = false`
- 5 `Substring(Append(S1, c1), Append(S2, c2)) =`  
     `Substring(S1, Append(S2, c2))` or  
     (`c1 = c2` and `Substring1(S1, S2)`)
- 6 `Substring1(S, Create) = true`
- 7 `Substring1(Create, Append(S, c)) = false`
- 8 `Substring1(Append(S1, c1), Append(S2, c2)) =`  
     `c1 = c2` and `Substring1(S1, S2)`

15.12 In order to somewhat shorten the text, we have abbreviated `Substring`, `Append` and `Create` to `S`, `A` and `C`, respectively. The proof then runs as follows:

$$\begin{aligned}
 & S(A(A(A(A(C, a), a), b), a), A(A(C, a), a)) \\
 = & \text{(axiom 5, first alternative of righthand side)} \\
 & S(A(A(A(C, a), a), b), A(A(C, a), a)) \\
 = & \text{(axiom 5, first alternative of righthand side)} \\
 & S(A(A(C, a), a), A(A(C, a), a)) \\
 = & \text{(axiom 5, second alternative of righthand side)} \\
 & a = a \text{ and } S1(A(C, a), A(C, a)) \\
 & \quad = \text{(axiom 8)} \\
 & a = a \text{ and } (a = a \text{ and } S1(C, C)) \\
 & \quad = \text{(axiom 6)} \\
 & a = a \text{ and } (a = a \text{ and true)} \\
 & \quad = \text{true}
 \end{aligned}$$

15.13 For the sequence type, we use the following notations:

- `()` denotes the empty sequence
- `~` denotes concatenation of two sequences
- `s[i]` denotes the *i*-th element of `s`. We assume that elements are numbered from 1 onwards.
- `s[i .. j]` denotes the subsequence of `s` containing elements `s[i]`, `s[i+1]`, `...`, `s[j]`. This subsequence is empty if `j < i`.
- `length(s)` denotes the number of elements in `s`.

In postconditions, a postfix accent (`'`) denotes the value of an object prior to the execution of the function (as is also done in, e.g. `Alphard`).

```

proc Create(var s: string);
  pre: true
  post: s = ()

proc Append(var s: string; c: char);
  pre: true
  post: s = s' ~

proc Delete(var s: string; i: integer);
  pre: true
  post: if i < 1 or i > length(s') then s = s'
        else s = s'[1 .. i-1] ~ s'[i+1 .. length(s')]

proc Substring(s1, s2: string): boolean;
  pre: true
  post: if  $\exists i, j: s1[i .. j] = s2$  then true else false

```

15.14 If only set operations are available, we must have a means to store duplicate elements, and we must remember the order in which elements are added. To do so, we do not just store the elements themselves but, rather, pairs  $(c, n)$ , where  $c$  is a character and  $n$  is its index in the string represented.  $n$  counts from 1 onwards. In the specification below, we use one auxiliary function `maxi`. `maxi` returns the largest index contained in the set given as parameter.

```

proc Create(var s: string);
  pre: true
  post: s =  $\emptyset$ 

proc Append(var s: string; c: char);
  pre: true
  post: s = s'  $\cup (c, \text{maxi}(s') + 1)$ 

proc Delete(var s: string; i: integer);
  pre: true
  post: s =  $\{(c, j) \mid (c, j) \in s' \wedge j < i\} \cup \{(c, j) \mid (c, j+1) \in s' \wedge j+1 > i\}$ 

proc Substring(s1, s2: string): boolean;
  pre: true
  post: if  $\exists 0 \leq j: \forall i \in \{1, \dots, \text{maxi}(s2)\}$ :
         $(\exists c \in \text{char}: (c, i) \in s2 \Rightarrow (c, i+j) \in s1)$ 
        then true
        else false

```

15.15 See [Gries, 1981, p 259-262] for a discussion of this problem.

15.16 See [Smit, 1982] for a correctness proof of KMP and various other string matching algorithms.

15.17 In the following specification, a postfix accent (') is used to denote the value of an object prior to the execution of the operation, rather than the backward pointing hook that is normally used in the VDM-notation.

```

ChangeAddr(ci: Client_Id, New_Addr: Address)
  ext wr clients
  pre mk-Client(-, -, c) ∈ clients
  post let name: Name, addr: Address •
    (let c = mk-Client(name, addr, ci) • c ∈ clients')
    in clients = (clients' - {c}) ∪ {mk-Client(name, New_Addr, ci)}

```

15.18 We may introduce a set `unavailable` containing the identifications of copies of books that are temporarily unavailable. We next introduce a map `available` with the same structure as the map `books`, which contains all available book copies. In the specification given in figure 15.12, we next replace `books` by `available` (from line 22 onwards). I.e., where we originally referred to the map `books`, we now refer to the map containing available book copies only. The library invariant is extended with a clause expressing the fact that `available` is a restriction of `books`: `available` does not contain copies that are unavailable. In the specification below, we have also added operations to make a copy available and unavailable, respectively. The changes then become as follows:

1. Add the following two components to the state:

```

unavailable: Book_Id-set
available: Book_Id  $\xrightarrow{m}$  Book

```

2. Add the following clause after line 21:

```

available = unavailable  $\triangleleft$  books

```

3. Replace `books` by `available` from line 22 onwards.
4. Add the following two operation specifications:

```

MakeAvailable (b: Book_Id)
  ext wr available, unavailable
  pre b ∈ unavailable
  post b ∉ unavailable

```

```

MakeUnavailable (b: Book_Id)
  ext rd books, borrowed
  wr available, unavailable
  pre b ∈ books ∧ b ∉ borrowed ∧ b ∉ unavailable
  post b ∈ unavailable

```

## 16 User Interface Design

16.12 Help systems come in two broad categories: passive help systems and active ones. A passive help system offers static information: an overview of commands/actions, information on the use of specific commands/actions, how-to like information (as in "how to

delete a paragraph”). The latter is especially important if the command mnemonics do not really fit their semantics, and in cases where command sequences have to be issued to realize a compound task.

A dynamic help system takes into account the current state and the history of the current job when offering help. It may give information on *admissible* commands/actions, offer hints as to how to achieve things better (replace recurrent command sequences by more powerful commands than the user is possibly aware of, and the like). Present-day desktop applications often offer both varieties.

- 16.13 Most of the adaptations are to the requirements engineering phase. Requirements engineering is likely to start with a feasibility study. Part of this feasibility study is to decide on the system boundaries: what is done by the computer, and what is left to the user. A global task analysis, possibly with a few user representatives only, may be done to clarify these issues.

Once this feasibility study is done, and a decision on a full requirements engineering phase is taken, a more elaborate task analysis step is conducted. Interviews, observations, and other techniques can be used to get at a full task catalog and task hierarchy. At this stage also, certain aspects of the interface functionality (dialog style, type of error messaging and help functions, dialog sequencing, and default behavior) is determined. This can be user-tested using rapid prototyping and screen walkthroughs.

During the design stage, several alignment issues deserve our attention, such as those between detailed data modeling and the objects that appear on the screen, between task structure and system structure, and the physical layout of screens.

Finally, testing should also include usability testing.

See [Summersgill and Browne, 1989] for a more detailed description of how to integrate user-interface issues with a classical waterfall-type development method, viz. SSADM.

- 16.14 – Manually constructed scenarios with prospective users. Advantages include: user involvement from the start, real-life examples that users feel comfortable with, expressed in the language of the user, no big investments needed, quick results. Possible disadvantages include: the extent to which the scenarios cover everything needed, how to document the results, the process need not converge, it is difficult to include dynamics, and the scenarios tend to be simple ones. See [Rettig, 1994] for a more elaborate discussion of a similar approach, viz. the use of paper prototypes in user interface design.
- Iterative prototyping. The advantages and disadvantages hereof are discussed in section 3.2 of this book.
  - Develop functional parts first, and only then the user interface. From a managerial point of view, this approach has definite advantages when it comes to control progress. Functionality is decided upon first, and the user interface is seen as a layer on top of this (so, while working on the user interface, no rework is needed because of wrong functionality). It also allows for a clear separation of concerns in the architecture (viz. the Seeheim model). The biggest disadvantage is that the result need not match the real user needs, and that it takes a long time before the users ‘see’ anything.

- Formal description and analysis of user interface. A major advantage of formal descriptions is that they allow for formal evaluation. However, it remains to be seen whether the user interface requirements can be sufficiently captured formally. Also, discussing formal specifications with users is not easy, and most developers are not familiar with formal techniques.

## 17 Software Reusability

17.7 When discussing data abstraction in chapter 11, we observed that general, domain-independent data abstractions often occur at the lower levels of the system hierarchy, while domain-dependent data abstractions show up at the intermediate and higher levels. Domain-independent data abstractions are limited in number: lists, queues, trees of various kinds, and the like. These are well-known, and their reuse is quite feasible. Domain independence is much harder to obtain at the higher levels of the system hierarchy. It is very difficult to describe such concepts (like "ledger" in a banking system, "alarm signal" in an elevator-control system, etc.) in a domain-independent way, since they derive their meaning from that domain. It is also very hard to retrieve them using domain-independent descriptions (if these exist): experts in a certain domain think in terms of concepts from that domain.

17.10 In a component-based software factory, explicit attention needs to be given to the library of reusable components. I.e., we have to actively look for reusable components during development (especially during design and implementation), and we have to assess components for inclusion in the library. Thus, the process model changes in at least two ways. During design and implementation, the library of reusable components has to be searched. Note that this not only involves a passive search. The design process itself will be different, since we wish to obtain a design in which reuse is indeed achieved: not just design with reuse, but design for reuse. Secondly, we must assess new components for inclusion in the library. This often incurs extra activities: extra testing and documentation, generalizing components, and the like. This had better be an activity clearly separated from the current project.

Managerially, incentives are needed to foster both the use of reusable components and their development. Furthermore, new roles and activities must be defined. A librarian is charged with the administration of the library (including the enforcement of coding and documentation standards, classification of components, and the like). New activities include assessment of existing components for inclusion in the library, as well as improvements to components before inclusion.

17.13 Reusable components (or templates, or designs) embody a certain amount of, either domain-dependent or domain-independent, knowledge. In order to assess their usefulness, we must be able to acquire sufficient knowledge about them. Codification of this knowledge eases this process. Component classification schemes are one means to do so. In a fancy classification scheme we may be given entities that are "close" to the one looked for. Codification of domain-dependent knowledge is a further step in supporting the retrieval of domain-dependent reusable artifacts.

17.15 Major aspects of this routine that need to be documented are:

- description of input and output formats (including the maximum size of the matrix);
- precision and error tolerancy information;
- information on the algorithm’s speed, and its memory requirements;
- how exceptions are handled (the possible singularity of the matrix);
- an identification of the numerical method used (such as LU-decomposition with complete or partial pivoting, QR-decomposition, Gauss-Jordan).

17.16 There are at least two reasons why this type of information may be relevant to the user:

- this information may be important in our assessment of the usability of the component. Finding an element in a binary search tree is faster than finding an element in an unsorted list. Sorting algorithms differ in speed and memory requirements. Numerical precision may depend on the algorithm used. Etc.
- in certain circumstances, we really **think** of entities in terms of their representation. For instance, we may think of a polynomial as a series of coefficients, or a series of zeroes plus a constant. We really do not think of a polynomial in a more abstract sense. A more elaborate discussion hereof can be found in [Sikkel and van Vliet, 1992].

## 18 Software Reliability

18.5 The major advantage of N-version programming is that we may hope for a larger probability that independently developed programs have independent failure behavior as well. If such is indeed the case, the reliability of the system will be increased as well. Unfortunately, one may argue that programmers tend to make similar mistakes, so that the reliability is not really increased by voting amongst different versions of the same program.

18.6 Suppose a simple program has two types of input. If the test set contains the same number of test cases for either type, faults in either execution path have the same probability of being revealed. If, on the other hand, the actual input during the operational use of the program is much more skewed (i.e., input of one type is much more likely to occur), the probability of faults to show up is also skewed. As a consequence, the actual reliability observed during execution is largely determined by the input type most often used, and this aspect should be considered when assessing the program’s reliability.

18.7 An extensive discussion is given in [Musa *et al.*, 1987]. In particular:

- if there is little manpower available to identify and correct faults, failures observed will not be corrected until manpower does become available. Increase in reliability would be higher if more manpower were available, while the actual reliability is the same in both cases: the same number of failures is observed in the same number of test cases.
- a similar argument holds if the identification and correction of faults is limited by the amount of computer time needed to do so.

- if we have two systems, one that is used 24 hours a day, and one that is used once a week, having experienced two failures in both systems during a time span of 2 weeks does not mean that the two systems have equal reliability.

18.8 Yes, given accurate data on failure occurrences, current reliability models give a quite reasonably objective assessment of software reliability.

18.9 – Strongly-typed languages lead to more robust programs. Illegal combinations in operands, assignments, actual parameters, etc., are detected by the compiler. Clerical typing errors often result in either type mismatches or undeclared objects as well. This all adds to the reliability of the resulting programs.

- Goto-less programming leads to clearer program structures, structures that are easier to comprehend and test, structures that are less complex. Again, this adds to the reliability of those programs.

- Abstract data types offer means for a clear description of the interfaces between the ADTs and the program using those ADTs. Next to that, ADTs hide representation details, details that cannot be (mis) used by the calling program. This increases possibilities for independent testing of program units and decreases mutual dependencies between program units.

- Object-oriented programming languages offer the advantage that less code needs to be written because of explicit code sharing between program entities. Also, program changes can often be accommodated through the addition of a new subclass, rather than having to tinker with existing code.

- If procedures have precondition true and have been adequately tested in isolation, we know that their use will never pose any problems. Again, the mutual dependencies between program elements is decreased, thus adding to the reliability of the system as a whole.

18.10 Ultimately, the actual occurrences of failures are what counts. Faults that never show up, for instance because they are located in a piece of code that never gets executed, are not important. A fault in a piece of code that gets executed many times a day is important, though. Thus, an assessment of the actual frequency of failure occurrences (= reliability) may be deemed more important than testing.

On the other hand, both testing and reliability assessment are needed. Testing, if started early on in the project, can help to prevent errors, and provides for systematic means to assess software quality. At a later stage, reliability assessment helps to assess the operational quality of the system.

18.11 See also the answer to exercise 12.16.

## 19 Software Tools

19.12 Artifacts like user documentation, specifications, and the like, have many features in common with source code modules:

- they too consist of a set of interrelated components;

- they are subject to change as well, resulting in different versions as well as revision histories;
- they also have some status, like: in development, being tested, frozen, etc.

Automatic support for configuration control of these artifacts thus offers similar help in the control these types of information.

19.13 There are a number of possible reasons for this discrepancy, such as:

- fully integrated support environments do not really exist yet. As a consequence, support for a number of activities is given, but not for all.
- support environments tend to concentrate on initial development. It is not clear to what extent systems developed using such an environment can also be easily maintained.
- there is no consensus yet as to how a support environment should look like. A lot of research and development in this area is still going on, and future environments will certainly differ from the present ones.
- because of their formality, support environments impose a certain way of doing things. Many development activities however are ill-formalizable, and the environment may then be a hindrance rather than a help. Also, tuning of a support environment to a specific situation often is not easy.
- sound ways to assess, compare, and judge the added value of support environments are not yet available.

## References

- [Bass *et al.*, 1998] L. Bass, P. Clements, and R. Kazman. *Software Architecture in Practice*. Addison-Wesley, 1998.
- [Bersoff and Davis, 1991] E.H. Bersoff and A.M. Davis. Impacts of Life Cycle Models on Software Configuration Management. *Communications of the ACM*, 34(8):104–118, 1991.
- [Boehm, 1981] B.W. Boehm. *Software Engineering Economics*. Prentice-Hall, 1981.
- [Brooks, 1995] F.P. Brooks. *The Mythical Man-Month*. Addison-Wesley, second edition, 1995.
- [Carmel *et al.*, 1993] E. Carmel, R.D. Whitaker, and J.F. George. PD and Joint Application Design: A Transatlantic Comparison. *Communications of the ACM*, 36(6):40–48, 1993.
- [DeMarco and Lister, 1985] T. DeMarco and T. Lister. Programmer Performance and the Effect of the Workplace. In *Proceedings 8th International Conference on Software Engineering (ICSE8)*, pages 268–272. IEEE, 1985.
- [DeMarco and Lister, 1987] T. DeMarco and T. Lister. *Peopleware*. Dorset House, 1987.
- [Gries, 1981] D. Gries. *The Science of Programming*. Springer Verlag, 1981.
- [Meyer, 1985] B. Meyer. On Formalism in Specifications. *IEEE Software*, 2(1):6–26, 1985.

- [Meyer, 1992] B. Meyer. Design by Contract. *IEEE Computer*, 25(10):40–51, 1992.
- [Musa *et al.*, 1987] J.D. Musa, A. Iannino, and K. Okumoto. *Software Reliability: Measurement, Prediction, Application*. McGraw-Hill, 1987.
- [Parnas, 1999] D.L. Parnas. Software Engineering Programs Are Not Computer Science Programs. *IEEE Software*, 16(6):19–30, 1999.
- [Perry and Kaiser, 1991] D.E. Perry and G.E. Kaiser. Models of Software Development Environments. *IEEE Transactions on Software Engineering*, 17(3):283–295, 1991.
- [Redmond and Ah-Chuen, 1990] J.A. Redmond and R. Ah-Chuen. Software Metrics: A User’s Perspective. *Journal of Systems and Software*, 13(2):97–110, 1990.
- [Rettig, 1994] M. Rettig. Prototyping for Tiny Fingers. *Communications of the ACM*, 37(4):21–27, 1994.
- [Sikkel and van Vliet, 1992] K. Sikkel and J.C. van Vliet. Abstract Date Types as Reusable Software Components: The Case for Twin ADTs. *Software Engineering Journal*, 7(3):177–183, 1992.
- [Smit, 1982] G. De V. Smit. A Comparison of Three String Matching Algorithms. *Software, Practice & Experience*, 12(1):57–66, 1982.
- [Summersgill and Browne, 1989] R. Summersgill and D.P. Browne. Human Factors: Its Place in System Development Methods. In *Proceedings Fifth International Workshop on Software Specification and Design (ACM SIGSOFT Engineering Notes, Volume 14, Number 3)*, pages 227–234. ACM, 1989.
- [Vincent *et al.*, 1988] J. Vincent, A. Waters, and J. Sinclair. *Software Quality Assurance, Vol I: Practice and Experience*. Prentice-Hall, 1988.