PART SIX

Management

22 Managing people

Objectives

The objective of this chapter is to discuss the importance of people in the software engineering process. When you have read the chapter, you will:

- understand simple models of human memory, problem solving and motivation and some implications of this for software managers;
- understand the key issues of team working, namely team composition, team cohesiveness, team communications and team organisation;
- understand some of the issues involved in selecting and retaining staff in a software development organisation;
- have been introduced to the P-CMM – a model that is a framework for enhancing the capabilities of software developers in an organisation.

Contents

The people working in a software organisation are its greatest assets. They represent intellectual capital and it is up to software managers to ensure that the organisation gets the best possible return on its investment in people. In successful companies and economies, this is achieved when people are respected by the organisation. They should have a level of responsibility and reward that is commensurate with their skills.

Effective management is therefore about managing the people in an organisation. Project managers have to solve technical and non-technical problems by using the people in their team in the most effective way possible. They have to motivate people, plan and organise their work and ensure that the work is being done properly. Poor management of people is one of the most significant contributors to project failure.

I base my discussion of management on cognitive and social factors rather than any, currently fashionable, management theory. Software engineering is a cognitive and social activity, so these factors are important in developing an understanding of how people write software. If managers have some understanding of these fundamentals, they can do a better job of getting the best from the people who work for them.

22.1 Limits to thinking

There is great diversity in individual abilities reflecting differences in intelligence, education and experience but most of us seem to be subject to some basic constraints on our thinking. These are a consequence of the way in which information is stored and modelled in our brains. You don't need to understand cognitive information processing in detail. However, I think that an awareness of the limits to the way we think is important. It helps explain why some software engineering techniques are effective and gives you insights into interactions between people in a software development team.

22.1.1 Memory organisation

Software systems are abstract entities and engineers have to remember their characteristics during the development process. For example, programmers must understand and remember the relationship between a source code listing and the dynamic behaviour of the program. They apply this stored knowledge in further program development.

The organisation of human memory seems to be hierarchical with three distinct, connected areas (Figure 22.1):

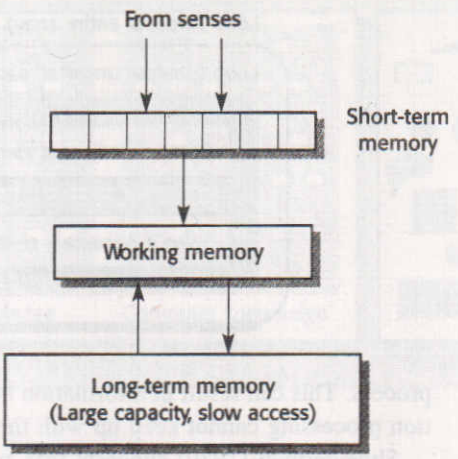


Figure 22.1
Human memory organisation

1. *A limited capacity, fast-access, short-term memory* Input from the senses is received here for initial processing. This memory is comparable with registers in a computer; it is used for information processing and not information storage.
2. *A larger capacity, working memory area* This memory area has a longer access time than short-term memory. It is used for information processing but can retain information for longer periods than short-term memory. It is not used for long-term information retention. By analogy with a computer, this is like RAM where information is maintained for the duration of a computation.
3. *Long-term memory* This has a large capacity, relatively slow access time and unreliable retrieval mechanisms (we forget things). Long-term memory is used for the 'permanent' storage of information. To continue the analogy, long-term memory is like disk memory on a computer.

Problem information is input to the short-term memory from reading documents and talking to people. This is integrated with other relevant information from long-term memory in the working memory. The result of this integration forms the basis for problem solutions. These are stored in long-term memory for future use. Of course, the solution may be incorrect. As more information becomes available, long-term memory must be modified. However, incorrect information is not completely discarded but is retained in some form. We know this because we learn from our mistakes.

The limited size of short-term memory constrains our cognitive processes. In a classic experiment, Miller (1957) found that the short-term memory can store about seven quanta of information. A quantum of information is not a fixed size but is rather a coherent information entity. It may be a telephone number, the purpose of an object or a street name. Miller also describes the process of 'chunking' where information quanta are collected together into chunks.

If a problem involves the input of more information than the short-term memory can handle, there has to be information processing and transfer during the input

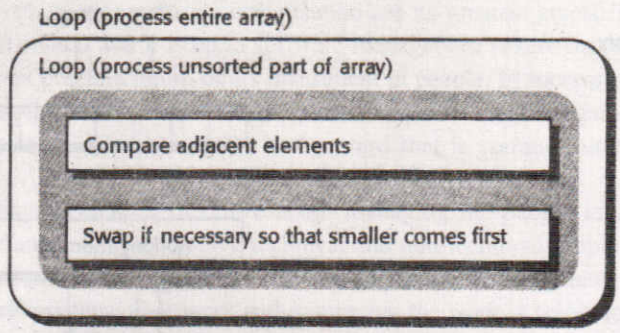


Figure 22.2
Cognitive 'chunks' in a sort program

process. This can result in information being lost. Errors arise because this information processing cannot keep up with the memory input.

Shneiderman (1980) suggests that an information chunking process is used in understanding programs. Program readers abstract the information in the program into chunks which are built into an internal semantic structure. Programs are not understood on a statement-by-statement basis unless a statement represents a logical chunk. Figure 22.2 shows how a simple sorting program might be 'chunked' by someone trying to understand it.

Once the internal semantic structure representing the program has been established, this knowledge is transferred to long-term memory. If it is regularly used, it is not usually forgotten. It can be reproduced in different notations without much difficulty. Therefore, we seem to learn in terms of high-level abstractions and not in terms of low-level details.

The knowledge acquired during software development and stored in long-term memory falls into two classes:

1. *Semantic knowledge* This is the knowledge of concepts such as the operation of an assignment statement, the notion of an object class, how a hash search technique operates and how organisations are structured. This knowledge is acquired through experience and learning and is retained in a representation-independent fashion.

2. *Syntactic knowledge* This is detailed representation knowledge such as how to write an object description in the UML, what standard functions are available in a programming language, whether an assignment is written using an '=' or a ':=' sign, etc. This knowledge seems to be retained in an unprocessed form.

This knowledge organisation is illustrated in Figure 22.3, adapted from Shneiderman's (1992) book on user interface design. This diagram suggests that semantic and task knowledge is organised and structured. Relationships between different knowledge fragments have been constructed. By contrast, syntactic knowledge is arbitrary and disorganised. It is therefore easier to forget and to make mistakes using this type of knowledge.

Semantic knowledge is acquired by experience and through active learning. New information is consciously integrated with existing semantic structures. Syntactic

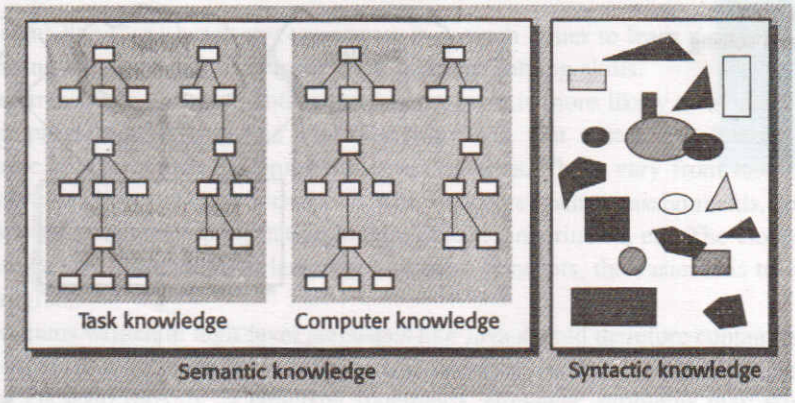


Figure 22.3 Syntactic and semantic knowledge

knowledge, however, seems to be acquired by memorisation. New syntactic knowledge is not immediately integrated with existing knowledge but may interfere with it. It is more easily forgotten than deeper semantic knowledge.

The different acquisition modes for syntactic and semantic knowledge help explain how experienced programmers learn a new programming language. They have no difficulty understanding language concepts such as assignments, loops, conditional statements, etc. The language syntax, however, tends to get mixed up with the syntax of familiar languages. Therefore, an Ada programmer learning Java might write the assignment operator as ':=' rather than '='.

As an understanding of a concept develops, it is stored in memory as semantic knowledge. Semantic knowledge appears to be stored in an abstract conceptual form. The concept details can be regenerated in a number of different concrete representations (Soloway *et al.*, 1982; Card *et al.*, 1983).

For example, consider the binary search algorithm where an ordered collection is searched for a particular item. This involves examining the mid-point of the collection and using knowledge of the ordering relationship to check if the key item is in the upper or the lower part of the collection. A programmer who understands this algorithm can easily produce a version in Java, Ada or other programming language.

This model explains why, for many people, learning to program is a skill that seems to arrive all at once, after a period of difficulties. Programming skills require an understanding of the semantic concepts and a separation of semantic and syntactic concepts. Instructors sometimes have difficulties understanding student problems. They have successfully understood and processed the semantic information, so only have to consider syntactic information. They may therefore find it hard to explain the semantic concepts in ways that novice programmers can understand.

22.1.2 Problem solving

Devising and writing a program is a problem solving process. To develop a software system, you must understand the problem, work out a solution strategy then

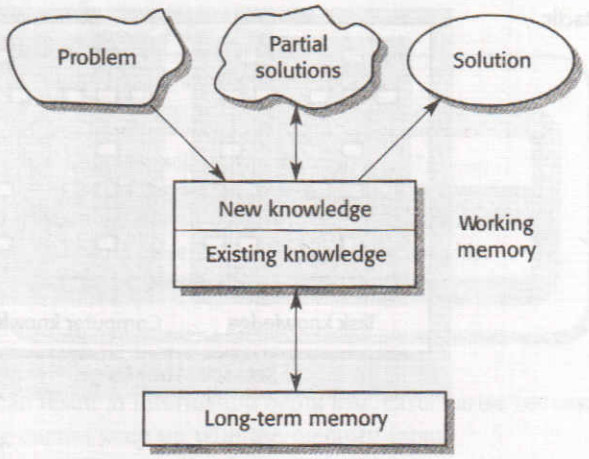


Figure 22.4
Problem solving

translate it into a program. The first stage involves the problem statement entering working memory from short-term memory. It is integrated with existing knowledge from long-term memory and analysed to work out an overall solution. Finally, the general solution is refined into an executable program (Figure 22.4).

Problem solving generally requires task and computer concepts to be integrated. Organisational factors such as the need to complete a solution within budget are important. Thus, a user may be expert in the task concepts, a software designer an expert in the computer concepts and a manager an expert in the organisation factors. During the software engineering process, all of this expertise may have to be used and integrated.

The development of the solution (the program) involves building an internal semantic model of the problem and a corresponding model of the solution. When this model has been built, it may be represented in any appropriate syntactic notation. The process of program design is therefore an iterative process involving several steps:

1. Integrate existing computer and task knowledge to create new knowledge and hence understand the problem. Curtis (Curtis *et al.*, 1988) suggests that application experience is particularly important at this stage.
2. Construct a semantic model of the solution. Test this against the problem and refine it until it is satisfactory.
3. Represent the model in some programming language or design notation.

When managers have to make decisions on who should be involved in a long-term project, they should consider overall problem solving ability and domain experience rather than specific programming language skills. Once a problem has been understood, experienced programmers will have roughly the same degree of difficulty in writing a program, irrespective of the programming language used. Language skills are necessary and take time to develop (particularly for complex

languages like C++) but in my experience it is much easier to learn a specific programming language than it is to develop problem solving skills.

The translation from semantic model to program is more likely to be error-free if the programming language includes constructs that match the lowest-level semantic structures that the programmer understands. These vary from individual to individual but probably correspond with concepts such as assignments, loops, conditional statements, information hiding, objects, inheritance, etc. The closer the fit between the programming language and these concepts, the easier it is to write the program.

Programs written in high-level languages like Java should therefore contain fewer errors than those written in assembly code because low-level semantic concepts can be encoded directly as language statements. However, problems may arise if functional and object-oriented concepts are mixed up. These can be a problem if a company has standardised on one type of method for analysis (e.g. SADT) but uses an object-oriented design and programming process.

The model also explains why structured programming (see Chapter 18) is the best strategy for organising program control. It is based on semantic concepts such as loops and conditional statements. The programmer's short-term memory is not overloaded and so he or she is less likely to make mistakes. Structured programs are easier to understand because you can read them from top to bottom. The abstractions involved in forming chunks can be made sequentially without looking at other parts of the program. Short-term memory can be devoted to a single section of code. Information from working memory about other parts of the program which interfere with that section does not have to be retrieved.

22.1.3 Motivation

One of the roles of project managers is to motivate the people who work for them. Maslow (1954) suggested that people are motivated by satisfying their needs and that needs were arranged in a series of levels as shown in Figure 22.5. The lower levels of this hierarchy represent fundamental needs for food, sleep, etc. and the need to feel secure in an environment. Social needs are concerned with the need to feel part of a social grouping. Esteem needs are the need to feel respected by others and self-realisation needs are concerned with personal development. Human priorities are to satisfy lower-level needs like hunger before the more abstract, higher-level needs.

People working in software development organisations are not usually hungry or thirsty and generally do not feel physically threatened by their environment. Therefore, ensuring the satisfaction of social, esteem and self-realisation needs is most significant from a management point of view:

1. Satisfying social needs means allowing people time to meet their co-workers and providing places for them to meet. Informal, easy-to-use communication channels such as electronic mail are very important.

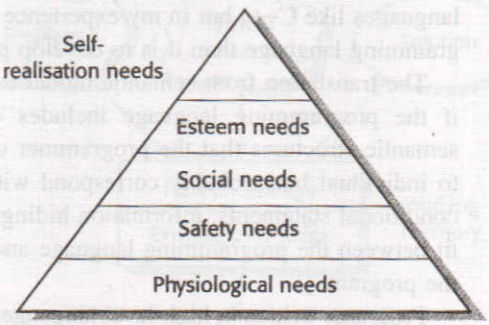


Figure 22.5 Human needs hierarchy

2. To satisfy esteem needs you need to show people that they are valued by the organisation. Public recognition of achievements is a simple yet effective way of doing this. Obviously, people must also feel that they are paid at a level that reflects their skills and experience.
3. Finally, to satisfy self-realisation needs you need to give people responsibility for their own work, assign them demanding (but not impossible) tasks and provide a training programme where people can develop their skills.

In a psychological study into motivation, Bass and Dunteman (1963) classified professionals into three types:

1. *Task-oriented* who are motivated by the work they do. In software engineering, they are technicians who are motivated by the intellectual challenge of software development.
2. *Self-oriented* who are principally motivated by personal success and recognition. They are interested in software development as a means of achieving their own goals.
3. *Interaction-oriented* who are motivated by the presence and actions of co-workers. As software development becomes more user-centred, interaction-oriented individuals are becoming more and more involved in software engineering.

Interaction-oriented personalities usually like to work as part of a group whereas task-oriented and self-oriented people often prefer to work alone. Women are more likely to be interaction-oriented than men. They are often more effective communicators.

Each individual's motivation is made up of elements of each class but one type of motivation is usually dominant at any one time. However, personalities are not static and individuals can change. For example, technical people who feel they are not being properly rewarded can become self-oriented and put personal interests before technical concerns.

The problem with Maslow's model of motivation is that it takes an exclusively personal viewpoint on motivation. It does not take adequate account of the fact

that people feel themselves to be part of an organisation, a professional group and, usually, some culture. People are not just motivated by personal needs but are also motivated by the goals of these broader groups. Being a member of a cohesive group is highly motivating to most people. People with fulfilling jobs often like to go to work because they are motivated by the people they work with and by the work that they do.

22.2 Group working

Most professional software is developed by project teams ranging in size from two to several hundred people. However, as it is clearly impossible for everyone in a large team to work together effectively on a single problem, these large teams are usually split into a number of groups. Each group is responsible for a sub-project that is developing some sub-system. As a general rule, software engineering project groups should normally have no more than eight members. When small groups are used, communication problems are reduced. The whole group can get round a table for a meeting and can meet in each other's offices. Complex communication structures are not required.

Putting together a group which works effectively is therefore a critical management task. It is obviously important that the group should have the right balance of technical skills and experience and personalities. However, successful groups are more than simply a collection of individuals with the right balance of skills. A good group has a team spirit so that the people involved are motivated by the success of the group as well as their own personal goals. Therefore, managers should promote explicit 'team building' activities to help establish this essential feeling of group loyalty.

There are a number of factors that influence group working:

1. *Group composition* Is there the right balance of skills, experience and personalities in the team?
2. *Group cohesiveness* Does the group think of itself as a team rather than as a collection of individuals who are working together?
3. *Group communications* Do the members of the group communicate effectively with each other?
4. *Group organisation* Is the team organised in such a way that everyone feels valued and is satisfied with their role in the group?

22.2.1 Group composition

Many software engineers are motivated primarily by their work. Software development groups, therefore, are often composed of people who have their own idea on

how technical problems should be solved. This is borne out by regularly reported problems of interface standards being ignored, systems being redesigned as they are coded, unnecessary system embellishments, etc. Selecting the right members for a project group is essential if these kinds of problem are to be avoided.

A group which has complementary personalities may work better than a group which has been selected solely on technical ability. People who are motivated by the work are likely to be the strongest technically. People who are self-oriented will probably be best at pushing the work forward to finish the job. People who are interaction-oriented help with communications within the group. I think that it is particularly important to have interaction-oriented people in a group. They like to talk to people and can detect tensions and disagreements at an early stage. They can help resolve personality problems and differences before they have a serious impact on the group.

It is sometimes impossible to choose a group with complementary personalities. In this case, the project manager has to control the group so that group members do not let their individual goals transcend organisational and group objectives. This control is easier to achieve if all group members participate in each stage of the project. Individual initiative is most likely when group members are given instructions without being aware of the part that their task plays in the overall project.

For example, say an engineer is given a program design for coding and notices possible design improvements. If these improvements are implemented without understanding the rationale for the original design, they might have adverse implications on other parts of the system. If the whole group are involved in the design from the start, they will understand why design decisions have been made. They may identify with these decisions rather than oppose them.

Within a group, the group leader has an important role. He or she may be responsible for providing technical direction and project administration. Group leaders must keep track of the day-to-day work of their group, ensure that people are working effectively and work closely with project managers on project planning.

Leaders are normally appointed and report to the overall project manager. However, the appointed leader may not be the real leader of the group as far as technical matters are concerned. The group members may look to another group member for leadership. He or she may be the most technically competent engineer or may be a better motivator than the appointed group leader.

Sometimes, it is effective to separate technical leadership and project administration. People who are technically competent are not always the best administrators. When they are given an administrative role, this reduces their overall value to the group. It is best to support them with an administrator who relieves them of day-to-day administrative tasks.

If an unwanted leader is imposed on a group, this is likely to introduce tensions. The members will not respect the leader and may reject group loyalty in favour of individual goals. This is a particular problem in a fast-changing field such as software engineering where new members may be more up to date and better educated than experienced group leaders. Some people with experience may resent the imposition of a young leader with new ideas.

22.2.2 Group cohesiveness

In a cohesive group, members think of the group as more important than the individuals in it. Members of a well-led, cohesive group are loyal to the group. They identify with group goals and with other group members. They attempt to protect the group, as an entity, from outside interference. This makes the group robust and able to cope with problems and unexpected situations. The group can cope with change by providing mutual support and help.

The advantages of a cohesive group are:

1. *A group quality standard can be developed* Because this standard is established by consensus, it is more likely to be observed than external standards imposed on the group.
2. *Group members work closely together* People in the group learn from each other. Inhibitions caused by ignorance are minimised as mutual learning is encouraged.
3. *Group members can get to know each other's work* Continuity can be maintained should a group member leave.
4. *Egoless programming can be practised* Programs are regarded as group property rather than personal property.

Egoless programming (Weinberg, 1971) is a style of group working where designs, programs and other documents are considered to be the common property of the group rather than the individual who developed them. If engineers think of their work in this way, they are more likely to offer it for inspection by other group members, to accept criticism, and to work with the group to improve the program. Group cohesiveness is improved because all members feel that they have a shared responsibility for the software.

As well as improving the quality of designs, programs and documents, egoless programming also improves communications within the group. It encourages uninhibited discussion without regard to status, experience or gender. Individual members actively cooperate with other group members throughout the course of the project. This all serves to draw the members of the group together and makes them feel part of a group.

Group cohesiveness depends on many factors, including the organisational culture and the personalities in the group. Managers can encourage cohesiveness in a number of ways. They may organise social events for group members and their families. They may try to establish a sense of group identity by naming the group and establishing a group identity and territory. They may get involved in explicit group building activities such as sports and games.

However, in my experience, one of the most effective ways of promoting cohesion is to ensure that group members are treated as responsible and trustworthy and given access to information. Often, managers feel that they cannot reveal certain information to all of the group. This invariably creates a climate of mistrust. Simple information exchange is a cheap and efficient way of making people feel that they are part of a group.

Strong, cohesive groups, however, can sometimes suffer from two problems:

1. *Irrational resistance to a leadership change* If the leader of a cohesive group has to be replaced by someone outside of the group, the group members may band together against the new leader. Group members may spend time resisting changes proposed by the new group leader with a consequent loss of productivity. Whenever possible, new leaders are therefore best appointed from within groups.
2. *Groupthink* Groupthink (Janis, 1972) is the name given to a situation where the critical abilities of group members are eroded by group loyalties. Consideration of alternatives is replaced by loyalty to group norms and decisions. Any proposal favoured by the majority of the group may be adopted without proper consideration of alternatives.

To avoid groupthink, formal sessions may be organised where group members are encouraged to criticise decisions. Outside experts may be introduced to review the group's decisions. People who are naturally argumentative, questioning, and disrespectful of the *status quo* may be appointed as group members. They act as a devil's advocate, constantly questioning group decisions, thus forcing other group members to think about and evaluate their activities.

22.2.3 Group communications

It is essential that there should be good communications between members of a software development group. The group members must exchange information on the status of their work, the design decisions that have been made and changes to previous decisions that are necessary. Good communications also strengthens group cohesiveness as group members come to understand the motivations, strengths and weaknesses of other people in the group.

Some key factors which influence the effectiveness of communications are:

1. *Group size* As a group increases in size, it becomes more difficult to ensure that all members communicate effectively with each other. The number of one-way communication links is $n * (n - 1)$ where n is the group size, so you can see that, with a group of seven or eight members, it is quite possible that some people will rarely communicate. Status differences between group members mean that communications are often one-way communications. Higher-status members tend to dominate communications with lower-status members who are often reluctant to start a conversation or to make critical remarks.
2. *Group structure* People in informally structured groups communicate more effectively than in groups with a formal, hierarchical structure. In hierarchical groups, communications tend to flow up and down the hierarchy. People at the same level may not talk to each other. This is a particular problem in a large

project with several development groups. If people working on different subsystems only communicate through their managers, this often leads to delays and misunderstandings.

3. *Group composition* If there are too many people in the group who have the same personality types, these may clash and communications may be inhibited. Communication is usually better in mixed-sex groups (Marshall and Heslin, 1975) than in single-sex groups. Women tend to be more interaction-oriented than men and female group members may act as interaction controllers and facilitators for the group.

4. *The physical work environment of the group* The organisation of the workplace is a major factor in facilitating or inhibiting communications. I discuss this in section 22.3.1.

22.2.4 Group organisation

Small programming groups are usually organised in a fairly informal way. The group leader gets involved in the software development with the other group members. A technical leader may emerge who effectively controls software production. In an informal group, the work to be carried out is discussed by the group as a whole and tasks allocated according to ability and experience. High-level system design is carried out by senior group members but low-level design is the responsibility of the member who is allocated to a particular task.

Informal groups can be very successful particularly where the majority of group members are experienced and competent. The group functions as a democratic unit, making decisions by consensus. Psychologically, this improves group spirit with a resultant increase in cohesiveness and performance. If a group is composed mostly of inexperienced or incompetent members, informality can be a hindrance. No definite authority exists to direct the work, causing a lack of coordination between group members and, possibly, eventual project failure.

An interesting organisational variant of democratic group organisation is described by Beck in his book on 'extreme programming' (Beck, 2000). In this approach, many decisions that are usually seen as management decisions such as decisions on schedule are devolved to group members. Programmers work together in pairs to develop code and take a collective responsibility for the programs that are developed. This approach has reportedly worked successfully but, like Cleanroom development discussed in Chapter 19, I suspect it requires highly qualified and motivated staff for it to be successful.

As I discuss in Chapter 23, individual ability has the most significant influence on programmer productivity. To make the most effective use of highly skilled programmers, Baker (1972) and others (Aron, 1974; Brooks, 1975) suggested that teams should be built around an individual, highly skilled chief programmer. The underlying principle of the chief programmer team was that skilled and experienced staff should be responsible for all software development. They should not be concerned

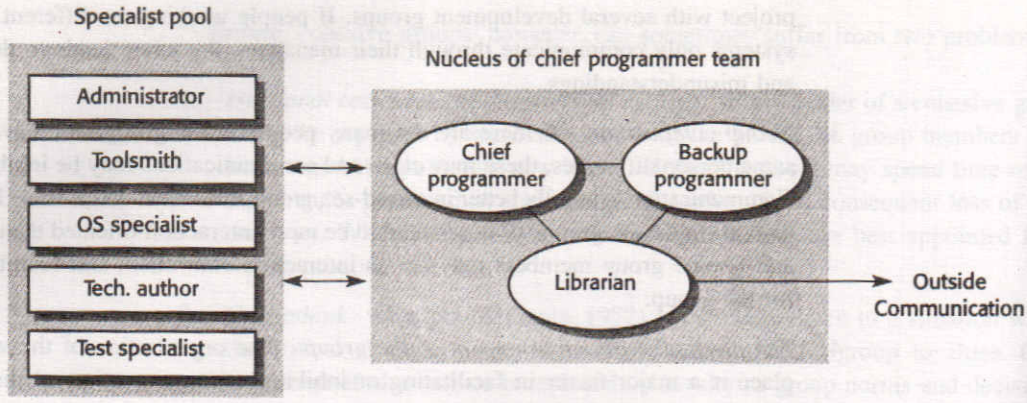


Figure 22.6 A chief programmer team

with routine matters and should have good technical and administrative support for their work. Their communications with people outside the group should be limited (Figure 22.6).

The key members of a chief programmer team are:

1. a chief programmer who takes full responsibility for the design, programming, testing and installation of the system;
2. an experienced backup programmer whose job is to support the chief programmer and take responsibility for software validation;
3. a librarian whose role is to assume all the clerical functions associated with a project such as configuration management, finalising documentation, etc.

Depending on the size and type of the application, other experts might be drawn from a specialist pool to work temporarily or permanently with a team. These might be administrators, operating system or language specialists, test engineers, etc.

The rationale for this approach comes from the fact that there are immense differences in programming ability amongst software engineers. The best programmers may be up to 25 times as productive as the worst programmers. It therefore makes sense to use the best people in the most effective way and to provide them with as much support as possible. Therefore, although the proposals for a chief programmer team are more than 25 years old, it is still an effective way to organise a small development group.

If the right people are available, using this very structured group organisation can be very effective. However, it suffers from a number of problems:

1. Talented designers and programmers are uncommon. The approach relies on an excellent chief programmer and deputy chief programmer. If they make mistakes, there is no one to question their decisions. In a democratic group, anyone can question decisions and hence may discover problems with them.
2. The chief programmer takes all the responsibility and can claim all the credit for success. However, group members may be resentful if their role in the

project is not recognised. Their need for esteem may not be satisfied if all the credit for success is given to the chief programmer.

3. Projects may be threatened if both the chief programmer and his or her deputy are ill or leave the organisation. Managers may not wish to accept this risk.
4. Organisational structures may not be able to accommodate the introduction of this type of group. Large companies may have a well-developed grading structure. Appointing chief programmers outside this structure may be very difficult. In small companies, it may simply be impossible for one of them to be completely devoted to one task.

Chief programmer teams may therefore be a risky strategy for organisations to adopt. However, we can learn from the experiences of this type of group. It makes sense to support technically talented people with project librarians, administrators, etc. Their abilities can therefore be used in the best way. Making specialists available for short times can be more productive than using programmers with general experience for a longer period.

22.3 Choosing and keeping people

One of the roles of a project manager is to choose staff to work on the project. In exceptional cases, project managers can appoint the people who are best suited to the job irrespective of their other responsibilities or budget considerations. More normally, however, project managers do not have a free choice of staff. They may have to use whoever is available in an organisation, they may have to find people very quickly and they may have a limited budget. This limited budget may constrain the number of (expensive) experienced engineers who may work on the project.

If a manager has some choice of staff who can work on the project, the factors which may influence his or her decision are shown in Figure 22.7. There is no way of rating these factors in terms of importance as this varies depending on the application domain, the type of project and the skills of other members of the project team.

The decision on who to appoint to a project is usually made using three types of information:

1. information provided by candidates about their background and experience (their resumé or CV);
2. information gained by interviewing candidates;
3. recommendations from other people who have worked with the candidates.

Some companies make use of various types of test to assess candidates. These include programming aptitude tests and psychometric tests. Psychometric tests are

Figure 22.7 Factors governing staff selection

Factor	Explanation
Application domain experience	For a project to develop a successful system, the developers must understand the application domain.
Platform experience	This may be significant if low-level programming is involved. Otherwise, not usually a critical attribute.
Programming language experience	This is normally only significant for short duration projects where there is insufficient time to learn a new language.
Educational background	This may provide an indicator of the basic fundamentals which the candidate should know and of their ability to learn. This factor becomes increasingly irrelevant as engineers gain experience across a range of projects.
Communication ability	This is important because of the need for project staff to communicate orally and in writing with other engineers, managers and customers.
Adaptability	Adaptability may be judged by looking at the different types of experience which candidates have had. This is an important attribute as it indicates an ability to learn.
Attitude	Project staff should have a positive attitude to their work and should be willing to learn new skills. This is an important attribute but often very difficult to assess.
Personality	This is an important attribute but difficult to assess. Candidates must be reasonably compatible with other team members. No particular type of personality is more or less suited to software engineering.

intended to produce a psychological profile of the candidate indicating attitude and suitability for certain types of task. Some managers consider these tests to be useless; others think they provide useful information for staff selection. As discussed above, problem solving ability seems to be related to the building of semantic models which is a long-term process. Aptitude and psychometric tests usually rely on the rapid completion of questions. It has not been convincingly demonstrated that there is a link between problem solving ability and aptitude tests.

Project managers are sometimes faced with difficulties in finding people with appropriate skills and experience. Teams have to be built using relatively inexperienced engineers. This may lead to problems because of a lack of understanding of the application domain or the technology used in the project.

One reason for this is that, in some organisations, technically skilled staff quickly reach a career plateau. To progress further, they must take on managerial responsibilities. Promotion of these people to managerial status means that useful technical skills are lost. To avoid this loss of technical skills, some companies have developed parallel technical and managerial career structures of equal worth.

Experienced technical people are rewarded at the same level as managers. As an engineer's career develops, they may specialise in either technical or managerial activities and move between them without loss of status or salary.

22.3.1 Working environments

The workplace has important effects on people's performance and their job satisfaction. Psychological experiments have shown that behaviour is affected by room size, furniture, equipment, temperature, humidity, brightness and quality of light, noise and the degree of privacy available. Group behaviour is affected by architectural organisation and telecommunication facilities. Communications within a group are affected by the building architecture and the organisation of the workspace.

There is a real and significant cost in failing to provide good working conditions. When people are unhappy about their working conditions, staff turnover increases. More costs must therefore be expended on recruitment and training. Software projects may be delayed because of lack of qualified staff (DeMarco and Lister, 1999).

Software development staff often work in large open-plan office areas, sometimes with cubicles, and only senior management have individual offices. McCue (1978) carried out a study that showed that the open-plan architecture favoured by many organisations was neither popular nor productive. The most important environmental factors identified in that design study were:

1. *Privacy* Programmers require an area where they can concentrate and work without interruption.
2. *Outside awareness* People prefer to work in natural light and with a view of the outside environment.
3. *Personalisation* Individuals adopt different working practices and have different opinions on decor. The ability to rearrange the workplace to suit working practices and to personalise that environment is important.

In short, people like individual offices that they can organise as they like. There is less disruption and fewer interruptions than in open-plan workspaces. In open-plan offices, people are denied privacy and a quiet working environment. They are limited in the ways that they can personalise their own workspace. Concentration can be difficult and performance is degraded.

Providing individual offices for software engineering staff can make a significant difference to productivity. DeMarco and Lister (1985) compared the productivity of programmers in different types of workplace. They found that factors such as a private workspace and the ability to cut off interruptions had a significant effect. Programmers who had good working conditions were more than twice as productive than equally skilled programmers who had to work in poorer conditions.

Development groups need areas where all members of the group can get together as a group and discuss their project, both formally and informally. Meeting

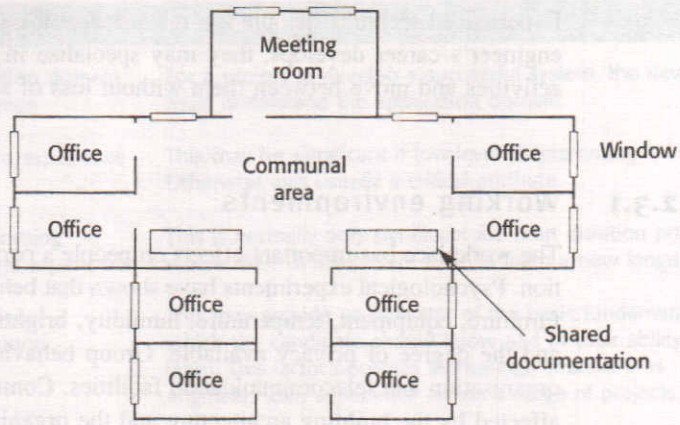


Figure 22.8 Office and meeting room grouping

rooms must be able to accommodate the whole group in privacy. Individual privacy requirements and group communication requirements seem to be exclusive objectives. McCue suggested that these conflicting needs could be accommodated by grouping individual offices round larger group meeting rooms (Figure 22.8).

A similar model is suggested by Beck (2000) in his description of an environment for 'extreme programming'. However, he suggests retaining an open-plan area with all programming activities taking place in the communal area and individual cubicles for the group members when they wish to work alone. Clearly, the key requirement is to provide both individual and group space so that people can work alone or as a group when necessary.

This type of communication helps people solve their problems and exchange information in an informal but effective way. Weinberg (1971) cites an anecdotal example of how an organisation wanted to stop programmers 'wasting time' talking to each other around a coffee machine. They removed the machine, then immediately had a dramatic increase in requests for formal programming assistance. As well as gossiping around the machine, people were solving each other's problems. This illustrates that companies need informal meeting places as well as formal conference rooms.

22.4 The People Capability Maturity Model

The Software Engineering Institute (SEI) in the USA is engaged on a long-term programme of software process improvement. Part of this programme is the Capability Maturity Model (CMM) for software processes which I discuss in Chapter 25. This is concerned with best practice in software engineering. To support this model, they have also proposed a People Capability Maturity Model (P-CMM)

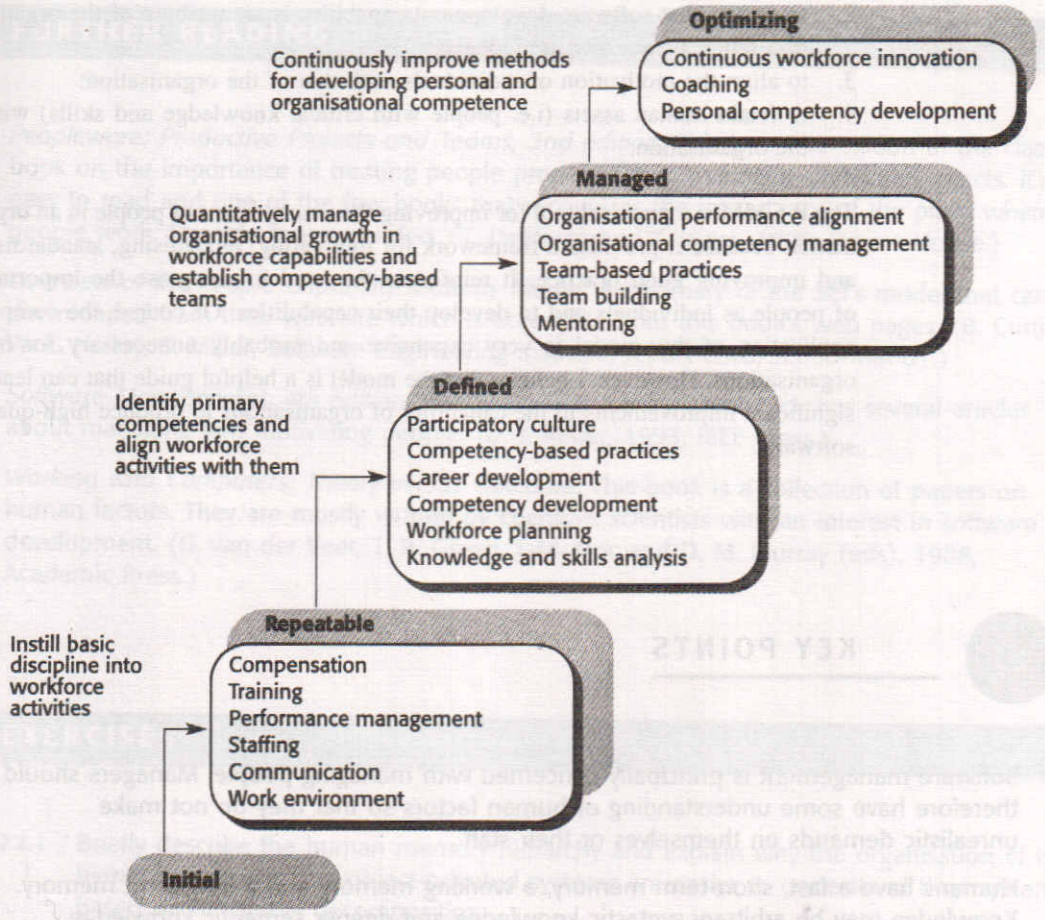


Figure 22.9 The people capability maturity model

(Curtis *et al.*, 1995). This can be used as a framework for improving the way in which an organisation manages its human assets.

Like the CMM, the P-CMM is a five-level model as shown in Figure 22.9. The five levels are:

1. *Initial* Ad hoc, informal people management practices used.
2. *Repeatable* Establishment of policies for developing the capability of the staff.
3. *Defined* Standardisation of best people management practice across the organisation.
4. *Managed* Quantitative goals for people management established and introduced.
5. *Optimising* Continuous focus on improving individual competence and workforce motivation.

Curtis *et al.* (1995) state that the strategic objectives of the P-CMM are:

1. to improve the capability of software organisations by increasing the capability of their workforce;

2. to ensure that software development capability is an attribute of the organisation rather than a few individuals;
3. to align the motivation of individuals with that of the organisation;
4. to retain human assets (i.e. people with critical knowledge and skills) within the organisation.

The P-CMM is a practical tool for improving the management of people in an organisation because it provides a framework for motivating, recognising, standardising and improving good practice. It reinforces the need to recognise the importance of people as individuals and to develop their capabilities. Of course, the complete application of this model is very expensive and probably unnecessary for most organisations. However, I believe that the model is a helpful guide that can lead to significant improvements in the capability of organisations to produce high-quality software.

KEY POINTS

- Software management is principally concerned with managing people. Managers should therefore have some understanding of human factors so that they do not make unrealistic demands on themselves or their staff.
- Humans have a fast, short-term memory, a working memory and a long-term memory. Knowledge may be arbitrary syntactic knowledge and deeper semantic knowledge. Problem solving involves integrating semantic information from the long-term memory with new information in the short-term memory.
- Factors which might be used to select staff include application domain experience, adaptability and personality.
- Software development groups should be small and cohesive. Group leaders should be technically competent and should have administrative and technical support.
- Communications within a group are influenced by factors such as the status of group members, the size of the group, the sexual composition of the group, personalities and available communication channels.
- Providing a private working environment with appropriate computing and communication facilities can improve the productivity and satisfaction of programmers.
- The People-Capability Maturity Model provides a framework and associated advice for improving the capabilities of people in an organisation and improving the organisation's capability to gain benefits from its human assets.

FURTHER READING

Peopleware: Productive Projects and Teams, 2nd edition. This is a new edition of the classic book on the importance of treating people properly when managing software projects. It's easy to read and one of the few books that recognises the importance of the place where people work. Strongly recommended. (T. DeMarco and T. Lister, 1999, Dorset House.)

Overview of the People Capability Maturity Model. A summary of the SEI's model that can be downloaded from their web site which is accessible from this book's web pages. (B. Curtis, W. E. Hefley, S. Miller, Software Engineering Institute, report CMU/SEI-95-MM-01.)

Software Management, 4th edition. This is an IEEE tutorial text which has several articles about managing and motivating people. (D. J. Reifer, 1993, IEEE Press.)

Working with Computers: Theory versus Outcome. This book is a collection of papers on human factors. They are mostly written by cognitive scientists with an interest in software development. (G. van der Veer, T. R. Green, J.-M. Hoc and D. M. Murray (eds), 1988, Academic Press.)

EXERCISES

22.1 Briefly describe the human memory hierarchy and explain why the organisation of this hierarchy suggests that object-oriented systems are easier to understand than systems based on functional decomposition.

22.2 What is the difference between syntactic and semantic knowledge? From your own experience, suggest a number of instances of each of these types of knowledge.

22.3 As a training manager, you are responsible for the initial programming language training of a new graduate intake to your company whose business is the development of defence aerospace systems. The principal programming language used is Ada, which was designed for defence systems programming. The trainees may be computer science graduates, engineers or physical scientists. Some but not all of the trainees have previous programming experience; none have previous experience in Ada. Explain how you would structure the programming training for this group of graduates.

22.4 What factors should be taken into account when selecting staff to work on a software development project?

22.5 Explain why keeping all members of a group informed about progress and technical decisions in a project can improve group cohesiveness.

22.6 Explain what you understand by 'groupthink'. Describe the dangers of this phenomenon and explain how it might be avoided.

22.7 You are a programming manager who has been given the task of rescuing a project that is critical to the success of the company. Senior management have given you an open-ended budget and you may choose a project team of up to five people from any other projects going on in the company. However, a rival company, working in the same area, is actively recruiting staff and several staff working for your company have left to join them.

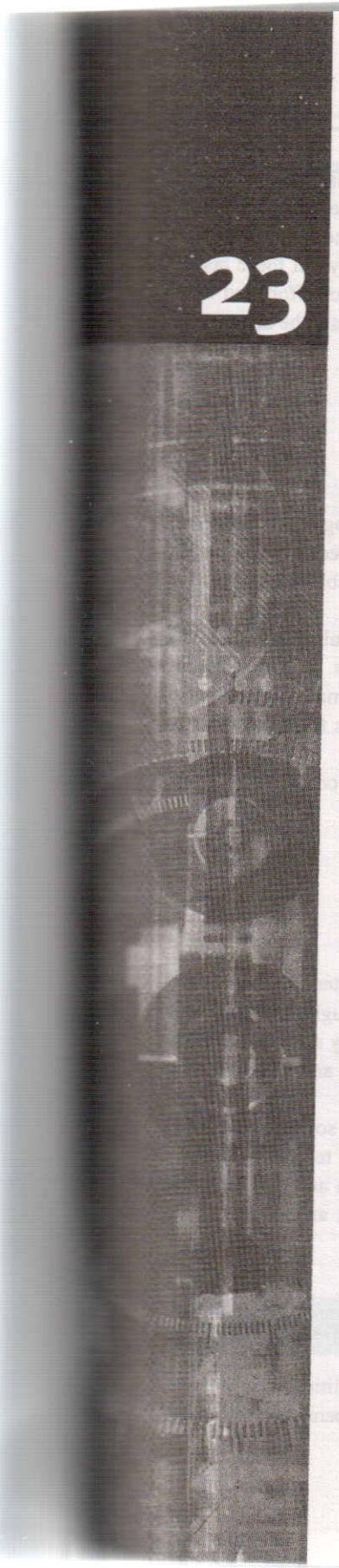
Describe two models of programming team organisation that might be used in this situation and make a choice of one of these models. Give reasons for your choice and explain why you have rejected the alternative model.

22.8 Why are open-plan and communal offices sometimes less suitable for software development than individual offices? Under what circumstances do you think that open-plan environments might be better?

22.9 Why is the P-CMM an effective framework for improving the management of people in an organisation? Suggest how it may have to be modified if it is to be used in small companies.

22.10 Should managers become friendly and mix socially with more junior members of their group?

22.11 Is it ethical to provide the answers which you think the tester wants rather than saying what you really feel when taking psychological tests?



23 Software cost estimation

Objectives

The objective of this chapter is to introduce techniques for estimating the cost and effort required for software production. When you have read this chapter, you will:

- understand the fundamentals of software costing and pricing and the complex relationship between them;
- have been introduced to three metrics that are used for software productivity assessment;
- appreciate that a range of different techniques should be used when estimating software costs and schedule;
- understand the principles of the COCOMO 2 model for algorithmic cost estimation.

Contents