Towards a software factory

DOI:
10.6100/IR356762

Document status and date:
Published: 01/01/1991

Document Version:
Publisher's PDF, also known as Version of Record (includes final page, issue and volume numbers)

TOWARDS A SOFTWARE FACTORY

Michiel van Genuchten

TOWARDS
A SOFTWARE FACTORY

Cover: Erik Knippers

ISBN: 90 900 4119 2

TOWARDS A SOFTWARE FACTORY

PROEFSCHRIFT

ter verkrijging van de graad van doctor aan de Technische Universiteit Eindhoven, op gezag van de Rector Magnificus, prof. dr. J.H. van Lint, voor een commissie aangewezen door het College van Dekanen in het openbaar te verdedigen op vrijdag 21 juni 1991 om 16.00 uur

door

Michael Jozef Ignatius Maria van Genuchten

Geboren te Eindhoven

druk: wibro dissertatiedrukkerij, helmond.

Dit proefschrift is goedgekeurd door de promotoren:

Prof.dr. T.M.A. Bemelmans

Prof.dr. J.C. van Vliet

en de copromotor:

Dr.Ir. F.J. Heemstra

CONTENTS

1 INTRODUCTION AND DEFINITION OF THE PROBLEM

1.1 Introduction

Software has become an important product during the first decades of its existence. It is now present in many forms. For example: embedded in electronic equipment or cars, in information systems that allow orders to be accepted automatically and in factory automation systems that enable a few men to run a chemical plant. The amount of software that needs to be developed has increased from virtually nothing around 1950 to an estimated value of 140 billion dollars worldwide in 1985 (Boehm 1988).

In spite of the huge investments in software, software development is often insufficiently controlled. Software development projects are usually late and software products repeatedly fail to meet the requirements. The control of software engineering is the subject of this book. We use the term control of software engineering instead of control of software development. The term engineering comprises both development and maintenance (Basili 1988). We will examine the current control of software engineering and explore the possibility of improving this control.

In the early days of software engineering, software products used to be isolated applications that were found in the secondary processes of an organization. They were usually not found in the organization's primary process or in its products. The software products had fairly stable specifications, because the stability of specifications was a major selection criterion for the applications to be automated. A typical system to be automated was a payroll system or an order handling system. The development process usually followed the lines of the waterfall model.

The control of software engineering was limited to the development phases. Development was usually done in projects. The definition of a project given by the Project Management Institute is "Any undertaking with a defined starting point and defined objectives by which completion is identified" (Wideman 1986). A project is a temporary organization form. This concept fitted in very well with software development in the early days: software development involved incidental, isolated development efforts, executed by specialists. The start and end of a project could be clearly identified. Things have changed over the years however. Three reasons will be discussed which explain the fact that traditional software control is no longer sufficient to control software engineering efforts.

The first reason is that development is only part of the job. Traditional software control usually considers only the development of software. Maintenance is regarded as an 'off-project activity' which is somebody else's problem. However, maintenance is a continuous activity that requires more effort nowadays than the initial development activities. Next to maintenance, other off-project activities are becoming more important. Examples are configuration management, process improvement and the development of methods and tools. A lot of continuously proceeding activities are required in software engineering nowadays. They cannot be controlled by a temporary structure such as project control.

The second reason is that the software product to be developed can no longer be considered an independent and isolated entity. Almost all the software products developed will have to fit to their environment. A new information system will have to fit to the existing information systems. For example, an order acceptance system was named as a typical system that was automated in the early days, whereas nowadays series of information systems that support the whole process of production control, from purchasing components to the delivery of an end product to a client, may be the goal of information systems development. The information systems have to be able to exchange or share data. As another example, a new release of embedded software will have to adjust to the hardware environment, and a new software package will have to be upwardly compatible with earlier releases. A software engineering department can no longer afford a 'greenfield approach'.

The third reason why traditional software control needs to be enhanced is that an exclusive emphasis on ongoing projects prevents from extensive reuse of software. The time and budgetary constraints that surround a project do not allow for the additional work required to make a software product available for future reuse.

We will argue in this book that software engineering control will have to adapt. The lines along which the changes are taking place will be described. Firstly, a change from development control to product control is envisaged. Development control limits itself to the initial development of a product. Product control involves control of a software product over its entire life cycle; it includes maintenance. Secondly, a change from product control to multiproduct control is pictured. Multiproduct control implies control of a range of software products over their life cycle. The expansion of the control focus allows to emphasize reuse of software, which is believed to be a major contributor to software productivity improvement. Multiproduct control is typified by the term software factory, which explains the title of this book. The discussion of control will research the analogy between production control and software engineering control. The experiences in production control will be reused as much as possible. Furthermore, we will pay special attention to data collection and information systems for software engineering control.

1.2 Importance of software engineering

Software has become an important product over the years and its importance is still growing. The annual cost of software in the United States was 70 billion dollars in 1985 (Boehm 1988). The average growth is estimated at 12 percent per year (Boehm 1988, Humphrey 1989), consisting of a five percent increase in personnel costs and a seven percent increase in the number of personnel. Some software cost trends are shown in Figure 1.1.

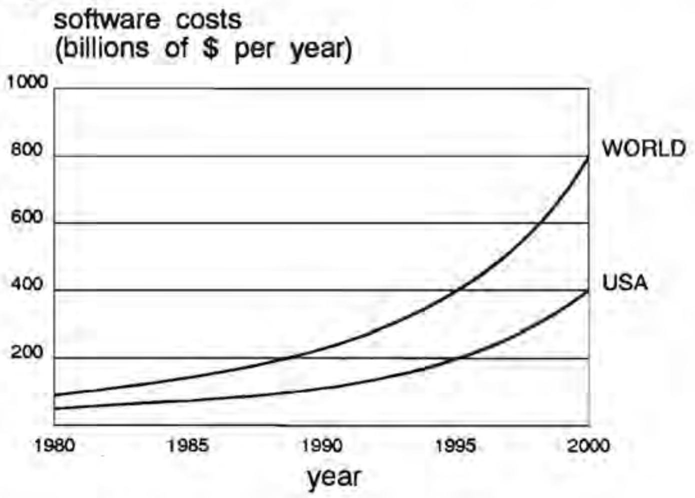

Figure 1.1 Software cost trends (Boehm 1987).

Figure 1.1 shows that a 12 percent growth rate would result in an 800 billion dollar software market in the year 2000. The importance of software will be illustrated with examples of three important application areas: information systems, factory automation and software products.

Information systems
The importance of information systems is illustrated by a survey mentioned by Davis (1987). This investigated the extent to which companies depend on the information processing capabilities of computers. The companies concerned were asked how long different business functions would be able to operate without information processing capabilities. The results are given in Figure 1.2.

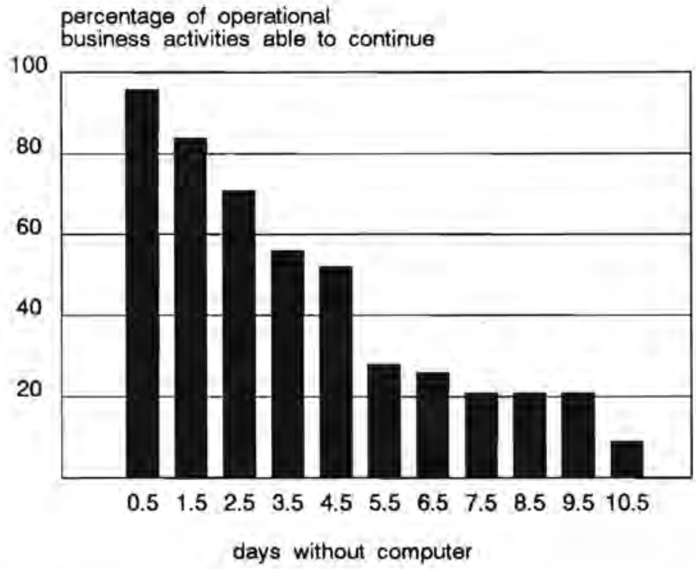

Figure 1.2 Decline in operational business activities following a complete data processing failure (Davis 1987)

Figure 1.2 shows that after 5.5 days only 28 percent of the activities would be functioning. In case of finance companies only 13 percent of the activities would still be functioning after the same period.

Factory automation
Another area in which software plays an important part is the automation of production processes or factory automation. Software costs account for already 40 percent or more of the factory automation facility costs in Japanese companies (Sakurai 1988). The same publication mentions an example of a Mitsubishi plant in which the software represents over 70 percent of the factory automation costs. The importance of controlling lead time in software development increases if factory automation is involved. If the software is late this will hold up production. The cost of the lost production may dwarf the software development costs.

Software products and services
The third category to be distinguished is software products and services. This can either be software embedded in products, software sold as an independent product, or all kinds of information services. The fact that software comprises a considerable part of the development cost of, for instance, electronic products is well-known. For example: 70 percent of the designers at Hewlett-Packard are involved in software and more than 50 percent of the new projects are software-based (Ward 1989). The

amount of software is less known. Four examples of embedded software are given in Table 1.1.

Table 1.1 Examples of embedded software (Schendler 1989).

PRODUCT	LINES OF CODE	LABOUR REQUIRED (in man years)	DEVELOPMENT COSTS (in millions of dollars)
Space Shuttle	5,600,000	22,096	1,200
Lincoln Continental	83,517	35	1.8
Citibank teller machine	780,000	150	13.2
IBM checkout scanner	90,000	58	3

The table demonstrates the importance of software in several areas. It was predictable that information processing and, as a consequence, software would be important for controlling and simulating the flight of a space shuttle. Software in electronic equipment, such as a teller machine and a checkout scanner, might also have been expected. The development costs of the software for the Lincoln Continental show that software has also found its way into products like cars and is rapidly becoming more important in areas where it was not even considered a short time ago.

Software can also be sold as an independent product. The packaged software sold in the United States in 1989 was $ 23.7 billion (Shaw 1990). It is projected to grow to $37.5 billion in 1992. Examples of package software are found in the personal computer environment. Examples are text processing packages, spreadsheets and database packages. The package software applications are now moving to all kind of computers. The degree of standardization will determine to a large extent the speed of proliferation of standard software packages in the mini and mainframe market. Standards should make application software less dependent on hardware. Information services are another new business that is driven by software. Examples are services such as public or proprietary databases, electronic data interchange and value added networks.

Summarizing this section, it might be said that software has evolved from playing a minor role in the secondary processes of organizations to a significant role in the primary processes and products of many organizations. This evolution and its consequences will be discussed later on.

1.3 Problems of software engineering

The control of software engineering has been unable to keep pace with the changes. Consequently, software engineering is now confronted with some serious control problems. Three of these will be addressed in this section:
- Delays in development,
- Maintenance, and
- Insufficient productivity improvements.

Delays in development
Delays in software development projects occur regularly, as do overruns of the planned cost. Some examples of delays in a number of well-known personal computer packages are shown in Table 1.2. The late deliveries were caused by overruns in development (Manual 1989).

Table 1.2 Examples of delays (Manual 1989)

Company	Product	Planned release date	Actual release date
Ashton Tate	dBase IV	7/88	6/89
Lotus	1-2-3, rel. 3.0	6/88	6/89
	1-2-3, rel. 2.2	6/88	Third quarter 89
Microsoft	Word 4.0 MAC	10/88	4/89
	Word 5.0 PC	1/89	5/89
	SQL server	12/88	4/89

Delays in software development can have serious consequences because so many of the operations are dependent on software. Delay in information systems development can cause an operation to close down, as already discussed in section 1.2. A consequence of factory automation software being late is postponement of the production. Embedded software that is late will postpone the shipment of the products in which it is embedded. This will be critical if the delay involves software embedded in thousands of products such as televisions or cars.

The same applies to companies which depend on the sales of their software products. Their revenues and profits are determined by these. Losses of major software development companies have been blamed on the late delivery of new releases of software packages. The stock prices of these companies are influenced by announcements of delays in shipping several products. An example is the Microsoft stock price which fell overnight after the company had announced delays (Nelson

1989). The delay in the release of the Lotus 1-2-3 software package is shown in Table 1.2. The effect on the company can be deduced from the contributions of the product to the 700 million dollar total revenues of the Lotus company shown in Figure 1.3.

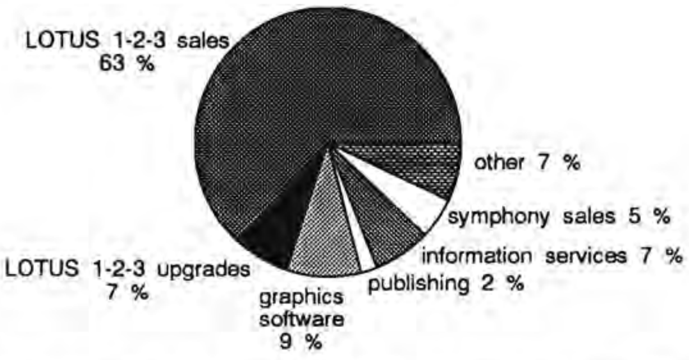

Figure 1.3 Estimated contributions to Lotus revenues (Wilke 1990)

The delays and cost overruns in software development were the reasons for the research described in this book. Chapter 2 discusses three empirical studies on delays in software development. The studies investigated the reasons for the delay in development projects and explored actions for improvement.

Maintenance

The second problem to be addressed is maintenance. Conte (1986) claims that 60 percent of the software engineering effort goes on maintenance. Lehman (1984) states that 70 percent of the expenditure on software is incurred after initial installation. The maintenance problem is best visualized by the software iceberg: the development costs are all we see and all we seem to care about. The main software engineering costs however, are hidden under the surface: the maintenance costs. The control of software development is only concerned with the minority of the software engineering effort if we assume that Conte's and Lehman's statements are true. This book will argue that control should be extended from development alone to the entire life cycle of the product. How this can be done will be explained in chapter 4.

Insufficient productivity improvements

The supply of software has not been able to keep up with the demand. This results in a software backlog that may rise to several years. The U.S. Air Force Data Systems Design Office has identified a four-year backlog of important data processing software functions (Boehm 1988). Boehm describes two problems related to the backlog. First, without software, it is impossible to achieve the potential productivity gains. It has been estimated that 20 percent of the productivity gains in the U.S.A. have been achieved through automation and data processing (Boehm 1988). The second problem is that the backlog creates a situation which yields a great deal of bad software. The backlog "creates a personnel market in which just about anybody can get a job to work off this software backlog, whether they are capable or not" (Boehm 1988, page 1463).

The productivity improvements in software development have been estimated by Boehm and are shown in Figure 1.4. The vertical axis shows the software productivity, given in equivalent machine instructions per man month. The horizontal axis gives the domain of applicability. This independent variable is introduced since the main productivity improvements have been achieved by exploiting the knowledge of particular application domains.

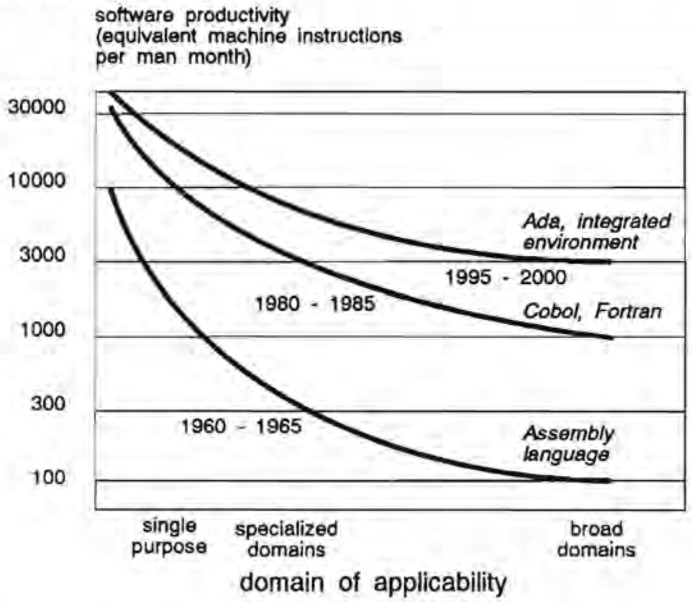

Figure 1.4 Software technology and productivity trends (Boehm 1987)

Figure 1.4 indicates that further productivity improvements are expected over the next decade. The productivity gain is estimated at an order of magnitude of two over a period of thirty years. However, the productivity improvement will be unable to keep up with the needs, assuming the growth estimates of over 10 percent per year are right. Boehm concludes therefore that the demand for new software is increasing faster than our ability to develop it.

Reuse of software is widely believed to be a key issue for improving software productivity. The fact that software is an intangible product has the advantage that it can be reproduced with negligible reproduction costs. Reuse has not really taken off, despite the fact that its advantages are clear and acknowledged (Biggerstaff 1987). The opportunities are there: Capers Jones (1984) claims that only 15 percent of the software written is unique. Lanergan and Grasso (1984) studied 5000 Cobol programs, 50 of them in detail. They concluded that 40 to 60 percent of the code was redundant and could be standardized. They claim that 60 percent reuse can be achieved. This not only results in a decrease in development costs, but also in a decrease in maintenance costs, because the reused components have already been tested in practice. The actual reported reuse rates are much lower. For instance, a study by Cusumano (1989) reported reuse rates of 15 and 35 percent in American and Japanese companies, respectively. Reuse of software will be one of the subjects in this book. Chapter 5 describes some of the problems that will have to be overcome to allow for extensive reuse. It will explore how the control of software engineering should be adapted to allow for extensive reuse of software.

1.4 Definition of the problem

The importance of controlled software engineering activities, as well as the current problems in control, lead to the following definition of the problem.

Definition of the problem
Software engineering has changed over the years due to changes in the products, the process followed and the resources used. The control of software engineering has not kept up with the changes and is therefore often insufficient to meet the demands. It is unclear how software engineering can be controlled in the new circumstances and what information is required to control it.

This definition mentions three sources of changes in software engineering activities: changes in the products engineered, the process followed and the resources used. These will discussed in more detail later on. The definition mentions two subjects that require research, namely the control concept and the information concept. The aim of this study can therefore be derived from the problem statement.

Aim
1 Determine the characteristics of the control concept of software engineering that fit in with the changed practices and demands
2 Derive the characteristics of an information system that supports the control concept.

The first goal is to determine the characteristics of a control system that is capable of controlling both current and future software engineering practices. The process of software engineering will have to be studied for this purpose. The second goal is to identify the characteristics of an information system that supports the control. These characteristics will be derived from the characteristics of the development process itself and its control.

1.5 The Process - Control - Information model

This book follows the lines of the Process - Control - Information model (abbreviated as the PCI model) as proposed by Bemelmans (1986). The PCI model distinguishes a process to be controlled, a control system and an information system within an organization. The distinction between the process, the control system and the information system are found in systems theory. The relations between the three elements mentioned are given in Figure 1.5.

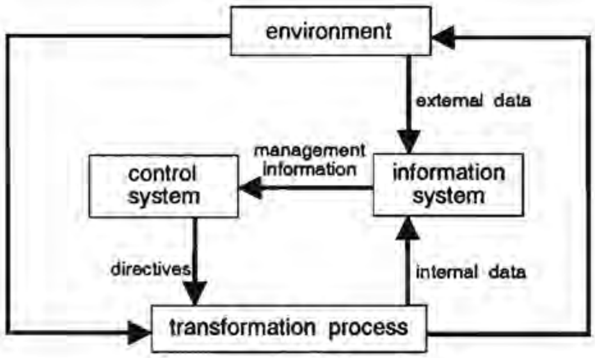

Figure 1.5 Process control

Figure 1.5 shows that a system receives input from the environment and produces output to the environment. The system to be controlled receives directives from the control system and supplies internal data to the information system. The information system translates the internal data and the external data into management information for the control system. Figure 1.5 is a descriptive model of the relation between a system to be controlled, a control system and an information system.

The PCI model goes one step further and could be considered as a constructive tool which supports the specification and development of control systems and information systems. It uses insights from systems theory and applies the contingency approach to select appropriate control systems and information systems for particular systems to be controlled. The PCI model states that the characteristics of a particular system determine which of the possible control systems is suitable for controlling that system. The selection of a control system is a matter of matching process characteristics with characteristics of the possible control systems. Examples of possible control systems in production control are job shop control versus flow shop control. In a similar way, the characteristics of the process and the control system determine the outlines of an appropriate information system. The PCI model is given in Figure 1.6.

$$\begin{tikzcd}
P \arrow[r] \arrow[rr, bend right] & C \arrow[r] & I
\end{tikzcd}$$

Figure 1.6 PCI model (Bemelmans 1986)

The PCI approach was originally developed as a support tool for the design of an information systems concept for production control in industrial companies. We will apply it to software engineering, following Heemstra (1989). The PCI model describes which characteristics of the primary process determine the choice of an appropriate control system. The characteristics are:

- Product characteristics, such as: the product range, the composition of the products and the demand volumes for the product.
- Process characteristics, such as: phased development, the limited measurability of the process and the involvement of the user in the development process.
- Resource characteristics, such as: specialization, engineering in teams and the availability of resources.

The PCI model also describes which control characteristics are a decisive factor in the specification of an information system. They include:

- The set of goals of an organization,
- The production control situation, and,
- The organization of production resources (Bemelmans 1986, Heemstra 1989).

The characteristics of the system to be controlled and the control concept determine the outlines of the information system concept, characterized by its functional and performance requirements.

1.6 Contents

This book consists of 10 chapters. It can be divided into three parts. The problem is explored in chapters 1, 2 and 3. Chapters 4, 5 and 6 describe how the control of software engineering should be improved to keep up with the changing demands. Chapters 7, 8 and 9 deal with the information required in order to be able to control software engineering, as described in the preceding chapters.

Chapter 2 describes empirical studies on reasons for delay in software development. The goal of these empirical studies is twofold: firstly, to gain an insight into reasons for delay. The insight guides the definition of the problem. The insight is also used to achieve a second goal, namely to support the software engineering departments concerned. The insights gained should lead to actions for improvement, enabling future projects in the department to follow the plan more closely. The evolving circumstances in which software development takes place are described in chapter 3. The consequences for control, both in the early days and in the current situation, are discussed.

Chapter 4 deals with the issues quality and maintenance. The issues are addressed to be able to describe the basics of what we define as product control, a stage in which the scope of control has been expanded from development only to the entire life cycle of a software product. Chapter 5 discusses the reuse of software and describes the basics of multiproduct control, a stage in which the scope of control has been further expanded to include the control of a range of software products. Chapter 6 describes how an organization can move from the current state of software engineering control to the software factory, as characterized in chapter 5. We argue that a certain level of process control has to be achieved before an organization can apply product or multiproduct control.

The chapters 7, 8 and 9 deal with information and information systems that support the control of software engineering. Chapter 7 describes a number of reference information systems, such as information systems that have been proposed in the literature to support the control of software engineering and information systems for production control in factories. The reference systems are compared to each other and assessed with respect to the software factory. Chapter 8 provides a data model for an information system for a software factory.

Chapter 9 brings information and data collection in software engineering down to earth and back to current software engineering practice. It shows examples of how several software engineering departments have collected information on their software engineering processes. The application of the data collection techniques described in chapter 9 are a first step on the way to a software factory, as described in this study.

Chapter 10 presents a summary of the major issues and provides some recommendations for future research.

2 WHY IS SOFTWARE LATE ?
Empirical studies of reasons for delay in software development

2.1 Introduction

In practice there is frequently a difference between the planned and the actual progress of a software project. A recent survey in the Netherlands (Siskens 1989) shows that in 30 percent of large projects the planned costs and lead time are overrun by more than 50 percent. The reasons why projects do not run according to plan are less clear, however. It is important to reveal the reasons for delay because an insight into these reasons can lead to actions for improvement enabling future projects to follow the plan more closely. This chapter describes a number of empirical studies regarding the reasons for differences between plan and reality which were carried out in 1988 and 1989 in various software development departments in a multinational organization.

This chapter consists of seven sections. Three surveys of overruns in development that have been described in the literature will be discussed in section 2.2. Section 2.3 explains the definition and planning of the studies. Sections 2.4 through 2.6 discuss the studies in three different departments. Each of these sections consist of three parts. The first part discusses the performance of the study in the particular department. The second subsection discusses the results and the third subsection deals with their interpretation. This includes the interpretation by the project leaders involved. The overall conclusions of the three empirical studies will be described in section 2.7.

2.2 Surveys on the overrun of development projects

Three empirical studies concerning the overrun of development projects will be discussed in this section. These studies will be referred to as surveys. The surveys will be compared with the studies described in sections 2.3 through 2.6 of this paper.

Survey by Jenkins, Naumann and Wetherbe (1984)
Jenkins e.a. interviewed the developers of 72 information system development projects in 23 major U.S. corporations. The aim of the survey was to collect empirical data on the systems development process in organizations. The average duration of the projects was 10.5 months. Over 70 percent of the projects took less than 1000 person days to finish. The users of the systems developed stated that they were 'satisfied' to 'very satisfied' with the result in 72 percent of the projects. The

relative effort overruns are given in Figure 2.1.

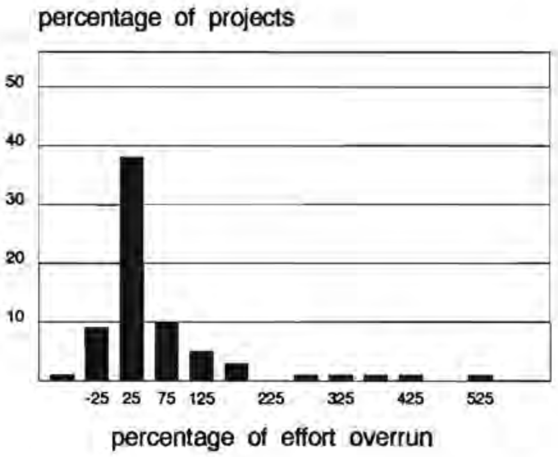

Figure 2.1 Distribution of relative effort overruns (Jenkins 1984)

The mean of the classes of overrun are given on the horizontal axes. For example: 38 percent of the projects had an overrun of between zero and 50 percent. Nine percent of the projects had an underrun of between zero and 50 percent. The average effort overrun was 36 percent. The relative schedule overruns are given in Figure 2.2.

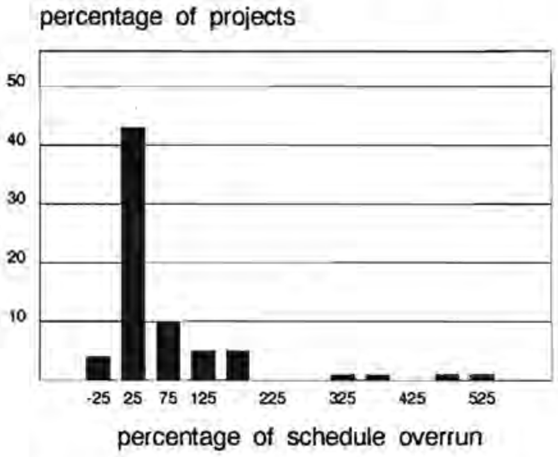

Figure 2.2 Distribution of relative schedule overruns (Jenkins 1984)

The average schedule overrun was 22 percent. Figure 2.2 shows that 40 percent of the projects had an overrun of between zero and 50 percent. One conclusion of Jenkins e.a. was that the cost and schedule overruns seem to be uniformly distributed among large, medium and small projects. They did not look into the reasons for delays and overruns.

Survey by Phan, Vogel and Nunamaker (Phan 1988, Phan 1989)
Researchers at the University of Arizona attempted to determine why the planned lead times and costs of information system development projects were overrun (Phan 1988, Phan 1989). Questionnaires were sent to 827 members of the American Institution of Certification of Computer Professionals. The survey yielded 191 responses. The respondents were involved in projects with an average duration of 102 person months. On average, the lead time was 14 months and 17 people worked on a project. The average cost overrun was 33 percent, similar to the 36 percent overrun reported by Jenkins e.a.

The survey comprised 100 questions. In relation to 72 of these the respondents were asked to recall the frequency with which the events occurred as a) always, b) usually, c) sometimes, d) seldom/rarely or e) never. Over 70 percent of the respondents claimed that user requirements and expectations were usually met. Figure 2.3 shows the prevalence of cost overruns.

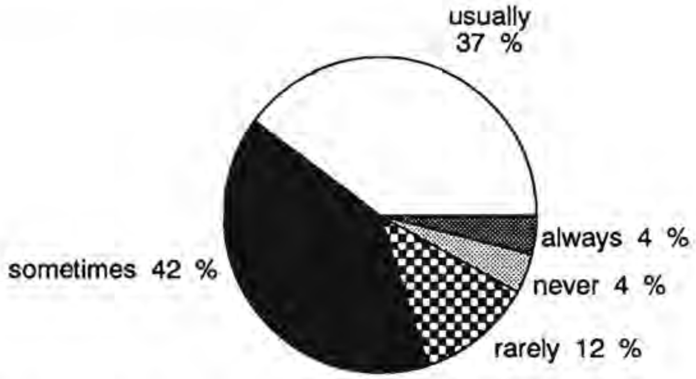

Figure 2.3 Prevalence of cost overruns (Phan 1989)

Only 16 percent of the respondents answered that they never or rarely had cost overruns. Cost overruns were usual for 37 percent of them. Figure 2.4 shows the prevalence of schedule overruns.

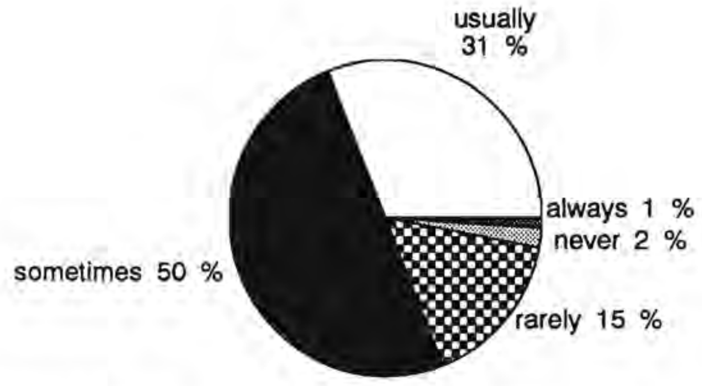

Figure 2.4 Prevalence of late deliveries (Phan 1989)

Figure 2.4 shows that more than 80 percent of the respondents stated that their projects were sometimes or usually late. The survey also addressed the reasons for cost overruns and late deliveries. According to 51 percent of the respondents over-optimistic estimation was usually a reason for a cost overrun. Almost 50 percent stated that frequent changes in design and implementation were usually a reason for a cost overrun. Nine percent stated they were always a reason. The survey also investigated why the product lead times were overrun. Over-optimistic planning was a reason to which 44 percent usually attribute the delay. Minor and major changes were usually a reason for 33 and 36 percent of the respondents, respectively. The lack of software development tools was only mentioned by 17 percent as a usual reason.

The four actions most frequently taken to regain control over delayed projects were:
1) upgrading the priority of the project,
2) shifting part of the responsibility and obligations to other groups,
3) renegotiating the plan and schedule, and,
4) postponing features and upgrades to the next release.

Survey by Thambain and Wilemon (1986)
Aim of a field study by Thambain and Wilemon was to investigate the practices of project managers regarding their project control experiences. The scope of the survey was not confined to software engineering projects; the leaders of electronics, petrochemical, construction and pharmaceutical projects were interviewed. Data was collected from 304 participants in project management workshops or seminars. Those questioned had an average of five years' experience in technical project management. The average lead time for the projects was one year and on average eight people worked on a project.

Among other things, the survey investigated what the project leaders and their superiors (such as senior functional managers or general managers) believed to be the reasons for cost and lead time overruns. The reasons for overruns were arranged in order of importance by project leaders and general managers. The results are given in Table 2.1.

Table 2.1 Directly observed reasons for schedule slips and cost overruns

RANK BY		PROBLEM	Agreement between general and project management
General managers	Project managers		
1	10	Insufficient front-end planning	Disagree
2	3	Unrealistic project plan	Strongly agree
3	8	Project scope underestimated	Disagree
4	1	Customer/management changes	Disagree
5	14	Insufficient contingency planning	Disagree
6	13	Inability to track progress	Disagree
7	5	Inability to track problems early	Agree
8	9	Insufficient number of checkpoints	Agree
9	4	Staffing problems	Disagree
10	2	Technical complexity	Disagree
11	6	Priority shifts	Disagree
12	10	No commitment by personnel to plan	Agree
13	12	Uncooperative support groups	Agree
14	7	Sinking team spirit	Disagree
15	15	Unqualified project personnel	Agree

It is striking to note that the project leaders and the general managers do not agree on the importance of nine of the fifteen reasons. According to the researchers, "the practical implication of this finding is that senior management expects proper project planning, organization, and tracking from project leaders. They further believe that the external criteria, such as customer changes and project complexities, impact project performance only if the project had not been defined properly and sound management practices were ignored. On the other hand, management thinks that some of the subtle problems, such as sinking team spirit, priority shifts and staffing are of lesser importance" (Thambain 1986).

The researchers also investigated the reasons that caused the problems referenced in Table 2.1. These less obvious reasons were called 'subtle reasons', which can be classified in five categories.

- Problems with organizing the project team
- Weak project leadership
- Communication problems
- Conflict and confusion
- Insufficient upper management involvement.

Obviously, the subtle reasons cited by the project leaders and general managers were not technical reasons, but related to organizational, managerial and human aspects.

2.3 Definition and planning of the study

Definition of the study

The framework of experimentation, as proposed by Basili, Selby and Hutchens (1986) will be used to define the study that is described in this chapter. According to this framework, a definition of an experiment consists of six parts: motivation, object, purpose, perspective, domain and scope. The motivation of this study was to gain an insight into the reasons for delay in order to be able to improve the control of future development projects. This new insight should lead to actions for improvement designed to enable future projects to follow their plan more closely. The object of the study was defined as the primary entity examined (Basili 1986). The object in this case was software development activities. Projects can be analyzed on various levels of detail, namely as a whole (as done by Jenkins e.a., see section 2.2), at phase level or at activity level. Data was collected and analyzed at the activity level in this study because experience has shown that a project generally does not overrun because of one or two main problems, but rather because of a large number of minor problems. According to Brooks: "How does a project get one year late? One day at a time" (Brooks 1975). These small problems could almost certainly be overlooked if data were collected at project level. In this study, an activity was defined as a unit of work that is identified in a plan and can be tracked during its execution. A typical activity may be the specification of a subsystem, the design of a module or the integration of some modules.

The purpose of the study was to evaluate the reasons for delay. This was done from the perspective of the project leader. The domain studied was software projects. The scope of the study covered six development projects in one software development department. The definition of the study is summarized in Table 2.2.

Table 2.2 The definition of the study

Motivation	To increase insight into the reasons for delay
Object	Software engineering activities
Purpose	To evaluate reasons for delay
Perspective	Project leader
Domain	Project
Scope	Six projects in one development department

Planning the study

Motivation of the study was to gain an insight into the reasons for delay in software development. The kind of questions the study aimed to answer were:

- What are the predominant reasons for delay?
- What is the distribution of the reasons for delay?
- How is the delay distributed over the phases of a project?
- Which actions for improvement can prevent delay in future projects?

The following basic principles were used for data collection.

1) The control of a project refers to the control of quality, effort and lead time. The study was based on the assumption that an activity is only completed when the (sub)product developed fulfils the specifications. In other words, if the quality of the product developed is adequate. In the department concerned this was monitored by reviews and testing. This assumption allowed attention to be focused on the collection of data relating to time and effort.

2) Data collection focused on the differences between a plan and reality. All planning data were obtained from the most recently approved plan. If a project was officially replanned, the new plan was taken as the starting point for the comparison between the plan and reality. The consequences of a replan will therefore not show up in the measurements. For example, the study described in section 2.4 involved six projects; one of them was not replanned, four were replanned once and one was replanned twice during the study. It might be argued that the differences between plan and reality were greater than the measurements will show.

3) The third principle was that data collection should not take the project leaders much time. This was a condition stated by the development department.

The definition of the study and the above principles resulted in a one-page data collection form. This consisted of a table with the data to be collected for each activity and a classification of reasons for delays. It is shown in Table 2.3.

Table 2.3 Data determined for each activity

	PLANNED	ACTUAL	DIFFERENCE	REASON
EFFORT	-	-	-	-
STARTING DATE	-	-	-	-
ENDING DATE	-	-	-	
DURATION	-	-	-	-

The planned and actual effort were expressed in hours. The starting and ending dates were given in weeks. The duration of an activity was defined as the calendar period between the starting and ending dates. All planning data were obtained from the most recent approved plan. The difference column indicated if there was any difference between the plan and reality. The reasons for three types of differences were distinguished in the final column:

- The reason for a difference between the planned and actual effort
- The reason for a difference between the planned and actual starting date
- The reason for a difference between the planned and actual duration.

A reason for the difference between the planned and actual ending date was not mentioned because this difference can be explained by the difference in the starting date and the difference in duration.

Obviously, many of the data in Table 2.3 were not only kept for the purpose of this study: the planned and actual hours and duration were also required for normal project control purposes. The survey mentioned earlier showed that in practice data of this kind are not kept as a matter of course; as many as 50% of the respondents claimed that they did not record progress data during the course of their projects (Siskens 1989). In this study, the project plans provided the planned effort, starting date and ending date. The clerical office provided the actual data, which was collected on the basis of time sheets. The actual data was validated in interviews with the participating project leaders every other week.

The final column was filled in specially for this study. This was performed by the project leader who, in consultation with the researcher, determined the reasons for differences between planning and reality. A classification was used to determine a reason. This was done for two purposes. First, the classification gave structure to the reasons identified and allowed results to be compared. Second, the classification saved time for thinking up reasons. Six groups of possible reasons for differences were identified in the classification. The division into six groups was based on a discussion with the project leaders concerned and a previous study (Heemstra 1989). The groups are listed in Table 2.4.

Table 2.4 Groups of reasons

Group of reasons	Description
	Reason relating to
capacity-related	the availability of the developers
personnel-related	the experience of the developers
input-related	conditions which must be fulfilled
product-related	the software product to be developed
organization-related	the organization in which the development takes place
tools-related	the tools used to develop the software
other	none of the previous categories

The division into six groups has proved to be valid for several (software) development departments. In fact, similar studies using the same groups of reasons were applied in a number of departments. About thirty reasons for delay were found within the groups. A first classification of reasons was identified after a discussion with the participating project leaders. Similar studies in other departments showed that the reasons were specific to the engineering environment in question because of differences among the software engineers, the type of software developed and the organization of the department. This confirms the measurement principle which states that metrics must be tailored to their environment, as formulated in (Basili 1988). A definite classification of reasons was identified after a pilot study. The classification of reasons, as used in the study described in section 2.4, is displayed in Table 2.5.

Table 2.5 The classification of reasons as used in this study.

CAPACITY-RELATED REASONS	
11	capacity not available because of overrun in previous activity
12	capacity not available because of overrun in other activity
13	capacity not available because of unplanned maintenance
14	capacity not available because of unplanned demonstration
15	capacity not available because of other unplanned activities
16	capacity not available because of other causes
19	other
PERSONNEL-RELATED REASONS	
21	too little experience with development environment
22	more inexperienced people in team than expected
29	other
INPUT-REQUIREMENTS NOT FULFILLED	
31	requirements late
32	requirements of insufficient quality
33	(specs of) delivered software late
34	(specs of) delivered software of insufficient quality
35	(specs of) hardware late
36	(specs of) delivered hardware of insufficient quality
39	other
PRODUCT-RELATED REASONS	
41	changing requirements during activity
42	changing of the interfaces during the activity
43	complexity of application underestimated
44	more problems than expected with performance requirements or memory constraints
45	product of insufficient quality developed (redesign necessary)
49	other
ORGANIZATION-RELATED REASONS	
51	less continuity in project staffing than expected
52	more interruptions than expected
53	influence of software Quality Assurance
54	bureaucracy
59	other
TOOLS-RELATED REASONS	
61	development tools too late or inadequately available
62	test tools too late or inadequately available
69	other
OTHER	
71-79	

A reason labelled "other" was included in each category because it was not exactly clear at the start of the study what reasons could be expected. During the study, however, it was found that the reason "other" only needed to be used rarely.

If the actual hours, starting dates and ending dates were recorded, little time was needed to determine the reason for any difference. In practice, determining the actual hours, starting and ending dates was found to take a great deal more time than determining the reasons. This was done in an interview once every other week with the project leader in question. It was important to analyze the data during the project because it would have been difficult to collect accurate data after the project had finished and validating the data would have been almost impossible. Several reasons could be given for each difference, with a maximum of four. In practice it was found that the difference could usually be ascribed to one reason.

Comparison of the study and the surveys
The study definition that was just described will be compared with the surveys, as in section 2.2. They will be compared with respect to their motivation, object, scope and the data collection technique used. The motivation of the survey by Jenkins e.a. was to conduct empirical research on the information systems development process in organizations. The survey by Phan e.a. aimed to collect factual data with regard to the management and control of software projects. Thambain e.a. investigated the practices of project managers in relation to their project control experience. The motivation of the study described in this paper was to gain an insight into reasons for delay.

The object of the three surveys was projects, Jenkins e.a. and Phan e.a. took information systems development projects as their object, while Thambain's survey was concerned with engineering projects. The object of the study described in this paper is the activities performed within a project. The scope of the surveys covered multiple projects in multiple organizations. This study is limited to development projects within three departments. The last and most obvious difference between the surveys and the study described in this paper is the data collection technique. Jenkins e.a. conducted interviews on 72 completed projects. Phan e.a. sent out a questionnaire and received 143 qualified responses. Thambain e.a. collected questionnaires from 304 participants in workshops and seminars. In the study described here, data were collected and validated during the execution of the projects on the basis of a number of interviews with the project leaders and the available project data. Because of the differences mentioned, the study and the surveys were complementary, rather than similar.

2.4 A systems software department

Carrying out the study

The study described in this section took place in a software development department in the second half of 1988 through the first half of 1989. The department was concerned with the development and integration of system software in the operating systems and data communications fields. The department employed 175 software engineers and covered a range of 300 products. Six representative projects in the department were selected for the study. A total of 160 activities in the projects were studied. The data in Table 2.3 were determined for each activity; these were the planned and actual hours and the starting and ending dates. The average duration of an activity was 4 weeks and the average effort was close to 100 person hours.

When determining the actual effort and the actual starting and ending dates, the existing registration was found to be of limited value because some of the data on the actual implementation of the project were not available in a usable form. Recording starting and ending dates was no problem because management emphasized the control of duration. Starting and ending dates were reported at the progress meetings. The number of hours spent on each activity was difficult to determine in the first part of the study for two reasons. First, the lack of reliability of the recorded hours. The validation of the data by project leaders showed that the difference between the recorded hours and the impression of the project leader was sometimes too large to be credible. Second, the numbering of the activities by the project leaders was found not to be unique in every case. This meant that the hours recorded could not be related to activities. The actual hours were not recorded if the effort could not be related to activities or the validation indicated that something was wrong. As a result, the planned and actual effort could only be compared for 97 of the 160 activities.

Results

The most important results of the study are presented in the form of four figures. Figure 2.5 shows the frequency distribution of the difference between the planned and the actual duration of the activities.

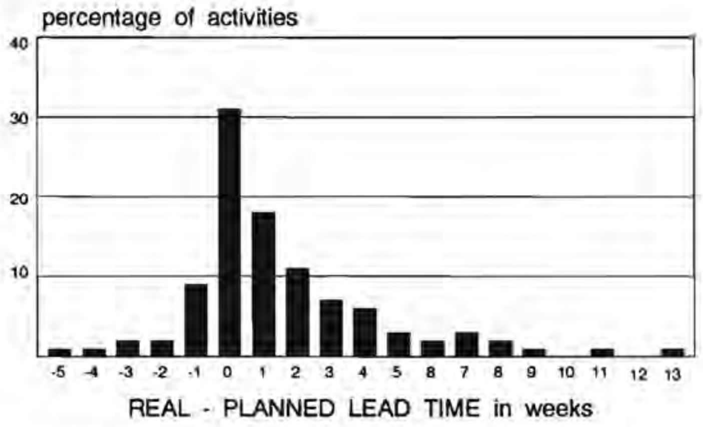

Figure 2.5 Frequency distribution of the difference between the planned and actual duration (N=160)

Figure 2.5 shows that over 30 percent of the activities were finished according to plan. Nine percent show a one week underrun, 17 percent show a one-week overrun. Figure 2.6 shows the relative difference between the planned and actual effort for 97 activities. This figure relates to only 97 activities due to the problems that occurred in the recording of hours for each activity.

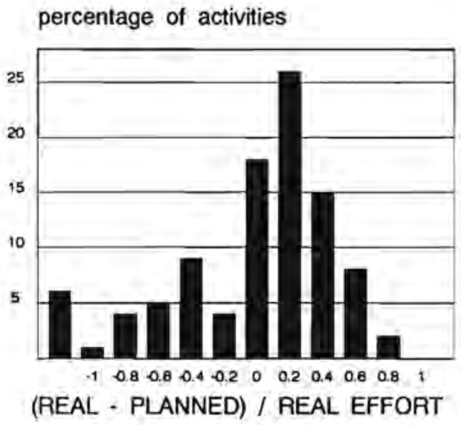

Figure 2.6 Frequency distribution of the relative difference between the planned and actual effort (N=97)

Figure 2.6 shows that about 50 percent of the activities overran their plan by more than ten percent. About 30 percent underran their plan by more than 10 percent. The comparison of the planned and actual figures yielded some useful insights. It showed, for instance, that the relative differences between planned and actual effort increased for the subsequent phases of the project; the delays and overruns increased towards the end of the project. The same result has been found in other engineering environments. This fact makes it possible to discourage the idea that delays can be overcome as the project progresses.

Figures 2.7 and 2.8 present the reasons for the delays and overruns. During the study it was found that many activities started too late. Figure 2.7 shows the distribution of the reasons for activities starting too late. These were divided into groups as identified in Section 2.3. Note that when an activity started too late because of a delay in a previous activity, it was recorded as reason 11, a capacity-related reason (see Table 2.5). This explains the large capacity section in Figure 2.7.

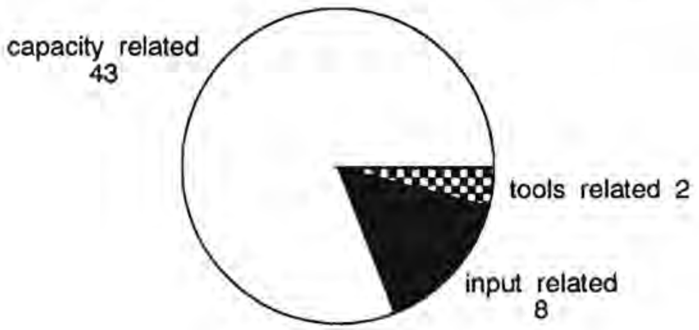

Figure 2.7 Distribution of reasons for differences between the actual and planned starting date (N=53)

The input-related reasons had to do with the late delivery of hardware components developed in parallel with the software. The start of the software development activities was also delayed because of this. The reasons for the differences between the planned and actual duration are listed in Figure 2.8.

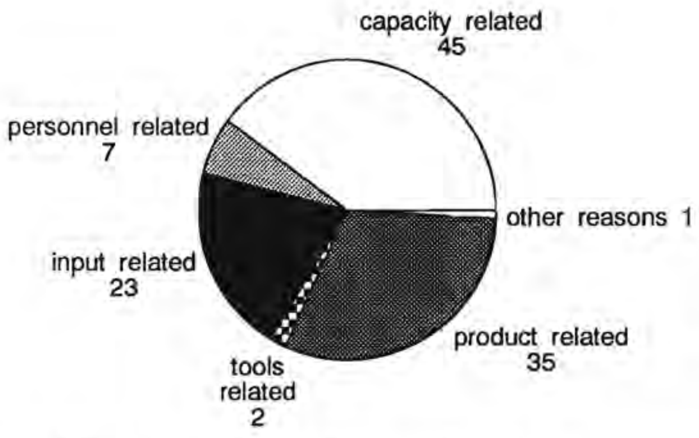

Figure 2.8 Distribution of the reasons for differences between the actual and planned duration (N=113)

Within the groups identified it was found that the most frequent reasons for differences between the planned and actual duration were:

- reasons 12 to 16: "more time spent on other work than planned". These reasons were named in 27 percent of the cases.
- reason 43: "complexity of application underestimated". Some outsiders blame all the software delays on underestimation. In this case, underestimation was given as an explanation in about 20 percent of the cases.

Interpretation of the results

The results were interpreted during a meeting attended by the project leaders taking part, the department manager and the researcher. In the researcher's opinion data of this kind should, in the first place, be analyzed together with the people involved in data collection. Six reasons for this are given. First, it is the engineers', project leaders' and manager's job to control software development. They should be supported with all the available data. Second, those involved represent the knowledge of software development in the department concerned; this knowledge is needed to interpret the results. Third, those involved can assess the feasibility of any actions for improvement. Fourth, actions which are decided on by members of the organization concerned will be accepted more easily and thus be implemented more quickly than actions recommended by an outsider. Fifth, interpretation of the results shows the people involved that the data is being used for their benefit. This should motivate them to participate in future analysis. Finally, a meeting like this can contribute to creating a common understanding among project leaders and general managers regarding problems within the department. Collective interpretation of the results can help to prevent different perceptions of the problems, as were reported by

Thambain and Wilemon (see section 2.2).

During the meeting it was found that the results of the study confirmed and quantified a number of existing impressions of project leaders and the manager. For some of those present the results provided new information. For instance, it was not clear to everyone that the amount of other work had such a significant effect on duration.

The following are examples of the possible actions for improvement discussed at the meeting.

- It was found that the amount of "other work" in the projects studied was underestimated. During the meeting it became clear that the other work consisted mainly of maintenance. Those present decided that in future projects more time and capacity should be set aside for "other work".
- During the meeting it became clear that the maintenance activities, in particular, constantly interrupted development. A number of possible ways of separating development and maintenance were discussed. The possibility of setting up a separate maintenance group was discussed and rejected. It was decided to schedule the maintenance work as far as possible in maintenance weeks and to include two maintenance weeks in each quarter. It was obvious that not all maintenance can be delayed for a number of weeks. Any defect that affected the customer's operation was resolved immediately, irrespective of the maintenance weeks. Defects of this kind were only a small fraction of the defects and correcting them involved only a small fraction of the maintenance effort. The vast majority of defects were found in products before they were released to customers. By carrying out most of the maintenance during maintenance weeks, it was hoped that development could proceed more quickly and with fewer interruptions during the other weeks. This suggestion was implemented by the department within one month after the meeting.
- The department wanted to gain more insight into the origin of maintenance. Another analysis study started. Its aim was to gain an insight into the origin of maintenance in order to be able to take improvement measures that could reduce future maintenance effort.

At the end of the meeting it was concluded that the study had yielded sufficient results for those involved. A considerable contribution was the fact that ongoing discussions could now be supported by facts.

2.5 A diversified software development department

Carrying out the study

The technique as described in section 2.3 was applied to four projects of another department in 1988 and 1989 (Lierop 1991). In three of the four projects, systems software was being developed. The fourth project considered the development of a time registration system. The software developers in this department are involved in one development project at a time. The maintenance effort on software developed earlier was negligible in the case of the software developers involved in the projects observed. A total of 80 activities were monitored; the majority of the activities are from the implementation and test phase of the development projects. The projects in this department are planned in greater detail than those discussed in section 2.4. The average planned effort per activity was 40 hours and the average planned lead time was 6 days.

The classification of reasons discussed in section 2.3 was slightly modified. The main differences are the addition of some causes and the fact that some are split up into more specific causes. An example of the latter is cause underestimation. In this study two kinds of underestimations are distinguished: underestimation of the complexity of the product and underestimation of the amount of work. One additional cause requires an explanation. During the study it became clear that activities scheduled sequentially were often carried out in parallel. Since the impact of this on the lead time is clear, it was distinguished as a separate cause.

Results

The results of this study will also be presented in tables and figures. Table 2.6 gives the average planned and actual effort for the 80 activities.

Table 2.6 The planned and actual effort in person hours

	total	average per activity
planned effort	3203 hours	40 hours
actual effort	3838 hours	48 hours

The study showed an average 20 percent overrun. The corresponding reasons for the differences shown in Table 2.6 are given in Figure 2.9. The majority of the causes are product-related.

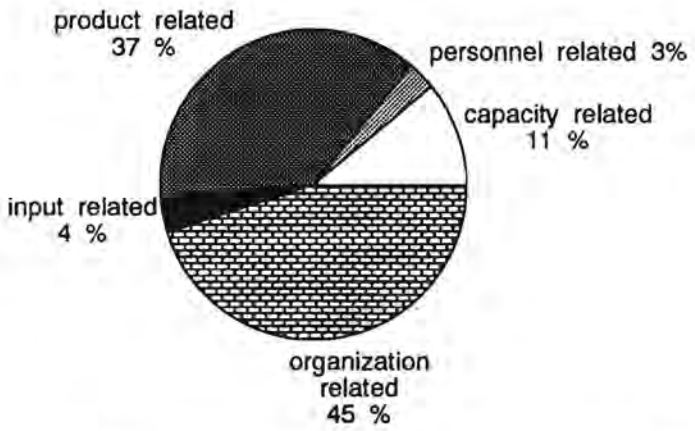

Figure 2.9 Reasons for differences between planned and real effort

Interpretation of the results
The evaluation by the participants did not result in actions for improvement. This is due to the differences in the projects observed and the fact that the majority of the causes are product-related. These causes are connected with the uncertainty in software development and are therefore hard to remedy with one or two clear actions for improvement. The evaluation did, however, yield some useful insights for the project leaders involved. The data showed that once an activity was late the number of hours worked on it per week decreased. Apparently, project members are unable to work full-time on an activity once it overruns its schedule because other work requires their attention. An insight like this is very useful when planning the remainder of a project that is late.

2.6 A CAD development project

Carrying out the study
The third empirical study concerned a project in which a components database has been developed. The system is intended to give designers an overview over the available electronic components and to support them in selecting components. The system had to substitute an existing database system. The project was carried out by four developers over a period of one and a half years and took 3500 hours to complete. This additional study is discussed because it is an example of the fact that software development projects do not have to be late. This project was finished in time and below budget.

The classification of reasons used in this study is a further modification of the original classification. There is one remarkable difference between it and the other two classifications: the reasons are not formulated as reasons for overruns but as reasons for differences between plan and reality. For example: reason 43 is not formulated as: 'complexity of application underestimated' but is stated as 'complexity of application inaccurately estimated'. This was done because after a few sessions with the project leader it became clear that the formulation in terms of overruns did not fit in with this project.

Results

A total of 115 activities have been carried out. The presentation of the results will be limited to the effort and will not include the lead time. The project members were only involved in this one project so the lead time in weeks can be calculated by dividing the effort in person-hours by the number of hours worked per week. The planned and actual number of person-hours are shown in Table 2.7.

Table 2.7 The planned and actual effort in man hours

	total	average per activity
planned effort	3952 hours	34 hours
actual effort	3528 hours	31 hours

The relative difference between planned and real effort is minus 9 percent. The reasons for the differences between plan and reality are given in Figure 2.10.

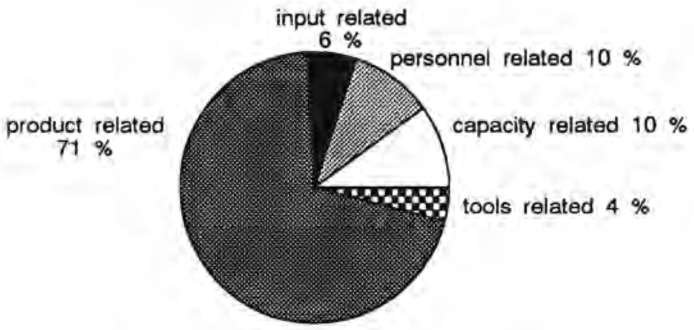

Figure 2.10 Distribution of reasons for relative differences between the planned and real effort

The reasons for the differences are presented in another way in Figure 2.11. The reasons are divided into reasons for underestimation and reasons for overestimation in Figure 2.11. The parts of the pie show the percentage of the overrun and underrun for which a group of reasons is responsible.

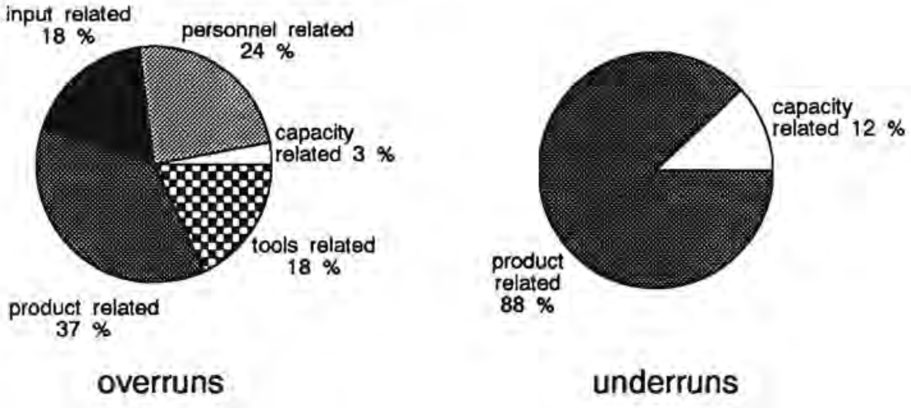

Figure 2.11 Reasons for differences between the planned and actual effort

Figure 2.12 presents the differences in the subsequent phases of the project. It shows, for example, that the overestimation of the global design phase was 360 hours. The underestimation was 32 hours. This resulted in an overestimation of 328 hours for the global design phase. Figure 2.12 also shows that the overestimations reduce towards the end of the project.

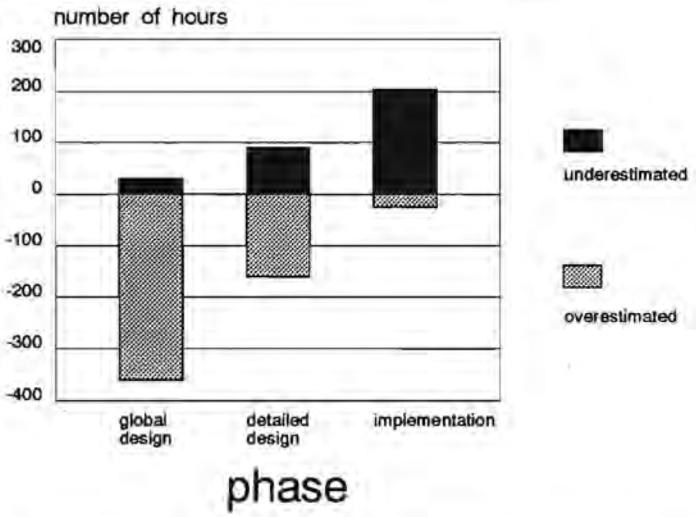

Figure 2.12 Difference between the plan and reality in each phase

Interpretation of the results

The interpretation of the results was not primarily aimed at actions for improvement. There is always room for improvement, but in this case that was not the first thing to look at. The interpretation was aimed at revealing the critical success factors for this project. Not only was it completed on time, but the product also fulfilled the specifications and was considered a success by the users. The major success factors identified were:

- The small and experienced project crew. The four persons concerned had considerable experience in the development of comparable systems.
- The thorough project preparation. The project had been preceded by a month preparation in which the product to be developed was agreed upon. The preparation included the detailed planning of the project. One critical remark is justified at this point. Parkinson's law may explain the success of this project to some extent. This law states that "Work expands to fill the available volume" (Boehm 1981). It is possible that too much time was planned, especially for the global design phase.
- Availability of the software engineers. The engineers were available full-time for this project. They were not bothered by other development or maintenance obligations.
- The clear responsibility of the project leader and the product manager. Three parties could be distinguished in the project: the clients who financed the development of the product, the project leader and the product manager. The product manager was a member of the development organization and functioned as an interface between the clients and the project leader. The product manager and the clients agreed upon a specification. The product manager and the project leader negotiated about a project plan and the required costs and lead time. The former informed the clients about the product under development and prepared them for the product. He also remained responsible for the software product after the initial installation. The product manager was responsible for the software product throughout the entire life cycle of the product, while the project leader was only involved during its development. The role of the project leader could be typified as 'making the product'. The role of the product manager could be typified as 'selling the product to the clients'. The clear responsibilities of the project leader and product manager contributed to the success of this project. The role of product management in the control of software development will be discussed in chapter 4 of this thesis.

2.7 Conclusions

The conclusions consist of three parts. Firstly, results of the studies will be compared to the three surveys that are described in section 2.2. Secondly, the results of the three studies will be compared. Thirdly, valuable insights obtained during the study with respect to software engineering control and information will be mentioned.

Three surveys on project control of projects were presented in section 2.2. As stated in section 2.3, the present study examined overruns in software development in a number of software departments in detail. As such, the study and its results are different from the surveys discussed. A comparison provides the following insights:

- The average overruns we found in two of the three cases approximate the overruns found by Jenkins, Naumann and Wetherbe (1984). The last case study showed that it is possible to develop software development according to plan, provided certain conditions are fulfilled.
- Over-optimistic planning was cited as a probable cause in all the studies that examined reasons for delay. Phan e.a. found that 44 percent of the respondents named over-optimistic planning as a reason. An unrealistic project plan and underestimation of the scope were named as major reasons in Thambain and Wilemon's survey. The studies described in this chapter also recorded underestimation of the complexity as a reason. The first study recorded it in 20 percent of the cases.
- Thambain and Wilemon's investigation of the subtle reasons for delay indicate that the reasons were not technical in nature, but were related to organizational, managerial and human aspects. The product- and tools-related reasons represent most of the technical reasons. The first study confirms Thambain and Wilemon's findings. The technical reasons comprise only one-third of the reasons mentioned in the first study. The latter two studies show a larger share for the 'technical reasons'.

It must still be noted that remarkably few comparable surveys or studies have been described in the literature. Moreover, this is true in general for empirical studies on the control of software development.

Secondly, the results of the three studies described in sections 2.4, 2.5 and 2.6 will be compared. The delays and the distribution of reasons for delay vary strongly per department. The effort overruns differ from minus 9 to plus 30 percent. The most important reason also varies from capacity-related reasons to underestimation and overestimation of the complexity of the application. The value of the data that is collected in one environment is limited in other environments due to the differences in software development and its control in distinct environments.

Thirdly, valuable insights obtained during the empirical studies with respect to control and information will be mentioned. The insights will be used in the remainder of this thesis. Three are as follows:
1) It has become clear that the control of software development cannot always be restricted to a development project. Some of the important reasons for delay originate outside the project but nevertheless affect the project. The first case study showed that the development was hindered by the maintenance activities that had to take place. Software maintenance should be taken into account in this department. The last study indicated that the function of the product manager may be valuable in supporting the control of software engineering activities throughout the entire life cycle of a product. The remainder of this thesis will show that the control of software development cannot be limited to projects.

2) The value of data on delays and the reasons for them is limited to the environment in which they are collected. 'Local for local' data collection seems to be appropriate in the current status of software engineering and its control. Since software development is not comparable at different sites, the data on development and its control are most useful at the site where they are collected. We recommend that every department should gain an insight into its reasons for delay in software development to enable adequate actions to be taken for improvement.

3) The empirical study can also be perceived as a means of collecting data on the development process. Hence, it may be concluded that the importance of closed loop information systems has again been confirmed. The closed loop principle (Bemelmans 1989) argues that information systems should be designed in such a way that those who provide input to an information system are main users of its output. Application of this principle results in feed-back to the data suppliers. The importance of closed loop information systems has been shown in two ways. Firstly, the case study discussed in section 2.5 revealed that the information provided by the existing hours registration system was inaccurate. One of the reasons was that the developers who provided the input never saw any output. The accuracy of the information improved during the study because it became clear that something useful was done with the data. Secondly, on the basis of the studies themselves we have shown on a small scale that software developers can be motivated to provide accurate data on condition that they benefit from the data collection.

3 CHANGES IN SOFTWARE ENGINEERING CONTROL

3.1 Introduction

The term software engineering has been in use since the late sixties. Software engineering is an example of a young engineering discipline. The circumstances in which software engineering has taken place have changed considerably over the first decades of its history and its evolution is the subject of this chapter. The software engineering process and its control will be characterized in terms of the Process-Control-Information model, as mentioned earlier. Software engineering used to be controlled in what will be called 'traditional' control. We will show why traditional control fitted in with the traditional software engineering process and examine to what extent it is still appropriate for the current, changed software engineering process.

The chapter consists of the five sections. Section 3.2 discusses the basic principles of traditional software development. Section 3.3 describes the traditional control of software development. Section 3.4 sketches the changes that occurred in the software engineering processes and discusses some of the problems that arise if control is not adjusted. Section 3.5 completes the chapter with a summary and conclusions.

3.2 Basic principles of traditional software development

Three basic principles will be named. They are the fact that isolated applications with stable specifications were developed, and the fact that development was based on isolated efforts by specialists. We do not argue that every single software development effort followed the basic principles outlined in this section. However, later on we will argue that traditional control assumes that software development enacts according to the given basic principles.

Isolated applications

In the early days, software products usually involved isolated applications. Typical information systems that were built were payroll systems and order acceptance systems. The systems that were developed did not have to take account of predecessors because there were none. They were specified and developed as systems that were to be used on their own (Looijen 1988). This kind of automation has an advantage from the development point of view: the systems do not have to allow for the interfaces to other systems, which might change. This kind of automation is usually referred to as island automation.

Stable specifications

The specifications of the systems developed were relatively stable. The main cause was not a more stable environment, but the fact that the applications which were selected, were the ones with unambiguous specifications, expected to be stable in the future. An information systems example illustrates this characteristic. The early information systems were transaction processing systems that were characterized by a great amount of data input, batch processing and a high volume of data output (Looijen 1988). They were usually structured decision systems that process data according to fully structured and formalized procedures (Ahituv 1982). An example is the payroll system. The applications selected had a low volatility. The chances of changing requirements were rare. This was, among other things, caused by the fact that the number of users was relatively low and the users could be considered expert users. The stability of the specifications was further enforced by the fact that applications were isolated and, as such, were unaffected by changes in other, related systems.

Isolated development efforts

Software development was new and was done by software specialists who defined their own working methods. Requirements were also mainly defined by the software specialists, who were even responsible for managing the development process. Generally speaking, the software developed did not affect the products of a company or its primary processes such as development, production, or sales. As a consequence, line management was not really involved in managing software development. The software specialists were organized as an independent unit in the organization. Separating them was in line with the efficiency goals: it was considered efficient to separate the expensive software specialists and let them get on with their work undisturbed.

Software developers could in most cases work on one project at a time. Software development departments did not have to deal with multiple projects simultaneously as they have to do now. Maintenance was not such a big problem since there was not much software around to be maintained and, as already mentioned, the kind of applications that were automated had stable specifications.

3.3 Traditional control

According to the PCI model, the characteristics of the primary process determine which control concept is suitable in a particular situation. This section will discuss some characteristics of the control of conventional software development. These are:
- The use of the waterfall model
- Project control
- The emphasis on control of time and cost.

The use of the waterfall model
The waterfall model was the model most frequently used in traditional software development. Its key features are a predefined list of deliverables in each phase and the introduction of milestones, usually at the end of each phase. Phases typically distinguished in the waterfall model are: specification, design, coding and testing. Testing was considered as one of the final activities in the life cycle. Its purpose was to ensure that the specified quality level had been achieved. At that time, the waterfall model replaced the so-called 'code and fix' model and could be considered an improvement from the viewpoint of control. The main contribution lies in the fact that progress becomes more measurable as a result of predefined phases and deliverables.

The waterfall model is not applicable to all software development efforts, however. For instance, if requirements are subject to change, developers using the waterfall model will in many cases find out after they have finished the final phase and delivered the product. Very important conditions for the appropriate use of the waterfall model are relative stability and clearness of specifications. These conditions were fulfilled in traditional software development. The waterfall model is limited to the development phases. Initially, this was not an obstacle since there was not much software to be maintained.

Project control
Software was usually controlled in projects. There are many ways of defining projects. Three important and frequently used definitions will be discussed shortly. The first is by Harrison who defines a project as "a non-routine, non-repetitive, one-off undertaking, normally with discrete time, financial and technical performance goals" (Harrison 1981). Harrison further states that projects are essentially temporary activities for those concerned, with typical durations of six months to five years. Management, organization and information systems have to be established anew for each project, and as a consequence there is a very limited learning curve for those involved.

Other definitions of projects have been given by the Project Management Institute (PMI) in their 'PMI Body of Knowledge Standards' (Wideman 1986). One of these definitions is: "Any undertaking with a defined starting point and defined objectives by which completion is identified. In practice most models depend on finite or limited resources by which the objectives are to be accomplished" (Wideman 1986). Another PMI definition of a project is: "A combination of human and nonhuman resources pulled together in a temporary organization to achieve a specified purpose with limited resources" (Beck 1986).

The definitions of a project concept have two things in common (see also Botter 1983 and Wijnen 1986): the fact that a project is a temporary and incidental organization form and the fact that the project's objective has to be achieved within certain constraints. Constraints usually concern the time and money that are available to execute the project. The discussion will focus on the incidental and temporary organization form and will show that this fitted in well with the control of traditional software development. Wijnen (1986) distinguishes three major approaches to a job: routine work, project work and improvisation. The routine work approach is chosen when the work to be done is repetitive in nature, does not change significantly over time, and the work process is clear. The main advantage of this approach is that it is embedded in daily routines. The opposite approach is improvisation. It is chosen when the work process or the objectives are unknown. This may be the case in, for instance, explorative research. The main advantage of this approach is its flexibility. The project approach is an intermediate approach. It is chosen, according to Wijnen (1986), if:

- The result is not completely new, but has some new aspects
- People from different disciplines have to cooperate
- The result can be specified beforehand and has to be achieved with limited resources.

The project work approach fitted in well with the control of traditional software development. The work was of a non-routine nature since software development involved a number of isolated, incidental efforts. On the other hand, the work was not done in an improvised way because most steps and activities in the process were known. The work was done according to the waterfall model, as discussed in section 3.2.

Emphasis on control of time and cost

Projects always have multiple objectives. The definition of the Project Management Institute (Wideman 1986) leads to a number of project control objectives being distinguished. PMI considers scope, quality, time and cost as project objectives

(Stretton 1989). Scope is defined as the work content of a project. The quality referred to is the quality of the product which is the result of the project (Wijnen 1986). Quality, cost and time may be considered as the key control aspects because these are what the world outside the project is interested in.

Project control usually follows the basic control loop that is introduced to put a number of activities into place that will be mentioned throughout this book. The basic activities are to state objectives, make a plan, execute a plan, measure progress and control. A simplified model of a control loop is given in Figure 3.1. Arrows indicate the sequence of activities.

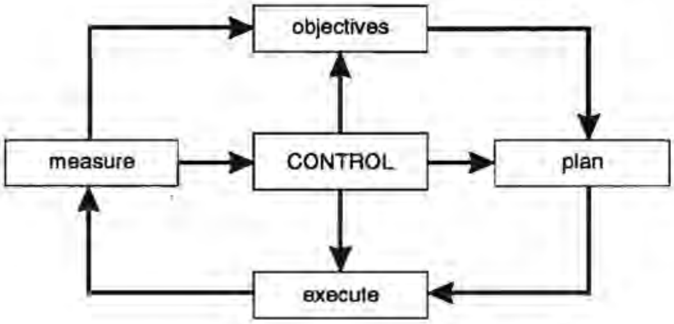

Figure 3.1 Control activities (Schaik 1985)

The control of a development cycle usually starts with a statement of objectives. The aspects of quality, cost and time have to be distinguished. Objectives will have to be specified for these three aspects. The statement of objectives is followed by planning. The execution of the project will be measured during development. Measurement should enable the actual progress to be compared with the plan. The comparison of plan and reality indicates whether some control action has to be taken. Three kinds of control actions are possible. The first one is to change the execution of the project. This can be done, for example, by exchanging people in the project team, using other tools or working overtime. The second kind of actions involves adjustments of the plan, such as rescheduling or reallocating resources. The third kind of action is aimed at modifying the objectives. Quality requirements can be altered, the delivery date can be postponed or the project may even be cancelled.

The problem with the control of a development project is not controlling the separate aspects but controlling them in an integral way. To develop a software product is one thing, to do this within a given time with limited resources is something different. Software engineering should be a matter of balancing quality requirements, on the one hand, and time as well as cost on the other (Bemelmans 1987). For example: performance requirements can considerably affect the development cost and lead time. Ideally, software engineering should ensure that the three are considered in an integral way. It should also ensure that engineers offer customers a number of alternatives. The customer should be given some design options to choose from. The engineers must be capable of balancing quality, time and cost in order to be able to provide reasonable alternatives. The client must be capable of making up his mind with regard to the alternative he prefers.

We mentioned earlier that in the early days software products were developed in isolation by software specialists. The definition of requirements was done by these specialists, as was the testing of a software product with respect to the requirements. The users did not have much influence on the aspect of quality. They did not have much knowledge about it either. Generally, only one alternative was specified, designed and implemented. The customer was often not even consulted during development. The aspects the customer could specify and control were time and cost. This led to an emphasis on the control of these two aspects and, consequently, an emphasis on the efficiency of the development process.

3.4 Changes in software engineering and its control

This section will discuss how the characteristics of the software engineering process have evolved over the years. Changes in the software engineering process that will be discussed are:
- New application areas
- Less isolated software products
- More maintenance
- An altered view of quality
- Less isolated development efforts
- More uncertain control situations.

The PCI model states that changes in the P characteristics do affect the choice of an appropriate control system. The discussion will show that traditional control, as described in section 3.3, does not always fit in with the changed circumstances in which current software engineering practices take place. We will discuss some of the problems that can arise if traditional control is practised, without taking account of the changed circumstances.

New application areas
The use of software has spread from the margin of the organization into its products and primary processes. Software has thus acquired strategic importance. Examples of this have already been given. The fact that software has become part of the primary process and products increases the importance of the control of software development with respect to quality, time and cost. The changed application areas for software result in an emphasis on a number of the quality attributes that were previously not so important. Because of the increased strategic importance, software specialists can no longer be given the responsibility for the software development process and its products. Software has become a core business for many companies and, as a result, it will have to become a responsibility of line management.

The new application areas introduced also involved more uncertainty in the application and development process. This trend can be illustrated by the following example. One of the new application areas includes decision support systems which support less structured decision processes. The fact that the decision process is less structured makes the specifications of the system less clear and more vulnerable to changes. This will lead to greater uncertainty regarding the product itself and the development process of such a product.

The waterfall model was the main process model in traditional software engineering control. A condition for using it appropriately is stability and clearness of specifications. This condition is no longer always fulfilled. The waterfall model is less appropriate if, for instance, the requirements are subject to change. Therefore alternative process models will have to be sought. Examples are prototyping and the evolutionary model.

Less isolated software products
In the early days, software products could be considered as isolated. Over the years however, they have become more integrated because of the new application areas and because of the fact that most software products have acquired a history. Today, a software product is very often not something new, but an enhanced product. It must therefore be compatible with its previous releases and must be able to function in different processing environments. The new application areas often require a software product to be able to interface properly with surrounding software products and exchange data according to specified interfaces. Nowadays, almost all software products are embedded in an environment to which they must adapt. The fact that software products are found everywhere, prevents the use of a greenfield approach to software development.

More maintenance

The increase in software maintenance can be attributed to a number of causes, the two most important of which will be mentioned here. The first is the growth in the amount of software and the increased size of software products. Software was incorporated in many products and at many places in the organization. The service life of the products increased. There is simply more software around to be maintained. A second cause of the increased need for maintenance is the fact that software products in new application areas are more vulnerable to change than the earlier applications were. As Brooks (1987) points out: all successful software products get changed. A first reason is that if a software product is found useful, people will try it at the edge or beyond its original domain. The software will either be adapted or its effectiveness will decline (Lehman 1983). One could question whether such adaptations should be called maintenance. We will address this question later on. A second reason for change is that software may survive the life of the environment for which it is written. The software will have to be adapted to be able to operate in a changed environment.

Nowadays over half of the effort comes after the initial installation of the software product (Martin 1983, Conte 1986, Lehman 1984). Control of software engineering cannot ignore maintenance that absorbs over half of its resources. Again, alternative life cycle models will have to be explored.

The use of a traditional control system can lead to an even greater increase in software maintenance. Traditional control was mainly organized in projects and it emphasized the control aspects of time and cost. A project team with clear goals in terms of time and money may be willing to sacrifice some quality goals, which are stated less specifically anyway. Quality may be considered less important since a lack of quality does not appear until the product is in use. By that time the project team has broken up and the members are working on their next development project. The additional work is referred to as maintenance and is considered as somebody else's problem.

Altered view of quality

The view of quality has altered for several reasons. A first reason is that the specification of the quality requirements can no longer be left to the software specialists since the use of the software is no longer limited to the specialists. Quality will have to be expressed in terms that can be communicated from users to engineers. A second reason is that customers are much more critical. They have become more familiar with information technology and its opportunities, as well as its constraints. The third reason is that quality has become an industry-wide subject. Efficiency was a major performance criterion in the industry in the sixties. Quality became an additional criterion in the seventies (Bolwijn 1990). It was acknowledged that quality cannot be built into a product by testing at the end of a process. Quality can only be assured by a quality-driven process. Testing as an isolated, final phase of the traditional waterfall model is far from enough to assure quality.

Less isolated development efforts

Software development has become a less isolated activity within organizations. Most of the time several projects are going on simultaneously. Those projects have to share the same scarce resources and can therefore no longer be considered as isolated development efforts. On top of that, projects can be related to each other as far as goals are concerned. Large projects are often divided into several, smaller projects. As a result goal coordination is required, in addition to resource coordination. This kind of coordination is usually referred to as program management.

For some organizations software development has become the main line of business. There are many organizations of hundreds or even thousands of people whose sole reason of existence is software engineering. They can certainly not afford to consider software engineering as a sequence of isolated development efforts. In addition to projects, some off-project activities will have to be performed and controlled. Examples of necessary, off-project activities are resource planning for projects, configuration management, the development of methods and maintenance. The maintenance problem discussed in one of the empirical studies was an example of the kinds of problems that arise if a project organization fails to realize that it will be faced with more and more off-project activities.

A related problem is the lack of software reuse. Software developers are often accused of the fact that they reinvent the wheel over and over again. One of the reasons for the lack of reuse is the organization of software development in projects. Time and cost constraints do not allow for additional work involved in making a software component available for future reuse, or in developing components that may be usable in future projects. One simply cannot afford to look beyond the project. Besides that, it is not in the interest of the participants: if they stumble over the same requirement in the near future, they will recognize it and reuse the accompanying software. They are not rewarded for the fact that others may be able to benefit from their experience as well. The result is that software products are not easily accessible for reuse and will therefore not be reused. The next project will start from scratch again.

More uncertain control situations.

Traditional control could be characterized as a situation of certainty. We will argue that the changes in the software engineering process have affected the level of uncertainty and, as a result, not all control situations can be characterized as certain any more.

The level of uncertainty of the process was taken as a starting point in a study by Heemstra (1989). He showed that the level of uncertainty should be a distinguishing factor in the choice of a control system. The study followed the lines of the PCI model. The level of uncertainty is determined by the degree of product uncertainty, uncertainty of development resources and process uncertainty. Heemstra distinguished between four control situations. They are given in Table 3.1.

Table 3.1 Four possible control situations (Heemstra 1989)

	level of uncertainty			
Variables	1	2	3	4
product uncertainty	low	low	low	high
process uncertainty	low	low	high	high
development resources uncertainty	low	high	high	high

The four different control situations have different characteristics with regard to the required control system. These characteristics will be described for the two extreme situations.

<u>1 The certain situation</u> (product, process and development resources uncertainty low)

In this situation, product requirements are known and stable, the development team knows the development process and there are enough control actions to react to unexpected events. Since it is clear what product is to be developed in which process, the required team capabilities and development tools can also be specified. Since both means and ends are specified, the control problem is mainly a problem of realization and the emphasis will be on efficiency: how to develop the specified product at the least possible costs within the shortest possible time. The project leader will be a controller among developers. The number of iterations in the planning and control cycle will be limited and development according to the waterfall model will usually do the job.

Traditional software development can be characterized as a certain situation: the product specifications were stable because the products were isolated applications and the process usually followed the lines of the waterfall model. The development resources uncertainty was low because the available software engineers usually worked full-time on a project and were not bothered by activities such as maintenance.

4 The uncertain situation (product, process and development resources uncertainty high)

The uncertain situation is the other extreme. In this situation, it is not yet clear what product has to be developed. The requirements are not only unknown, they are also evolving. The process of development is less clear so the waterfall model will not be appropriate. Other process models will have to be used in order to be able to handle changing requirements. Control actions are scarce and their effect is hard to measure. The wishes with respect to the team's development capabilities are unknown and the required development tools have not yet been chosen. The main problem in this situation is an exploration problem: how to identify alternatives with respect to the product and development process. The team leader will have to be an engineer among engineers and will have to be a technological leader. Engineers will have to be able to deal with a lot of uncertainty. The objectives of a development effort in an uncertain situation will differ from the objectives in the certain situation. The goal in an uncertain situation may be: 'to explore which design alternatives are available within a fixed period of six months with a fixed resource of five engineers. A new budget will be decided upon after six months'.

The study by Heemstra shows that the control of software engineering should depend on the level of uncertainty. The level of uncertainty has risen over the years in terms of the product developed, the process applied and the resources used. We have argued that the requirements became less stable as a consequence of entering new application areas. As a result, the product uncertainty increased. The process uncertainty also increased over the years: instead of the waterfall model, which was used as if it suited all purposes, alternative process models had to be employed to cope with the increased product uncertainty. The resource uncertainty has increased because different tasks make claims on the same, scarce resources. Examples of such tasks are: development activities for projects executed in parallel, maintenance, configuration management, the development of methods and process improvement.

The description showed that traditional control could generally be characterized as a situation of certainty. The changes in the software engineering process have affected the level of uncertainty as regards the product, process and resources. Consequently, not all current control situations can now be characterized as certain. A whole variety of control situations can be encountered in software engineering processes nowadays. It is necessary to identify the level of uncertainty, after which an appropriate control system must be selected. Project management, the waterfall model and an emphasis on the control of time and cost were identified as characteristics of traditional control. They can still be appropriate in a control situation where certainty exists. They will however be counterproductive in an uncertain control situation.

3.5 Summary and conclusions

Looking back at all the remarks on the previous pages, we can summarize the main changes in software engineering and its control as follows.

Table 3.2 Characteristics of the software engineering process

Characteristic	Traditional process	Current process
application	- isolated - secondary processes of the organization	- integrated with other applications - in products and primary processes
specifications	- stable	- vulnerable to change
development effort	- isolated	- integrated with other projects and maintenance
quality	- testing at the end of the development process	- towards quality assurance and attention for process quality
level of uncertainty	- in general: a certain control situation	- the whole range from certain to uncertain control situations

The changes in the software engineering process require changes in control. Problems will arise if these changes are neglected and traditional control is employed over the whole range of software engineering processes that have emerged. Some of the main problems have been discussed. They were:

- the fact that maintenance is not taken into consideration
- the fact that off-project activities such as configuration management, process improvement and maintenance will suffer from a lack of attention
- lack of reuse
- inappropriate application of traditional software engineering control.

Our conclusion is that the use of the waterfall model and an emphasis on time as well as cost alone are incapable of coping with the variety of control situations that have developed over the years. This does not imply that all that has been learned about the control of software engineering must be forgotten and that we have to start all over again. In some situations traditional control will be sufficient. In others the control of software engineering needs to be altered, extended or improved. Other control systems will have to be explored. This will be the subject of the following chapters.

4 QUALITY AND MAINTENANCE; towards product control

4.1 Introduction

Software quality has become a key issue over the last years. It is the subject of the sections 4.2 and 4.3. A related subject is software maintenance. Some people still think software engineering ends with the initial installation of a software product. It is, however, a well-known fact that the software will call for a great deal of effort after its initial installation. This effort is usually referred to as maintenance. Section 4.4 explores the relation between quality and maintenance. Software quality and maintenance need to be addressed to be able to discuss the main subject of this chapter: the change from traditional development control, which disregards maintenance, to product control, that does take maintenance into account. Product control will be described in sections 4.5 and 4.6. Section 4.7 ends the chapter with a summary and conclusions.

4.2 Quality attributes

Software quality has been mentioned several times in the previous chapters. This section discusses quality attributes which can make software quality more operational. These attributes will be discussed here to provide a common frame of reference. Two main studies of software quality are those by Boehm (1977) and Cavano and McCall (1978). The quality framework of Cavano and McCall will be introduced as an example because it classifies the quality attributes and gives a comprehensible impression of the meaning of the quality attributes distinguished. The framework will be presented first, followed by a definition of quality attributes identified. The end of the section will discuss the consequences of the changes in software development, as presented in chapter 3, in relation to the importance of the quality attributes which have been distinguished.

According to McCall (1978), the concept of quality is based on the three viewpoints on the basis of which a manager interacts with an end product: its operation, revision and transition. The framework is given in Figure 4.1.

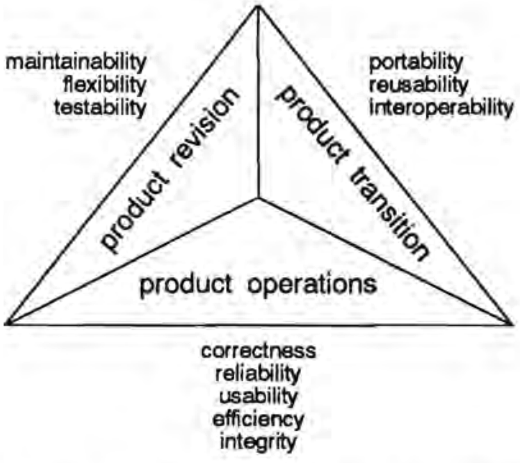

Figure 4.1 The software quality triangle (McCall, 1978, page 191)

The viewpoints of operation, revision and transition correspond to the life cycle of a product. The software quality attributes are associated with the viewpoints. A product is judged on its operational quality attributes after development. The operation of a product determines whether it meets the client's needs. Maintenance of software products starts after delivery. The revision quality attributes become important at this point, especially flexibility and maintainability. The transition of components from an existing product to a new product becomes important when the aim is to reuse software.

A typification and a definition of the attributes are given in Table 4.1.

Table 4.1 The definitions of the quality attributes

Attribute	Typification	Definition
Correctness	Does it do what I want?	extent to which a program satisfies its specifications and fulfils the user's mission objectives
Reliability	Does it do it accurately all the time?	extent to which a program can be expected to perform its intended function with required precision
Efficiency	Will it run on my hard-ware as well as it can ?	the amount of computing resources and code required by a program to perform a function
Integrity	Is it secure?	extent to which access to software or data by unauthorized persons can be controlled
Usability	Can I run it?	effort required to learn, operate, prepare input, and interpret output of a program
Maintaina-bility	Can I fix it?	effort required to locate and fix an error in an operational program
Testability	Can I test it?	effort required to test a program to ensure it performs its intended function
Flexibility	Can I change it ?	effort required to modify an operational program
Portability	Will I be able to use it on another machine?	effort required to transfer a program from one hardware environment to another
Reusability	Will I be able to reuse some of the software?	extent to which a program can be used in other applications - related to the packaging and scope of the functions that programs perform
Interoper-ability	Will I be able to interface it with another system?	effort required to couple one system with another

The factors and their definitions largely correspond to the factors of software quality distinguished by Boehm (1977). The changes in software development, as described in chapter 3, affect the importance of the various quality attributes. In general it can be argued that the importance of quality attributes has increased because of the growing importance of software applications and the increasing interest in quality. The attributes particularly affected will be discussed.

Correctness, reliability and integrity have become more important for those applications which affect the primary process of a company or are part of its products. The quality attribute which McCall names integrity is often referred to as security (Myers 1976). The importance of the efficiency of software has become less important, due to price-performance ratios of the computing resources that have become available as a result of a rapid evolution in technology. The usability of software has become more important since the use of software is no longer limited to specialists: many people use software products nowadays.

The second group of quality attributes are the product revision attributes. The importance of testability has increased because of the fact that the size of software products have grown in volume and the fact that software products have become more integrated, as already discussed. The correctness of a program has to be tested, as well as the integration of the program with other programs being developed concurrently. Maintainability has become of interest because of the fact that over half of the engineering effort is required after the initial installation of a program and there are a lot of software products to be maintained. Flexibility has become important because applications which are automated nowadays are more vulnerable to changing specifications, as was argued in section 3.4.

The third group of attributes is the transition quality attributes, namely portability, reusability and interoperability. Portability has become an important item over the years because of the rapid hardware developments and the lack of standard hardware platforms. Portability should ensure the value of software, despite the migration to other platforms. Reuse has become a major issue, promising an improvement of the productivity of software development and a reduction of the lead time involved in development efforts. Interoperability is important because of the increasing integration of software products (section 3.4).

The conclusion of this short overview is that software quality has become a multi faceted concept over the years. Correctness, especially functional correctness, may have been the most important quality attribute in conventional software development. Nowadays software development will have to take account of many quality attributes simultaneously.

4.3 Quality definitions

This section will first make a distinction between different viewpoints on product quality. Next, the concept of process quality will be introduced. Finally, a basic approach will be suggested for handling the different viewpoints on quality in practice.

Product quality
The need for different quality definitions will first be illustrated with an example: 'A product has been developed according to the specifications, but the users are not satisfied because the product does not fit their needs. Developers conclude that the users are not able to explain what they want and users conclude that the developers are not able to understand what they want. Both parties end up being unsatisfied with the result of the development effort and it is not easy to say who is right and who is wrong.'

The following discussion of quality definitions will show that both parties are right, according to their own definitions. Garvin (1984) distinguishes between five quality definitions in an important paper entitled "What does 'Product Quality' really mean?". The five definitions are a:
- Transcendent definition,
- User-based definition,
- Product-based definition,
- Manufacturing-based definition, and,
- Value-based definition.

The five quality definitions will be discussed.

The transcendent definition says that quality is synonymous with innate excellence. According to this definition, quality is absolute and universally recognizable, despite the fact that it cannot be defined precisely. Only experience can teach to recognize quality.

The second quality definition is user-based. An example of a user-based definition is Juran's definition: "Quality is fitness for use" (Juran 1988a). A user-based definition starts from the assumption that individual customers have different needs and those goods that best satisfy their needs are considered to have the highest quality. According to the user-based quality definition, quality is a subjective concept. The user-based definition of software is often the only definition that is acknowledged by the users of software products.

The product-based definition views quality as a precise and measurable variable. Differences in quality reflect a difference of some ingredient or attribute possessed by a product. The product-based definition views quality as an inherent characteristic of goods, rather than as something ascribed to them. This group of definitions consists of objective quality definitions. Most of the work on software quality to date uses the product-based quality definition. The quality framework that was presented in section 4.2 was mainly founded on the product-based quality definition.

The fourth group of definitions to be discussed is the manufacturing-based definitions. These definitions identify quality as conformity with specifications. Whereas user-based definitions focus on the user's side, manufacturing-based definitions focus on the supply side. The primary focus is internal. Software engineers often use manufacturing-based quality definitions.

The four groups of quality definitions so far discussed dealt with product quality in isolation. The fifth group of definitions are the value-based definitions. They define quality in relation to costs. A value-based definition regards a quality product as one that provides performance at an acceptable price or conformance at an acceptable cost. Section 3.3 argued that software engineering should be a matter of balancing quality requirements on the one hand, and time as well as cost on the other hand. The value-based quality definition appeals to the balancing capabilities of both engineers and users.

So much for the description of five groups of product quality definitions, as described by Garvin. The groups of definitions will be classified to enable some of the problems which arise in connection with product quality in software engineering to be illustrated. Table 4.2 classifies the definitions as users' or engineers' definitions because these are the two main parties in the engineering process. Table 4.2 also distinguishes between objective and subjective definitions.

Table 4.2 Classification of the quality definitions

DEFINITIONS OF PRODUCT QUALITY	OBJECTIVE	SUBJECTIVE
USER'S VIEWPOINT		transcendent, user-based, value-based
ENGINEER'S VIEWPOINT	manufacturing-based, product-based	transcendent, value-based

We have classified the transcendent, user-based and value-based quality definitions as subjective. They are subjective, both from the user's and the engineer's viewpoint. The manufacturing-based and product-based quality definitions are classified as objective because they can be quantified in accepted terms. The definition of quality attributes in section 4.2 can be used as a product-based quality definition. An objective product-based quality definition is the basis for an objective manufacturing-based quality definition. The manufacturing-based and product-based definitions are classified as engineers' quality definitions because they are stated in the terms used by engineers.

Engineers and users will have to agree on the quality requirements to be met by a product. In traditional development, the system specifications were drawn up by specialists for whom the manufacturing-based definition of quality was enough: quality meant conformance to the specifications. We described how the software applications gained importance. This has led to an increased interest in the specification of the software product on the part of users and line management. The emphasis on user-based quality definitions has increased, resulting in the need for a common language.

Table 4.2 shows that no common, objective definition exists. The value-based and transcendent definition is used by both users and engineers. The fact that these are subjective definitions detracts from the value of those definitions as a language for expressing user requirements. The subjective character of the definitions and the different background make it very unlikely that users and engineers will be able to communicate precisely on the basis of those definitions. Here we are clearly discussing a key problem in product development which we do not intend to solve at this point. We only identify the problem and discuss an attempt to solve part of it.

Some attempts have been made to narrow the gap between users and engineers. One way to solve the problem is to provide users and engineers with a common, objective language in which they can express the quality requirements. We will discuss one example of such a language. Gilb (1988) advocates quantifying all the product's attributes in terms the users can relate to. An example of the application of this idea to the specification of a banking system will be given. The specification of the quality attributes of reliability and portability are given in Table 4.3.

Table 4.3 Examples of the specification of quality attributes in users' terms (Gilb 1988, page 140-141)

	SCALE	TEST	PLAN
Reliability	errors reported in lines of source code per year	reported errors in official log versus official size data	less than 0.1 errors per 100 lines of code per year (in 19xx)
Portability	salvage use versus new build cost estimate when ported to IBM COBOL	calculation and sample of conversion	0.95 (within 3 years)

Similar definitions are provided for all the quality attributes. A scale, a test and a planned level are given for each attribute. The test describes the way in which the value of the attribute will be measured. The language proposed by Gilb may provide users and engineers with a common language to express user requirements. This represents an attempt to narrow the gap between two of the five quality definitions; i.e. the user- and the product-based definition.

Process quality

The discussion so far has been limited to product quality. It would however be incomplete, without considering process quality for it is clear that these two aspects are related. The relation is shown, for instance, in Juran's definition of a product as the output of any process (Juran 1988a). Juran (1988b) defines a process as a systematic series of actions directed at the achievement of a goal. Obviously, quality processes are essential for making quality products.

So far, quality has been approached from the product side in this section. Quality and quality improvement can also be approached from the process side. This is done, for example, by employing standards such as ISO 9000 (1987) which tries to ensure that development processes comply with certain standards and tries to ascertain that the conditions for a quality process are provided. ISO 9001 for software states that "It is intended to provide guidance where a contract between two parties requires the demonstration of a supplier's capability to develop, supply and maintain software products. The guidelines ... are intended to describe the suggested controls and methods for producing software which meets purchaser's requirements. This is done primarily by preventing nonconformity at all stages from development through maintenance" (ISO 9000, 1987). Humphrey (1989) focuses on the quality of the development process as well; his software process maturity framework will be discussed in chapter 6.

Knowledge of software and its development is at present inadequate to translate product characteristics into characteristics of an appropriate engineering process. We do not know enough about the product-related and process-related variables. It is, for example, not clear what the relations are between:

- The experience of the developer and the quality of the product
- The complexity of the product and development lead time
- The introduction of new development methods and the quality of the product.

Some attempts have been made to improve the knowledge and understanding of software and its development. Two of these will be mentioned. Firstly empirical studies, examples of which are:

- An investigation that studied the relation between module size and error proneness (Basili 1983)
- A study that evaluated the effectiveness of software engineering technologies such as tool use, chief programmer teams and code reading (Card 1987).

An overview of empirical studies is provided in Basili (1986).

Another attempt involves software cost models. They do not research one relation at a time, but try to provide a descriptive model of software development. The best known example is COCOMO (Boehm 1981). The value of software cost models lies mainly in the fact that they show relations between product-related and process-related variables and that they force development departments to collect information on the relations in their own development environment (Genuchten 1991).

An approach to handle quality

So far, we discussed five groups of product quality definitions and introduced the notion of process quality. How should we deal with these definitions in practice? It should be clear that multiple definitions are required to understand and improve quality. Different definitions should however be emphasized in different phases of the development life cycle. Garvin (1984) states that one needs to shift one's approach to quality as products move from design to market. Firstly, quality must be defined according to user-based quality definitions. Characteristics that relate to users' wants and needs must be identified. The characteristics must be translated into product attributes, which are derived from a product-based quality definition. For example, the definition of software quality attributes that was presented in section 4.2. The manufacturing or engineering process must be organized in such a way that products are made according to their specifications. The foregoing is presented in Table 4.4.

Table 4.4 A shifting emphasis on quality definitions

Quality emphasis	quality definition used
quality characteristics as perceived by user	user-based
↓	
balancing quality and cost	value-based
↓	
identifiable product characteristics	product-based
↓	
organization of engineering process that ensures products are engineered to specifications	manufacturing-based & process-based

A distinction should be made between one-of-a-kind production for a known client and mass production for anonymous users. In the latter case, large investments in a repetitive manufacturing process are required. This necessitates an explicit shift to a manufacturing based quality definition somewhere on the line from product to market. The reproduction of software does not involve an expensive repetitive manufacturing process. This is an explanation of the fact that the shift to a manufacturing based quality definition is, rightly or wrongly, often made less explicit in software engineering.

In case of one-of-a-kind production a product is made for one known client. The different quality viewpoints will have to be balanced throughout the engineering phases. One cannot afford to come up with a product that does meet the requirements, but does not satisfy the needs of the only client. We already indicated that a software engineer should be able to balance quality on the one hand and cost as well as time on the other. We now state that in case of one client, the software engineer must be able to balance the different quality viewpoints throughout development as well.

Shifting the emphasis on different quality definitions represents one attempt to understand and improve product quality. Improved understanding of software and its development will teach us how to translate the specifications based on one quality definition to specifications based on another definition. At the same time we can, however, attack the problem from the opposite angle: from process improvement to product improvement. It is not a matter of choosing one of the possibilities, but of doing both and striving for the maximum result.

4.4 Causes of increasing maintenance

Chapter 4 has so far dealt with software quality in general. This section will discuss software maintenance, a related subject. Maintenance is often perceived as a consequence of lack of quality. It should be obvious that this is an unacceptable simplification. The discussion of software quality in section 4.3 showed that reality is more complicated. This section will distinguish between three kinds of software maintenance and examine the causes of the increase of software maintenance.

Different kinds of maintenance

Three kinds of maintenance will be distinguished using the classification first made by Swanson (1976). The first kind of maintenance is corrective maintenance, i.e. maintenance performed in response to processing and performance failures. An obvious type of failure is the processing failure. The IEEE glossary distinguishes between errors, faults and failures (IEEE 1983). An error is defined as a defect in the human thought process. Faults are the concrete manifestations of errors within the software. One error may cause several faults. Failures are the departures of the software system from software requirements. A performance failure is the failure to meet a specified performance criterion, for example the specified response time.

Perfective maintenance is the second kind of maintenance. This is maintenance performed to eliminate processing inefficiencies, enhance performance or improve maintainability. The changes that are made in perfective maintenance take place within the limits of established specifications.

The third kind of maintenance is adaptive maintenance. This is maintenance in response to changes in data and processing environments. Adaptive maintenance involves an adaptation of the original specification. It is difficult to distinguish between adaptive maintenance and prolonged development. An objective specification of the quality of a software product is required in order to be able to tell the difference. However, it is important to be able to judge the difference because the software supplier may be held responsible for the adaptive maintenance. The supplier and the client may have agreed that the supplier will pay the bill if it is a case of adaptive maintenance and the client will pay if prolonged development is involved. In everyday practice, prolonged development is often confused with adaptive maintenance.

Causes of the increase in maintenance

Conte, Dunsmore and Shen (1986) claim that maintenance takes over 60 percent of the total software engineering effort. Lehman (1984) states that 70 percent of the expenditure on a program is incurred after initial installation. Martin (1983) estimates that development costs are only a quarter to one third of the total costs. Six reasons for increase in maintenance and their effect on the amount of corrective, adaptive and perfective maintenance will be described.

Increase of the volume of software

The number of operational software products, as well as their size, has increased over the years. The amount of corrective, adaptive and perfective maintenance has also increased. Some departments are completely absorbed by their maintenance work and might even change their name from development to maintenance department.

Technological developments

Technological developments that have taken place over the years have made a lot of adaptive maintenance essential to keep the applications up to date with the processing environment.

New application areas

Applications which are automated evolved from applications with stable specifications to applications that are more vulnerable to changes. This has led to an increase in adaptive and perfective maintenance: the data and processing environments are more vulnerable to change (adaptive) and it pays to increase the maintainability of the applications because of their importance (perfective).

<u>More integrated software products</u>
Software products have become more integrated over the years. This increased complexity and made the applications more prone to faults, resulting in more corrective maintenance.

<u>Increased emphasis on user-based quality definitions</u>
The previous section has discussed different quality definitions and argued that the emphasis on user-based quality definitions has increased. This makes it more important to ensure a proper translation from user-based, via product-based to manufacturing-based quality definitions. We have argued that the current knowledge of software and its development is too limited to permit a smooth translation from one group of quality definitions to another. This means there is a greater possibility of translation problems and misunderstandings about quality definitions. The difficulties that arise are responsible for an increase in corrective maintenance. The fact that users are gaining more influence allows them to enforce changes in the software product if their needs change. This increases the adaptive maintenance.

<u>Inadequate maintenance</u>
The emphasis, both in software engineering practice and in theory, has hitherto been on software development. Maintenance used to be the kind of activity that had to be done in addition to, or in between, development activities. There is a need for proper maintenance organization, as well as tools and methods. The lack of interest in maintenance has led to poor maintenance practices which, in turn, inevitably lead to more maintenance.

4.5 A response to the maintenance problem

A potential response to the maintenance problem and its consequences for software development and maintenance will be discussed. The following subjects will be addressed:

- The incorporation of maintenance into software engineering control
- An extension of the life cycle
- Alternative life cycle models

The incorporation of maintenance into software engineering control

Maintenance will have to be incorporated into software engineering control for two reasons. The effort involved in maintenance is a first reason to integrate the control of post-delivery activities with the control of software development. One cannot restrict one's attention to approximately half of the expenditure and consider that the other half is somebody else's problem. A development organization should accept the responsibility for maintaining the products it has developed in the past.

A second reason to incorporate development and maintenance control is provided by Lehman (1984). He states that the term 'maintenance' is inappropriate in the context of software. His line of reasoning will be summarized. He starts with the distinction of S-type and E-type software. S-type software is defined as software for which the only criterion of success in its creation is equivalence to a specification. E-type software is one embedded in its operational environment and implementing an application in that environment. The success of an E-type system can only be determined by its use; i.e. after installation of the software product in its environment.

The criterion for the success of an E-type program is determined by user satisfaction (a user-based quality definition). Continued satisfaction demands continued change since the system will have to be adapted to the changing environment, changing needs, developing concepts and advancing technologies. A system will either evolve, or its effectiveness - and that of the application it supports - will decline. Software is by nature evolutionary and it is pointless to distinguish between initial development and maintenance. Software does not deteriorate by itself and does not need to be maintained in the traditional engineering sense (Fox 1982). Lehman therefore concludes that the terms 'development' and 'maintenance' should be replaced by evolution. We agree that there should be no distinction between development and maintenance and that the control of software development should therefore be replaced by control of the evolution of a software product. This is abbreviated as product control; control of the software engineering activities over the entire life cycle of the product.

Extension of the life cycle

A clear consequence of the changeover to product control is the extension of the life cycle. The entire life cycle of the product is now taken into account instead of its development alone. This is shown in Figure 4.2. The terms development, maintenance and prolonged development are used to distinguish between evolution before and after the initial installation.

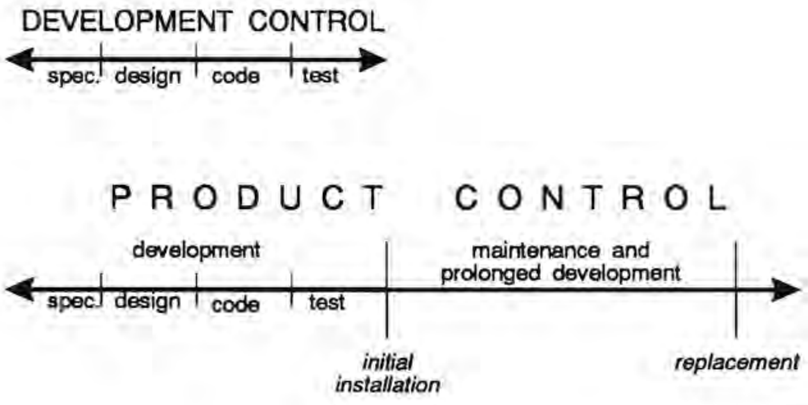

Figure 4.2 Development cycle versus product life cycle

Responsibility for the maintenance of products and processes requires additional insight into the quality of the developed product and the accompanying development process. Insight is needed to be able to estimate the maintenance effort. The department requires insight both into the quality of the products it is developing and the products it has developed in the past.

Different life cycles for software evolution
The extension of the life cycle has implications for the life cycle models used. Chapter 3 has described how the waterfall model was usually employed in traditional software development. This does not include maintenance. Alternative life cycle models will have to be explored. Three software development life cycle models will be described and their applicability to software maintenance will be examined. The development life cycle models are the linear model, prototyping and evolutionary delivery.

Linear development
Linear development is often associated with the waterfall model discussed in section 3.3. The scope of the waterfall model is limited to development phases only. The maintenance process model usually employed in combination with the waterfall model is what Basili (1990) calls the quick-fix model. The existing system, usually just the code, is taken as the starting point in the quick-fix model. The necessary changes are made in the code and should also be made in the other documents that are affected. The quick-fix model is shown in Figure 4.3.

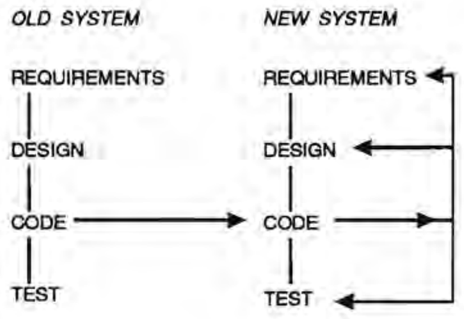

Figure 4.3 Quick-fix process model (Basili 1990)

Upstream documents must be updated to prevent future problems. It is a well-known fact that this is not always done in practice. The short-term advantage of the quick-fix model is the limited time required to address a customer's problem. This is important in the event of a failure that hinders the customer's operation. Disadvantages are the fact that the design and structure of the product are harmed. The maintenance and enhancement are usually performed on implementations rather than on problem-oriented specifications. This leads to inefficient and often ineffective maintenance.

Prototyping

The second life cycle model which can be used in the product control situation is prototyping. Prototyping is often employed in information system development, particularly to determine user's wishes. In the long term, this should reduce the amount of maintenance. Prototyping may be appropriate in the context of maintenance if considerable changes in the system are involved. Two kinds of prototyping are distinguished: throwaway prototyping and evolutionary prototyping. In throwaway prototyping a part of the system is developed to be able to cover some requirements. Once they are determined, the prototype is thrown away and development starts. Evolutionary prototyping (Davis 1988) begins with the development of a first version of the system. Users are confronted with this system, after which additional requirements are determined and subsequently implemented. There is a risk that evolutionary prototyping will deteriorate into the code and fix strategy.

Evolutionary delivery

The third life cycle model to be discussed is evolutionary delivery. Evolutionary delivery (Gilb 1988) abandons the idea that a software product should be delivered in one piece after the complete project has been finished. Other names for this and related strategies are incremental development, iterative enhancement (Basili 1975), evolutionary development or the spiral model (Boehm 1987). We choose the name evolutionary delivery because delivery is what distinguishes this strategy from others such as prototyping. The principle of this strategy is:

- deliver something to the real end user,
- measure the added value,
- adjust both design and objectives based on observed realities (Gilb 1988, page 84).

The accuracy of and changes in specifications are monitored through the intermediate deliveries. Evolutionary delivery may be fruitful if a lot of uncertainty is involved. Uncertainty is reduced because large jumps are replaced by smaller steps. In section 3.5, a distinction was made between uncertainty related to the product, the process and the development resources. Product uncertainty is reduced, because after each delivery the engineers can check whether they have understood the requirements properly and whether the users requirements have changed. Measurements can ensure whether the chosen process model is effective and how many resources have been used for developing the first increment. The main problem in using this method is to come up with an overall design which the evolutionary deliveries fit into. Basili (1975) provides an early example of the fact that evolutionary delivery can be effectively used in practice. Other examples are provided by Gilb (1988).

Evolutionary delivery fits in very well if the time horizon has been prolonged from the development phases alone, to the entire life cycle of the software product. When one considers the entire life cycle of a product it seems reasonable to aim at evolution and the delivery of intermediate increments instead of revolution and the instantaneous delivery of a complete and perfect product. Development can be perceived as the first increment while the maintenance updates can be seen as successive evolutionary deliveries. Evolutionary delivery shows a lot of similarities with what Basili (1975, 1990) calls iterative enhancement. He argues that the iterative enhancement model for software development can be readily applied to software maintenance. Iterative enhancement starts with the existing system and evaluates it for redesign and modification. This model assumes a complete and consistent set of documents describing the system. Iterative enhancement modifies the set of documents, starting with the highest level document affected by the changes. The iterative enhancement process model is given in Figure 4.4.

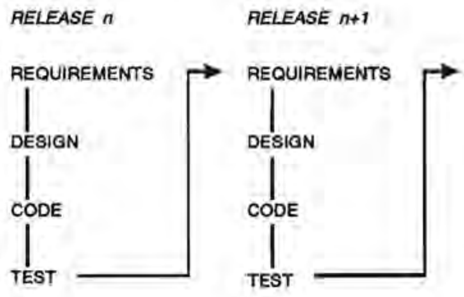

Figure 4.4 The iterative enhancement model

An advantage of the iterative enhancement model is that the design evolves with the product. Another advantage is that the process model is compatible with the development process model. It might be argued that the evolutionary delivery model never results in a complete product and that the distinction between development and maintenance has therefore disappeared. A disadvantage of the use of an iterative enhancement model is that it takes time to go through the phases at a time when a customer is possibly in desperate need of a working software product.

<u>A contingency approach</u>

The process models have been presented as if they were mutually exclusive. In practice, however, they are often complementary. Some examples: the activities specification, design, code and test can be identified in all three process models. Using evolutionary delivery, the life cycle will be gone through a number of times. Each time the life cycle is gone through, a delivery to the real end user will end the development life cycle. The basic activities can also be identified in prototyping. The life cycle is, however, nested within the development activities of specification and design. The examples show that the different process models have several aspects in common and that, depending on the situation, a development organization and its clients should choose the process model that fits in best with their needs. It should be clear that the waterfall model does not suit all purposes and that other process models are available.

4.6 Consequences for control

The change from development control to product control has several consequences for control. The changed goal and two organizational issues will be discussed.

Goal

The goal of an organization whose sole responsibility is development control is to deliver a product according to its specifications within the given time at the planned costs. This has been referred to as traditional development control in this thesis. The goal of an organization which is responsible for the control of a software product over its entire life cycle could be formulated as: maximize quality at minimal cost over the entire life cycle of the product within the given time constraints. This type of control is called product control. The goals of product control differ in two ways from traditional development control. Firstly, the horizon of control has been extended from development alone to the entire life cycle of the product. Secondly, the quality definition of the product has changed. The organization that employs development control can afford to stick to a manufacturing-based quality definition such as: 'quality is conformance to specification'. An organization that employs product control needs to establish a relationship with its clients over the life cycle of a product. A user-based quality definition such as 'quality is fitness for use' will be required to establish and maintain such a relationship.

The new goals have a number of consequences. A characteristic of any organization which is accountable for both the development and maintenance of its product is that the quality of the product will be a very important control aspect. A department which is not responsible for maintenance may benefit when it delivers a product early under given time constraints. The short term benefits of delivering a product early will be dwarfed by the loss on maintenance for an organization which is responsible for maintenance. Section 4.2 discussed how McCall (1978) divided quality attributes into classes relating to the operation, revision and transition of a software product. Operational quality attributes such as correctness, usability and integrity are the most important aspects when one is only responsible for developing a software product. Product control includes the maintenance of the software. Consequently, revision quality attributes such as maintainability, flexibility and testability become important features.

Organization

Two organizational issues which arise as a result of the change from development control to product control will be discussed. The first issue is the introduction of a separate function which is responsible for the evolution of a software product over the entire life cycle of a product. The second issue is the fact that a department should be organized in such a way that it controls both development and maintenance. Three possible organizations will be described.

Software product management
An organization which employs traditional development control usually acknowledges two parties in the development of a software product: the client and the project manager. The client expresses his requirements and the project manager develops the software product that meets them. In the case of product control, the responsibility involved goes beyond that of traditional project management. A new function which controls a product over its life cycle has to be acknowledged. We will call this function product management, as opposed to project management. We refer to management instead of a manager to make clear that the functions does not necessarily need to be performed by two different persons. The name product management has been chosen because it is a well known term for such a function. Again, the distinction between one-of-a-kind production for a known user and mass production for anonymous users should be made. Product management is normal for products that are produced or developed for anonymous users. Think, for example, of canned food, cars or televisions. Product management is known in software in situations where products are developed for anonymous users. An example is standard personal computer packages. We will argue that product management can also play an important part in the control of a software product developed for one or a few known clients. The role of product management became clear to us during the empirical study that was described in section 2.6. One of the critical success factors in the project was the cooperation of the clients, the product manager and the project manager. This section will describe the cooperation in detail and compare the responsibilities of the parties involved.

In product control, project management remains responsible for the development of a specified product within the given constraints. Its role is temporary: after the initial installation, project management will go on to its next development project. As discussed in this section, over half of the effort is incurred after the initial installation. The incurrence of this effort will have to be controlled. This comes under the responsibility of product management, which is part of the software supplying organization. Product management is involved at the start of the product's life cycle and agrees with the client on the requirements of the software product to be developed. The user states his requirements in a user-based quality definition. Product management uses a user-based quality definition and a value-based quality definition to assess the quality because it intends to establish a relationship with the client throughout the life cycle of a product. Product management translates the user-based quality requirements into a product-based quality definition. It must have the ability to evaluate the software product in both user terms and in product terms. The product-based quality definition is the starting point for the negotiations between product management and project management on the constraints under which the product will have to be engineered. Project management can use a manufacturing-based quality definition to assess the quality.

Product management is the interface between the client's organization and the development team during the development of the software product. It is the only function allowed to adapt the specifications after consultation with the client and the development team. Product management is responsible for preparing the client to use the software product which will result from the development project. After initial installation it is responsible for user support. Product management has to arrange a facility for user questions, change requests and problem reports. The task of project management can be characterized as 'making the product'. The task of product management can be characterized as 'selling the product to the client and supporting the client in the use of the product'.

The tasks of the client, product management and project management in the various phases of the product life cycle are summarized in Table 4.5.

Table 4.5 The tasks in the phases of the product life cycle

PHASE IN LIFE CYCLE	CLIENT	PRODUCT MANAGEMENT	PROJECT MANAGEMENT
specification	* determine user requirements	* translate into specifications	* make plan
development		* track specifications * interface between client & development team * prepare users	* execute project within given constraints
introduction	* attend training * prepare organization	* arrange training * help desk	
use	* report problems * determine additional requirements	* arrange maintenance * translate requirements into specifications	

The differences between product management and project management are listed in Table 4.6.

Table 4.6 Differences between project management and product management

PRODUCT MANAGEMENT	PROJECT MANAGEMENT
continuous function during product life cycle	temporary function during development
user-based quality definition & value-based quality definition	manufacturing-based quality definition
interface to users	isolated from users
sell product to client	make product

To be clear: we do not argue that the problems involved in controlling software engineering can be solved by creating a new function or appointing yet another official. This is not enough, just as it is not enough to appoint a quality manager to solve quality problems or an automation manager to solve automation problems. The way the project manager, product manager and clients play their parts will be the decisive factor for the success or failure of the project.

We have said that product management is often required in an organization that is in product control. It is required if either an unambiguous interface between the client's and the user's organization is required, or if the control of the software product goes beyond the control of the development phase alone. This is shown in Table 4.7.

Table 4.7 The necessity of product management

	One user	More users
Development control	NO	YES
Product control	YES	YES

The first condition was fulfilled in the empirical study, as described in section 2.6. The software product was developed for a number of clients and in a number of development projects. The second condition will be increasingly satisfied in the years to come. As has been argued in this chapter, software suppliers will be held responsible for the maintenance of the software products they have developed in the past. Clients will establish long term relationships with software suppliers to protect their investments in software products. Product management will be the counterpart in these long-term relationships. One of the two conditions will be satisfied most of the time. We expect that software product management will become common practice in software engineering efforts in the near future.

Whether or not software product management needs to be performed by a product manager is something that has still to be determined. It is possible that an existing executive will take the role of product manager.

Organization of development and maintenance

An organization employing product control should be structured in such a way that it controls both development and maintenance. This can be done in several ways, three of which will be considered.

The first way is to separate the development and maintenance people into different units. This could be called a specialization approach. This allows development people to do their work undisturbed. A disadvantage is that developers may want to shift the responsibility for the quality of the developed product to the maintenance group. A developer is not involved and is not made responsible for the products he has developed. Another disadvantage is that the organization cannot cope with the varying offer of maintenance work. Another potential disadvantage is that maintenance is perceived as a second rate activity. People involved in maintenance should not be perceived as being engineers who are not qualified to do development work (Swanson 1989). Maintenance is much too important for such a qualification. Rotating staff between development and maintenance can prevent these problems to some extent.

A second way to organize development and maintenance is to involve all the engineers in maintenance and development. The people who do the development work also maintain the products in an organization like this. The development work will have to wait as soon as a maintenance job becomes more important. In theory, there is a maximum of involvement and accountability: the developer who caused the need for maintenance will have to carry it out. This has the advantage that developers are confronted with the consequences of lack of quality. They should feel responsible for the product either they or their department have developed. In practice, however, it is often unclear where the fault was caused and who, if any one, is responsible. The need for corrective maintenance is caused by defects which can sometimes be assigned to a developer. Adaptive and perfective maintenance cannot usually be assigned to a developer. Capacity planning requires special attention in this situation. Care should be taken to prevent the total amount of work from exceeding the available capacity. This can happen if all the maintenance work is released to the department without taking the available capacity into account.

Some kind of regulation is required to prevent an engineering department from becoming overloaded with work. Experience with production control has shown that a production unit which is overloaded with work becomes less productive. The variation in lead time increases when the workload is too high. The empirical study described in section 2.4 shows what effects uncontrolled maintenance interference can have. Production has identified the function called work load control to regulate the work released to a production unit (Bertrand 1990). A similar function will be required to control the amount of work in a department which is involved in both development and maintenance. Work load control will have to judge which engineering job needs to be released to the shop floor. Product management, as discussed earlier in this section, may have a say in the prioritization of engineering jobs. The control complexity will increase if both development and maintenance are performed within one organization.

The third way is to disperse maintenance and development work over a period of time. Section 2.4 describes how maintenance weeks were introduced to regulate the maintenance activities. This prevents continuous interruptions, while on the other hand involvement and accountability remain. Fixed maintenance days or even hours have a similar effect to some extent. Dispersion in time has the disadvantage that control becomes more complicated. In this case, maintenance jobs are introduced on the shop floor, in addition to the development jobs that are already there. If development is not under control, development and maintenance in one organization never will be under control. The advantages and problems of the three options distinguished are summarized in Table 4.8.

Table 4.8 Advantages and problems of organizational options

Development and maintenance	ADVANTAGE	DISADVANTAGE
Separated	- clear responsibilities - efficiency of specialized maintainers - less interruptions in development	- lack of involvement - no response to varying workload - lack of product know-how
United	- involvement - accountability	- fight for scarce resources - responsibilities - increased control complexity
Dispersed in time	- involvement - accountability	- increased control complexity

The table shows that, as usual, there is no ideal solution for all circumstances. It might, however, be said that if an organization can afford it, it should integrate maintenance and development because the distinction between these two aspects is, in fact, artificial and the involvement of developers is crucial. An organization that does not control its development well will deliver products that require a lot of maintenance. At the same time, their development will usually be late. These organizations cannot afford to control development and maintenance in one organization because the fight for the scarce resources will make things worse and the whole organization may end up doing maintenance. On the other hand, organizations of this kind have the greatest need for the involvement of the engineers with the products they have developed in order to be able to improve the quality of the products in the long run.

4.7 Summary and conclusions

This chapter has discussed the control of software engineering activities over the entire product life cycle. This has been called product control. A detailed discussion of software quality and maintenance was required. We distinguished between operation, revision and transition quality attributes. Next is that several quality definitions are required to understand the quality conflicts that arise. User-based, product-based, manufacturing-based and value-based quality definitions were distinguished. To achieve quality improvement, user needs must be properly translated into identifiable product attributes. Product attributes need to be translated into process attributes that can ensure the development of the required product. The lack of understanding of software and its development is an obstacle to a proper translation at present. Most progress can be expected if the problem is tackled from both sides at the same time: via better product definitions and via process improvement. Quality and maintenance were discussed to be able to justify the change from development control to product control. Maintenance takes up most of software engineering capacity nowadays and, as a consequence, control of software engineering cannot afford to confine itself to development only. Another reason for integrating the control of development and maintenance is the fact that the division between the two is artificial. A consequence is the need for alternative process models such as prototyping and evolutionary delivery. Section 4.6 discusses the consequences for control. The subjects that have been discussed are: the difference between the goal of development control and product control and the organization of a department that employs product control. Two organizational aspects have been discussed in detail, first of which was the role of product management. Product management is required because the control of a software product needs to be extended beyond the control of software development, the province of project management. Secondly, the pros and cons of the organization of development and maintenance activities in one or in separate departments have been discussed.

5 REUSE OF SOFTWARE; towards multiproduct control

5.1 Introduction

The previous chapter discussed product control which differs from development control in that it acknowledges responsibility for the maintenance of products developed in the past. Control of the product is extended from development only to its entire life cycle. This chapter discusses yet another change; that from product control to multiproduct control. Multiproduct control expands its control focus from one product to a family of products. Characteristics of multiproduct control will be discussed in section 5.6 and 5.7. One important characteristic is the emphasis on software reuse. The sections 5.2 to 5.5 discuss various aspects of reuse.

5.2 The necessity of reuse

This section argues that reuse of software is required to meet the increasing software demand. Section 1.3 mentioned the software backlog and the fact that the demand for software is increasing faster than the ability to supply it. The gap between demand and supply will grow in the future if productivity does not increase.

An important characteristic of software is that it is intangible. This causes some problems in development. The fact that software is invisible and not visualizable impedes the design process within one mind, it also severely hampers communication among minds (Brooks 1987). On the other hand, the intangibility of software makes it possible to copy software with negligible reproduction costs. Reusability of software is widely believed to be a key to improving software development productivity and quality (Biggerstaff 1987). Opportunities for the reuse of software are there: Capers Jones (1984) claimed that 15 percent of the code written is unique, while the remaining 85 percent appears to be common, generic and concerned with putting applications into computers. Advantages of reuse are obvious and the reuse of software has been aimed at for many years. Despite this, reuse of software has not really taken off as yet (Biggerstaff 1987). Cusumano (1989) reported reuse rates in North American and Japanese companies. The rates were based on a survey involving 51 companies. The data should be interpreted carefully because of the differences in determining the amount of reuse in different firms. The reuse rate in American companies was 15 percent, while it was 35 percent in Japanese firms.

There are some empirical indications that reusability does increase productivity. Three examples will be discussed. Lanergan and Grasso (1984) classified over 5000

Cobol programs, 50 of which were studied in detail. The study team concluded that 40 to 60 percent of the code in the programs examined was redundant and could be standardized. The organization decided to make logic structures a standard for new program development. At the time of publication, "5500 logic structures were reused, averaging 60 percent reusable code" (Lanergan 1984). They believe that this translates into a 50 percent increase in productivity in the development of new programs.

Prieto Diaz (1990) reports on the implementation of a library of reusable software components. During the first year, a reuse factor of 14 percent was achieved, realizing an estimated 1.5 million dollar overall saving. The reuse rate was calculated by dividing the lines of code reused by the total lines of code produced. The reuse factor is assumed to reach a level of 20 percent after the second year and 50 percent after five years.

Further empirical evidence is provided by Selby (1989), who examined almost 3000 modules from a NASA software production environment. He classified the modules into four categories based on the degree of reuse from previous systems. The categories were:

- Complete reuse without revision,
- Reuse with slight revision ($<$ 25 percent changes),
- Reuse with major revision ($\geq$ 25 percent changes),
- Complete new development.

The average development effort (in hours) divided by the final implementation size (in source lines of code) can be perceived as an indicator of productivity and is given in Table 5.1. The effort required for the development of the original development is not included.

Table 5.1 Amount of reuse versus development effort (in tenths of hours) per source line of code (Selby 1989)

	effort (in tenths of hours) per source line of code	number of modules examined
new	1.1	1629
major revision	0.76	205
slight revision	0.60	300
reuse	0.05	820
all	0.73	2954

The difference in productivity for the classes of reuse are clear and statistically significant. The shortcomings of effort per lines of code as a productivity indicator are known. Despite this, the results are perceived as an indication of the fact that reuse increases productivity.

5.3 Reuse in industry

The benefits of reuse have been acknowledged in industry. This section describes how changing customer demands forced industry to emphasize reuse by standardization and modularization of its products. As a consequence, the production organization was adapted. We will argue that the software industry will evolve along the same lines.

Modularization and standardization
Two extreme methods of organizing production resources are usually distinguished: the flow shop and job shop (Bemelmans 1986). The flow shop involves the production of large numbers of standard products. Its production resources are usually organized in production lines for specific products. A flow shop is not flexible to changes in demand. A job shop on the other hand specializes in the production of small lots of client-specific products. The job shop emphasizes flexibility. A job shop is flexible towards changing demands, both in the number of products and the kind of products. A job shop is called a project shop if production times are relatively long.

Market demands have evolved over the years. The market required higher quality products, more complex products and more client specific-products. This called for flexibility in production processes. At the same time the market required shorter lead times and cheap products. This called for efficiency in production processes. But increasing either efficiency or flexibility was no longer good enough; both had to be provided at the same time. Market demands and the response from industry are presented in Figure 5.1.

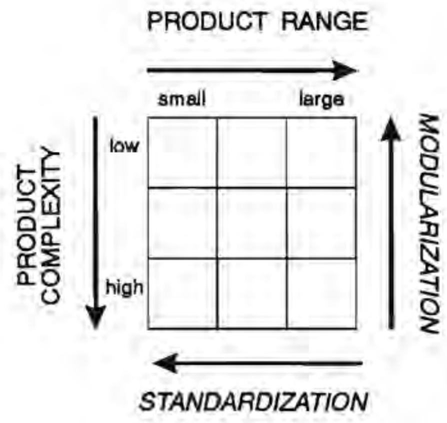

Figure 5.1 Changing process requirements (Bemelmans 1986)

The product range increased because products become more client-specific and the product life cycle becomes shorter. The increase in product variety resulted in a decrease in lot size. The complexity of products increased to meet the more demanding customer requirements which resulted in more components of more complex composition. Both drives forced the production organization to emphasize flexibility and adopt a job-shop or project-shop like structure. This hampered efficiency which was a concomitant requirement. Two responses to the demand for efficiency are standardization and modularization. Standardization is a reaction to the smaller lot size. It is, in general, not in line with the more client specific demand. Clients may, however, be willing to sacrifice some of their specific requirements for a more standardized solution at a lower price. Standardization reduces the product range and allows the design of products and processes to be reused.

Modularization of products is a response to more complex products. The traditional product structure could be represented by a pyramid. A great number of components and materials was assembled into a unique end product. Over the years most production organizations identified key modules or subassemblies which appeared in different products. Modularization can be in line with more client specific demands. The essential feature of modularization is that a great number of client-specific products are assembled from a limited number of key components which are used in many products. The designs of the components are reused. The view of a product and its development changes from a number of isolated pyramids to an hourglass, as is presented in Figure 5.2.

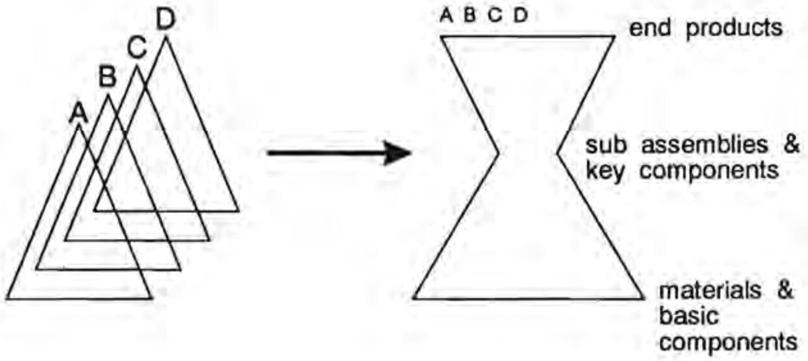

Figure 5.2 Different product views

The great number of client-specific products are reflected by the top of the hourglass and the limited number of key components by its neck. The variety of materials and parts is modelled by the bottom of the hourglass. Key components can be manufactured from a variety of materials and parts.

Production control situations

The distinction between engineering, components manufacturing and assembly leads to different production control situations. The notion of the decoupling point is introduced to differentiate four different control situations. The customer order decoupling point determines which part of a process is driven by customer orders. The process is driven by customer orders from the decoupling point downstream. Upstream the process is driven by forecasts of customer orders. The location of the decoupling point is determined by factors such as: the lead time that is acceptable to the market, the production (or development) lead time and the product structure. One reason to move the decoupling point downstream (i.e. to produce more independently of customer orders) is that the lead time for the complete production or the development is too long to be acceptable to the market.

The location of the decoupling point determines whether a production control situation is characterized as engineer-to-order, make-to-order, assemble-to-order or make-to-stock. The control situations are shown in Figure 5.3.

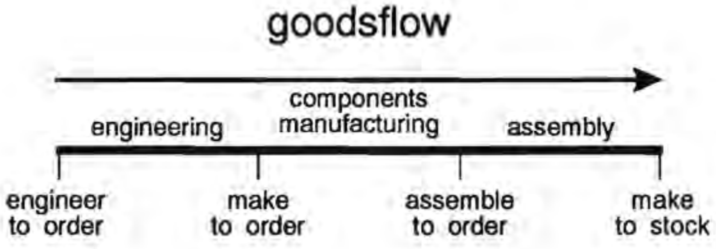

Figure 5.3 Four production control situations

Figure 5.3 distinguishes between engineering, component manufacturing and assembly as production phases. The decoupling indicates to what point the production process is customer-order driven. For example: in the case of engineer-to-order the engineering, component manufacturing and assembly are customer-order driven. In the case of assemble-to-order production, only the assembly is customer order driven. The relation between Figure 5.2 and Figure 5.3 should be clear. If we assume that the product structure is represented by a pyramid, the decoupling point of engineer-to-order is located at the bottom of the pyramid. Component engineering and manufacturing as well as assembly are done on a customer-order-driven basis. The decoupling point in assemble-to-order production is located at the neck of the hourglass. The four production control situations will be discussed.

Engineer-to-order

Engineer-to-order production is characterized by the fact that products are largely developed according to the specifications agreed upon in a particular order. An engineer-to-order situation has defined a product range in which it is willing to operate. Production is customer-driven and the customer order decoupling point is located upstream. Engineer-to-order production is flexible towards customer needs. This does not mean that engineer-to-order production starts from scratch with every new customer order. Engineer-to-order emphasizes the reuse of its experience by means of reference products and reference processes which are used as much as possible. It can do so because it has identified a product range in which it is willing to operate.

Make-to-order

The second situation to be discussed is make-to-order. In this situation, the client has a specified product that has to be manufactured. A make-to-order company does not provide standardized products from which a customer may choose. Two types of make-to-order companies are distinguished. The first type produces the same customer-defined order repeatedly. The design of the product supplied by the customer is reused in every order. The second type of make-to-order company sells capacity and is sometimes referred to as a jobber. A capacity selling company is willing to accept any customer order they can cope with.

<u>Assemble-to-order</u>
The third production control situation which is usually distinguished is assemble-to-order. The decoupling point is moved downstream: assembly is done for specific client orders and the manufacturing of components is client-order independent. This situation is often found when the variety of final products is large in comparison with the sales volume, or if products cannot be developed from scratch because the time to engineer the product to order is too long to be acceptable and, on the other hand, products are too client-specific to be sold off the shelf. It is better to put the semi-finished products into stock and assemble them on a client-specific basis in many different final products.

<u>Make-to-stock</u>
The fourth production control situation is make-to-stock. The decoupling point is moved further downstream and, in fact, the complete production process is not based on actual customer orders, but on demand forecasts. Final products are often made to stock because the market requires zero delivery lead times. A customer cannot affect the specification of the product. The emphasis in make-to-stock production is on efficiency.

5.4 A parallel in software development

Modularization and standardization
Traditional software development can be characterized as a project shop. Capacities were organized in functional groups and usually not around product lines. Development lead times were relatively long because all the software products were developed from scratch. Software development also faced new customer requirements, as already discussed. Examples of new customer requirements:
- More integrated software products,
- More client-specific software products due to the emphasis on user based quality definitions,
- Shorter lead times and higher quality products,
- More productive software development.

This led to the same trends that are observable in industry: the complexity of products increased due to new application areas for software and to the fact that software products became more integrated (see Figure 5.1). At the same time, user-based quality definitions of software products led to an increased number of client-specific products. The larger product range has consequences for control. There is one difference between the software industry and industry. As far as industry is concerned, an important consequence of an increased product range is the decrease in the lot size which makes production control more complicated. Lot size is not relevant for software since, due to its intangibility, the difference between a lot size of ten versus a lot size of 100 is negligible from the viewpoint of control.

The software industry has to come up with a response to the two trends perceived: increased product complexity and a larger product range. The trends require flexibility in the development process and do not allow other customer requirements, such as a reduction in lead time and an increase in productivity to be met. The responses can be similar to the response in industry: standardization and modularization.

Standardization is a response to the increasing product range. A trade-off between quality and cost has taught clients that it may be wise to sacrifice some of their specific requirements in exchange for a standard solution. The standard software product has the advantage of a short lead time, a higher quality level and a lower price. Standardization is accepted by clients for an increasing number of products. For example: not too many people nowadays will take the trouble to develop their own operating system or programming language. The same holds for text processing software or communication protocols. On the supply side, a lot of software suppliers limit their attention to one or a few products. Others restrict their product range by limiting the number of releases. Standardization of software products might be perceived as a form of reuse: every time a copy is sold the whole product is reused.

Modularization is another response to the increasing complexity of products. It allows a lot of end products to be produced from a limited number of key components. Identification of such key components and standardization based on these components is one way to achieve modularization. Just as in industry, the pyramid view of a product is replaced by an hourglass view of a family of products. Modularization of software leads to the reuse of components. Standish (1984) states that a field of engineering has to undergo considerable evolution and considerable traffic in application before a useful practice of componentry emerges. First of all, reusable components have to be discovered, standardized and widely taught.

Production control situations
Four production control situations have been distinguished. They were make-to-order, engineer-to-order, assemble-to-order and make-to-stock. The relevance of the production control situations for the control of software development will be addressed.

Engineer-to-order
Engineer-to-order production is characterized by the fact that products are largely developed on the basis of specifications agreed upon in a particular order. Most software development nowadays is done in what could be called an engineer-to-order control situation: an internal or external client comes along, a specification is made and a project is started that should result in the specified product. A difference with industry is that software development is often started from scratch. Engineer-to-order production puts a lot of emphasis on the exploitation of experience. It uses reference products and processes as much as possible. It might be said that a lot of software organizations act like engineer-to-order organizations, without emphasizing the reuse of products and processes.

Make-to-order
Two types of make-to-order companies were distinguished in section 5.3. The first type produces the same customer order repeatedly. This production situation is not relevant for software, because reproduction of the same software product involves a negligible effort. The other type of make-to-order company was the one which sells capacity. Those kinds of companies are found in the software industry. A lot of software houses do still operate as capacity-selling companies: a client supplies a specification and the software house provides the engineers who develop the product. A software house typically employs development control, the control stage discussed in chapter 3. They are responsible for delivering (part of) the product. Maintenance is usually not their responsibility, nor is the reuse of software.

Assemble-to-order
The third production control situation distinguished is assemble-to-order. The decoupling point is moved downstream: assembly is done for specific client orders, the manufacturing of components is client-order independent. The assemble-to-order situation is also relevant to software development. Many software products cannot be developed from scratch because the time to engineer the product to order is too long to be acceptable. On the other hand, such products are also too client-specific to be sold off the shelf. One way to meet the conflicting needs is to develop key components client-independently and assemble an end product on a client-specific basis. The fact that software components can be copied for nothing is another good

reason to reuse software components and assemble them for a customer order. The client can still specify the product to some extent, as long as it is possible to assemble the software products from the key components. Assemble-to-order production is usually restricted to one or a few phases of a larger production system. The same will be true for assemble-to-order software engineering. The components assembled will have to be engineered by the department at an earlier stage or will have to be bought. The amount of additional development involved in assembly can vary to some extent. Customer-specific enhancements may be necessary.

Make-to-stock

The fourth production control situation distinguished is make-to-stock. The decoupling point is moved further downstream. The make-to-stock situation is also relevant to software development. The product is made to stock for an anonymous client that is not involved in its specification. The appropriate term for make-to-stock software development would be engineer to stock, because we only have to put one item into stock. Many software products nowadays are made to stock. Among the so-called off the shelf products are operating systems, personal computer packages such as text processors and data base management packages.

Engineer-to-order production as a reference

The relevance of the four production control situations to software engineering has been discussed. Engineer-to-order production will be used as the main reference for software engineering control in this thesis for three reasons. The first is that most software organizations nowadays act as engineer-to-order organizations. Most of the engineering process is driven by specific customer orders; the decoupling point is located upstream. The second reason to use engineer-to-order as a reference is that software engineering can learn a lot from engineer-to-order manufacturing. Experience has taught production to reuse reference products and processes. Production control and information systems are designed to support that reuse, as we shall see in chapter 7. They can act as a valuable reference for the software industry.

The third reason to use engineer-to-order as the main reference is that the reproduction of software products does not require as much emphasis of control as the reproduction of hardware. Due to its intangibility, subjects such as material coordination, lot sizes, scheduling and stock control are less relevant for software engineering control. Assemble-to-order, and especially make-to-stock, production focus on these 'reproduction' issues. The engineering activities are not considered by assemble-to-order and make-to-stock production control. Engineering is usually located and controlled by a separate department. Software control will emphasize engineering control and can therefore gain most insights from the use of engineer-to-order production control as a reference. Of course this was already suggested by the use of the word engineer in 'engineer-to-order'. In other words: even software companies which act as assemble-to-order or make-to-stock companies can benefit from the use of engineer-to-order as a reference.

5.5 Reuse of software

This section discusses some software-specific aspects of reuse. The subjects are: views of reusability and reuse dilemmas.

Views of reusability

There are many different views of reusability. A limited view is the definition of reuse as the reapplication of code. An expansive view of software reuse is the one defined by Biggerstaff (1989): reuse is the reapplication of a variety of kinds of knowledge about one system to a similar system in order to reduce the effort of developing and maintaining that other system.

The expansive view of software reuse includes both composition and generation technologies. Composition techniques are characterized by the fact that the components to be reused are atomic and, ideally, unchanged in their use. A prerequisite for doing this is the availability of reusable building blocks. A component library is a typical tool for the composition technology. Object-oriented programming and software engineering environments are often associated with the composition approach.

Generation technologies are also perceived as a reuse technology by Biggerstaff (1989). The components which are reused are difficult to characterize and are described as patterns. Examples of techniques belonging to this technology are very high level languages and application generators. Whereas the composition technology focuses on components and their composition, the generation technology focuses on languages and generation facilities. In other words, the composition technology focuses on products while the generation technology focuses on production resources. The framework for reusability that represents Biggerstaff's expansive view of software reuse is presented in Table 5.2.

Table 5.2 A framework for reusability technologies (Biggerstaff 1987)

FEATURES	APPROACHES TO REUSABILITY			
component reused	building blocks		patterns	
nature of reuse	atomic and immutable passive		diffuse and malleably active	
principle of reuse	composition		generation	
emphasis	application component libraries	organization & composition principles	language based generator	application generator
typical systems	libraries of subroutines	object oriented	very high level language	

The concept of reuse can be expanded even further. Basili (1989) and Rombach (1990) argue that not only products can be reused, but that processes and other knowledge can be reused as well. For example: reuse of design inspections or test procedures can be considered as reuse of process knowledge. Rombach (1990) also mentions the reuse of cost models as an example of knowledge reuse.

Reuse of software components, generation tools, reuse of processes and reuse of knowledge can be distinguished as identifiable subjects in software engineering. Obviously the four are related. The different subjects can, however, be distinguished and improvement in one of the areas can be achieved. 'Start small, expand later' is a valid starting point for the reuse of software development, as well as for research on the subject. The remainder of the thesis will take a much more limited view of reuse. We will restrict ourselves to the reuse of components. The thesis will focus on the lessons that software development has already learned and those that can be learned from other industries.

Reuse dilemmas
Biggerstaff (1989) identifies three reuse dilemmas that an organization has to face. The first is the generality versus applicability payoff. In general one can say: the more general the component, the less payoff for a specific application. On the other hand: the more a software product is specified towards one particular application, the less applicable it will be for reuse. A system aimed at a broad domain of application will be less powerful for a specific domain than a system whose focus is narrowed down to this one domain of application.

The second dilemma is component size versus reuse potential. The bigger the component, the higher the payoff if the component is reused. The probability that the component can be reused will, however, be reduced because it will become more and more specific as it becomes bigger. Another decision that has to be taken is whether the department is aiming at reusing code or reusing more integrated software artifacts.

The third dilemma is the setup and cost of a components library. Considerable efforts and investments have to go into a software library before it starts to pay off. An example where a reuse factor of 14 percent was achieved within a year has been mentioned. This is an indication of the fact that reuse can pay off, but that the payoff period can be considerable. Such a payoff period does not fit in very well with the short term interests of development departments in particular and some businesses in general.

5.6 Multiproduct control

We described the change from an organization that is responsible for development to a company that is responsible for a software product over its the entire life cycle. This was called the change from development control to product control. This chapter has so far discussed the reuse of software. The control focus of an organization should be expanded further if an organization wants to emphasize reuse. It is not enough to control one software product over its life cycle. A number of software products will have to be taken into consideration to allow for reuse. This chapter does in fact describe the change from product control to multiproduct control. An organization that is employing multiproduct control is typified as a software factory. We will discuss the consequences of the change to multiproduct control in terms of product life cycles, goal and organization.

Product life cycles
An emphasis on the reuse of software requires adaptations of the development life cycle. The development of reusable components and the reuse of components in development are distinguished.

Creating components
There are two ways to create reusable modules. One can generalize components resulting from a specific development effort and one can develop components especially for reuse. The former is termed the reactive approach and the latter the proactive approach. Generalizing a component after development implies an additional phase after the traditional life cycle. The emphasis in this phase will be on quality attributes that McCall typified as transition quality attributes (see section 4.2). The attributes were portability, reusability and interoperability. An advantage of generalization after development is that the costs of creating reusable components

can be relatively low because the component has been developed anyway and the only additional costs are the generalization costs. A disadvantage is that the creation of reusable components depends on customer orders. Reuse is a reaction to the products under development. The software department does not take the initiative to develop components which will become of interest in the future.

An organization can also proactively create reusable components. In production control terms the department decides to engineer components to stock. An example is the development of a file handler before any document retrieval system is ordered. Another example is the development of tax routines independently of an order for a bookkeeping system by a specific client. In principle, the proactive development of components will follow the traditional life cycle. An advantage of this proactive approach is that the components are specifically developed for reuse. The revision attributes have been taken into account from the beginning. Effective anticipation will shorten the lead time for the development of new products because a lot of effort has already gone into developing key components. A prerequisite is a clear product policy; the department has to decide on the specialization towards certain (application) domains and on the kind of products it intends to deliver, both now and in the future. A difficulty is that the components have to be made to stock without having the certainty that they will meet future demands. Taking account of all the difficulties involved in requirements analysis for concrete customer orders, it will be clear that anticipating the long-term demand will be even more difficult.

Reusing components

Not only developing the components, but also reusing available components requires adaptations to the traditional life cycle. The specification phase will remain the first phase. Test and maintenance will remain phases as well. Design and code, in particular, will be affected by reuse. In general, four reuse steps can be identified: finding components, understanding components, modifying components and composing components (Biggerstaff 1987).

A critical issue in finding a component is matching the module required to the modules available in a component library. Finding and matching software components depends on the description and the accessibility of the component. This appears to be a major obstacle to the extensive reuse of software, according to Biggerstaff and Richter (1987) as well as Sikkel and Van Vliet (1988). Horowitz and Munson (1984) address four subjects that have to be studied to make the concept of reusable software a reality. The subjects are: mechanisms for identifying components, a method for specifying components, the actual form of the components (implemented in programming language or described by a program design language?)

and a way to catalogue the components. We regard them as an information issues, which will be discussed later on.

Goal

The second aspect of multiproduct control to be discussed is the goal of a multiproduct company. The goal of an organization employing multiproduct control differs from an organization using traditional development control or product control. The aim of multiproduct control is to minimize cost and maximize the functionality and quality of the product range that is being, and will be, developed. Such a goal is more ambitious and requires control of the products in development and in use, as well as a policy for the current and future product range. A software company cannot afford to meet any customer requirements, because reuse can only be accomplished if the company sticks to a certain product range which allows the reuse of components. The availability of resources and capacities is no longer the only criterion for accepting orders. The way in which the new order fits into the defined product range will be one of the primary acceptance criteria.

It is of the utmost importance for a software factory to state its reuse goals explicitly. The intended reuse rates should be specified. This can be done in terms of the proportion of reused software in new software products. It is also possible to specify the reuse goals in terms of the number of times that reusable components are incorporated in new products. Reuse goals must be made operational to be able to get an insight into the benefits of reuse, as opposed to the additional costs.

Organization

Regarding the organization of multiproduct control, two subjects will be addressed: work order release and organization of development, maintenance and development for reuse in one or more departments.

<u>Work release</u>

A software engineering organization employing multiproduct control faces three lines of work, namely development, maintenance and development for reuse. The work has to be done by the same scarce capacities. The division of work over the scarce resources must be coordinated. Production control has faced situations with similar characteristics and has identified the task 'work order release' for this purpose. We will introduce some production control principles in order to put work order release into perspective.

In production control, different aggregation levels are distinguished. These are the goods flow control level and the production unit level. A production unit is a department which on short term is self-contained with respect to the use of its resources, and which is responsible for the production of a specific set of products from a specific set of materials and components (Bertrand 1990, page 13). A production unit does, however, face constraints such as limited capacity. A production unit reaches agreement with the goods flow control level about the products to be produced and the production lead time. This brings us to the goods flow control level, which includes:

- The coordination of production levels of a number of production units
- The coordination of production and sales.

Goods flow control is broken down into aggregate production planning and material coordination. The three levels of production control are given in Figure 5.4, which shows aggregate production planning, material coordination and production unit control.

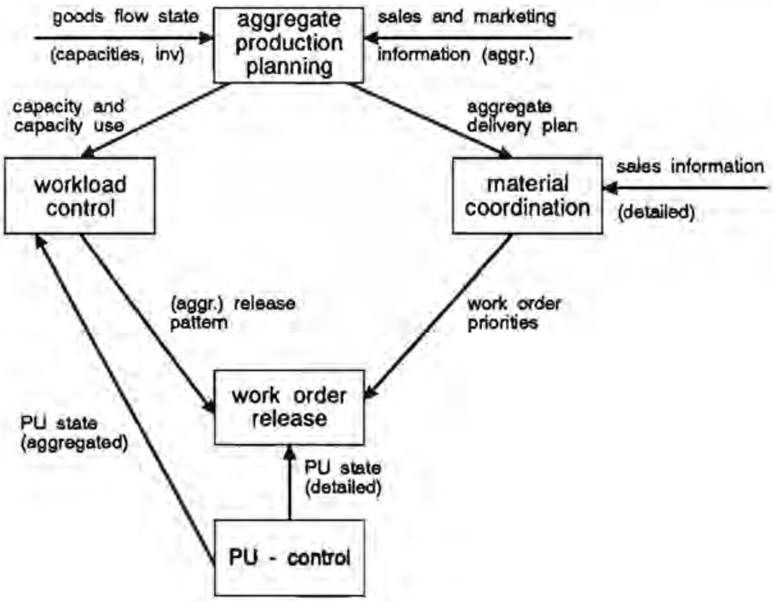

Figure 5.4 The Global goods flow control structure (Bertrand 1990, page 57)

Material coordination indicates the work order priorities, based on an aggregate delivery plan and sales information. Work order release determines which orders are released to a production unit. The release decision is based on the work order priorities, the aggregate release patterns and the status of the production unit.

It is interesting to note that production control distinguishes between different aggregate levels and puts a lot of emphasis on capacity allocation. Let us apply the production control concepts to software engineering. Capacity allocation attracts less attention in software engineering. This can be explained by the fact that work order release is not required if engineers work on one project at a time, as was usually the case in traditional control. The next activity is defined in the project plan and the engineer starts the next task as soon as the previous task is finished. Work order release becomes important if several lines of activity use the same scarce resources, as is the case in multiproduct control.

Experience shows that the lead time for a work order increases if too much work is released to a production unit. The output of the unit will decrease and the lead time of work orders will increase if the input is bigger than the output. The empirical study discussed in section 2.4 can be perceived as an example of a lack of control of work order release. The maintenance orders were released to the engineering department irrespective of the available resources, the current workload and the output of the system. The input outnumbered the output. As a consequence, the lead times for both development and maintenance increased. The introduction of maintenance weeks can be perceived as an attempt to regulate the input to the production unit.

Work order release is a function that should be fulfilled at an aggregate level of control in the engineering department. Available work orders, current work load and the priority of work orders have to be evaluated in order to be able to do the job properly. Work order release can be done by the departmental management, possibly supported by a planning department. Obviously, work order release cannot be done by a project leader because his focus is and should be limited to a project.

Organization of development, maintenance and development for reuse

Three lines of activity are performed in multiproduct control. The issue at hand is whether the three lines of activity are to be performed by one or more departments. As in the organization of product control, three possible solutions are distinguished:
- One in which projects and development for reuse are united
- One in which they are divided over different organizations
- One in which they are together, but dispersed in time.

The advantages and disadvantages for all three situations will be discussed. The discussion on work order release has led to the introduction of the term production unit. The decision upon the number of production units is not the issue here. The issue at hand is whether or not departments should specialise in one of the three lines of work.

The first situation is the one in which the traditional software development organization is given the assignment to reuse software as an additional control goal. An advantage in this situation is that the developers of software products for specific customers are also involved in the creation of reusable components. Here the reactive method of creating reusable components is employed. A developer who has designed a piece of software in a project that can potentially be reused later, is allowed to improve the software further with regard to aspects such as portability, reusability and interoperability. In using the proactive approach to generate reusable components, the development of the component will be a separate development effort, with special emphasis on transition quality attributes. The fact that the people who are in charge of producing customer-specific software also produce the components will improve the acceptance of the component. The components are not threatened by the 'not invented here' syndrome.

Of course there are also disadvantages in combining development and reuse. They can be witnessed in most software organizations nowadays. The two have always been together: the potential of reusing software has been acknowledged for decades now, but reuse has not really taken off in most organizations. Basili (1989) reveals some of the causes in a paper called 'Software development; a paradigm for the future'. He stresses the necessity to do several off-line activities to allow for the reuse of software, process models and development experiences. The traditional organization is usually a project organization geared to the development of a specific system. A project cannot afford to spend extra time on tailoring experience for another project within the budgetary constraints. Consequently, reuse cannot take off. Basili states that if there is no separate organization to look after the exploitation of experience, short-term interests will always predominate at the cost of the long-term interests. Basili states that the off-line activities require a different focus, a different set of processes and an independent cost base. He also states that two functions that differ so much are to be put into two different organizations.

This brings us to the second possible organization of development, maintenance and development for reuse. Basili calls the first organization the project organization and the second organization the experience factory. A project organization is an organization that makes products according to customers' specifications. The experience factory packages experience that is obtained during projects and makes it available for future reuse. It builds and maintains the experience base. The experience factory can also act as a component factory if the project organization requires a certain component.

An advantage of this organizational structure is that there is an independent organization which is committed to looking after reuse. At least there are some people who are not bothered by the short term problems and can afford to look beyond the horizon of the project at hand. Another advantage may be that engineers in the experience factory can judge the reusability potential of components more objectively because they are not biased by personal involvement in the development of customer packages.

There are also some disadvantages, however. Firstly, the initiative for the development of a reusable component is reactive: either the project organization delivers a software product that is worthwhile generalizing, or the project organization orders a component that is not yet available. An organization that wants to control its product range in order to be able to exploit the potential for reuse should, in our opinion, take the initiative to develop key components. Secondly, there is also a danger that separating customer-specific software development and the development of reusable components may be fighting the symptom, but not attacking the disease. The symptom is that reuse is not taking off. The disease is that software engineers are so preoccupied by current problems that it is not possible to strive for long-term goals, such as achieving productivity improvements by reusing software. The experience factory may be viewed by the project people as another staff department. A department that has no clear responsibilities, such as meeting the deadline the project people are facing. At the same time, the experience factory is allowed to bother the project organization. As a separate entity, it faces the same danger as some quality improvement departments: they are perceived as a burden by the people who do the actual work for customer orders.

The third organization to be distinguished is a middle course between the two previous organizations. Both projects and development for reuse are in one organization, but the two are dispersed in time. The involvement of the engineers can be obtained, because they do both kinds of activities and are expected to reuse both the software items and the experiences they have made available for reuse themselves. A disadvantage of this situation is that the control of such an organization will be complex. The same organization carries out development, maintenance and development for reuse activities. There is a risk that short-term interests will again prevail. Goals should therefore be stated clearly and those responsible for maintenance and reuse should have the power to exact concessions which will allow them to do their job. The control will probably have to be broken down into different levels to be able to handle the complexity, just as in the case of production control. It should be possible to decide upon the distribution of capacities over development, maintenance and development for reuse at a higher level of control. The distribution should be stated to the production unit with lower and upper bounds to allow the engineering organization to react to unexpected events. The production units themselves can decide upon the distribution of the work over time.

The advantages and problems of the various organizational options are shown in Table 5.3.

Table 5.3 Advantages and problems of organizational options

Development, maintenance and development for reuse	ADVANTAGE	DISADVANTAGE
Separated	- clear responsibilities - objective judgement of reusability potential	- not invented here syndrome - another staff department - fighting the symptom
United	- involvement - use and reuse belong together	- short term interest may predominate
Dispersed in time	- involvement - accountability	- increased control complexity

5.7 A specific software factory

Multiproduct control can be achieved in many ways; a lot of decisions have to be taken with regard to the control. We will specify one software factory which will be taken as a starting point in the remainder of the thesis. We do not argue that this software factory is the ultimate one. An organization will have to arrange its multiproduct control in the way that fits their purposes best. The software factory specified will be taken as a starting point for the design of an information system in chapters 7 and 8. The control characteristics which will be made specific are: goal, production control situation and organization.

The goal of the software factory we have in mind is to minimize cost and maximize the functionality and quality of the product range that is being, and will be, developed. To be able to achieve this goal, the company has limited its product range. It has done this because it wants to distinguish itself from other software suppliers. The department is held responsible for the products it has developed in the past; i.e. it has to control maintenance on products that have been developed in the past.

The production control situation can be characterized as an engineer-to-order situation with an emphasis on reuse of software components. This means that product specifications are agreed upon in particular customer orders. Customers orders must, however, fit into the defined product ranges. Software engineering uses references as much as possible. This holds for reference products and reference processes. Software components are engineered for future reuse, both proactively and reactively. The department invests in the reuse of software components, both by generalizing software products for potential future reuse and by developing software components in advance for future reuse. Descriptions of products that have been developed in the past must be available to the engineers in order to be able to reuse them. The same holds for process know-how. A customer order is called a project in this environment. Projects need to be controlled with respect to the aspects of quality, time and money. The organization is well aware of the fact that there is more to be done than simply completing projects.

The organization consists of a number of production units that perform development, maintenance and development of reusable components for the products that belong to their product range. The department has evaluated the advantages and disadvantages of specialization of departments in development, maintenance or development for reuse. It has decided that in its particular situation, the involvement of engineers in all three activities is very important and the activities are therefore performed in one organization, dispersed in time. Consequently, work order release requires special emphasis.

6 TOWARDS A SOFTWARE FACTORY

6.1 Introduction

Three stages in the control of software engineering have so far been discussed. The main difference between the stages is the object of control. The stages were called development control, product control and multiproduct control. In development control, the object of control was limited to the development phases only. Product control extended the object of control to the entire product life cycle. Multiproduct control extended the control object from one product to multiple products. An organization that is in the multiproduct control stage has been typified as a software factory. Up to now the discussion has concentrated on the differences between the stages. The requirements to be fulfilled in order to be able to advance from one stage to the next have not yet been discussed. It might be thought that progress from development control via product control to multiproduct control would be just a matter of time. The opposite is true. A certain level of process control is required to be able to advance from one stage to the next, as will be demonstrated below.

Humphrey (1988, 1989a, 1989b) developed a software process maturity framework, which can be used to assess the capabilities of a software engineering organization and to identify the major areas for improvement. His framework will be used to outline the relation between the level of process control and the stage of control.

6.2 Levels of process control

The relation between product and process quality has already been addressed. This section concentrates on process quality and discusses the five levels of process quality distinguished by Humphrey (1988, 1989a). The levels of process control are discussed in detail because they will be used to describe the steps towards a software factory.

Humphrey (1989) defines a process as a set of tasks which when properly performed, produces the desired result. He further states that an important first step in addressing software problems is to treat software engineering as a process that can be measured, controlled and improved. Humphrey has applied Deming's work on statistical control to software engineering. Deming states that once a process is under statistical control, a consistently better result can only be achieved by improving the process (Deming 1986). Deming applied his ideas to manufacturing. Humphrey states that despite the differences between manufacturing and software engineering, there is no apparent reason why Deming's approach should not work for software.

The maturity framework was developed to be able to enable the capabilities of software engineering organizations to be characterized. The framework can also be used by a software organization to assess its own capabilities and identify the most important areas for improvement. Humphrey distinguishes between and describes five levels of process maturity. These are given in Table 6.1. The main condition to be satisfied in order to advance to the next level is also included.

Table 6.1 The levels of process control

Level of process control	Main condition to be satisfied to be able to advance to the next level
Optimizing	
Managed	Process control
Defined	Process measurement
Repeatable	Process definition
Initial	Basic management control

The five levels will be discussed in greater detail. The characteristic of the process level and the key actions for advancement to the next level will be addressed.

Initial process control
The initial process can be characterized as ad hoc and chaotic. The organization operates without explicit procedures, cost estimates and plans. An organization at this level of process control is usually driven from crisis to crisis by unplanned priorities and unmanaged change. Organizations like this typically do not meet their commitments. Staffing the organization is not the main problem, managing the organization is. Problems that occur are, for instance: missed deadlines, missing specifications and failure to integrate components because of inadequate configuration management. A test for an organization is its behaviour in a crisis. If it abandons all procedures and rushes back into code and fix practices, it is likely to be at the initial level of process control. Humphrey characterizes this level as follows: "When projects do succeed, it is generally because of the heroic efforts of a dedicated team, rather than the capability of the organization." Humphrey based an assessment method on the distinction between the five levels of process control. Surveys based on this assessment method, revealed that 76 percent of the contractors working for the United States Department of Defense operate at a initial level of process control (Humphrey 1989b).

Improvement areas to focus on are: change control, project planning, project management and software quality assurance. Uncontrolled changes are a well known cause of problems in software engineering. If changes are not controlled, a product will become unchangeable and unreliable. Change control should ensure that changes are tackled in the right way. Project planning and management should ensure that realistic plans are made and that their progress is tracked. They should enforce effective control of commitments.

Repeatable process level
The second control level distinguished is the repeatable process level. Humphrey states that it may take between one and three years to advance from the initial level to a repeatable level, even with dedicated management commitment. The main difference between it and the initial level is that it has established basic project controls such as project management, management oversight, product assurance and change control. Humphrey summarizes this by the statement that organizations at the repeatable process level provide 'commitment control'. The process at this level is under statistical control, though the process still depends heavily on individuals. The organization is frequently faced with quality problems. Humphrey's surveys reveal that 22 percent of the organizations operate at the repeatable level of process control. The capabilities of the organization stem from their prior experience with similar work. A similar development effort would have more or less the same result. The process is vulnerable if changes occur. Examples of changes that entail risk are new tools and methods, the development of a new kind of product and major organizational changes.

Training is a key activity at the repeatable level for spreading experiences and working practices among the individuals of the organization. The focus of control should change from individual projects and products to the software process. An organization which operates at the repeatable process level lacks a framework for improvement. Other key actions are the installation of a process group which focuses exclusively on improving the engineering process. While software quality assurance is busy with enforcing the current process, the software process group aims at improving it. Humphrey provides a rule of thumb: a process group should be about 1 to 3 percent of the size of the engineering organization. Another key action is the establishment of a software process architecture which describes technical and management activities for the proper execution of the engineering process. A third key action is the introduction of a family of software engineering methods and technologies such as software inspections, design methods, modern implementation languages, library control systems and testing methods.

Defined process level

The defined process level is the stage in which the foundation is established for examining the process and deciding how to improve it. The advancement from the repeatable to the defined process level may take another one to three years, according to Humphrey. The process has been defined, installed and institutionalized and the organization will stick to the process, even when it is facing a crisis. Insight into the control of the process is still qualitative, however, and there is as yet little data to indicate how effective the process is. Only one percent of the organizations that have been assessed by Humphrey has achieved a defined process level. Quantification is a major step to the next level of process control. Key actions are: the start of process measurement and assessment of the relative quality of the products. Quantification and data collection are important issues for improvement.

Managed process level

The advancement to the managed level of process control will bring considerable improvements to the engineering organization. The process knowledge that was available in qualified terms will be quantified at this level. Management's attention should now be focused on quantitative planning and process control. The focus will shift from problem solving to problem analysis and even to problem prevention. Collection of process data is an important activity here. Humphrey states that the greatest potential problem at this level is the cost of gathering data. So far, Humphrey has found no entire organizations that operate on the managed or optimizing level of process control.

Optimizing process level

The optimizing level provides process control. Data for tuning the process are available in the optimizing process. The emphasis has shifted from problem-solving to problem prevention. The emphasis is on the improvement of the process itself. The optimizing process helps the manager to understand opportunities for improvement. One of the opportunities is automation of the software engineering process. Humphrey mentions automation as a key area at the optimizing level. Tools may be helpful at lower levels of process control, but it is only after process control has been put into practice that automation will result in significant productivity and quality gains. In other words: "Automation of poorly defined processes will result in poorly defined results" (Humphrey 1989a, page 23). Professionals can communicate in quantitative terms on the process of software engineering. The optimizing process provides a disciplined environment for professional work. Humphrey puts the emphasis on the difference between discipline and regimentation: discipline does not affect the actual conduct of the work, while regimentation does. Humphrey ends with the remark that "Discipline thus enables creativity by freeing the most talented software professionals from the many crises that others have created" (Humphrey 1988, page 78).

Use of the framework

The five levels Humphrey describes give a comprehensible characterization of the levels of process control. The characteristics of the levels and their key problem areas are summarized in Table 6.2.

Table 6.2 Another characteristic of levels of process control (Humphrey 1989)

Level of process control	Characteristic	Key problem areas
Optimizing	Improvement fed back into process	automation
Managed	(quantitative) Measured process	problem analysis, problem prevention
Defined	(qualitative) Process defined and institutionalized	process measurement, process analysis, quantitative quality plans
Repeatable	(intuitive) Process dependent on individuals	training, technical practices (reviews, testing), process focus (standards, process groups)
Initial	(Ad hoc, chaotic)	project management, project planning, configuration management, Software quality assurance

The distinction between the different levels should not be interpreted in an absolute sense. For example, it may be that process measurement is applied at the initial level of process control or that configuration management is still a problem at the repeatable process level. Obviously, an organization does not advance two or three levels by merely adopting the techniques that Humphrey prescribes. The framework does, however, give an idea of the different levels of process control in software engineering and provides clues about how an organization can advance from one level to another.

6.3 Steps towards the software factory

The stages development control, product control and multiproduct control have been described in the preceding chapters. This section will discuss how an organization could proceed from one stage to the next. In other words; it will outline the way to a software factory. It might be supposed that the advancement from one control stage to the next is just a matter of time. The opposite is true. We will argue that two requirements must be fulfilled to be able to advance from one control stage to the next: an explicit decision by the organization concerned and a certain level of process control in terms of Humphrey.

A first requirement is the decision to advance to the next stage of control. It is not essential for all the existing software engineering organizations to advance from development control to product or multiproduct control. For example, an organization may choose to limit itself to development control and capture a niche in a market. An organization may also decide to limit itself to the development and maintenance of one product, or one well-defined product family. The product control stage will suffice in this case.

The second requirement for the advancement to product or multiproduct control is a certain level of process control. It takes a minimum level of process control to be able to achieve product or multiproduct control. An example: an organization that cannot control the development of a single product, will certainly not be able to control both the development and maintenance of that product. The remainder of this section will outline the relation between the level of process control and the stage of control an organization can attain.

Development control is the most limited focus of control we have identified. This stage requires an organization that operates at a repeatable level of process control, for it is impossible to control development at an initial level of process control. The initial level has been typified as chaotic. Organizations which operate at the initial level of process control often do so without project plans, ex post facto costing and accepted working procedures. They hardly track the progress of their projects in relation to the plans, let alone that being able to control their projects with respect to the aspects of quality, time and cost. Humphrey argues that the introduction of basic management skills is a minimum requirement for achieving a repeatable level of process control. Examples of basic management skills are change control and project planning. The same kinds of skills are a requirement for controlling development projects at the stage we have called development control.

The repeatable level of process control may be appropriate for controlling development projects, but it is insufficient for controlling all the engineering activities required during the life cycle of a product. Organizations which operate at a repeatable level of process control "typically have their cost and schedules under reasonable control. They generally do not have orderly methods for tracking, controlling and improving the quality of their software process" (Humphrey 1989b, page 178). Quality requirements are insufficiently acknowledged and controlled at the repeatable level. An organization that chooses to be responsible for both development and maintenance cannot afford to relinquish its responsibility for quality. For that reason, the level of process control will have to be raised to be able to control both development and maintenance.

Multiproduct control extends the responsibility of an organization even further. The goal of product control is to maximize quality at minimal cost over the lifetime of a certain product within the given time constraints. The goal of multiproduct control is to minimize cost and maximize the functionality and quality of a certain range of software products. The goal of an organization which strives for multiproduct control is quite ambitious. A high level of process control is therefore required. An organization that is employing multiproduct control intends to exploit the potential of reuse. It has identified a product range in which it operates. This organization will invest in components it intends to reuse later. This means that it must look beyond the products under development and have a clear view of its future product range. An organization that needs to look beyond its current projects and products has to control its operations without being confronted by the day-to-day problems which beset most of the current software engineering processes. It is questionable whether the defined level of process control is sufficient for multiproduct control. At this level, the knowledge of the process is mainly qualitative. A more quantitative knowledge of the software engineering process is probably required to allow for multiproduct control. Advancement to the managed or optimizing level of process control would appear to be necessary. Statements about the defined, managed or optimizing level of process control must be restrained, because so far only a few organizations operating at these levels of process control have been found.

The preceding argument is visualized in Figure 6.1.

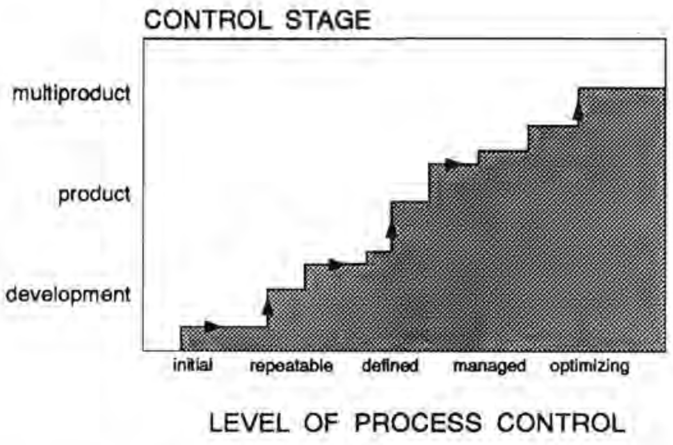

Figure 6.1 The relation between the level of process control and the control stage

It should be obvious that Figure 6.1 is not normative but indicative. The figure only visualizes the steps to a software factory. All we aim to show is that process improvement is necessary for the advancement from development control through product control to multiproduct control. The shape of the steps shows we believe in incremental process improvement and in incremental expansion of the control focus. An example of incremental improvement of the stage of control is as follows: an organization in the development control stage can enter into a maintenance agreement with one of its clients. It can then enter into similar contracts with all of its clients after it has gained some experience and become an organization employing product control. The level of process control should be raised simultaneously.

6.4 Summary and conclusions

This chapter has outlined the relation between the level of process control and the stage of control an organization can attain. Section 6.2 discussed the software process maturity framework, developed by Humphrey (1988, 1989a, 1989b). He applied Deming's work to software engineering and distinguished between five levels of process control. The five levels of process maturity are the initial, repeatable, defined, managed and optimizing level of process control. The key problem areas at the different levels of process control are described by Humphrey. Quantification and data collection are among the actions at different levels of process control. The framework is used in section 6.3 to be able to outline the relation between the level of process control and the stage of control an organization can attain.

The description of development, product and multiproduct control showed that there is a relation between the advancement in process control and the stage of control an organization can accomplish. The relation between the advancement in the process control level and achievable control stage is visualized in the steps towards the software factory (Figure 6.1). The main point is that the relation between process control and the stage of control is acknowledged. Organizations aiming at product or multiproduct control should be aware of the existence of the steps and deliberately work their way upwards.

7 INFORMATION SYSTEMS IN SOFTWARE ENGINEERING CONTROL AND PRODUCTION CONTROL

7.1 Introduction

The next three chapters will discuss information requirements and information systems for software engineering control. This chapter describes the search for reference information systems and the assessment of the reference systems with respect to software engineering control in the software factory. The description of control in the software factory given in section 5.7 will be used as a starting point. The reference systems that will be explored first are information systems for software engineering control as they are proposed in the literature. Three systems will be described in section 7.2. They are the Project Management Data Base (Penedo 1985), The TAME system (Basili 1988, Jeffrey 1987) and the experience base as described by Noth (1987). They have been selected because we consider them to be important information systems for software engineering control which have been described in the literature. The three systems will be compared with one another and will be assessed with respect to control in the software factory in section 7.3. An information system for engineer-to-order production control, as described in (Bertrand 1990), will also be considered as a reference system. This system will be described in section 7.4. It will be compared to the reference systems already discussed and will be assessed with respect to control in the software factory in section 7.5. Section 7.6 contains the summary and conclusions of the seventh chapter.

7.2 Information systems for software engineering control

This section describes three information systems which support the control of software engineering activities, as proposed in the literature. They will be used as a reference system for designing an information system for a software factory. The systems discussed are:

- The TRW Project Management Data Base, as described by Penedo and Stuckle (1985),
- The TAME system as described by Basili and Rombach (1988), Jeffrey and Basili (1988),
- An experience data base, as described by Noth (1987).

The description of the information systems will focus on their goals and data models. The data models represent the skeleton of the information systems and allow a comparison to be made. This section contains a description of the three information systems. A comparison of the system and the assessment with respect to the software factory will be made in section 7.3.

TRW Project Management Data Base

The TRW Project Management Data Base (abbreviated as PMDB) was one of the results of TRW's studies of software engineering environments. The goal of PMDB is to provide a project library that captures the information generated during software development. Four steps were defined to accomplish this task. They were:

1 Identification of a PMDB model
2 Synthesis of procedures and methods associated with PMDB
3 Identification of user views
4 Prototyping and implementation of the items above.

The results of the first steps are reported upon in the only paper about PMDB that is publicly available.

PMDB is aimed at supporting large projects. It is intended to store and relate the output of a project. The process by which the components in the database are entered, modified and controlled should be automated. PMDB includes products, resources and plans. It aims to support everyone involved in a project in their work, including programmers, managers, engineers and secretaries.

The PMDB data model will be discussed. PMDB is represented in an entity relationship model containing 31 objects, 220 attributes and 170 relationships. An entity relationship model of eleven key objects and their relations is given in Figure 7.1.

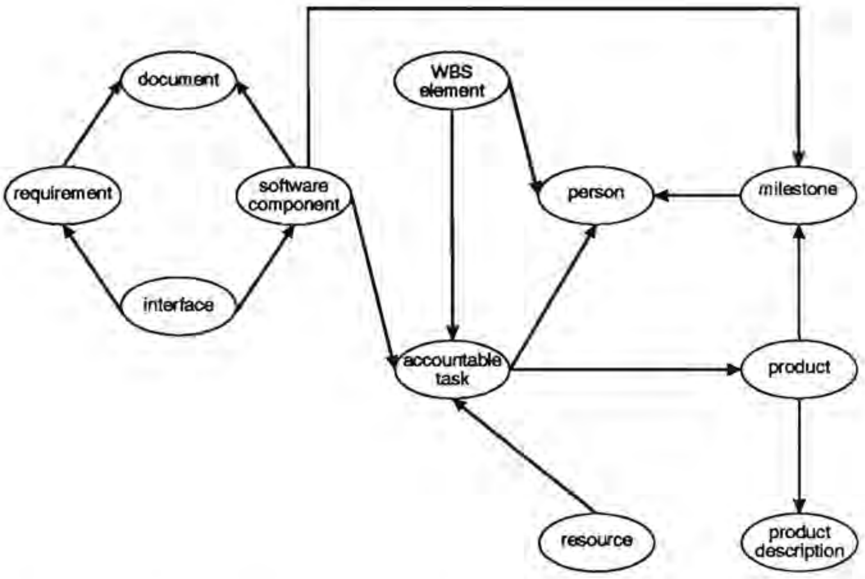

Figure 7.1 An entity relationship model of PMDB

The data model will be described from left to right. A development effort usually starts with requirements. Figure 7.1 shows that the requirement will define interfaces to the world outside the project and be described in a document. Requirements are allocated to software components which may vary from high level descriptions of a product to a routine. Software components can be associated with accountable tasks. According to Figure 7.1, the structure of a project is represented by a Work Breakdown Structure (WBS) consisting of accountable tasks. As we shall see, the entities WBS and accountable task are similar to the entities project and activity in data models of other information systems for software engineering control. Tasks consume resources and produce a product. A product is described in a product description and versions of the product are supposed to be finished at product milestones. Milestones are managed by persons, as are accountable tasks and the WBS elements.

The data model given in Figure 7.1 distinguishes between 11 objects only. As already mentioned, a complete data model would consist of all 31 objects. It would, for example, show that resources can be applied to purchasing software, hardware or consumables. It would also show that a product is related to objects such as hardware architecture, tools, hardware components and software components. A list of all objects and their description is given in Table 7.1.

Table 7.1 The objects as distinguished in PMDB

NAME	DESCRIPTION
Accountable task	Characterizes the accountable tasks/jobs to be performed by the project
Change item	Characterizes the changes that occur in a project, caused by customer request, a problem report or other reason
Consumable purchase	Characterizes the purchasing of consumables used by a project (or by a data processing facility within a project)
Contract	Characterizes the contract or sub-contracts which are part of the project
Data component	Characterizes the data items which are part of the software development
Dictionary	Characterizes the project dictionary, e.g. terms and acronyms.
Document	Characterizes any documentation, e.g. plans, specification documents, manuals, trade-off analysis, hardware lay-outs etc.
Equipment purchase	Characterizes purchasing of equipment, e.g. computers, parts etc.
External component	Characterizes systems which are external to the systems being designed/built by the project and interface with it
Hardware architecture	Characterizes hardware configurations
Hardware component	Characterizes each hardware component (or firmware) which is either part of a hardware design project or is part of the hardware in a software design project
Hardware component description	Describes characteristics of different types of hardware, either bought or designed and built by the project
Interface	Characterizes interfaces between software/hardware and hardware software components, or software and software components
Milestone	Characterizes each of the project's major milestones and internal schedules
Operational scenario	Characterizes scenarios which reflect acceptance that the system performs under required performance criteria
Person	Characterizes project personnel
Problem report	Characterizes problems which have been reported against baselined information
Product	Characterizes versions of deliverable products
Product description	Describes characteristics of different types of products
Requirement	Characterizes project requirements
Resource	Characterizes project resources, e.g., travel, reproduction, equipment, etc.
Risk	Characterizes elements which have been identified as risks to the project
Simulation	Characterizes simulation runs of software hardware architectures
Software component	Characterizes each software component (e.g. module, task, unit, routine etc.) of a project
Software configuration	Characterizes software configurations
Software executable task	Characterizes the packaging of software components
Software purchase	Characterizes the purchasing of software components, e.g. sub-systems or tools
Test case	Characterizes test cases
Test procedure	Characterizes test procedures which may use one or more test cases
Tool	Characterizes tools or programs utilized by a project during its life cycle
WBS element	Characterizes elements of the work breakdown structure for a project

The list of 31 objects includes the objects problem report and change request. These objects are used to report problems and changes during the life cycle of the product. We conclude from this that PMDB looks beyond development control and captures data during the whole product life cycle.

The authors mention some research issues related to PMDB. One is version control. They state that it will be difficult to capture the data from all the different versions because of the sheer volumes of data. The support of collection and retention of history is another research issue. Support of histories was one of the most requested user requirements. PMDB therefore aims to provide mechanisms for the collection and retention of historical data. Again, the large volumes of data may cause problems. Interfaces to other databases are also a research issue. Other databases which are considered relevant are personnel and company cost databases as well as a database with reusable software components. The last one is not considered as part of PMDB.

We conclude that the PMDB paper provides a thorough description of the data model of a project database and the issues that need additional research. PMDB shows the comprehensiveness of data collection in software engineering control. Unfortunately, PMDB is less specific about some subjects which are important for software engineering control in a software factory, such as the collection and retention of historical data as well as reusable software components.

TAME system
TAME stands for Tailoring A Measurement Environment. TAME is a project from the Department of Computer Science of the University of Maryland. The main goal of the TAME project is to create a corporate experience base which incorporates historical information, packaged in such a way that it is useful for future projects. TAME is based on years of research by the University of Maryland into the analysis of software engineering. The TAME project has a lead time of several years. So far a first prototype has been reported upon. The researchers intend to build a number of evolving prototypes in which the components distinguished will be built in gradually. A description of the TAME project is given in Basili and Rombach (1988). The TAME process model will be discussed to give an overview of this system. Next, the discussion will focus on its resource data model, as described by Jeffrey and Basili (1988).

The TAME system is supposed to provide the support for the TAME process model given in Figure 7.2.

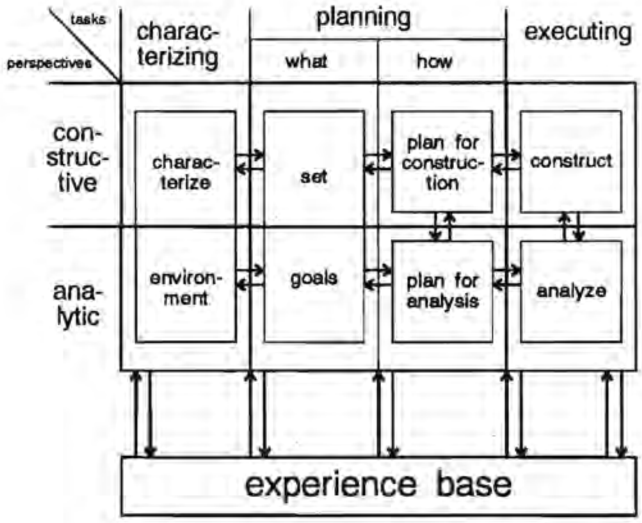

Figure 7.2 The TAME process model

The TAME process model distinguishes between three major tasks, namely characterizing, planning and execution. Characterizing is necessary to understand the factors that influence the environment in which the engineering effort takes place. Planning includes goal-setting in the TAME process model. Execution involves the construction of the software product. The two rows in the model distinguish between a constructive and an analytic aspect of software engineering. Software construction needs to be improved to be able to generate higher quality software. Improving the construction process requires analysis of the current software engineering process and products. Experience on ongoing projects is fed into the experience base. Future projects should benefit from the available experience, which contains information about both products and processes. The TAME system aims to support all components of the process model, except for the execution of the construction.

The TAME research so far has put a great deal of emphasis on modelling resource data. The TAME resource data model, as described by Jeffrey and Basili (1988), will be discussed. It is the only TAME data model that is available at this point of time. The heart of the data model consists of four entities and three relations. It is given in Figure 7.3.

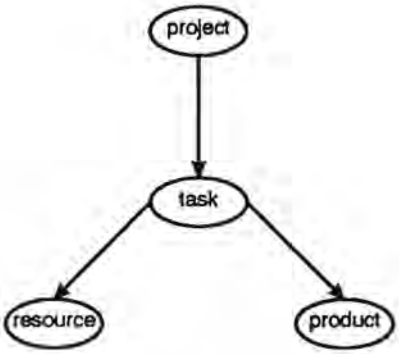

Figure 7.3 A model of a software project

Figure 7.3 shows that a project consists of a number of tasks. Tasks consume resources and produce a product. The same four entities and relations can be isolated in the data model that was proposed by Penedo and Stuckle. The comparable entities are: WBS element, accountable task, resource and product. The resource data model is refined. Firstly, resource type and resource use are distinguished. Four resource types are separated: hardware, software, human and support resources. Hardware resources contain all the equipment that is used or will potentially be used in the environment under consideration. Software resources encompass all the previously existing programs and software systems that are used or will potentially be used in the environment under consideration. Human resources encompass all the people used, or who can potentially be used, for engineering or operation. Support resources comprise additional facilities such as materials, communications and supplies.

Resource use is described by resource incurrence, resource availability and resource use descriptors. Incurrence allows a distinction to be made between estimated and actual resource use. Resource availability allows a distinction to be made between desirable, accessible and utilized resources. Desirable resources are all the resources within the organization that could be of value for the project. Accessible resources are those that can be used on the project, utilized resources are a subset of the accessible resources and are defined as the resources actually in a project. Resource use descriptors mentioned are: the nature of the work (designing, coding, inspecting), the point in calender time and resources utilized (for example, in hours or dollars).

Four dimensions of resources have been identified: resource type, resource use, resource incurrence and resource availability. The structure proposed is presented in Figure 7.4. This model is a refined version of the model presented in Figure 7.3.

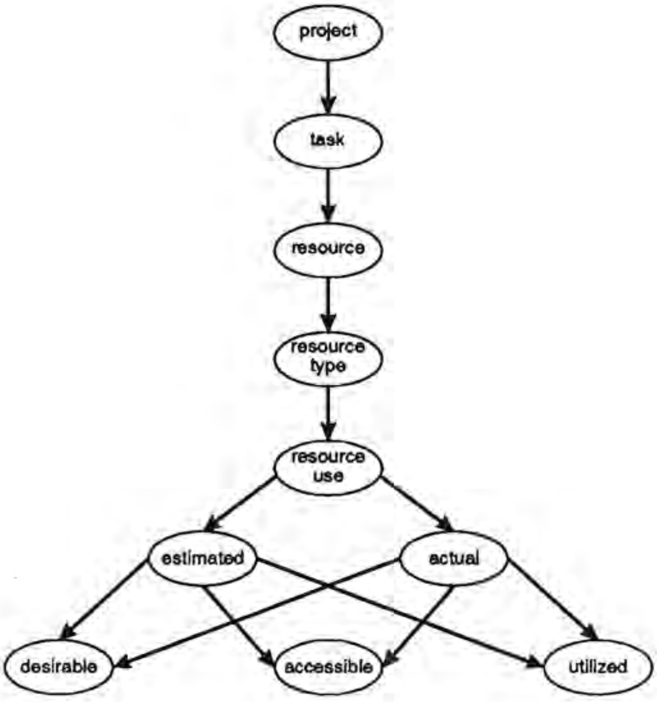

Figure 7.4 A model of resource data

The TAME process model in Figure 7.2, as well as the data model in figures 7.3 and 7.4, will be used to compare the TAME system with the other proposed information systems. At this point we conclude that TAME has a sound basis and holds a lot of promises. TAME claims it should be able to support maintenance work and contains a database with reusable components. The currently available publications are less specific about the development status of these features.

Noth's experience database

Noth (1987) describes an experience database for the support of software project management. The goal of the database is to provide data on completed projects to support the planning of similar projects. The database design is derived from an extensive analysis of the information requirements of the parties involved in the control of software engineering. Noth analyzed the requirements of management, administration, planning, control, engineering, quality control and configuration management.

Noth distinguishes between a database and a methods base. The database is divided into a text, a data and a knowledge base. The data are accumulated from the projects through data collection after the project is finished. The database can be accessed through an information retrieval system. The methods bank contains methods and techniques for operating on the data in the database. The methods bank and information retrieval system are activated by information requests from the organization. Examples of methods and techniques in the methods bank are: statistical analysis tools, risk analysis tools and estimation models.

Noth describes a project in what he calls a logical data model. He distinguishes between products, resources and environment as object classes. The data model consists of a list of objects which are in Table 7.2.

Table 7.2 Objects and attributes of Noth's database

PRODUCTS		
description and classification	risk estimation and evaluation	
size	change requests and failures	
resource use	problem reports	
cost	reference to persons and functions involved	
lead time	reference to related products.	
RESOURCES		
- PERSONAL:	- TEAM:	- HARDWARE/SOFTWARE:
name	name	name
profile	profile members	classification
function in project	fluctuation	function
department	absence due to illness	applicability
problem involvement	productivity	supplier
productivity	product involvement	reference to buy decision
absence due to illness	problem involvement.	problem involvement.
product involvement		
reference to job description.		
ORGANIZATION		
place within organization	problem involvement	
function	experience	
product involvement	political interest	
	preferences.	

Among the product attributes that are distinguished are the usual ones such as size, resource use, cost and lead time. Noth also considers risk, change requests, failures and problem reports as objects in his experience base. He does not only intend to capture development experience, but also experiences with the use of the product that has been developed.

Within the object class resources a distinction is made between the objects personnel, software and hardware. The resource data attract the attention because they can easily be abused for personal appraisal. Examples are data such as absence due to illness, personal productivity and the involvement with problems in the software are examples.

The last class of objects is the environment. It can be divided into the environment within the company and outside the company. Both can be subdivided into organizations, departments and persons. Some of the environmental attributes, such as the function and experience can be measured in objective terms. Some others are highly subjective, such as political interest and preferences.

Noth's experience database limits itself to the collection of experience. It does not support tracking of ongoing development of maintenance activities and it does not support the reuse of software. The experience data are mainly used for planning purposes.

7.3 Comparison of the systems and assessment with respect to the software factory

The information systems described in the previous section will be compared with one another. Next, we will assess the value of an information system for software engineering control as a reference for this activity in a software factory.

Comparison

The three systems will be compared with respect to three criteria. These criteria have been chosen because they typify the information systems and the available descriptions allow us to compare the systems with respect to them. The criteria are the goal of the information systems, their data model and their maturity.

<u>Goal</u>

A feature common to the three systems is that they all aim to capture software engineering data to support software engineering control. All the systems capture both product and process data. However, the software engineering control tasks that are supported and, as a consequence, the kind of data collected differs considerably. PMDB is intended to provide a project library that captures the data that is generated during a software engineering project. It is aimed at supporting the control of ongoing projects. Support of the collection and retention of history is mentioned as an issue for future research. TAME is intended to support the control of current construction projects by analyzing the current project. It also aims to supply experience from its experience base to support the control of future projects. Noth's database does not support tracking of current projects. It mainly supports the planning of future projects, by supplying of experience data.

We conclude that the top level goal of the systems may be the same. As soon as the goals become more specific, considerable differences become clear.

Data models

A comparison of the data models shows that PMDB and Noth's data model comprise the entire system, while the TAME data model only encompasses the resource data - a limited part of the TAME system. Other TAME data models are not yet available. This makes comparison difficult. What is clear, however, is that all the systems take a project as the starting point for their data model. PMDB is aimed at large software projects. Noth's experience database is aimed at projects as well. The TAME system does not specifically aim at projects and can also be tailored to other engineering activities. The TAME resource model, however, takes the project as its starting point. Another similarity is that the 'heart' of a data model, as identified by the TAME resource model, can be identified in all the systems. All three data models show a project that is broken down into activities which consume resources and produce a product.

Maturity

The maturity of the three systems is the last criterion that will be considered in the comparison. Practical experiences with implementing the proposed information systems have not yet been published. The progress of the TAME system is regularly reported upon (Basili 1988, 1989, 1990 and Rombach 1990). A first prototype has been developed and more will become available in the years to come. We are not aware of any reports on the progress of PMDB or Noth's experience database.

The comparison of the three information systems has yielded a number of similarities and differences. The most important of these are summarized in Table 7.3.

Table 7.3 Similarities and differences between the systems discussed

SIMILARITIES
Capture software engineering experience by means of data collection
Collect data on software product and process
Software projects as a starting point
'Heart' of the data model
DIFFERENCES
Support of control versus planning support
The use of product data
Representation of the systems
Maturity

Assessment
The information systems will be assessed with regard to some requirements which are derived from the control characteristics of a software factory, as given in section 5.7. The most important characteristics will be repeated here. Control in the software factory was characterized in terms of goal, production control situation and organization. The goal of the software factory is to minimize cost and maximize the functionality and quality of the product range that is being, and will be, developed. The software factory has limited its product range. The department is held responsible for the products it has developed in the past, i.e. it has to control the maintenance on products that have been developed in the past. The production control situation can be characterized as engineer to order. This means that specifications of products are agreed upon in particular customer orders. Customers orders must, however, fit in with the product ranges defined. Software components are engineered for future reuse, both proactively and reactively. The organization is made up of several production units. Development, maintenance and development of reusable components for a product range are done within the same production unit.

The description of the control system allows us to derive some minimal requirements that a software factory information system will have to meet. The minimal requirements allow an assessment to be made. They are:
- Support of the control of current projects,
- Support of the planning of future projects,
- Support of the control of maintenance activities, and
- Support of the reuse of software components.

The three information systems discussed will be assessed with respect to these minimal requirements.

PMDB

PMDB aims to support data collection and retention in ongoing software projects. It is not clear how reference information for future software development efforts is provided. A database with software components which are available for potential future reuse is not considered as part of PMDB. Maintenance activities are not distinguished as separate activities. We conclude that PMDB is not yet capable of supporting software engineering control in a software factory.

TAME

TAME supports both the control of current and the planning of future projects. Different kinds of reference data are acknowledged and TAME is tailorable to maintenance activities. Provision will be made for a library with reusable components. TAME is very promising, thanks to its sound basis and its tailorability. It is difficult to assess TAME with respect to the software factory in its current state of development, since so far only a first prototype has been reported upon. At this time, we cannot be sure that TAME is able to support software engineering control in a software factory.

Noth's experience database

The experience base captures data on ongoing projects to support the planning of future projects. It does not support the control of ongoing projects. A database with reusable components is not considered part of the experience database. We conclude that Noth's experience base is not suitable as an information system for a software factory.

The three systems have been discussed as potential reference systems for an information system for software engineering control in the software factory. They provide useful insights to those involved in software engineering control. The ideas behind the systems are valuable and parts of the systems may be useable. The fact that the systems do not (yet) meet all the requirements and that the systems are still being researched, has led us to incorporate another reference system in the discussion. It is an information system that supports engineer-to-order production. So far we have used information systems for software engineering control as reference systems for the software factory. In the next section we will try to use information systems for production control as a reference.

7.4 Information systems in production control

Section 5.4 discussed the control of multiproduct software engineering. It distinguished between the following production control situations: engineer-to-order, make-to-order, assemble-to-order and make-to-stock. We argued that the software factory shows most similarities with engineer-to-order production. This section will describe the outlines of an information system for engineer-to-order production control. The outlines described will act as a reference for an information system for a software factory.

The outlines will be described in terms of the architecture and a data model. The architecture is described in (Bertrand 1990). The description consists of four concentric circles which are given in Figure 7.5.

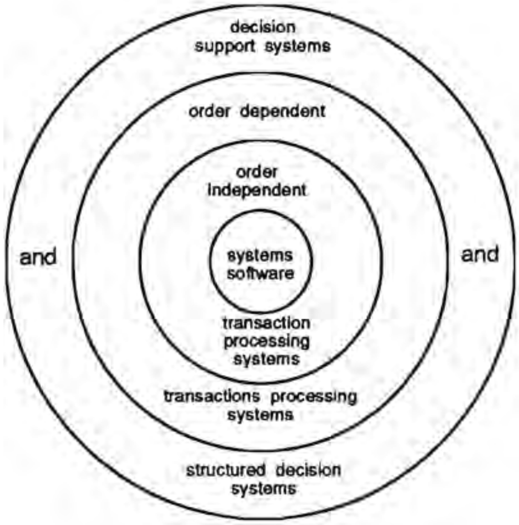
decision support systems
order dependent
order independent
and
systems software
and
transaction processing systems
transactions processing systems
structured decision systems

Figure 7.5 The four concentric circles of production control software (Bertrand 1990, page 117)

The inner circle contains application-independent software which enables application programs to run. It consists of an operating system, a data base management system, an input/output monitor and a query language facility. The second layer consists of the state-independent transaction processing systems. It is often referred to as order-independent data. The terms order-dependent and order-independent data will be used in this book because state dependent and state-independent are reserved words in software engineering. The second layer constitutes all kinds of recording data relating to products, technology, equipment and personnel that are independent of the flow of orders and the flow of materials. For example, the order-independent data contains the description of reference products and resources.

The third layer represents order-dependent transaction processing. The application software which monitors the state and the state transitions of the materials and orders is located in this layer. For example, it contains planning data. It also tracks the status of orders which may start as prospects and be transformed later into confirmed orders, shipped orders and finished orders. The fourth and outer layer is the layer of the various decision support systems. They should support the human decision-maker and may contain, for example, project and activity scheduling systems. The discussion of the information systems will focus on the two middle circles: the order-dependent and order-independent data. They represent the skeleton of information systems for production control (Bertrand 1981) and are regarded as essential for the fourth layer (Bertrand 1990).

The contents of the concentric layers differ considerably for the production control situations distinguished in section 5.4. The information system in a make to stock environment is mainly a tool for planning and control. Information systems for customer-order-driven production become a tool for engineering, in addition to their decreased role in planning and control. The architecture of customer-order-driven production will be described in more detail. In terms of the Process-Control-Information model: the Control of a software factory shows similarities with engineer-to-order production, as was argued in section 5.4. Therefore, the design of an Information system for a software factory can benefit from a study of an Information system for engineer-to-order production.

The study of the analogy and differences will focus on a model of the architecture of the information system, as well as on data models of the information system. The architecture of an information system for customer order driven production is given in Figure 7.6. The circles of the architecture will be discussed from the inside outwards. The inner circle is not specific to customer-order-driven production. It contains software which enables the application software to run, as well as a calender and certain parameters that must be initialized before the software is used.

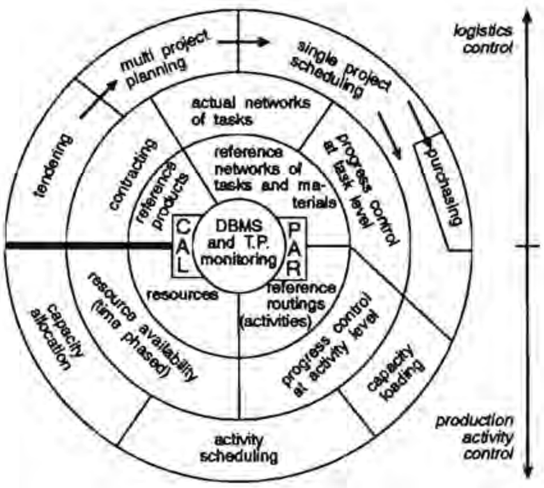
logistics control
multi project planning
single project scheduling
actual networks of tasks
reference networks of tasks and materials
tendering
contracting
reference products
progress control at task level
purchasing
DBMS and T.P. monitoring
C A L
P A R
resources
reference routings (activities)
resource availability (time phased)
capacity allocation
progress control at activity level
capacity loading
activity scheduling
production activity control

Figure 7.6 Architecture of an information system for customer order driven production (Bertrand 1990, page 162)

The order-independent data are used as an aid in engineering. They are mainly used as a reference. The references include reference products, reference networks of tasks and materials as well as reference routings. A reference product is a product which shows a certain degree of similarity with the product to be engineered and manufactured. In software terms: a reference product is a piece of software that can possibly be reused. A reference network describes how similar products have been produced in the past; it describes the subsequent steps to be taken. A reference routing describes the production operations in detail. The reference routing includes the capacity unit, the set-up time, the run time and the waiting time. Information on the resources completes the order-independent data.

The order-dependent data are represented by the next concentric circle in Figure 7.6. The order-dependent data are connected to customer orders. The order-dependent data will not be known completely at the start of the production. The bill of material and the routings will become known during execution. The customer will gradually fill in the blank spots that have been left open at the start of the engineering of the system. The order-dependent data consist of contracting data, actual networks of tasks and data to control the progress. Progress control is done at an aggregate level called the task level and at a more detailed level called the activity level. Data on the availability of resources in time constitute the last group of data represented as order-dependent.

The outer circle in the architecture of Figure 7.6 represents the decision support systems. The systems distinguished support the tendering of new orders, multi and single project planning, capacity loading, activity scheduling and capacity allocation.

Data structure diagrams constitute the skeleton of a production control information system (Bertrand 1990). Data structure diagrams will be given for the order-dependent and the order-independent data of the information system. The pictorial notation for representing the data structures is the one used by Martin (1987). It is explained in Figure 7.7.

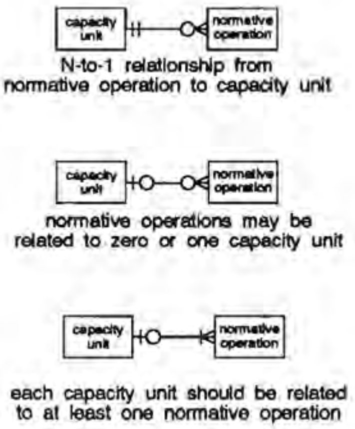
capacity unit
normative operation
N-to-1 relationship from normative operation to capacity unit
capacity unit
normative operation
normative operations may be related to zero or one capacity unit
capacity unit
normative operation
each capacity unit should be related to at least one normative operation

Figure 7.7 The notation used.

The data structure diagram for the order-independent part is given in Figure 7.8.

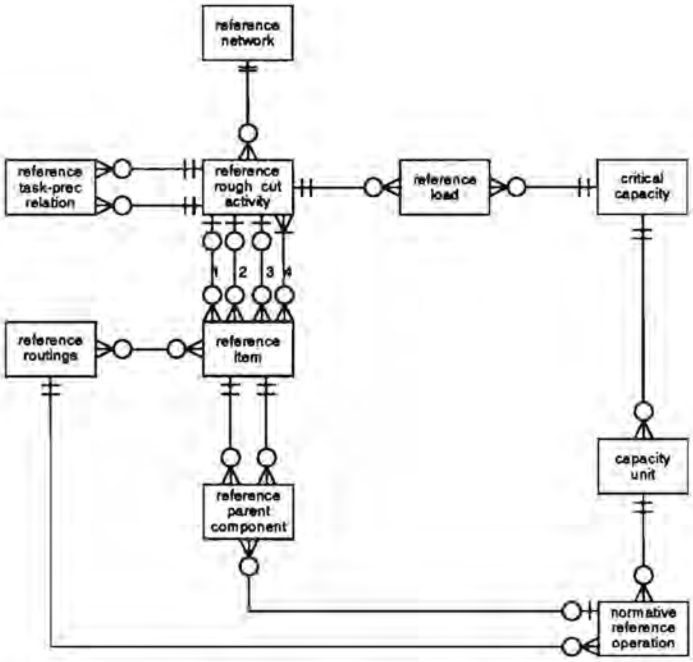
reference network
reference task-prec relation
reference rough cut activity
reference load
critical capacity
1
2
3
4
reference routings
reference item
capacity unit
reference parent component
normative reference operation

Figure 7.8 Data structure diagram of the order-independent data (Bertrand 1990, page 156)

The data structure diagram represents three levels of production control that have already been distinguished in section 5.6. They are: aggregate production planning, material coordination and production unit control. A rough cut capacity planning is made at the aggregate production planning level. The rough cut capacity planning uses estimates of the labour contents and lead times of networks of tasks. The material coordination level activities operate on detailed networks of activities, which may be described as detailed project plans. Detailed scheduling of work operations takes place at the shop floor control level. It is important to acknowledge the fact that at the different levels of production control, planning is done with different precision. Long term planning is done roughly, based on aggregate data while short term planning of known activities is detailed and precise. It should be obvious that different data are needed to plan at these levels.

The discussion of the entities distinguished in the data model in Figure 7.8 starts with the order-independent data. The description of the diagram begins with the reference network of activities. Reference networks of activities are stored to support the engineering and manufacturing if a product is needed that shows similarities with a product previously developed. A network consists of a number of reference activities. It can be related to several critical capacities by means of the entity type reference load. Critical capacities are related to a number of capacity units. An activity may have a precedence relation with other activities. An activity is connected to a reference item through four relationships. Three relationships indicate which items are supplied, prepared and consumed in the activity involved. The fourth relationship indicates in which activity an item is designed.

The reference parent component relation represents the bill of material. A bill of material for a parent is a list of components required for this parent item; it represents a list of parent component relationships. The bill of material is an important carrier of product information in production control. The remaining entity types are related to the detailed operations. The reference routing describes the operations of an item in detail. A reference routing consists of a list of normative reference operations. These may be used in more than one reference routing and consume capacity from a certain capacity unit. A description of the available resources completes the description of the order-independent data.

We now proceed to the order-dependent data. A data model in which the order-dependent data are added to the order-independent data is given in Figure 7.9. The left half of the picture is a copy of Figure 7.8.

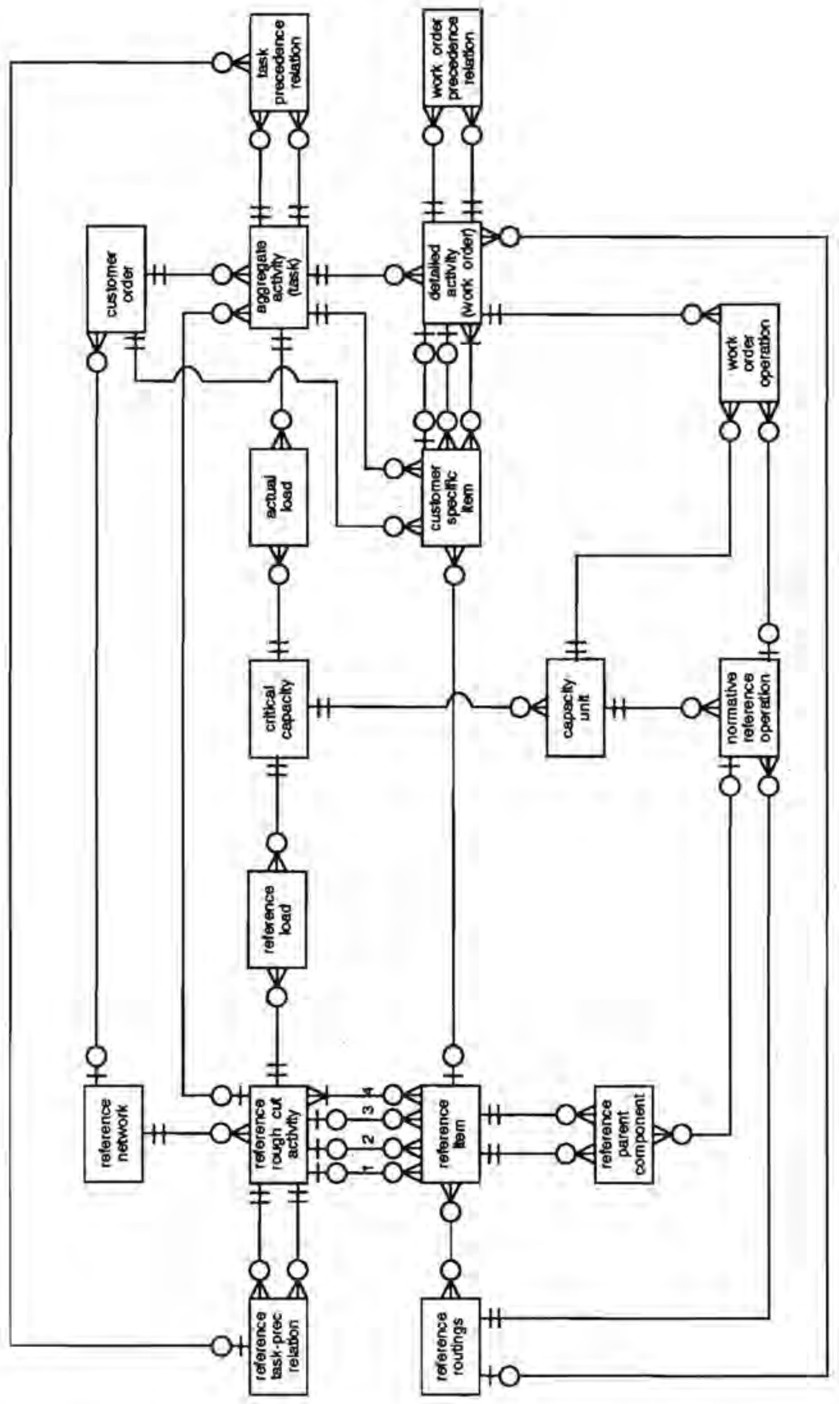
task precedence relation
work order precedence relation
customer order
aggregate activity (task)
detailed activity (work order)
work order operation
actual load
customer specific item
critical capacity
capacity unit
normative reference operation
reference load
reference network
reference rough cut activity
reference item
reference parent component
1
2
3
4
reference task-prec relation
reference routings

Figure 7.9 A data model for engineer-to-order production (Bertrand 1990, page 159)

The data structure diagram comprises both the order-independent data and the order-dependent data. The former are shown at the left of the figure, while the latter appear at the right. In general, it might be said, that an order-dependent equivalent of the order-independent entity types is added. A customer order that is entered into the system is usually related to a reference network. A customer order plays a central role in an information system for engineer-to-order production. An engineer-to-order company will only accept customer orders if the product fits into the defined product range. The customer order is related to one or more customer specific items, which can be derived from a reference item. The customer order consists of a number of reference tasks, which can be broken down into detailed activities or work orders. Both the task and the work orders are restricted by precedence relations. The work orders consist of a list of work order operations. The work order operations may be derived from a normative reference operation and consume capacity from a capacity unit. This concludes the discussion of the data model. The engineer-to-order information system will be assessed with respect to the software factory in the next section.

7.5 Assessment with respect to the software factory

This section consists of two parts. The first part compares the information systems for software engineering control with the information system for engineer-to-order production control. The second part assesses the information system for production control with respect to the software factory.

Comparison
The focus will be on the lessons software engineering control can learn from production control. The fact that software engineering actually does have lessons to learn is a question of maturity. Production control information systems are based on decades of experience in production control, while software engineering control has been practised for only a limited number of years. Three lessons will be discussed:
- The levels of planning distinguished
- The acknowledgement of order-independent data
- Different kinds of reference data.

<u>The levels of planning distinguished</u>
Production control information systems support planning at different aggregate levels. Three information systems have been proposed for the control of software development and were discussed in section 7.2. Two of them, TAME and PMDB, are aimed at supporting control of ongoing projects. They do not distinguish between

different levels of planning. The PMDB data model shows 170 relationships between 31 objects but only provides a relationship between resources and accountable tasks. It does not provide a relationship between resources and, for example, products, work breakdown structure items or software components. This means that work has to be planned in detail before it can be matched with the available resources. The TAME resource model only relates tasks to resources. Consequently, it is impossible to plan at more aggregate levels.

Planning at more aggregate levels is required for software engineering control. Chapters 3, 4 and 5 have shown that control of software engineering involves more than the execution of isolated projects. Three lines of work which use the same scarce resources have been distinguished. They are development, maintenance and development for reuse. One cannot plan these activities a detailed level only. Production control has taught us that control must be broken down over several levels in order to reduce control complexity. Software engineering will arrive at the same conclusion. At present, many software engineering departments only plan at the most detailed levels. They plan activities within the context of projects. A summary of current project plans will not allow to answer questions like such as:

- How many engineers will be available next month to start development of the next release?
- Which engineers can be moved to this project without consequences for release dates of other projects?
- Will we need more people or equipment within the next three months?

The need to answer questions as these increases as the number of engineers within a department grows and if development, maintenance and development for reuse are executed within the same department.

The acknowledgement of order-independent data

Another insight that can be gained is the distinction between order-independent and order-dependent data. The distinction shows the acknowledgement of the fact that there is a body of knowledge that is independent of the current activities, projects and products. The order-independent data do in fact represent the experience base of the organization and are used as an aid in engineering new products.

Software engineering organizations often lack an experience base of this type. This may be due to the fact that many software engineering organizations are capacity-selling companies. Such a company is often only responsible for the provision of resources and not for controlling the engineering effort. It does not need the experience data as a reference. In fact order-independent data are absent in information systems for capacity selling production control (Bertrand 1990).

The three information systems discussed in section 7.2 try to accommodate an organization with an experience base. Noth's experience database has been especially developed for that purpose and aims to collect what might be called order-independent data. The purpose of PMDB is to provide data on ongoing projects, i.e. order-dependent data. It studies the possibilities for supporting the collection and retention of historical data. TAME acknowledges that order-dependent and order-independent data are related. Its process model explicitly distinguishes the data on ongoing projects from the data in the experience base. In other words, TAME captures both order-independent and order-dependent data.

Different kinds of reference data

Production control puts a lot of emphasis on references. It seizes every opportunity to derive a product, a network or a routing from a reference. The repetitive character of manufacturing has taught production control to exploit experience. Software engineering gave the impression that every activity was new and unique and that it was therefore not possible to exploit experience. The benefits of exploiting experience in a software environment can be even greater than the benefits in industry. Software has the potential for reuse that allows a software product to be copied with negligible reproduction costs.

The order-independent data are divided over reference products, reference networks, reference routings and resources. Software products can be described as reference products. Reference networks may be used to capture software engineering experience. A new engineering effort may benefit from the knowledge captured in, for example, the time required to learn how to operate a new tool or the sequence of activities that has to be applied when a new process model is used. The same holds for reference routings. As already described, reference routings indicate the average time spent to execute the subsequent operations involved in manufacturing a product. The time stated in a reference routing for manufacturing a hardware item may be regarded as a norm. The time in a reference routing for the development of software should be considered as an indication of the expected time required for this purpose. The non-repetitive character of software engineering causes the difference in the use of the reference routing. Software engineering could certainly use this kind of reference for estimating the extent of future engineering efforts. The lack of references is believed to be a main reason for the estimation problems in software engineering (Heemstra 1989).

The information systems discussed in section 7.2 are all intended to supply reference data. Noth distinguishes between products, resources and environment data. Collection and retention of history is a research issue in PMDB. One of TAME's principles is that data collection should be tailored for an environment, because software engineering practices differ from place to place. The reference data to be collected, have still to be determined. TAME makes it clear, however that it is interested in more than just reference products. Basili (1989) points that it is not only products which can be reused but, for instance, experience with the application of processes, the use of development methods and the use of tools as well.

Assessment

The remainder of this section will assess the information system for production control with respect to the software factory. A production control information system supports the control of current projects through its order-dependent data. It also supports the planning of future projects through its order-independent data. Reference networks, activities and operations can be used as planning support. There are, however, some differences between software engineering control and engineer-to-order production control. One difference is that in the case of software, the product itself can be perceived as information and as part of the information system. The information system must provide for storage of software products. Another difference is that at least two kinds of products need to be distinguished in the information system because they have to be controlled differently. Products being used by a customer cannot be modified in the same way as products under development. Additional precautions will have to be taken, because the customer's interest in the product in use must be protected.

This relates to the support of maintenance activities. Maintenance of hardware products is usually not done by and within the engineering department. One reason for this is that the product that has to be maintained is at the client's site. An engineer-to-order information system generally does not make any provisions for maintenance orders. In the case of software, things are different. The product to be maintained is usually both at the client's site and at the supplier's site. The supplier usually retains a copy of the software product. Furthermore, maintenance can take up to 50 percent of the engineering resources. In the software factory we have specified, maintenance and development take place within the same production unit. It is obvious that the information system should make special provisions for maintenance orders.

Another minimal requirement is support for the reuse of software components. The production control information system provides for this by its reference products. It still has to be determined whether software products can be described in the same way as hardware products. It must be noted that product representation is a research topic in the field of production control. In the case of customer order driven production the representation of a product is particularly difficult, because parts of the product may only be specified during the project. There are similarities between the representation issue in production control and software representations which are a major research issue in the reuse of software, as already indicated.

From this assessment, we conclude that an information system for engineer-to-order production control fits in reasonably well with the software factory. This is not surprising because chapter 5 already concluded that software engineering control in the software factory showed similarities with engineer-to-order production control. According to the Process - Control - Information model, it could be expected that the information requirements would also show similarities. The information system for production control will be used as a reference system instead of the information systems for software engineering control. The choice is based on the assessments and the fact that the information systems for production control represent decades of experience. There is a lack of experience with the control of software engineering.

7.6 Summary and conclusions

This chapter has described and assessed a number of information systems which could be used as a reference for an information system for software engineering control. The description of multiproduct control in the software factory described in section 5.7 was used as a basis for the assessment. Firstly, three information systems proposed in the literature for the control of software engineering were described. These systems were PMDB, TAME and Noth's experience database. The goal of the Project Management Data Base (PMDB) from TRW is to provide an environment base that includes products, resources and plans while TAME from the University of Maryland aims to create a corporate experience base. Noth describes an experience base of completed projects that is intended to support the control of software engineering. The assessment of the systems with respect to the minimal information requirements of multiproduct control in the software factory showed that PMDB and Noth's experience database are too incomplete to act as useful reference systems. TAME may, however, be able to support control in the software factory to a great extent, thanks to its tailorability. The state of development of TAME and the limited descriptions that are available detract from its potential as a reference system. Another reference system was studied: an information system for engineer-to-order production, as described in Bertrand (1990). The architecture showed the distinction between order-dependent and order-independent data. The order-independent data can be perceived as the experience base of the organization. The order-independent data consist of reference products, reference routings and reference networks. The conceptual data model constitutes the skeleton of the information system.

The comparison of the 'software' and 'production' information system provided three lessons for software engineering control. Firstly, it showed that the production information systems support planning at different aggregate levels, unlike the information systems for software engineering control. The second lesson was the distinction between order-dependent and order-independent data. TAME is the only information system for software engineering control which distinguishes between both data on ongoing projects and an experience base. The third lesson to be learned is the distinction between different kinds of order-independent data or reference data. The production control information systems contains data on products it has developed in the past and processes it has employed, as well as the use of methods and tools.

We also assessed the usability of the production control information system to the software factory. The assessment with respect to the software factory showed that this system fitted in fairly well. A more detailed assessment also showed some misfits. We chose to adapt the information system for engineer-to-order production control in such a way that it becomes useful for the software factory. The main reasons to opt for the production control information system were the results of the assessment and its maturity as compared with the information systems for software engineering control.

8 A DATA MODEL FOR AN INFORMATION SYSTEM FOR MULTIPRODUCT CONTROL

8.1 Introduction

It is a well-known fact that software engineering and its control differ considerably from place to place. An information system that supports this control should therefore be adaptable to different circumstances. The goal of this chapter is not to find the outlines of an information system which supports all kinds of software engineering, but to provide a reference framework for an information system for a software factory. The characteristics of a specific software factory have already been described.

A data model for an information system for the control of a software factory will be derived from a data model for an information system for engineer-to-order production control. The overall architecture of an information system for production control consists of four concentric cycles. The circles represent systems software, order-independent data, order-dependent data and decision support systems. This chapter focuses on the two middle circles of the architecture: order-independent data and dependent data. As already mentioned, they represent the skeleton of the information system. Consequently, this chapter will not address decision support systems such as planning tools, analysis tools or cost estimation models.

8.2 Additional requirements

This section will derive the requirements of an information system for a software factory from two sources. The first source is the description of the characteristics of software engineering control in a software factory. The second source is an information system for engineer-to-order production control, which will act as a reference system. The data model which was presented as Figure 7.9 will be used. The fact that, in this case, the product concerned is software results in a number of additional information requirements. The most important will be mentioned in this section. The data model itself will be presented in the next section.

The system to be controlled is an engineering organization engaged in development, maintenance and development for reuse. The products which are being currently developed and the products that have been developed in the past are considered part of the system. Customer's orders are also regarded as part of the system. Reference products and processes which constitute the experience base of the organization are included in the system as well. Customers themselves, and the actual use of the software products do not come within the system's boundaries.

The fact that the product concerned is software affects the data model in the following five ways:

1 Storage of software products
Information systems for engineer-to-order production contain product information. They will record, for example, the bill of material and design drawings of a product. On top of that, in case of software the supplier will keep a copy of the actual software product. In the case of hardware products, maintenance is usually done at the client's site, whereas in the case of software it is often done at the supplier's site. An information system for a software factory will therefore need a facility for storing software products that are in use at clients' sites.

Software products in a software factory can pass through different stages. These are: under development, in use, and under maintenance. The information system should allow an explicit distinction to be made between the stages because products should be treated differently at each successive stage. For example: a product under development is subject to changes until it is released. After release it becomes a product in use. A product in use may not be changed. The status of a product in use changes into a product under maintenance if the engineering department and the client agree upon a maintenance change. After the changes have been made, the product is again released and may not be changed any more.

We propose to make the following distinction between the software products. The data model of the production control information system distinguishes between order-dependent and order-independent data. A product in use is order-independent from the control point of view. An item under development is subject to change and belongs to the order-dependent information. An item under maintenance is order-dependent as well. In this way, there is a clear distinction between the different software products. The transitions of a product from being order-dependent to order-independent and vice versa are also clear. This will be reflected in the data model of the information system.

2 Distinction of maintenance orders
Maintenance orders should be distinguished since they are different from development orders. We distinguished between corrective, adaptive and perfective maintenance orders. We will argue that corrective maintenance orders should be treated as a separate entity, whereas adaptive and perfective maintenance orders can

be treated as customer orders. A corrective maintenance order is an order to correct a fault that has been found in a software product in use. These orders must be distinguished from customer orders because they represent different information. A maintenance order usually consists of a description of the failure that has occurred, the product concerned, client data and possible other products and releases which might be affected by the fault that is found. It may very well fit on one sheet of paper. A customer order on the other hand includes a specification of the product required and is a far more extensively described order.

Adaptive and perfective maintenance orders can be dealt with as customer orders because the adaptation of the software needs to be formulated as a requirement. Functional or quality enhancements to products in use can also be treated as customer orders.

3 Distinction of component orders

The software factory engineers components for future reuse client-independently. A component order could be distinguished from a customer order because it is an order from within the organization. The attributes are therefore different from the attributes of a customer order. They do not, for example, contain extensive client or order data. We will not classify component orders as a separate entity because we consider the similarities between customer orders and component orders more important than the differences. We will refer to customer orders and presume that the engineering department itself acts as one of the customers.

4 Internal change requests

Software engineering often requires the adaptation of an order because things that have been specified at a higher level turn out to be impossible or undesirable at lower levels of development. The adaptations will be referred to as internal change requests to distinguish them from external change requests which result from changing customer requirements. Internal change requests are generated during engineering activities and are therefore considered part of the control system discussed. External change requests originate from outside the system boundaries and can be dealt with as enhanced customer orders, just as they are dealt with in production control. (An internal change request is not similar to the 'engineering change' that is known in production control. An engineering change is a transition from a specific way of manufacturing to another way (Bertrand 1990, page 130). It is a change that concerns the manufacturing process. An internal change request concerns the order, i.e. the product to be engineered.)

5 Superfluous relations
Some relations which were distinguished in the data model in section 7.4 are superfluous. This concerns the four relations between a reference item and a reference activity and, as a consequence, the relations between a specific work order and a specific item. A reference activity is connected to a reference item through four relationships in the production control information system. Three relationships indicate which items are prepared, manufactured and used for assembly in the reference activity concerned. The fourth relation indicates in which activity an item has been designed. The work preparation and manufacturing relation are superfluous for software engineering control. Reproduction of software involves copying. It is superfluous to distinguish between a manufacturing and a work preparation relation in the data model. It is sufficient to know in which activity an item was designed and in which activity an item is reused. A software reference item and a software engineering reference activity are connected via only two relationships: the 'engineered by' and 'reused in' relationship.

8.3 A data model

A data model for an information for software engineering control in a software factory is presented in this section. It is an enhanced version of the data model of an information system for engineer-to-order production, which was presented in Figure 7.9. The outlines of the data model are presented in two ways in Figures 8.1 and 8.2 to facilitate the understanding of Figure 8.3, that contains the complete data model. Figure 8.1 shows that the data model consists of an order dependent and an order independent part. It further shows that three levels of aggregation can be distinguished: the aggregate, the intermediate and the detailed level. They resemble the three levels of aggregation distinguished in production control. The marked entities are new in comparison to the data model for engineer to order production. The six new entities are: corrective maintenance order, maintenance order reference network, product in use as well as parent-component, specific product in use and internal change request. They will be discussed in more detail later on.

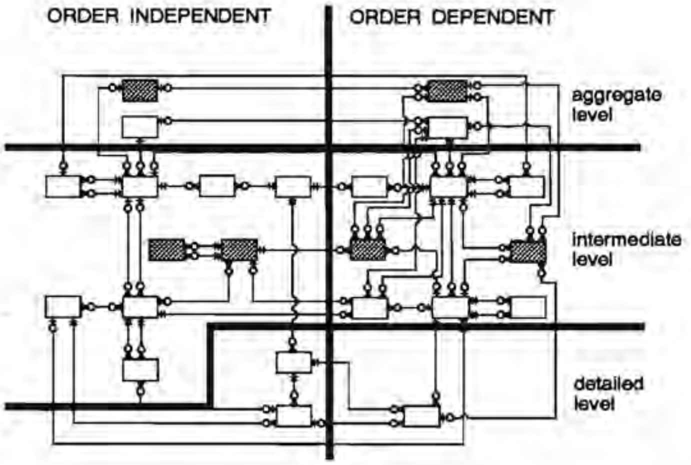

Figure 8.1 Three levels of aggregation

Figure 8.2 shows the data model from another point of view. It distinguishes between order data, product data, planning data and reference data.

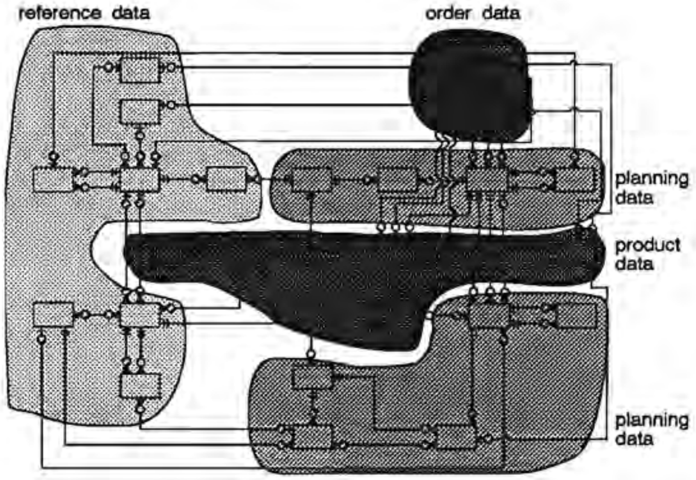

Figure 8.2 Order data, product data, planning data and reference data.

Figure 8.3 shows the data model for software engineering control in a software factory.

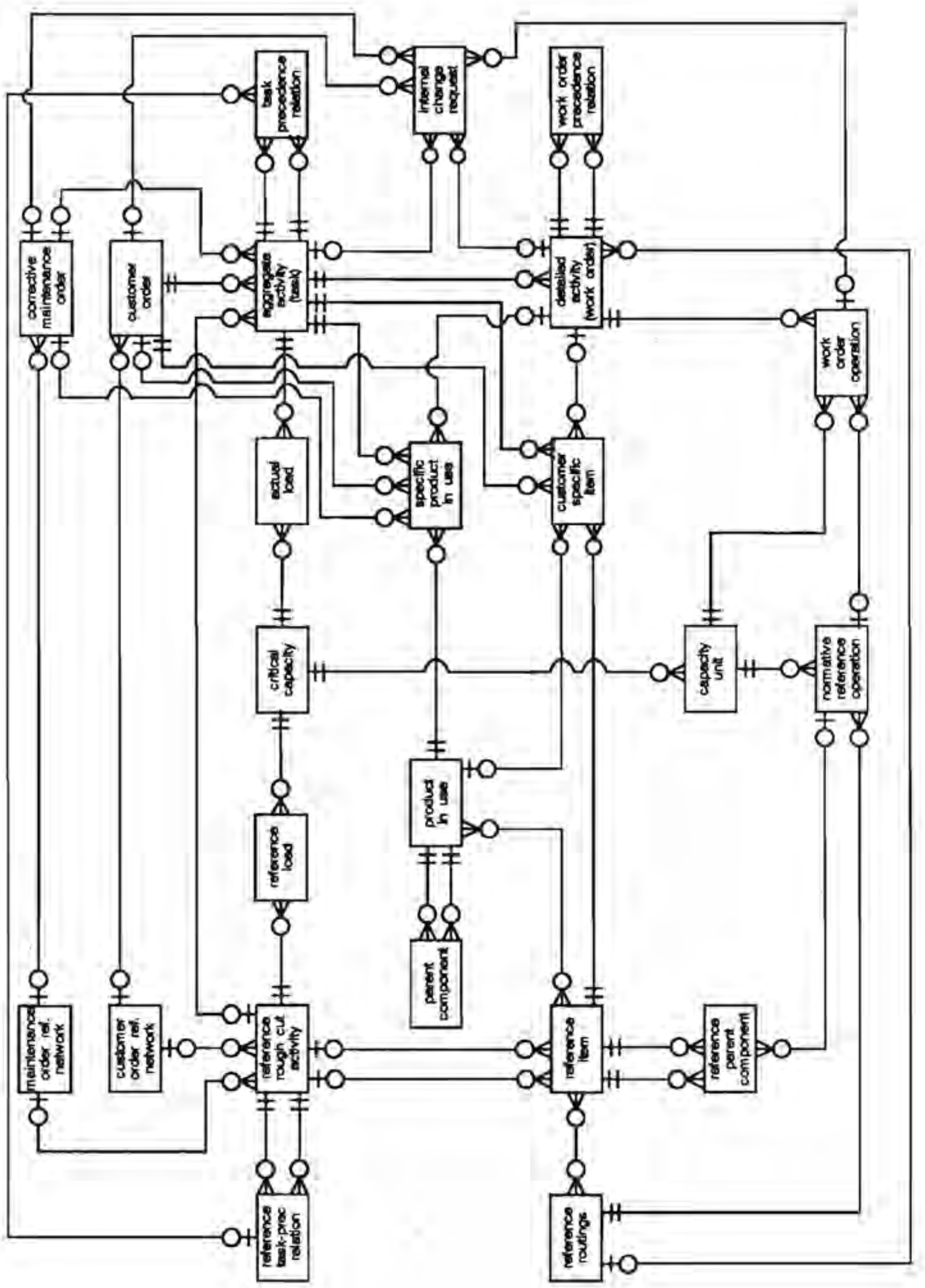
corrective maintenance order
customer order
task precedence relation
internal change request
work order precedence relation
aggregate activity (task)
detailed activity (work order)
work order operation
actual load
specific product in use
customer specific item
critical capacity
capacity unit
normative reference operation
product in use
reference load
parent component
maintenance order ref. network
customer order ref. network
reference rough cut activity
reference item
reference parent component
reference task-prec relation
reference routings

Figure 8.3 A data model for an information system for a software factory

The main difference between the 'production' data model and the 'software' data model lies in the description of orders and products. The description of the resources is largely similar. The discussion of the data model starts at the top of the order-dependent part of the data model. Two kind of orders are distinguished: customer orders and corrective maintenance orders. They are distinguished for the reasons already discussed. The orders are related to aggregate activities because they can both take up a substantial share of the available resources. Maintenance orders are known to make considerable demands on the available resources. If corrective maintenance orders consume only a small amount of these resources they can be neglected in aggregate planning. The relation to aggregate activities is superfluous in that case. Both kinds of orders are related to their reference networks.

The data model shows that an aggregate activity is broken down into detailed work orders. Both aggregate activities and work orders can be represented in a network. A work order is further broken down into work order operations which are related to a capacity unit. A customer order is related to a specific item under development or a specific product in use. The former applies if it concerns the first release of a product and the latter is the case if it concerns a functional or quality enhancement to a product that has been developed before and is in actual use. A corrective maintenance order is related to a specific product in use. The relation between the product in use and the detailed work order is a 'maintained by' relation. One entity remains to be clarified in the order-dependent part of the data model: the internal change request. As explained, software engineering brings along change requests that are generated during engineering work, i.e. during aggregate activities, detailed work orders or work order operations.

The discussion of the order-independent part of the data model starts with the reference networks. These networks are broken down into in rough-cut activities that are related via reference loads to critical capacities. A reference rough-cut activity is related to a reference item via an 'engineered by' and 'reused in' relationship. The engineering prescriptions are given in a reference routing, which is broken down into a number of reference operations.

The order-independent part of the data model shows some new entities. The first one is the maintenance order reference network, that is distinguished from the customer order reference network. Another one is the entity product in use. It represents the products that are in use at customers' sites. They are order-independent and may not be changed. The bill of material presents the parent/component relations. A product in use can be composed of a number of reference items. The data model for engineer-to-order production shows only one relationship between a reference item and a customer-specific item. The data model for the software factory shows three relations between a product in use as well as reference items, on the one hand, and specific products in use as well as specific products under development, on the other hand. The three relations will be dealt with in succession.

1 The relation between a reference item and a customer-specific item. This relation is identical to the one in the original data model. It is a 'specified item' relation. A reference item is usually specified in order to use it in the development of a customer order. A reference item can also become an item under development once it is decided that it needs additional development. A specific item can become a reference item once it is released as a reusable component.

2 Once the development of a customer-specific item is completed, the item is released and becomes a product in use; it becomes order-independent from the control point of view. The release of the software involves the generation of the software product. Generation of large software products requires considerable effort and computer resources. The production environment that generates a software product should be captured, in order to be able to regenerate the product at a later point in time. We do not pay special attention to the generation issue, due to our primary interest in the control of software engineering activities. Therefore entities such as 'production environment' are not included in the data model.

   A product can also become an item under development if it is decided that a product in use should be made available for future reuse. This is what has been called reactive development for reuse in section 5.5.

3 A product in use can become a specific product in use when it needs to be maintained or enhanced. Once the maintenance or enhancement is completed, the product is released and becomes a product in use again.

8.4 A description of some entities.

In this section the differences between the data model for the information systems for production control and software engineering control will be discussed in greater detail. Entities that are new or whose meaning has been significantly altered compared with production control will be described. These are the:

- Customer order
- Corrective maintenance order
- Specific product in use

- Reference product
- Parent/component relation
- Reference routing and normative reference operation.

Customer order

A customer order contains the customer requirements in terms of quality, time and money. Quality is usually the most difficult item to specify. A starting point for a product based quality definition is given by the 'IEEE guide to software requirements specifications' (IEEE 1983). The outline of a software requirements specification is given in Table 8.1.

Table 8.1 Outlines of a software requirements specification (IEEE 1984)

1 Introduction	3 Specific requirements
1.1 purpose	3.1 functional requirements
1.2 scope	3.1.1 Functional requirement 1
1.3 definitions, acronyms and abbreviations	3.1.1.1 introduction
1.4 references	3.1.1.2 inputs
1.5 overview	3.1.1.3 processing
	3.1.1.4 outputs
2 General description	3.1.2 Functional requirement 2
2.1 product perspective	...
2.2 product functions	3.2 External interfaces
2.3 user characteristics	3.2.1 User interfaces
2.4 general constraints	3.2.2 Hardware interfaces
2.5 assumptions and dependencies	3.2.3 Software interfaces
	3.2.4 Communications interfaces
	3.3 Performance requirements
	3.4 Design constraints
	3.4.1 standards compliance
	3.4.2 hardware limitations
	...
	3.5 Attributes
	3.5.1 security
	3.5.2 maintainability
	...
	3.6 Other requirements
	3.6.1 database
	3.6.2 operations
	...

The introduction should specify the audience, delineate the purpose and define the scope of the software requirement specification. The general description should discuss the general factors that affect the product and its requirements. The specific requirements should contain all the details needed to create a design. The specific requirements specification is drawn up according to a product-based quality definition. It pays attention to both functional and quality requirements. The quality attributes are similar to those discussed in section 4.2. The IEEE standard shows clearly that a specification should involve more than just a list of functional requirements. The IEEE standard shows that the entity-type customer order represents quite an extensive document. Nevertheless, it should be perceived as a minimum reference network. The adjective 'minimum' is used to point out that better references will usually be available. For example, a description of a similar product that has been developed in the past and was specified according to the IEEE standard. Such a reference not only shows how a product should be specified, but may also contain pointers to reference items that can be reused.

An order for the enhancement of an existing product was also termed a customer order in section 8.2. An enhancement to an existing product can be defined by specifying the difference between the existing and the required system.

Corrective maintenance order
A corrective maintenance order indicates a failure that has occurred or a fault that is found. It contains different information from a customer order. A typical maintenance report should at least have the attributes that are given in Table 8.2.

Table 8.2 Typical attributes for the entity corrective maintenance order

Order data (name, number)
Client data (name)
Product data (name, release, software environment, hardware environment)
Failure that occurred
Severity of failure
Perceived fault
Other software products that might be affected

The corrective maintenance order contains attributes that identify the order, the client concerned and the product concerned. The product data should comprise information on the software and hardware environment in which the software product operates. This information should enable the failure that has occurred at the client's site to be reproduced. This is often a prerequisite for finding the fault. The typical corrective maintenance order further contains the client's description of the failure and his perception of the fault that caused it. Finally, the order describes other products that might possibly be affected by the same fault. The description shows that a corrective maintenance order differs significantly from a customer order, as described earlier in this section.

Specific product in use
The description of a specific product in use will contain the complete description of the order-independent product in use. This includes:
- The software itself,
- The software and hardware environment in which the product operates,
- The milestone documents such as the software requirement specification and design documents,
- Updates made to the product in the current enhancement activity. Obviously, the descriptions of the product should be updated for every enhancement made to a software product after its initial installation.

Details on the documentation of a software product can be found in the standard life cycle models as they are currently applied.

Reference product
Reference products must be described in such a way that they can be identified and retrieved for future reuse. The key problem in reuse according to Biggerstaff and Richter (1987) as well as Sikkel and Van Vliet (1988) is the representation of the software. Horowitz and Munson (1984) identify four problems that must be addressed to make the concept of reusable software reality. The problems are:
- Mechanisms for identifying components; how can one determine and specify components which are generally useful?
- A method for specifying components; how can one describe a component so that others can understand it?
- The form of the components; should components be described in natural language, design language or programming language?
- Cataloguing the component; how should a component be catalogued so that it can be retrieved easily?

All four problems relate to the representation of software. The second and fourth subjects address the search problem: how can a component that is useful for future reuse be identified and retrieved? It should be clear to an engineer that identification and retrieval, followed by reworking, involves less effort than development from scratch. Many engineers will follow the usual habit and develop a new component from scratch if this is not perfectly clear. The identification and retrieval problem is one of the main obstacles to be overcome. Locating and retrieving an item from a large collection is certainly not unique to software engineering. Locating and retrieving an item from a large collection is a problem in, for example hardware engineering, all kinds of libraries and patent registration. One way to enable location and retrieval is classification. The most advanced work on the classification of software products has been done by Prieto Diaz (1987, 1990). His work will be discussed as an example of how the identification and retrieval problem can be overcome.

The goal of Prieto Diaz's research is to provide an environment that helps to locate components and estimate the adaptability and conversion effort. Prieto Diaz used a faceted classification scheme. The faceted method is often used in library science. It relies on building up or synthesizing from the subject statements of particular documents. The other, frequently used classification scheme is enumerative. It divides a universe of knowledge into successively narrower classes. The faceted scheme is chosen by Prieto Diaz to classify software components because collections of software components are large and constantly growing. An enumerative classification is less suitable for such an environment because the growing universe of knowledge will make it necessary to redefine the classes time after time.

The scheme classifies software components by its functionality and its environment. Functionality is described by its function, objects and medium. Function is a synonym for the action performed. The objects of the function can, for example, be arrays, expressions and files. The medium is the locales where the action is executed, such as tables, files or trees (Prieto Diaz 1987). The environment is characterized by the system type, the functional area and the setting. System types refer to functionally identifiable, application independent modules, such as database management. Functional areas describe application-independent activities, such as cost control. The setting characterizes the environment in which the application is exercised. A setting can be 'advertising' or a 'car dealer'. The six facets that characterize software modules and some examples are given in Table 8.3.

Table 8.3 The faceted classification schedule (Prieto Diaz 1987)

FACET	EXAMPLE
Function	add, create, exchange, join
Objects	arrays, characters, files
Medium	files, screen, table
System type	database management, file handler, line editor
Functional area	batch job control, CAD, bookkeeping
Setting	advertising, computer store, car dealer

Prieto Diaz has integrated this classification scheme into a prototype library system. A thesaurus is provided by the prototype to avoid duplicate and ambiguous descriptions of similar components. The concept of conceptual closeness was introduced to be able to find similar components in case an identical component is not available. Another feature of the prototype library system is a mechanism that evaluates the components with respect to the estimated reuse effort. It could be characterized as a reuse cost estimation model. Five attributes were selected as indicators of the reuse effort. The attributes and their metrics are given in Table 8.4.

Table 8.4 Reuse attributes and their metrics (Prieto Diaz 1987, page 15)

ATTRIBUTE	METRIC
Program size	Lines of code
Program structure	Number of modules, number of links and cyclomatic complexity
Program documentation	Subjective overall rating
Programming language	Relative language closeness
Reuser experience	Proficiency levels in two areas: programming language and domain of application

The attributes program size and structure are measured in objective terms. The other attributes are measured in more subjective terms. The quality of the documentation is rated subjectively on a scale from one to ten. The language is considered as a variable because it affects the size of a component. The experience of the reuser is rated since it affects the effort that will be required to reuse a component. The evaluation mechanism ranks the components with respect to the estimated effort.

The faceted classification scheme, the conceptual closeness model and the evaluation mechanism have been integrated in a prototype. A system analyst was assigned to classify the components. This is an example of the organizational support that is required. A total of six program support functions had to be created to develop a 'reuse culture' in the organization. They are:

- Management support to provide initiative and funding.
- An accessible, densely populated, fully supported, easy to use library system.
- An identification and qualification group responsible for the contents of the library and procurement of new modules.
- A maintenance group that maintains reusable components
- A development group that creates reusable components
- A reuse support group (Prieto Diaz 1990, page 302).

A prototype was developed and used in the organization. During its first year 38 percent of the items in the library were reused. A reuse factor of 14 percent was achieved. The reuse factor is defined as the number of lines reused, divided by the total number of lines of code produced by the organization. The estimated savings were 1.5 million dollars. The goal of the organization was a 50 percent reuse factor by the end of the fifth year.

Parent/component relation
The composition of a product is represented in what is called a 'bill of material' in production control. A product is represented as a parent with a number of components. A bill of material for a parent item is a list of its components. Software products can be represented in the same way. A software bill of material should at least contain the following attributes:
- Parent identification,
- Component identification,
- Starting effectivity date,
- Ending effectivity date.

The parent and component identification enable the two related software items to be identified. The starting and ending effectivity dates limit the time for which the relation is considered to be effective. An item that occurs as a parent in one bill of material can act as a component in another bill of material. In this way, multilevel bills of material can be defined.

An engineer-to-order environment is characterized by the fact that orders are largely customer specific. Three types of product data can be distinguished (Veen 1990): standard data, customer-specific data and historical reference data. The order-independent bill of material can contain the standard data and the historical reference data. The customer-specific data become available during engineering and are stored as attributes of the order-dependent entity 'customer specific order under development'. An extensive study of bills of material in different production control environments can be found in (Veen 1990).

The number of parent/component relations is further increased because software products exist in many versions and releases, as do software components. The fact that software appears to be more easily adaptable is one of the reasons for the large number of versions of software products (Brooks 1987). Every adaptation to an order-independent software item results in a new version of a software item, as does an adaptation to the hardware or software environment of a software product. In terms of the data model: every release of an order-dependent product under development, product in use or component results in a new version of a software product.

As stated in section 5.4, the number of specific end products is increasing. Software configuration management has become an important issue, both in theory and practice. We will not discuss the issue in detail but we will introduce one attempt to address the configuration problem. All end products might be described as parents with their components. This description would be highly redundant because the only difference between two versions may be one or a few components. One way to address this problem is to use a generic bill of material that allows a lot of variants to be described with a limited number of data. The generic bill of material is described in (Bertrand 1990, Veen 1991, Hegge 1990). The bill of material is split into a generic and a specific bill of material. The generic bill of material describes the components each product of a family contains. The specific bill of material describes the choices that should be made to configure a specific product out of its components. Configuration using the generic bill of material is currently supported by a prototype. The approach looks valuable for software with its high number of similar products. Future research will be required in order to reveal its applicability to software.

Reference routings

A routing in production control is a list of the normative operations required for manufacturing a product out of its components. A routing contains data such as the capacity units whose resources are used, the sequence of operations, the set-up time and the run time. The set-up and run time are considered normative times. The throughput time of an item can be computed on the basis of the set-up and run time. In software terms, a routing can be considered as a list of reference engineering activities that have to be executed to develop a software product. A routing will be an aid in engineering and planning instead of a basis for computing the throughput time. A general description of a routing is given in an engineering method. It can consist of the activities specify, design, code and test. This kind of routing gives no additional information on top of the information that is already available in an engineering method. Additional information that can make the reference routing worthwhile is, for example, reference times. These can be used to make an analogy estimation (Boehm 1981) of the required development time. A routing of a software product could look like the example given in Table 8.5. This is the routing for a very simple reference product called SW1 consisting of reference module 1 and reference module 2.

Table 8.5 Example of a software reference routing

Item	Reference item	Sequence	capacity unit	reference effort
SW1	x1	10	integration	2 hours
		20	integration test	10 hours
Module 1	x11	10	specification	20 hours
		20	design	40 hours
		30	coding	15 hours
		40	test	8 hours
Module 2	t23	10	specification	5 hours
		20	design	20 hours
		30	code	10 hours
		40	test	12 hours

The routing gives the capacity units from which the items require resources. This routing gives one reference module per item, but it is also possible to use several references to arrive at an estimate.

Normative reference operation
A normative operation can be perceived as one line out of a routing. The normative reference operation gives more detailed information for the operation concerned. The reference operation 'integration test' of the product SW1, as given in Table 8.5 will be discussed by way of example. The reference operation integration test can contain information about the document descriptions that have to be available before the test can start. It can also give directions for the kind of tests that have to be performed for the integration of these types of modules. In some cases, the reference operation may even specify which tests have to be executed. In addition to these directions, reference efforts and lead times may be given in the reference operation. A reference operation's value lies in the directions for carrying out the operation and the fact that it provides reference times.

8.5 Use of the proposed data model

This section discusses the possible use of the proposed data model. We envisage two ways to use it: as a reference model for development or as a test for proposed information systems.

A reference for an information system in a software factory
This chapter has provided a data model for an information system for a software factory. A data model is an important result in the development of an information system according to a data-driven method. Such a model can be taken as a starting point in the development of an information system for the control of a software factory. This approach has several advantages. Firstly, a considerable amount of effort is saved because one does not have to start from scratch. Secondly, the experience that has been accumulated in production control over a number of decades will be reused.

This could be taken one step further. One could introduce an information system for engineer-to-order production control in a software engineering department and adapt it to some of the typical software engineering characteristics. The adaptation to the software factory involves serious reworking. Changes to the data model are required, as was shown in section 8.3. Consequently, the following actions are necessary (Bertrand 1990):
- Change of the physical data model
- Recreation of the physical database
- Redesign of the transaction processing software
- Investigation of the application software and, if necessary, reworking of application software.

The question of whether existing information systems for engineer-to-order production could be used in the software factory needs additional research. The similarities are obvious and some of the consequences of the differences have been pointed out in this chapter. The additional research required should include an examination of the available production control information systems.

Using the data model to assess available information systems
The data model as described, can be used to assess the possibilities and shortcomings of information systems which are proposed for software engineering control. The data model presented in this chapter is based on the lessons learned in industry. Information systems for the control of software engineering, which will be suggested in the future can be compared with the data model presented in this chapter. In this way, possible shortcomings can be found in the proposed systems or the data model presented here.

9 FIRST STEPS TOWARDS A SOFTWARE FACTORY

9.1 Introduction

So far, we have focused attention on information systems for supporting multiproduct control in a software factory. It should be obvious that the vast majority of software engineering organizations do not operate as software factories as yet. Most of the current organizations are still struggling at the initial or repeatable level of process control, as was indicated by Humphrey. Most of them still have to achieve development control or product control before they can aim at multiproduct control. We are convinced that analysis of the current software engineering process can lead to the improvement that is required to get going on the way to the software factory. Data collection is required to be able to analyse the software engineering process.

The necessity of data collection for software engineering control is widely acknowledged, both in theory and practice. This, however, has still not led to extensive data collection in software engineering organizations. A recent survey in the Netherlands showed that 50 percent of the software engineering organizations do not collect any data on their engineering process (Siskens 1989). This chapter describes four examples of data collection techniques. The aim is to show that data collection in software engineering is feasible and can produce useful results. The examples given are intended to stimulate software engineering organizations to improve their data collection and thus advance to an improved level of process control. To be able to achieve this, we consider it more appropriate to provide practical examples than to give an overview of all the metrics that could be applied in theory. A thorough, theoretical description of metrics can, for example, be found in Conte (1986).

9.2 Basic principles

This section describes some of the principles we have applied in data collection. These are based on publications by Basili (1988) and Bemelmans (1989), as well as on a number of our own practical experiences. The principles are:
- The distinction between construction and analysis in software engineering
- The 'closed loop' principle in information systems
- 'Local for local' data collection
- A focus on continuous improvement

The distinction between construction and analysis in software engineering
Basili (1988) distinguishes between an analytic and a constructive aspect in software engineering. The distinction leads to analytic and constructive activities with the associated analytic and constructive methods and tools. Whereas constructive methods and tools are concerned with building products, analytic methods and tools are concerned with analyzing the constructive process and the resulting products. Basili states: "We need to clearly distinguish between the role of constructive and analytic activities. Only improved construction processes will result in higher quality software. Quality cannot be tested or inspected into software. Analytic processes (e.g. quality assurance) cannot serve as a substitute for constructive processes but will provide control of the constructive processes" (Basili 1988, page 759). Humphrey (1988) is talking about the same subject when he states that a project has two results: the software product and the knowledge of how the software product could have been developed better.

Let us look at data collection from the perspective of construction and analysis. The goal of software engineering improvement is to upgrade software construction since only improved software construction can result in better quality software. Analysis is required to control and improve software construction processes. Data on the engineering process are again necessary to analyse software engineering. The data collection techniques discussed in this chapter are intended to provide data which allow the analysis of the software engineering process. The analysis should result in actions for improvement which lead to better software construction processes. The relation between construction and analysis is illustrated in another way in Figure 9.1, which represents construction and analysis as the two wheels of a bicycle. The left picture shows the way most organizations approach software engineering nowadays. This could be called construction driven engineering.

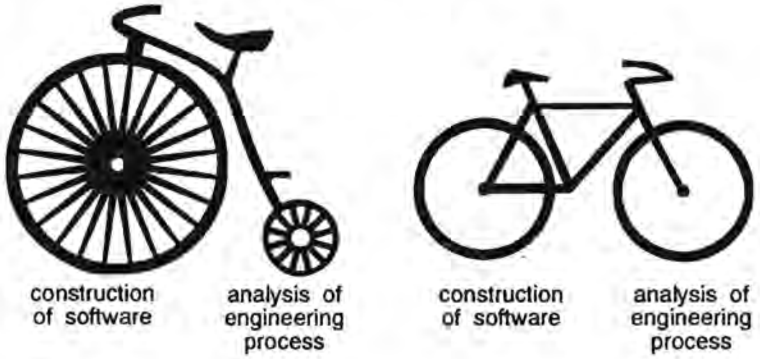
construction of software
analysis of engineering process
construction of software
analysis of engineering process

Figure 9.1 Construction versus analysis

The right-hand picture shows another, more modern approach to software engineering that might be called balanced engineering. A similarity is that the construction wheel is the driving wheel in both pictures. That is justified since only improved construction can result in improved software quality. The left-hand picture shows that the construction wheel is also the steering wheel. This should be considered a design defect because it results in an unstable engineering process. The future direction of the engineering process should be determined by the analysis of the current engineering process and new technological possibilities.

The 'closed loop' principle in information systems
The second principle is what Bemelmans (1989) has called the closed loop information supply. The principle states that information systems should be designed in such a way that those who provide input to the information system are main users of its output. Application of this principle has to result in feedback to the data supplier. This has a number of advantages of which we will mention the two most important. Firstly, it forces the data supplier to provide accurate and complete input. The supplier will harm himself as a user of the system if he does not do this. Secondly, the principle prevents users of information systems from asking for more information than they actually need. They will again harm themselves, because they will have to provide a lot of input. The closed loop principle forces the members of an organization to restrict themselves to the data they really need for control.

The closed loop principle has been applied in software engineering. One consequence is that the data collected by engineers should primarily support these engineers in the control of their work. Time sheets which are filled in every week without any feedback are a example of an information system that has not taken the closed loop principle into account. These kinds of systems often provide an organization with a lot of inaccurate and useless data. Another consequence of the closed loop principle is that data suppliers should know for what purpose the data will and will not be used. It should be obvious that the data should not be used against the supplier. For this reason, we discourage the use of productivity data for personnel appraisal. If data suppliers find out that data are being abused, they will do everything to corrupt the data, which will soon resemble the expected instead of the real situation and will have become useless. As Grady (1990) points out: you run the risk of distorting the idea of data as a helper and perverting it into data as a weapon unless the data are interpreted under the same ground rules by engineers and project managers.

The empirical studies described in chapter 2 can be perceived as examples of the application of the closed loop principle. The data were collected by project leaders. The same data were analyzed by project leaders and their managers in a joint meeting. The analysis increased the insight into reasons for delay among project leaders and their managers. It also resulted in actions for improvement which enabled future projects to follow the plans more closely, an outcome in the interest of the project leaders.

'Local for local' data collection

The approach to software engineering varies from department to department. The differences relate to products, development processes, resources, tools, goals and organization structures. Examples are the number of hours worked per month, productivity differences among applications and the development environment. As a result, it makes little sense to collect data in one environment and use it in another. A software engineering department can gain most insight from the data collected in its own environment. The fact that organizations are different becomes clear, for example, if one compares the distribution of the reasons for delays in three different departments, as presented in chapter 2.

A focus on continuous improvement

The data collection efforts were aimed at events that were perceived as deficiencies in the software process. The techniques focused on events such as delays and defects. Data collection and analysis should provide more insight into the causes of the perceived deficiencies, resulting in actions for improvement. A remark should be made at this point. We deliberately use the phrase 'perceived deficiency'. Most people would agree that delays and defects are examples of deficiencies. However, it is sometimes unclear whether a one week delay should be regarded as a deficiency or as an achievement by a project team that has taken just a little more time than was allowed for in the unrealistic schedule. We consider every deficiency as an opportunity for improvement. As a result it is not a matter of 'who was right or wrong'. It becomes a matter of 'how can we prevent this from happening again'.

The techniques employed did not require massive data collection. The data collection forms usually consisted of only one page. We are convinced that a number of small incremental steps towards improvement are better than one big leap. The improvement measures resulting from the study were incremental rather than revolutionary. The results of one analysis study will lead to some actions for improvement and will probably pinpoint to the next analysis study.

9.3 Two examples of data collection with regard to time and money

Reasons for delay
The empirical studies as described in chapter 2 can be considered as one example of a data collection technique. The data collection focused on the reasons for delay in software engineering. The analysis pays off because they result in actions for improvement that should allow future engineering efforts to follow their plan more closely.

Estimate delay in the remainder of a project
The insight in reasons for delay can be used to estimate the delays in the remainder of an ongoing project. This is a second analysis technique to be discussed. The fact that the collected data can be used to estimate the remainder of the project allows the people involved to benefit from the data collection and the analysis. The closed loop principle is applied and the participants of the project benefit from the data collection during the ongoing project.

Insight in the delay in the first phases of a project can be used to estimate the delay in the remainder of the project. It is important to be able to make a convincing estimation early in the project because control actions can still have effect at that time. An example of a control action that must be taken early is the modification of functional requirements. If this can be done in an early phase it will result in less development effort. Changing the product in the implementation or testing phase is less effective since most of the development work is already done and the changes will effect parts of the system that have already been completed. Another example of an action that only makes sense early in the project is to add people, although even early in the project care must be taken that the additional communication burden imposed by enlarging the project team does not become counter-productive (Brooks 1975).

One way to look at delays and overruns in projects is the S-curve. This technique has been in use in non software development for decades and is, for instance, discussed in Harrison (1977). The application of the S-curve is an example of the fact that techniques and methods that have been developed in non software environments, can be applied to software development. The S-curve compares the planned and actual cost of an ongoing project. The curve usually takes the shape of a S because a project often starts with a limited number of people, followed by a period of many participants and concluded by a period in which less people are involved. An example of the S-curve is given in Figure 9.2.

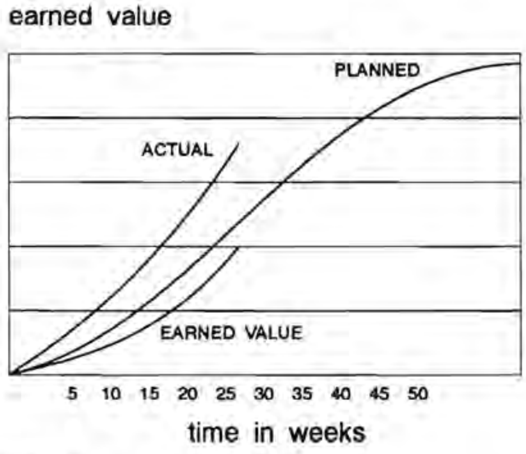

Figure 9.2 The S-curve

The S-curve consists of three lines:

- The Planned line represents the cumulative planned effort against the planned end date of the activities. The line can be computed with the data that is available in the project plan. The data collection that has been described in chapter 2 will provide the data to compute this line.
- The Actual line represents the cumulative actual effort against the actual end date of the activities. The data collection that has been described in chapter 2 will provide the data to compute this line.
- The Earned value line is needed because the Planned and Actual lines cannot be directly compared because they often differ on both axes; they show a delay in lead-time and a cost overrun. The earned value line shows the planned effort against the actual end date of the activity.

The Planned, Actual and Earned value line can now be compared in pairs. The vertical difference between the Actual line (actual effort versus actual end date) and the Earned value line (planned effort against actual end date) gives the difference in effort. The horizontal difference between the Planned line (planed effort versus planned end date) and the Earned value line (planned effort versus actual end date) gives the difference in lead time.

The extrapolation of the S-curve can be based on the insights the project leaders have gained into the project so far. Suppose that the specification phase of a project shows a 20 percent delay in lead time and a 10 percent cost overrun. Extrapolation of the delay and overrun in the remainder of the project can be based on the data collected in two ways. The first way uses the insight in the distribution of the delay over the subsequent phases of the project. The empirical studies in chapter 2 show that the relative differences between planned and actual effort increase toward the end of the project. Being able to show this fact enables to discourage the idea that delays can be made up as the project progresses. It is more reasonable to expect that the delay will increase from 20 to, say, 30 percent than to expect it to decrease to 10 percent. This insight can be used to extrapolate the S-curve.

The second way to extrapolate the delay and overrun is to use the insight into the reasons for delay. This way is preferred because it uses more of the information available on the progress of the project. The data collection, as it was presented earlier, comprised differences between planned and actual lead time as well as the reasons for the differences. The insight in the reasons can be exploited. For example, suppose that a 20 percent lead time delay is found in the first phase of the project and that ten of the 40 reasons that were mentioned are related to the lack of experience with a new development method. Five percent (one fourth of 20 percent) of the lead time delay can be ascribed to the lack of experience with the new development method. The project leaders can be interviewed on their view of the impact of the various reasons for delay in the remainder of the project. Suppose they expect the problems of lack of experience with the development method to double in the remainder of the project. The delay because of the lack of experience can be expected to be 10 percent. Applying this method of reasoning to all the (groups of) reasons for delay and cost overrun results in an extrapolation of the expected delay in the remainder of the project.

It is not important whether the expected delay is specified in one or two digits. Key to the value of the extrapolation is the fact that the project leaders are involved in the extrapolation and that the method of reasoning is clear and can be verified by anyone involved. The insight into both the delays in different phases and into the insight in the reasons for delays should be used in extrapolating the delay in the remainder of the project. It is also recommended that several extrapolations are made, based on different assumptions. We have experienced that different extrapolations stimulate a discussion on the measures that can be taken to avoid the delay that is estimated (Lierop 1991). Goal of the extrapolation is not to predict the delay in the project, but to encourage an early discussion on measures that can avoid additional delay.

9.4 Two examples of data collection with regard to quality

The analysis technique that has been presented in chapter 2 and elaborated in section 9.3 focuses on the control aspects time and cost. The analysis techniques in this section focus on the control aspect quality. Pettijohn (1986) pointed out that there are two primary sources of quality data: inspection data and maintenance reports. The first analysis technique aims at maintenance reports; a sign of lack of quality that has not been detected during development. The second analysis technique focuses on inspections and gains insight from the faults that are detected during development.

Analysis of problem reports

The analysis so far has been aimed at development. The importance of maintenance has been stressed in chapter 4. Software maintenance has been analysed within a large software engineering department (Boomen 1990). The analysis was one of the results of a study of the reasons for delay that was described in section 2.5. This study of reasons for delay showed that maintenance was a main reason for delay in development. Analysis of maintenance was a natural continuation. The questions we were mainly interested in were:

- Where does the maintenance originate?
- How can we reduce the effort that is required to do the maintenance?

Maintenance reports represent the information that is gathered on faults that are detected and solved. The department concerned calls them problem reports. If a problem occurs, a problem report is written that describes the problem perceived and its correction. The goal of the analysis was to gain insight in the origin of the problems and to investigate several relations that were expected. It was, for example, expected that faults made early in the development life cycle take more effort to fix. A correlation was also expected between the phase in which a fault originates and the phase in which it is detected. It was expected because the department had adopted the V model of development and testing. The V model is shown in Figure 9.3. It shows the phases as they were distinguished by the department concerned: exploration, requirements, design, implementation, integration test, verification test and validation test.

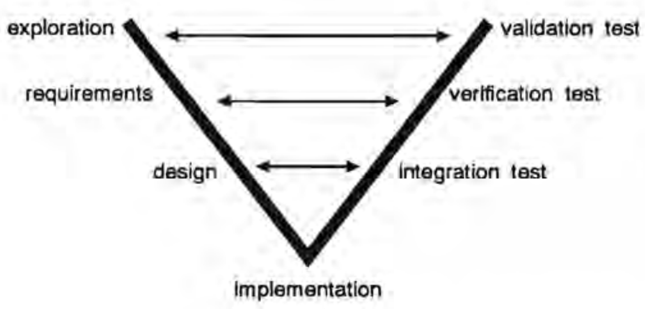

Figure 9.3 The V model of development and testing (Boomen 1990).

The V model shows that the validation tests are designed to detect faults in the exploration results, verification tests are designed to detect faults in the requirements results and that integration tests are designed to detect faults in the design results.

Analysis of the problem reports required additional data collection. Some multiple choice questions were added to the existing problem reports. Three of these questions are shown in Table 9.1.

Table 9.1 Three questions related to problem reports

1) How many hours did it take to solve the problem? o Less than one hour o 1 to 2 hours o 2 to 4 hours o 4 to 8 hours o over 8 hours
2) In what phase did the error occur ? o exploration o requirements o design o implementation o other, ...
3) In what test was the fault detected ? o integration test o verification test o validation test

Over 400 problem reports were analysed. Some relations that were expected were not found. For example, we found no relation between the phase in which a fault originated and the effort required to solve it. Table 9.2 shows the solution time of the fault versus the kind of fault.

Table 9.2 Solution time versus the kind of fault

	SOLUTION TIME						
KIND OF ERROR	< 1	1-2	2-4	4-8	> 8	Total	%
Requirements	22	10	7	0	1	40	10
Design	12	8	6	1	4	31	7
Implementation	93	37	15	6	12	163	40
Other	103	25	16	7	26	177	43
Total	230	80	44	14	43	411	100
Percentage	56	20	11	3	10	100	

The results of the study were analysed by the software engineers, project leaders, the manager and members of the quality assurance department. The analysis yielded some useful insights. Maintenance was perceived as an activity that took a lot of effort in the department concerned. It was not expected that over half of the problem reports are solved within an hour. Analysis of the data revealed that the maintenance problem was more of a lead time than an effort problem. It took more time to get the problem to the appropriate engineer than to solve the problem.

The analysis yielded more unexpected results. Some correlations that were expected were not found. Figure 9.4 shows the number of requirements, design and implementation failures found in the various tests.

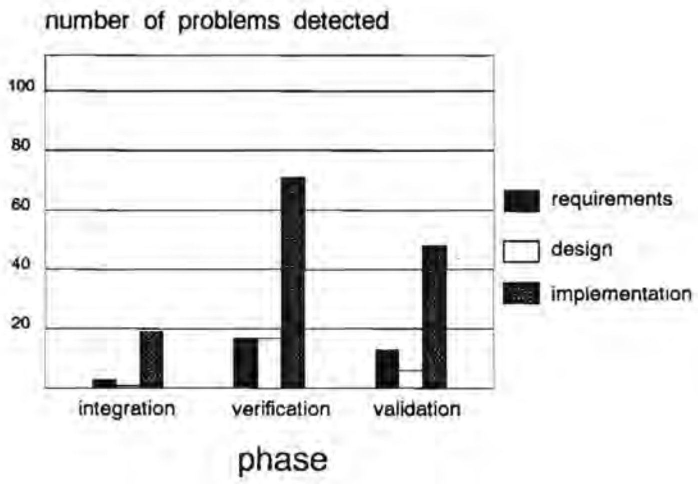

Figure 9.4 The kind of faults that are found in the various tests

Figure 9.4 shows integration, verification and validation on the horizontal axis. The number of faults detected are distributed over the phases in which the errors occurred. For example, three requirements errors, one design error and 19 implementation errors were found in the integration test. At first glance, there is no clear correlation between the kind of error and the phase in which the fault was detected. Apparently, the V model of development and testing does not work.

What can be concluded from these results? Certainly, they were not what was expected; there is no apparent correlation between source of error and either time to fix or point of detection. The answer is, in part, that measurements of the effectiveness of a working process can hardly be useful if the process is not actually in place and followed. However, it is also clear from the results of this study that any measurements are useful, and, if properly interpreted, give a good insight into the processes which are actually being followed. In this case, the results were unexpected and surprising, but they focused attention on what is perhaps the real problem - that the methods and procedures which were prescribed were not in fact being followed - and thus opened the opportunity for improvement. The study led to a reconsideration of the relation between the engineering department and the tools and methods department. It is clear that something will have to change if the tools and methods that are developed are not applied. It is also clear that both sides will have to adjust to improve the situation.

Answers to the questions stated in this study enable to take improvement measures to assure less errors in the future. The analogy with the analysis described in the chapter 2 and section 9.3 is clear. Both analysis methods had a clear goal. The data collection form fits on one sheet of paper. The data should result in additional insight that in its turn results in actions that improve the quality of the engineering process. The improved engineering process should result in improved quality of software.

The analysis of inspection or walk through data

The analysis of problem reports looks into the errors that remain undisclosed during development. It is obvious that it is better to detect faults earlier. A lot of faults are disclosed during development in what are called inspections or walk throughs. Fagan's inspection method will be discussed as an example. Fagan's method has been chosen for three reasons. Firstly, it is a technique that has been applied since 1972 and has proven its value. Secondly, it puts a lot of emphasis on data collection during the inspection. Thirdly, it is well engineered and documented (Fagan 1976, 1986, Gilb 1988, Humphrey 1989). An inspection is a meeting at which a document produced by a developer is inspected by colleagues to reveal possible defects. Errors, faults and failures have been distinguished in section 4.4. The term defect is introduced because it is often used in connection with inspections. Fagan defines a defect as 'An instance in which a requirement is not met' (Fagan 1986). The inspected document can be source code, a design document, a specification or any other documentation. Fagan's inspection method consists of six phases or activities: planning, overview, preparation, inspection, rework and follow-up. The planning activity confirms that the documents to be inspected fulfil the entry criteria and sets up the meeting with the required participants. People with different roles will participate in the inspection: moderator (the coach of the inspection team that manages the inspection process), administrator (who collects the inspection data), author (of the subject of the inspection), reader (who reads the document as if he will have to implement it) and tester (who inspects the document from the test point of view). The assignment of inspection roles is made at an overview meeting, where the documents to be inspected are handed out.

The actual inspection takes place after individual preparation by the participants. The goal of the inspection is to find defects, not to discuss or fix defects. Defects are classified as major or minor; a major defect is one that would cause a malfunction or unexpected result if uncorrected, a minor defect will not cause malfunction but is more in the nature of poor workmanship, such as spelling errors that do not lead to erroneous product performance (Fagan 1986). Something is called a defect if one of the participants perceives it as a defect, even if it is not a defect. The idea behind this is that if one of the few people in the inspection team can misinterpret a (part of the) document, it is not clear enough and may lead to similar misinterpretation later in the development process. After the inspection, the author reworks the defects and the document is inspected again in a follow-up session.

Inspection has some similarities and some clear differences with walk-throughs. Some differences are in Table 9.3.

Table 9.3 Differences between inspections and walk-throughs (Fagan 1976)

PROPERTIES	INSPECTION	WALK-THROUGH
- formal moderator training	yes	no
- definite participant roles	yes	no
- who drives the inspection or walk through	moderator	owner of document
- use 'How to find errors' checklist	yes	no
- use distribution of error types to look for	yes	no
- follow-up to reduce bad fixes	yes	no
- detailed error feedback to individual programmer	yes	incidental
- improve inspection efficiency from analysis of results	yes	no
- analysis of data -▸ process problems -▸ improvements	yes	no

One difference will be discussed in detail. Data collection is an important aspect of inspection. The data is used in the first place to give the author feed-back on his work. It is clear that inspection should never be used against developers because this would conflict with one of the inspection's prerequisites: colleagues that are motivated to find defects. Inspection data can also be used to analyse software development. The data that is collected during inspection can be used to answer questions like:

- Are more defects made in larger modules?
- Has the number of defects reduced since we introduced structured programming?
- Which percentage of the defects do we find in inspection and what does that mean for the number of defects that remain in the software product? In other words: how many defects do we include in our product when we deliver it to our client?
- How many pages of documentation should we inspect per hour to maximize the number of defects found?
- Has the number of defects reduced since we introduced Fagan inspections?

The last question will be addressed. Fagan (1986) estimates that all design and code inspection costs amount to 15 percent of project cost. Examples of the insights that can be gained by analysis of inspection data can be found in the literature. Some examples:

- A major aerospace contractor that has a rigorous and comprehensive inspection program reported an after-release defect rate of less than 0.11 defects per thousand lines of code.

- A major government systems developer reported 12.25 defects found per thousand lines and 1.5 defect found per men hour spent on inspection.
- Another development group reported 42 defects per thousands lines of code.
- A banking computer services firm found that it took 4.5 hours to eliminate a defect by unit testing compared to 2.2 hours by inspection (these four examples can be found in (Ackerman 1989))
- An IBM development department installed a defect detection and prevention process. Analysis of inspection data led to actions for improvement that resulted in a 50% reduction of defects at a cost of 0.4% of the resources of the department concerned (Mays 1990). The analogy with the other analysis techniques is clear: data collection followed by analysis led to actions for improvement. The actions in this case had impressive results.

The examples show three things. First, inspections pay off if they are applied properly. Second, it is clear that data collection is an important aspect of inspections. Analysis of the data provides insight into the software process and pinpoints at flaws in the process. Third, the data collected at various sites differs considerably. Therefore, it is necessary for every software engineering department to gain insight into its own software engineering process to enable it to take adequate improvement measures.

10 CONCLUSIONS AND RECOMMENDATIONS

10.1 Introduction

The last chapter allows us to summarize the book and recommend some subjects for future research. The summary presented in section 10.2 includes the main conclusions. Section 10.3 comprises four subjects which are recommended for future research.

10.2 Summary and conclusions

The subject of this book is the control of software engineering. The problem is explored in chapters 1, 2 and 3. Next, chapters 4, 5 and 6 describe how the control of software engineering can be improved. Finally, the chapters 7, 8 and 9 deal with the information required to control software engineering.

Chapter 1 contains the problem statement and the aim of the book. The aim is described as:

1 Determine the characteristics of the control concept of software engineering that fit in with the changed practices and demands
2 Derive the characteristics of an information system that supports the control concept.

Chapter 2 discusses empirical studies of reasons for delay in software engineering. Delay can be perceived as a consequence of lack of control. Studies in three different engineering departments show that the delays and the reasons for delay varied from one department to another. The studies also show that the control of software engineering cannot be restricted to a development project. Some of the important reasons for delay originate from outside the project but nevertheless affect it. The third empirical study describes a project with a lead time of 1.5 years that was finished in time and under budget. This is an example of the fact that software engineering can be controlled, provided certain conditions are fulfilled.

Chapter 3 concludes the exploration of the problem. It describes the circumstances in which software engineering takes place. A distinction has been made between traditional and current software engineering. Characteristics of traditional software engineering were the fact that isolated applications with stable applications have been developed and that engineering efforts were isolated efforts by specialists.

The control of software development was characterized by the use of the waterfall model, project control and an emphasis on efficiency. Over the years the circumstances have changed. Examples of those changes are the fact that new application areas have been entered, more maintenance has become necessary and both software products and engineering efforts have become less isolated. Another change is that the quality of products and processes has become an important subject over the years. Control of software engineering has not adapted itself well enough to the changed circumstances. Traditional control will not be appropriate in all the engineering situations that occur.

The enhancement of control will take place in two steps. The first step is the progression from development control to product control. The explanation of this step requires a closer look at software quality. It reveals that several quality definitions are required to understand the quality conflicts that arise. User-based, product-based, manufacturing-based and value-based quality definitions are distinguished. The use of several valid quality definitions such as 'quality is fitness for use' and 'quality is conformance to specifications' can lead to conflicts. Quality improvement means that user needs are properly translated into identifiable product attributes. Product attributes have to be translated into process attributes which can ensure the engineering of the required product. The lack of understanding of software and its engineering is an obstacle to a proper translation at this time. Maintenance takes up most of the capacity of software engineers nowadays. We conclude that control of software engineering cannot afford to limit its concern to development alone. The integrated control of development and maintenance is called product control. The consequences for control are the difference between the goal of development control and product control and the organization of a department employing it. Two organizational aspects are discussed in detail. Firstly, the function of product management has been examined. Secondly, the pros and cons of the organization of development and maintenance activities in one or more departments are discussed.

The next step towards enhancing the control of software engineering is the transition from product control to multiproduct control. Chapter 5 discusses multiproduct control, i.e. the control of a number of software products over their life cycle. An organization employing multiproduct control is typified as a software factory. The enhancement is required in order to be able to exploit the reuse potential of software. Reuse is necessary because major productivity improvements are required to allow the software supply to keep up with the rapidly increasing demand for software. Chapter 5 reveals that the software industry has evolved along the same lines as the industry in general. The market requires both flexibility and efficiency from the suppliers. Two ways to meet these conflicting demands are standardization and modularization. The key to the latter is that a limited number of components can be assembled into a variety of client-specific products. The aspects of multiproduct control are discussed: life cycles, goal and organization. Two organizational subjects are discussed, namely a) the organization of work release, and b) the organization of development, maintenance and development for reuse in one or more departments. Multiproduct control can be organized in many ways. One specific software factory is described in chapter 5. This software factory is taken as a starting point in the remainder of the book.

Chapter 6 discusses the road to the software factory. The discussion so far has concentrated on the differences between the stages and on the need to advance from one stage to the next. It might be concluded that the progression from project control via product control to multiproduct control is just a matter of time. The opposite is true. One cannot expect to control development and maintenance if one is not able to control development alone. The progression from one control situation to the next requires an explicit decision to do so and a certain level of process control. We use the levels of process control, as distinguished by Humphrey (1989a), to describe the steps to the software factory. The steps show that an organization can only afford the expansion of the control focus, if it has achieved a certain level of process control.

Chapter 7 is the first of three chapters that deals with the information requirements and information systems in a software factory. Chapter 7 assesses a number of information systems which can be used as a reference for an information system for software engineering control. Firstly, three information systems proposed in the literature for the control of software engineering are described. These systems are the Project Management Data Base from TRW, TAME from the University of Maryland and the experience base described by Noth. Valuable insights have been gained from the assessment of the systems. Another reference system has been studied: an information system for engineer-to-order production, as described in (Bertrand 1990). The assessment with respect to the software factory shows that the production control information system fits in fairly well. A more detailed assessment also shows some misfits. We have chosen to adapt the information system for engineer-to-order production control in such a way that it becomes useful for the software factory. The main reasons to opt for the production control information system are the results of the assessment and its maturity as compared with the information systems for software engineering control.

In chapter 8 we propose a data model for an information system for software engineering control in a software factory. Additional information requirements are derived from the fact that in this case the product concerned is software. The data model itself is presented in Figure 8.1 in section 8.3. The main differences between it and the data model for production control are the product description and the many relations between the order-dependent and order-independent part of the data model. Products in use and reference items are considered as order-independent, while products and components under development are order-dependent. Product representation and the identification of software components are identified as key issues. The data model can be used as a starting point in the development of an information system for software engineering control. Another possibility is to use the data model to assess the information systems proposed for software engineering control.

Chapter 9 is the third and last chapter that addresses the information issue. Most of the software engineering organizations nowadays are not software factories. Data collection is important for improving the control of software engineering and advancing to higher levels of process control. Chapter 9 shows practical examples of data collection in software engineering. Examples have been given because many organizations do not collect data on software engineering and it is important to show that it is possible to collect data and analyze software engineering. Four examples of analysis in software engineering are discussed, beginning with the basic principles used in data collection. We consider data collection techniques a means for setting out on the road to the software factory. But they are certainly not a map of this road. Such a map cannot be provided since software engineering differs greatly from one place to another and few organizations have reached the software factory as yet. The ongoing analysis of software engineering should determine the path an organization will choose in progressing towards the software factory.

10.3 Recommendations for future research

A researcher is allowed to make recommendations for future research at the end of a thesis. Four possible themes will be mentioned in this section.

Analysis of software engineering

This book has discussed a number of empirical studies on the control of software engineering. The studies addressed questions such as:

- Why is software late? and

- Where does the maintenance originate?

More similar studies are required because insight into the control of software engineering is still limited. Examples of questions that need to be addressed are:
- To what extent do object-oriented techniques increase productivity?
- How many faults do we include in our software when we deliver it to our customer?

The questions need to be addressed because we require more insight into software engineering and its control. The level of control is still insufficient, while software engineering is becoming increasingly important. There is another reason why the questions should be addressed. Before long, clients of software engineering departments will start to ask questions like this because they want to know where their money is going. The software engineering community should be able to answer the questions by that time. The question of the costs and benefits of information technology will become crucial and should be a topic of interest for those involved in software engineering.

The answers to the questions stated can be found by analysing software engineering. An advantage of empirical studies is that the knife cuts both ways: both research and the participating department or company benefit. The software engineering departments which cooperate benefit from the data collection because they gain additional insight into their engineering process. The researcher benefits from the insight into software engineering processes gained in practice.

The human factor in control

The empirical studies in chapter 2 show that human factors play an important part in the control of engineering. The success of engineering is and always will be determined by the quality and commitment of the people who are doing the job. The human factor is also important in the control of the engineering process. Software engineering control depends on the commitment of engineers to the stated goals. It is unclear how such a commitment can be obtained and how it can be maintained throughout the project. We have seen that the way in which quality goals are stated, and time as well as effort are estimated, affects the level of commitment. It has become clear that the estimation of software projects is not just a matter of techniques and models (Kusters 1990). The research group Management Information Systems and Automation of the University of Technology in Eindhoven has started a study that will examine estimation as a group process. The estimation process will be investigated and the possibilities of information technology support will be explored.

The way to a software factory
This book has argued that software engineering organizations should broaden the scope of their control from development, via product to multiproduct control. Multiproduct control has been typified by the name software factory. Software factories have been a topic of interest for many years now (Cusumano 1989, Boehm 1989). Descriptive studies of the characteristics of the software factory and the changes an organization is going through are required. An example of such a study is (Cusumano 1989). The insights that are gained from the studies will point the way to further improvements and will guide other organizations on their way to improved control of software engineering.

The logistics of engineering
This book has attempted to apply concepts from logistics or production control to software engineering. Examples are the concept of the decoupling point, the hourglass product structure, planning at different aggregate levels and the use of information systems from the field of production control as a reference system for the software factory. The concepts originate in production, i.e. the factory. Their application to engineering will be fruitful for both engineering and logistics. Engineering can benefit from the fact that the field of logistics has been studying primary process control for years. Of course there are differences between repetitive manufacturing and software engineering. The differences and similarities will require additional research and will lead to many discussions.

Both parties can benefit. Logistics will benefit because engineering is becoming increasingly important as compared with production because of the shortening of product life cycles and the rise of intangible products like software. Logistics should consider software engineering as a growing market for its ideas and concepts. Logistics will also benefit because some problems that software engineering has faced for years will become relevant for production. An example is the 'one of a kind' production that is becoming progressively important in production control. Software engineering has always been 'one of a kind production'. Logistics may be able to gain fresh insight from the ways in which the software community has sought to solve the problems that occur in 'one of a kind' production.

One specific topic for research will be mentioned. It is the issue of product representation. This issue is a key problem in software reuse. Product representations are also a key subject for information systems in production control. The research currently going on in both logistics and computer science should benefit from the exchange of ideas and concepts.

REFERENCES

Ackermann, A.F., Buchwald, L.S., Lewski, F.H., "Software inspections: an effective verification process", IEEE Software, May 1989.

Ahituv, N., Neumann, S., "Principles of information systems for management", Wm. Brown Co. Publ., Dubuque, Iowa, 1982.

Basili, V.R., Turner, A.J., "Iterative enhancement, a practical technique for software development", IEEE Trans. Software Eng., Vol. SE-1, no. 4, pp 390-396, 1975.

Basili, V.R., Perricone, B.T., "Software errors and complexity: an empirical investigation", Communications of the ACM, January 1984.

Basili, V.R. Selby, R.W. Hutchens, D.H., "Experimentation in software engineering", IEEE Trans. Software Eng., Vol. SE-12, no. 7, pp 733-743, 1986.

Basili, V.R., Rombach, H.D., "Tailoring the software process to project goals and environments", International Conference on Software Engineering, 1987.

Basili, V.R., Rombach, H.D., "The TAME project: towards improvement oriented software environments", IEEE Trans. Software Eng., Vol. SE-14, no. 6, pp 758-773, 1988.

Basili, V.R., "Software development; a paradigm for the future", Key note address, Proceedings of the thirteenth annual international computer software and applications conference, Orlando, FL, 1989.

Basili, V.R., "Viewing maintenance as reuse-oriented software development", IEEE Software, January 1990.

Beck, J.R., "Time management", Project management special summer issue, August 1986.

Bemelmans, T.M.A., "Bestuurlijke informatiesystemen en automatisering", Hoofdstuk 8, derde druk, Stenfert Kroese 1987 (in Dutch).

Bemelmans, T.M.A., "Bedrijfskundig ontwerpen van bestuurlijke informatiesystemen", published in "Automatisering met een menselijk gezicht", in P.A. Cornelis, J.M. van Oorschot, Kluwer, Deventer, 1986 (in Dutch).

Bemelmans, T.M.A., "Informatiekunde; vragen, geen antwoorden", Special issue Informatie, June 1989 (in Dutch).

Bemelmans, T.M.A., "Informatiemaatschappij: heerlijke nieuwe wereld?", published in "Arbeid en management in de informatiemaatschappij", Stenfert Kroese, Leiden, 1986 (in Dutch).

Bertrand, J.W.M., Wortmann, J.C., Wijngaard, J., "Production control; a structural and design oriented approach", Elsevier, Amsterdam, 1990.

Biggerstaff, T.J., Richter, C., "Reusability framework, Assessment and directions", IEEE Software, pp. 41-49, March 1987.

Biggerstaff, T.J., Perlis, A.J., "Introduction", in Software reusability, Volume 1, concepts and models, ACM Press, New-York, 1989.

Boehm, B.W., "Software engineering economics", Englewood Cliffs, NJ, 1981.

Boehm, B.W. "A spiral model of software development and enhancement", IEEE Computer, May 1987.

Boehm, B.W. "Improving software productivity", IEEE Computer, September 1987.

Boehm, B.W., en Papaccio, P.N. "Understanding and Controlling Software Costs." IEEE transactions on software engineering, volume SE-14, no. 10 October 1988.

Boehm, B.W., "Software factories in the USA", in Proceedings of the 11th World Computer Congress, San Francisco, August 28 - September 1, 1989.

Bolwijn, P., Kumpe, T., "Production in the 1990's; efficiency, flexibility and innovation.", Long range planning, August 1990.

Boomen, T., Brethouwers, G., "On the analysis of software development", Masters Thesis, University of Technology Eindhoven, April 1990.

Botter, C.H., "Organisatie rond de produktinnovatie", Kluwer, Deventer, 1982 (in Dutch).

Brooks, F.B., "The Mythical Man-Month, Essays on software engineering", Addison Wesley Publishing Company, London, 1975.

Brooks, F.P., "No silver bullet, essence and accidents of software engineering", IEEE Computer, April 1987.

Cavano, J.P., McCall, J.A., "A framework for measurement of software quality", proceedings of the ACM software quality assurance workshop, November 1978.

Conte, S.D., Dunsmore, H.E., en Shen, V.Y., "Software engineering metrics and models", Benjamin Cummins, 1986.

Crossman, T.D., "Inspection teams, are they worth it?", in proceedings of the 2nd national symposium EDP quality assurance, Chicago, Il., March 24-26, 1982.

Cuelenaere, A.M.E., van Genuchten, M.J.I.M., and Heemstra, F.J.,"Calibrating a software cost estimation model: why and how". Information and software technology, volume 29, no. 10, December 1987.

Cusumano, M.A., "The software factory: a historical interpretation", IEEE Software, March 1989.

Davis, G.B., Olsen, M.H., "Management Information Systems, Conceptual foundations, Structure and Development", Mc Graw-Hill, New-York, 1984.

Davis, A.M., Bersoff, E.H., Comer, E.R., "A strategy for comparing alternative software development life cycle models", IEEE transactions on software engineering, October 1988.

Deming, W.E., "Quality, productivity and competitive performance", Massachusetts Institute of Technology, Centre for advanced engineering studies, 1982.

Fagan, M., "Design and code inspections to reduce errors in program development", IBM systems journal, no.3 1976.

Fagan, M., "Advances in software inspections", IEEE transactions on software engineering, July 1986.

Fox, J.M., "Software and its development", Prentice Hall, Englewood Cliffs, 1982.

Garvin, D.A., "What does 'product quality' really mean", Sloan Management review, Fall 1984.

Genuchten, M.J.I.M. van, Fierst van Wijnandsbergen, M., "An empirical study on the control of software development", Proceedings of the conference on Organization and Information Systems, pp 705-718, Bled, Yugoslavia, September 13-15, 1989.

Genuchten, M.J.I.M. van, Koolen, J.A.H.M., "On the use of software cost models", Information and Management, July 1991.

Genuchten, M.J.I.M. van, "Why is software late? An empirical study of reasons for delay in software development", to be published in IEEE Transactions on Software Engineering, June 1991.

Gilb, T., "Principles of software engineering management", Addison Wesley, 1988.

Grady, R.B., "Dissecting software failures", Hewlett-Packard Journal, April 1989.

Grady, R.B., "Work product analysis: the philosopher's stone of software?", IEEE Software, March 1990.

Harrison, F.L., "Advanced project management", Gower publishing company limited, Aldershot, England.

Heemstra, F.J., "Hoe duur is programmatuur?", Kluwer, 1989 (in Dutch).

Heemstra, F.J. "Wat bepaalt de kosten van software", Informatie, volume 29, extra edition, 1987 (in Dutch).

Hegge, H.M.H, Wortmann, J.C., "Generic bill-of-material; a new product model", working paper, Department of Industrial Engineering, Eindhoven University of Technology, 1990.

Humphrey, W.S., "Characterizing the software process: a maturity framework", IEEE Software, pp. 73-79, March 1988.

Humphrey, W.S., "Managing the software process", Addison Wesley, 1989a.

Humphrey, W.S., Kitson, D.H., Kasse, T.C., "The state of software engineering practice: a preliminary report", Proceedings of Conference on Software Engineering, Pittsburgh, 1989b.

IEEE, "IEEE standard glossary of software engineering terminology", New York, Rep. IEEE-std-729-1983, 1983.

ISO, "ISO 9000", International Organization for Standardization, Reference number ISO 9000:1987 (E).

ISO, "Guidelines for the application of ISO 9001 to the development, supply and maintenance of software", International Organization for Standardization, 1990.

Jeffrey, D.R., Basili, V.R., " Validating the TAME resource model", Proceedings of the eleventh International Conference on Software Engineering, pp 187-201, Singapore, 1988.

Jenkins, A.M., Naumann, J.D., Wetherbe, J.C., "Empirical investigation of systems development practices and results", Information & management, 7, 1984.

Jones, T.C., "Reusability in programming: a survey of the state of the art", IEEE transactions on software engineering, Vol. 10, no. 5, pp. 488-494, September 1984.

Juran, J.M., Gryna, F.M., "Juran's quality control handbook", Fourth edition, McGraw-Hill book company, 1988a.

Juran, J.M., "Juran on quality planning", Juran institute 1988b.

Kusters, R., van Genuchten, M.I.J.M., and Heemstra, F.J., "Are software cost estimation models accurate?", Information and software Technology, Volume 32, no. 2, March 1990.

Lanergan, R.G., Grasso, G.A., "Software engineering with reusable designs and code", IEEE transactions on software engineering, Vol. 10, no. 5, pp. 498-501, September 1984.

Lehman, M.M., "Programs, lifecycles, and laws of software evolution", Proceedings of the IEEE, 9, 1980.

Lehman, "Program evolution", Information processing and management", Vol. 20, no.1-2, 1984.

Lierop, F.L.G. van, Volkers, R.S.A., van Genuchten, M.I.J.M., and Heemstra, F.J., "Heeft iemand de software al gezien? Inzicht in het uitlopen van softwareprojecten", Informatie, March 1991 (in Dutch).

Looijen, M., "Management en organisatie van automatiseringsmiddelen", Ph.D. Thesis, Eindhoven University of Technology, 1988 (in Dutch).

Manual, T., "What's behind all the software delays", Electronics, June 1989.

Martin, J., "Recommended diagramming standards for analysis and programmes", Prentice Hall, 1987.

Martin, J., McClure, C., "Software maintenance the problem and its solution", Prentice Hall, Englewood Cliffs, 1983.

Matsumoto, Y., "Management of industrial software production", IEEE Computer, pp. 59 - 71, February 1984.

Mays, R.G., Jones, C.L., Holloway, G.J., Studinski, D.P., "Experiences with defect prevention", IBM systems journal, no 1, 1990.

McCall, J.A., "The utility of software metrics in large scale software systems development", IEEE Second software life cycle management workshop, August 1978.

Mills, H.D., Dyson, P.B., "Using Metrics to quantify development", IEEE Software, March 1990.

Mintzberg, H., "The structuring of organizations", Prentice-Hall, 1979.

Moad, J., "Cultural barriers slow reusability", Datamation, November, 15, 1989.

Myers, G.J., "Software reliability;principles and practices", Wiley interscience, John Wiley and sons, 1976.

Nelson, R., "Software's midlife crisis", Electronics, June 1989.

Noth, T., "Unterstutzung des Management von Software-Projekten durch eine Erfahrungsdatenbank"", Springer Verlag, 1987 (in German).

Penedo, M.H., Stuckle, E.D., "PMDB, A project master database for software engineering environments", Proceedings of the eighth International Conference on Software Engineering, pp 150-157, London, August 1985.

Pettijohn, C.L., "Achieving quality in the development process", AT&T Technical journal, Volume 65, March April 1986.

Phan, D., Vogel, D., Nunamaker, J., "The search for perfect project management, Computerworld, September 1988

Phan, D., "Information systems project management: an integrated resource planning perspective model", Ph.D. Thesis, department of Management Information Systems, University of Arizona, Tucson, AZ, 1990.

Prieto Diaz, R., Freeman, P. "Classifying software for reusability", IEEE Software, pp. 6-16, January 1987.

Prieto Diaz, R., "Implementing faceted classifying for software reuse", Proceedings of the International Conference on Software Engineering, pp 300-304, Nice, 1990.

Rombach, H.D., "A comprehensive approach towards reuse", Keynote address, 2nd international conference on Software Quality Assurance, Oslo, Norway, May 1990.

Sakurai, M., "Cost accounting for software", Cost management system seminar, June, 15-16, Dallas.

Schaick, E.A., "A management system for the information business", Prentice Hall, 1985.

Schendler, B.R., "How to break the software logjam", Fortune, September, 25, 1989.

Selby, R.W., "Quantitative studies of software reuse", in Software reusability, Volume 2, application and experience, editors Biggerstaff, T.J., Perlis, A.J., ACM Press, New-York, 1989.

Shandle, J., "It's time to grow up", Electronics, June 1989.

Shaw, M., "Prospects for an engineering discipline of software", IEEE Software, November 1990.

Sikkel, K., Van Vliet, J.C., "Kan software langer mee? Een overzicht van hergebruik van software", Informatie, Jaargang 30, no 7/8, pp. 464-483, August 1988 (in Dutch).

Siskens, W.J.A.M., Heemstra, F.J., van der Stelt, H. "Cost control in automation projects, an empirical study", Informatie, volume 31, January 1989 (in Dutch).

Standish, "An essay on software reuse", IEEE Transactions on Software Engineering, no. 5, 1984.

Stretton, A., "A consolidation of the PMBOK framework and functional components", Project management journal, December 1989.

Swanson, E.B., "The dimensions of maintenance", proceedings of the second international conference on software engineering, San-Francisco, pp. 492-497, 1976.

Swanson, E.B., Beath, C.M., "Organizational foundations for maintenance", Software maintenance, research and practice, Vol. 1, pp. 47-58, 1989.

Thambain, H.J., Wilemon, D.L., "Criteria for controlling projects according to plan". Project Management Journal, June 1986.

Veen, E. van, "Modelling product structures by generic bills-of-material", Ph.D. thesis, University of Technology Eindhoven, 1991.

Ward, T.M., "Software measures and goals at Hewlett-Packard", Conference proceedings Juran seventh annual conference on quality management, Atlanta, 1989.

Weiss, D.M., Basili, V.R., "Evaluating software development by analysis of changes: some data of the SEL", IEEE transactions on software engineering, February 1985.

Wideman, R.M., "The PMBOK report, PMI Body of Knowledge standards", Project management special summer issue, August 1986.

Wilke, J.R., "Lotus expects more delays on software for Macintosh", Wall Street Journal, January, 17, 1990.

Wijnen, G., Storm, P., Renes, W., "Projectmatig werken", Marka paperback series, 1984 (in Dutch).

SAMENVATTING

Het onderwerp van het proefschrift is het beheersen van software-ontwikkeling. Het onderzoek is gestart naar aanleiding van het grote economische en maatschappelijke belang van software enerzijds, en de huidige problemen met de beheersing anderzijds. Voorbeelden van problemen zijn:

- herhaalde overschrijdingen van kosten en doorlooptijd van softwareprojecten,
- het grote aandeel van de onderhoudskosten in de totale softwarekosten, en
- de beperkte produktiviteitsverbeteringen in de software-ontwikkeling.

Het proefschrift vormt de volgende bijdrage aan het oplossen van de gesignaleerde problemen:

1 Door middel van empirisch onderzoek zijn oorzaken van vertraging van software-ontwikkeling in een aantal ontwikkelafdelingen vastgesteld. Het verworven inzicht stelde de betreffende afdelingen in staat maatregelen te nemen die toekomstige uitloop kon voorkomen. Het inzicht gaf tevens richting aan het onderzoek naar verbeterde beheersconcepten voor software-ontwikkeling.

2 In het proefschrift wordt een beheersconcept voorgesteld op basis waarvan de beheersing van software-ontwikkeling beter in staat zal zijn de huidige problemen het hoofd te bieden. Door het beheersconcept wordt expliciet aandacht besteed aan de beheersing van onderhoud en hergebruik van software.

3 Het proefschrift beschrijft de informatievoorziening in een softwarefabriek. Het blijkt dat software-ontwikkeling gebruik kan maken van de ervaringen zoals die zijn opgedaan in de produktiebeheersing.

Het proefschrift bestaat uit drie delen. Het eerste deel inventariseert het probleem. Het beschrijft ondermeer de genoemde empirische onderzoeken. Het eerste deel wordt afgesloten met een karakteristiek van de traditionele beheersing van software-ontwikkeling. Die beheersing gaat uit van de veronderstelling dat men aan geïsoleerde projecten werkt waarin systemen met relatief stabiele specificaties worden ontwikkeld. De wijze waarop men dergelijke projecten beheerst verloopt overeenkomstig het watervalmodel, met een nadruk op efficiency. De wijze van beheersing is op de dag van vandaag niet meer adequaat, immers de omstandigheden zijn gewijzigd. Zo zijn de huidige software-applicaties niet langer geïsoleerde produkten, maar zijn ze veelal onderdeel van een groter, geïntegreerd geheel. Het merendeel van de huidige softwarekosten zijn onderhoudskosten en geen ontwikkelkosten.

Het tweede deel van het proefschrift beschrijft hoe de beheersing zich zou moeten

uitbreiden, opdat deze beter toegerust is voor de huidige omstandigheden. De beheersing zal zich uitbreiden van traditionele 'ontwikkelbeheersing', via 'produktbeheersing' naar 'multiproduktbeheersing'. Het verschil tussen de het traditionele concept en produktbeheersing is dat beheersing zich uitbreidt over de gehele levenscyclus van een softwareprodukt. Onderhoud wordt derhalve in de beschouwing betrokken. De gevolgen van deze uitbreiding voor de organisatie worden aangegeven.

De stap van 'produktbeheersing' naar 'multiproduktbeheersing' leidt opnieuw tot een uitbreiding van de beheersing. Men beperkt zich niet langer tot het ontwikkelen en onderhouden van een aantal afzonderlijke produkten, maar men kijkt naar het samenstellen van specifieke eindprodukten binnen een gedefinieerd produktenpakket op basis van eerder ontwikkelde onderdelen. Multiproduktbeheersing legt de nadruk op het hergebruik van software via modularisatie en standaardisatie. Een organisatie die deze vorm van beheersing toepast wordt getypeerd als een softwarefabriek. De gevolgen van multiproduktbeheersing worden aangegeven. Zo wordt geschetst dat in een softwarefabriek drie soorten werk plaatsvinden, te weten ontwikkeling op klantenorder, alsmede onderhoud en ontwikkeling van herbruikbare componenten. Werklastbeheersing wordt besproken als een manier om te komen tot een goede allocatie van de schaarse capaciteit over de drie soorten werk. Tevens worden mogelijke organisatievormen van de softwarefabriek aangegeven.

Het derde deel van het proefschrift beschrijft de informatievoorziening in een softwarefabriek. Er wordt gestart met een vergelijking van reeds voorgestelde informatiesystemen voor de beheersing van software-ontwikkeling met informatiesystemen voor produktiebeheersing. Die vergelijking leidt tot de aanbeveling om een informatiesysteem voor 'engineer-to-order' produktie als referentiemodel te gebruiken. Aangegeven wordt hoe zo'n informatiesysteem aangepast dient te worden om het geschikt te maken voor de softwarefabriek. Een datamodel voor een informatiesysteem voor de softwarefabriek wordt voorgesteld. Het voorgestelde datamodel kan gebruikt worden als uitgangspunt bij de ontwikkeling van een informatiesysteem voor het beheersen van software-ontwikkeling. Het kan ook gebruikt worden als toets voor systemen die in de literatuur en in de praktijk worden voorgesteld.

De discussie over informatievoorziening spitst zich toe op de softwarefabriek. Het grootste deel van de huidige software-afdelingen opereren echter nog niet als zodanig. Daarom wordt het proefschrift afgesloten met de beschrijving van een aantal technieken die afdelingen op weg kunnen helpen naar een softwarefabriek. Het betreft hier een aantal technieken die het inzicht in de huidige software-ontwikkeling kunnen vergroten. Het vergrote inzicht dient te leiden tot een verbeterde beheersing als eerste stap op weg naar een softwarefabriek.

DANKWOORD

Een promotie-onderzoek is geen eenmansactie. Ik wil op deze plaats een aantal personen bedanken die in de afgelopen jaren een belangrijke bijdrage aan het onderzoek hebben geleverd. Het zijn in de eerste plaats de promotoren Theo Bemelmans, Hans van Vliet en Fred Heemstra. Ik heb veel van jullie geleerd.

Binnen Philips wil ik twee groepen met name bedanken. Ten eerste mijn collega's binnen Innovation Management Consulting. Ten tweede de software-ontwikkelaars met wie ik in de afgelopen jaren heb samengewerkt.

Binnen de Technische Universiteit Eindhoven wil ik twee groepen noemen. Ik denk met plezier terug aan de BAP, bestaande uit Rob Kusters, Fred Heemstra en mijzelf. Verder wil ik de leden van de vakgroep BISA bedanken. BISA is voor mij een voorbeeld van een groep waar met inzet en plezier wordt gewerkt.

CURRICILUM VITAE

Michiel van Genuchten is geboren op 29 september 1963 te Eindhoven. De middelbare school heeft hij doorlopen in Meppel en Epe. In 1981 haalde hij het diploma Atheneum-B. Hetzelfde jaar begon hij de studie Technische Bedrijfskunde aan de Technische Universiteit Eindhoven. Het afstudeerproject vond plaats bij de vakgroep Bestuurlijke Informatiesystemen en Automatisering (BISA), het afstudeerbedrijf was N.V. Philips Electronics, afdeling EDP-Industriële Toepassingen.

Na zijn afstuderen op 1 april 1987 is hij in dienst getreden bij de vakgroep BISA en bij de afdeling Corporate Organization & Efficiency van Philips. Het promotie-onderzoek is gedurende de afgelopen vier jaren in dienst van beide werkgevers uitgevoerd. Binnen Philips zijn door hem adviesprojecten uitgevoerd binnen grote software-ontwikkelafdelingen van diverse produktdivisies. Het onderzoek binnen de vakgroep BISA is uitgevoerd als lid van de onderzoeksgroep BAP (Beheersen van AutomatiseringsProjecten). Het onderzoek heeft ondermeer bestaan uit empirische studies in diverse bedrijven en heeft geleid tot publikaties in nationale en internationale tijdschriften.

STELLINGEN

behorende bij het proefschrift

Towards a software factory

van

Michiel van Genuchten

Eindhoven, 21 juni 1991

X

Mensen zijn niet zo zeer beperkt door wat zij niet kunnen. Hun mogelijkheden worden vooral begrensd door wat zij niet kunnen leren.

XI

Academische titels dienen te verjaren.