Software Engineering
Third Class
Lecture 1

Introduction to Software Engineering

1.1 Introduction
Virtually all countries now depend on complex computer-based systems. Most electrical products include a computer and controlling software. Industrial manufacturing and distribution is completely computerized, as is the financial system. Therefore, producing and maintaining software cost-effectively is essential for the functioning of national and international economies.

Software engineering is an engineering discipline whose focus is the cost effective development of high-quality software systems. Software is abstract and intangible. It is not constrained by materials, or governed by physical laws or by manufacturing processes. In some ways, this simplifies software engineering as there are no physical limitations on the potential of software. However, this lack of natural constraints means that software can easily become extremely complex and hence very difficult to understand.

The notion of software engineering was first proposed in 1968 at a conference held to discuss what was then called the 'software crisis'. This software crisis resulted directly from the introduction of new computer hardware based on integrated circuits. Their power made hitherto unrealizable computer applications a feasible proposition. The resulting software was orders of magnitude larger and more complex than previous software systems.

Early experience in building these systems showed that informal software development was not good enough. Major projects were sometimes years late. The software cost much more than predicted, was unreliable, was difficult to maintain and performed poorly. Software development was in crisis. Hardware costs were tumbling whilst software costs were rising rapidly. New techniques and methods were needed to control the complexity inherent in large software systems.

These techniques have become part of software engineering and are now widely used. However, as our ability to produce software has increased, so too has the complexity of the software systems that we need. New technologies resulting from the convergence of computers and communication systems and complex graphical user
interfaces place new demands on software engineers. As many companies still do not apply software engineering techniques effectively, too many projects still produce software that is unreliable, delivered late and over budget.

The tremendous progress since 1968 and that the development of software engineering has markedly improved software. Effective methods of software specification, design and implementation have been developed. New notations and tools reduce the effort required to produce large and complex systems.

There is no single 'ideal' approach to software engineering. The wide diversity of different types of systems and organizations that use these systems means that we need a diversity of approaches to software development. However, fundamental notions of process and system organization underlie all of these techniques, and these are the essence of software engineering.

### 1.2 Software and Software Engineering

Software is (1) instructions (computer programs) that when executed provide desired function and performance, (2) data structures that enable the programs to adequately manipulate information, and (3) documents that describe the operation and use of the programs.

Software engineers are concerned with developing software products, i.e., software which can be sold to a customer. There are two fundamental types of software product:

1. **Generic products** these are stand-alone systems that are produced by a development organization and sold on the open market to any customer who is able to buy them. Examples of this type of product include software for PCs such as databases, word processors, drawing packages and project management tools.

2. **Customised (or bespoke) products** these are systems which are commissioned by a particular customer. A software contractor develops the software especially for that customer. Examples of this type of software include control systems for electronic devices, systems written to support a particular business process and air traffic control systems.

An important difference between these types of software is that, in generic products, the organization that develops the software controls the software specification. For custom products, the specification is usually developed and controlled by the
organization that is buying the software. The software developers must work to that specification.

However, the line between these types of products is becoming increasingly unclear. More and more software companies are starting with a generic system and customizing it to the needs of a particular customer.

Software engineering is an engineering discipline that is concerned with all aspects of software production from the early stages of system specification to maintaining the system after it has gone into use. In this definition, there are two key phrases:

1. *Engineering discipline* Engineers make things work. They apply theories, methods and tools where these are appropriate. But they use them selectively and always try to discover solutions to problems even when there are no applicable theories and methods. Engineers also recognize that they must work to organizational and financial constraints, so they look for solutions within these constraints.

2. *All aspects of software production* Software engineering is not just concerned with the technical processes of software development but also with activities such as software project management and with the development of tools, methods and theories to support software production.

In general, software engineers adopt a systematic and organized approach to their work, as this is often the most effective way to produce high-quality software. However, engineering is all about selecting the most appropriate method for a set of circumstances and a more creative, less formal approach to development may be effective in some circumstances. Less formal development is particularly appropriate for the development of web-based systems, which requires a blend of software and graphical design skills.

### 1.3 Software Characteristics

To gain an understanding of software (and ultimately an understanding of software engineering), it is important to examine the characteristics of software that make it different from other things that human beings build. When hardware is built, the human creative process (analysis, design, construction, testing) is ultimately translated into a physical form. If we build a new computer, our initial sketches, formal design drawings, and prototype evolve into a physical product (chips, circuit boards, power supplies, etc.).
Software is a logical rather than a physical system element. Therefore, software has characteristics that are considerably different than those of hardware:

1- Software is developed or engineered; it is not manufactured in the classical sense. Although some similarities exist between software development and hardware manufacture, the two activities are fundamentally different. In both activities, high quality is achieved through good design, but the manufacturing phase for hardware can introduce quality problems that are nonexistent (or easily corrected) for software. Both activities are dependent on people, but the relationship between people applied and work accomplished is entirely different. Both activities require the construction of a "product" but the approaches are different.

2- Software doesn't "wear out."

Figure 1: failure rate for hardware

Figure 1 depicts failure rate as a function of time for hardware. The relationship, often called the "bathtub curve," indicates that hardware exhibits relatively high failure rates early in its life (these failures are often attributable to design or manufacturing defects); defects are corrected and the failure rate drops to a steady-state level (Ideally, quite low) for some period of time. As time passes, however, the failure rate rises again as hardware components suffer from the cumulative effects of dust, vibration, abuse, temperature extremes, and many other environmental maladies. Stated simply, the hardware begins to wear out.
Software is not susceptible to the environmental maladies that cause hardware to wear out. In theory, therefore, the failure rate curve for software should take the form of the “idealized curve” shown in Figure 2. Undiscovered defects will cause high failure rates early in the life of a program. However, these are corrected (ideally, without introducing other errors) and the curve flattens as shown.

![Figure 2: Idealized and actual failure curves for software](image)

The idealized curve is a gross oversimplification of actual failure models for software. However, the implication is clear—software doesn't wear out. But it does deteriorate. This seeming contradiction can best be explained by considering the “actual curve” shown in Figure 2. During its life, software will undergo change (maintenance). As changes are made, it is likely that some new defects will be introduced, causing the failure rate curve to spike as shown in Figure 2. Before the curve can return to the original steady-state failure rate, another change is requested, causing the curve to spike again. Slowly, the minimum failure rate level begins to rise—the software is deteriorating due to change.

Another aspect of wear illustrates the difference between hardware and software. When a hardware component wears out, it is replaced by a spare part. There are no software spare parts. Every software failure indicates an error in design or in the process through which design was translated into machine executable code.
Therefore, software maintenance involves considerably more complexity than hardware maintenance.

3- Although the industry is moving toward component-based assembly, most software continues to be custom built.

Considering the manner in which the control hardware for a computer-based product is designed and built. The design engineer draws a simple schematic of the digital circuitry, does some fundamental analysis to assure that proper function will be achieved, and then goes to the shelf where catalogs of digital components exist. Each integrated circuit (called an IC or a chip) has a part number, a defined and validated function, a well-defined interface, and a standard set of integration guidelines. After each component is selected, it can be ordered off the shelf.

As an engineering discipline evolves, a collection of standard design components is created. Standard screws and off-the-shelf integrated circuits are only two of thousands of standard components that are used by mechanical and electrical engineers as they design new systems. The reusable components have been created so that the engineer can concentrate on the truly innovative elements of a design, that is, the parts of the design that represent something new. In the hardware world, component reuse is a natural part of the engineering process. In the software world, it is something that has only begun to be achieved on a broad scale.

A software component should be designed and implemented so that it can be reused in many different programs. In the 1960s, scientific subroutine libraries that were reusable in a broad array of engineering and scientific applications had been built. These subroutine libraries reused well-defined algorithms in an effective manner but had a limited domain of application. Today, the view of reuse has been extended to encompass not only algorithms but also data structure. Modern reusable components encapsulate both data and the processing applied to the data, enabling the software engineer to create new applications from reusable parts. For example, today's graphical user interfaces are built using reusable components that enable the creation of graphics windows, pull-down menus, and a wide variety of interaction mechanisms. The data structure and processing detail required to build the interface are contained with a library of reusable components for interface construction.
1.4 The attributes of good software

As well as the services that it provides, software products have a number of other associated attributes that reflect the quality of that software. These attributes are not directly concerned with what the software does. Rather, they reflect its behavior while it is executing and the structure and organization of the source program and associated documentation. Examples of these attributes (sometimes called nonfunctional attributes) are the software’s response time to a user query and the understandability of the program code.

The specific set of attributes that you might expect from a software system obviously depends on its application. Therefore, a banking system must be secure; an interactive game must be responsive, and so on. These can be generalized into the set of attributes shown in Table 1, which are the essential characteristics of a well-designed software system.

<table>
<thead>
<tr>
<th>Product characteristic</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Maintainability</td>
<td>Software should be written in such a way that it may evolve to meet the changing needs of customers. This is a critical attribute because software change is an inevitable consequence of a changing business environment.</td>
</tr>
<tr>
<td>Dependability</td>
<td>Software dependability has a range of characteristics, including reliability, security and safety. Dependable software should not cause physical or economic damage in the event of system failure.</td>
</tr>
<tr>
<td>Efficiency</td>
<td>Software should not make wasteful use of system resources such as memory and processor cycles. Efficiency therefore includes responsiveness, processing time, utilization, etc.</td>
</tr>
<tr>
<td>Usability</td>
<td>Software must be usable, without undue effort, by the type of user for whom it is designed. This means that it should have an appropriate user interface and adequate documentation.</td>
</tr>
</tbody>
</table>

Table 1: Essential attributes of good software
**Software Engineering**

**Third Class**

**Lecture 2**

**The Software Life cycle**

**2.1 Software Processes**

A software process is a set of activities that leads to the production of a software product. These activities may involve the development of software from scratch in a standard programming language like Java or C. Increasingly, however, new software is developed by extending and modifying existing systems and by configuring and integrating off-the-shelf software or system components.

Software processes are complex and, rely on people making decisions and judgments. Because of the need for judgment and creativity, attempts to automate software processes have met with limited success. Computer-aided software engineering (CASE) tools can support some process activities. However, there is no possibility, at least in the next few years, of more extensive automation where software takes over creative design from the engineers involved in the software process.

One reason the effectiveness of CASE tools is limited is because of the immense diversity of software processes. There is no ideal process, and many organizations have developed their own approach to software development. Processes have evolved to exploit the capabilities of the people in an organization and the specific characteristics of the systems that are being developed.

Although there are many software processes, some fundamental activities are common to all software processes:

1. **Software specification.** The functionality of the software and constraints on its operation must be defined.

2. **Software design and implementation.** The software to meet the specification must be produced.

3. **Software validation.** The software must be validated to ensure that it does what the customer wants.

4. **Software evolution.** The software must evolve to meet changing customer needs.

Although there is no 'ideal' software process, there is scope for improving the software process in many organizations. Processes may include outdated techniques or may not take advantage of the best practice in industrial software engineering.
Indeed, many organizations still do not take advantage of software engineering methods in their software development.

Software processes can be improved by process standardization where the diversity in software processes across an organization is reduced. This leads to improved communication and a reduction in training time, and makes automated process support more economical. Standardization is also an important first step in introducing new software engineering methods and techniques and good software engineering practice.

2.2 Software process models

A software process model or a software engineering paradigm is an abstract representation of a software process. Each process model represents a process from a particular perspective, and thus provides only partial information about that process. In the sections that follow, a variety of different process models for software engineering are discussed.

2.2.1 The Linear Sequential Model

Sometimes called the classic life cycle or the waterfall model, the linear sequential model suggests a systematic, sequential approach to software development that begins at the system level and progresses through analysis, design, coding, testing, and support. Figure 1 illustrates the linear sequential model for software engineering.
The linear sequential model encompasses the following activities:

**System/information engineering and modeling.** Because software is always part of a larger system (or business), work begins by establishing requirements for all system elements and then allocating some subset of these requirements to software. This system view is essential when software must interact with other elements such as hardware, people, and databases. System engineering and analysis encompass requirements gathering at the system level with a small amount of top level design and analysis.

**Software requirements analysis.** The requirements gathering process is intensified and focused specifically on software. To understand the nature of the program(s) to be built, the software engineer ("analyst") must understand the information domain for the software, as well as required function, behavior, performance, and interface. Requirements for both the system and the software are documented and reviewed with the customer.

**Design.** Software design is actually a multistep process that focuses on four distinct attributes of a program: data structure, software architecture, interface representations, and procedural (algorithmic) detail. The design process translates requirements into a representation of the software that can be assessed for quality before coding begins. Like requirements, the design is documented and becomes part of the software configuration.

**Code generation.** The design must be translated into a machine-readable form. The code generation step performs this task. If design is performed in a detailed manner, code generation can be accomplished mechanistically.

**Testing.** Once code has been generated, program testing begins. The testing process focuses on the logical internals of the software, ensuring that all statements have been tested, and on the functional externals; that is, conducting tests to uncover errors and ensure that defined input will produce actual results that agree with required results.

**Support.** Software will undoubtedly undergo change after it is delivered to the customer (a possible exception is embedded software). Change will occur because errors have been encountered, because the software must be adapted to accommodate changes in its external environment (e.g., a change required because of a new operating system or peripheral device), or because the customer requires functional or performance enhancements. Software support/maintenance reapplies each of the preceding phases to an existing program rather than a new one.
The linear sequential model is the oldest and the most widely used paradigm for software engineering. However, criticism of the paradigm has caused even active supporters to question its efficacy. Among the problems that are sometimes encountered when the linear sequential model is applied are:

1. Real projects rarely follow the sequential flow that the model proposes. Although the linear model can accommodate iteration, it does so indirectly. As a result, changes can cause confusion as the project team proceeds.

2. It is often difficult for the customer to state all requirements explicitly. The linear sequential model requires this and has difficulty accommodating the natural uncertainty that exists at the beginning of many projects.

3. The customer must have patience. A working version of the program(s) will not be available until late in the project time-span. A major blunder, if undetected until the working program is reviewed, can be disastrous.

In an interesting analysis of actual projects Bradac, found that the linear nature of the classic life cycle leads to “blocking states” in which some project team members must wait for other members of the team to complete dependent tasks. In fact, the time spent waiting can exceed the time spent on productive work! The blocking state tends to be more prevalent at the beginning and end of a linear sequential process. Each of these problems is real. However, the classic life cycle paradigm has a definite and important place in software engineering work. It provides a template into which methods for analysis, design, coding, testing, and support can be placed. The classic life cycle remains a widely used procedural model for software engineering. While it does have weaknesses, it is significantly better than a haphazard approach to software development.

2.2.2. The Prototyping Model

Often, a customer defines a set of general objectives for software but does not identify detailed input, processing, or output requirements. In other cases, the developer may be unsure of the efficiency of an algorithm, the adaptability of an operating system, or the form that human/machine interaction should take. In these, and many other situations, a prototyping paradigm may offer the best approach.

The prototyping paradigm (Figure 2) begins with requirements gathering. Developer and customer meet and define the overall objectives for the software, identify
whatever requirements are known, and outline areas where further definition is mandatory. A "quick design" then occurs. The quick design focuses on a representation of those aspects of the software that will be visible to the customer/user (e.g., input approaches and output formats). The quick design leads to the construction of a prototype. The prototype is evaluated by the customer/user and used to refine requirements for the software to be developed. Iteration occurs as the prototype is tuned to satisfy the needs of the customer, while at the same time enabling the developer to better understand what needs to be done.

Figure 2: The prototyping paradigm

Ideally, the prototype serves as a mechanism for identifying software requirements. If a working prototype is built, the developer attempts to use existing program fragments or applies tools (e.g., report generators, window managers) that enable working programs to be generated quickly.

But what do we do with the prototype when it has served the purpose just described? Brooks provides an answer:

"In most projects, the first system built is barely usable. It may be too slow, too big, awkward in use or all three. There is no alternative but to start again, smarting but smarter, and build a redesigned version in which these problems are solved . . . When
a new system concept or new technology is used, one has to build a system to throw away, for even the best planning is not so omniscient as to get it right the first time. The management question, therefore, is not whether to build a pilot system and throw it away. You will do that. The only question is whether to plan in advance to build a throwaway, or to promise to deliver the throwaway to customers . . ."
The prototype can serve as "the first system." The one that Brooks recommends we throw away. But this may be an idealized view. It is true that both customers and developers like the prototyping paradigm. Users get a feel for the actual system and developers get to build something immediately. Yet, prototyping can also be problematic for the following reasons:

1. The customer sees what appears to be a working version of the software, unaware that the prototype is held together “with chewing gum and baling wire,” unaware that in the rush to get it working no one has considered overall software quality or long-term maintainability. When informed that the product must be rebuilt so that high levels of quality can be maintained, the customer cries foul and demands that "a few fixes" be applied to make the prototype a working product. Too often, software development management relents.

2. The developer often makes implementation compromises in order to get a prototype working quickly. An inappropriate operating system or programming language may be used simply because it is available and known; an inefficient algorithm may be implemented simply to demonstrate capability. After a time, the developer may become familiar with these choices and forget all the reasons why they were inappropriate. The less-than-ideal choice has now become an integral part of the system.

Although problems can occur, prototyping can be an effective paradigm for software engineering. The key is to define the rules of the game at the beginning; that is, the customer and developer must both agree that the prototype is built to serve as a mechanism for defining requirements. It is then discarded (at least in part) and the actual software is engineered with an eye toward quality and maintainability.
Software Engineering
Third Class
Lecture 3

3.1 The Evolutionary Process Models

There is growing recognition that software, like all complex systems, evolves over a period of time. Business and product requirements often change as development proceeds, making a straight path to an end product unrealistic; tight market deadlines make completion of a comprehensive software product impossible, but a limited version must be introduced to meet competitive or business pressure; a set of core product or system requirements is well understood, but the details of product or system extensions have yet to be defined. In these and similar situations, software engineers need a process model that has been explicitly designed to accommodate a product that evolves over time.

The linear sequential model is designed for straight-line development. In essence, the waterfall approach assumes that a complete system will be delivered after the linear sequence is completed. The prototyping model is designed to assist the customer (or developer) in understanding requirements. In general, it is not designed to deliver a production system.

The evolutionary nature of software is not considered in either of these classic software engineering paradigms. Evolutionary models are iterative. They are characterized in a manner that enables software engineers to develop increasingly more complete versions of the software.

3.1.1 The Incremental Model

The incremental model combines elements of the linear sequential model (applied repetitively) with the iterative philosophy of prototyping. Referring to Figure 1, the incremental model applies linear sequences in a staggered fashion as calendar time progresses. Each linear sequence produces a deliverable “increment” of the software. For example, word-processing software developed using the incremental paradigm might deliver basic file management, editing, and document production functions in the first increment; more sophisticated editing and document production capabilities in the second increment; spelling and grammar checking in the third increment; and
advanced page layout capability in the fourth increment. It should be noted that the process flow for any increment can incorporate the prototyping paradigm.

![Incremental Model Diagram](image)

**Figure 1: The incremental model**

When an incremental model is used, the first increment is often a *core product*. That is, basic requirements are addressed, but many supplementary features (some known, others unknown) remain undelivered. The core product is used by the customer (or undergoes detailed review). As a result of use and/or evaluation, a plan is developed for the next increment. The plan addresses the modification of the core product to better meet the needs of the customer and the delivery of additional features and functionality. This process is repeated following the delivery of each increment, until the complete product is produced.

The incremental process model, like prototyping and other evolutionary approaches, is iterative in nature. But unlike prototyping, the incremental model focuses on the delivery of an operational product with each increment. Early increments are stripped down versions of the final product, but they do provide capability that serves the user and also provide a platform for evaluation by the user.

Incremental development is particularly useful when staffing is unavailable for a complete implementation by the business deadline that has been established for the project. Early increments can be implemented with fewer people. If the core product is well received, then additional staff (if required) can be added to implement the next
increment. In addition, increments can be planned to manage technical risks. For example, a major system might require the availability of new hardware that is under development and whose delivery date is uncertain. It might be possible to plan early increments in a way that avoids the use of this hardware, thereby enabling partial functionality to be delivered to end-users without inordinate delay.

### 3.1.2 The Spiral model

The *spiral model*, originally proposed by Boehm, is an evolutionary software process model that couples the iterative nature of prototyping with the controlled and systematic aspects of the linear sequential model. It provides the potential for rapid development of incremental versions of the software. Using the spiral model, software is developed in a series of incremental releases. During early iterations, the incremental release might be a paper model or prototype. During later iterations, increasingly more complete versions of the engineered system are produced. A spiral model is divided into a number of framework activities, also called *task regions*. Typically, there are between three and six task regions. Figure 2 depicts a spiral model that contains six task regions:

![Figure 2 A typical spiral model](image-url)
Customer communication—tasks required to establish effective communication between developer and customer.

Planning—tasks required to define resources, timelines, and other project related information.

Risk analysis—tasks required to assess both technical and management risks.

Engineering—tasks required to build one or more representations of the application.

Construction and release—tasks required to construct, test, install, and provide user support (e.g., documentation and training).

Customer evaluation—tasks required to obtain customer feedback based on evaluation of the software representations created during the engineering stage and implemented during the installation stage.

As this evolutionary process begins, the software engineering team moves around the spiral in a clockwise direction, beginning at the center. The first circuit around the spiral might result in the development of a product specification; subsequent passes around the spiral might be used to develop a prototype and then progressively more sophisticated versions of the software. Each pass through the planning region results in adjustments to the project plan. Cost and schedule are adjusted based on feedback derived from customer evaluation. In addition, the project manager adjusts the planned number of iterations required to complete the software.

Unlike classical process models that end when software is delivered, the spiral model can be adapted to apply throughout the life of the computer software.

The spiral model is a realistic approach to the development of large-scale systems and software. Because software evolves as the process progresses, the developer and customer better understand and react to risks at each evolutionary level. The spiral model uses prototyping as a risk reduction mechanism but, more important, enables the developer to apply the prototyping approach at any stage in the evolution of the product. It maintains the systematic stepwise approach suggested by the classic life cycle but incorporates it into an iterative framework that more realistically reflects the real world. The spiral model demands a direct consideration of technical risks at all stages of the project and, if properly applied, should reduce risks before they become problematic.
But like other paradigms, the spiral model is not a panacea. It may be difficult to convince customers (particularly in contract situations) that the evolutionary approach is controllable. It demands considerable risk assessment expertise and relies on this expertise for success. If a major risk is not uncovered and managed, problems will undoubtedly occur. Finally, the model has not been used as widely as the linear sequential or prototyping paradigms. It will take a number of years before efficacy of this important paradigm can be determined with absolute certainty.

3.2 The formal methods model

The formal methods model encompasses a set of activities that leads to formal mathematical specification of computer software. Formal methods enable a software engineer to specify, develop, and verify a computer-based system by applying a rigorous, mathematical notation. A variation on this approach, called cleanroom software engineering, is currently applied by some software development organizations.

When formal methods are used during development, they provide a mechanism for eliminating many of the problems that are difficult to overcome using other software engineering paradigms. Ambiguity, incompleteness, and inconsistency can be discovered and corrected more easily, not through ad hoc review but through the application of mathematical analysis. When formal methods are used during design, they serve as a basis for program verification and therefore enable the software engineer to discover and correct errors that might go undetected.

Although it is not destined to become a mainstream approach, the formal methods model offers the promise of defect-free software. Yet, the following concerns about its applicability in a business environment have been voiced:

1. The development of formal models is currently quite time consuming and expensive.

2. Because few software developers have the necessary background to apply formal methods, extensive training is required.

3. It is difficult to use the models as a communication mechanism for technically unsophisticated customers.

These concerns notwithstanding, it is likely that the formal methods approach will gain adherents among software developers who must build safety-critical software
(e.g., developers of medical devices) and among developers that would suffer severe economic hardship should software errors occur.

### 3.3 Component Based Software Engineering

In the majority of software projects, there is some software reuse. This usually happens informally when people working on the project know of designs or code which is similar to that required. They look for these, modify them as needed and incorporate them into their system.

This informal reuse takes place irrespective of the development process that is used. However, in the last few years, an approach to software development called component-based software engineering (CBSE), which relies on reuse, has emerged and is becoming increasingly used.

This reuse-oriented approach relies on a large base of reusable software components and some integrating framework for these components. Sometimes, these components are systems in their own right (COTS or commercial off-the-shelf systems) that may provide specific functionality such as text formatting or numeric calculation. The generic process model for CBSE is shown in Figure 3.

![Figure 3 Component-based software engineering](image)

While the initial requirements specification stage and the validation stage are comparable with other processes, the intermediate stages in a reuse-oriented process are different. These stages are:

1. **Component analysis.** Given the requirements specification, a search is made for components to implement that specification. Usually, there is no exact match, and the components that may be used only provide some of the functionality required.
2. **Requirements modification.** During this stage, the requirements are analyzed using information about the components that have been discovered. They are then modified to reflect the available components. Where modifications are impossible, the component analysis activity may be re-entered to search for alternative solutions.

3. **System design with reuse.** During this phase, the framework of the system is designed or an existing framework is reused. The designers take into account the components that are reused and organize the framework to cater to this. Some new software may have to be designed if reusable components are not available.

4. **Development and integration** Software that cannot be externally procured is developed, and the components and COTS systems are integrated to create the new system. System integration, in this model, may be part of the development process rather than a separate activity.

Component-based software engineering has the obvious advantage of reducing the amount of software to be developed and so reducing cost and risks. It usually also leads to faster delivery of the software. However, requirements compromises are inevitable and this may lead to a system that does not meet the real needs of users. Furthermore, some control over the system evolution is lost as new versions of the reusable components are not under the control of the organization using them.
Agile methods

Agile methods allowed the development team to focus on the software itself rather than on its design and documentation. Agile methods universally rely on an iterative approach to software specification, development and delivery, and were designed primarily to support business application development where the system requirements usually changed rapidly during the development process. They are intended to deliver working software quickly to customers, who can then propose new and changed requirements to be included in later iterations of the system. Probably the best-known agile method is extreme programming.

Agile methods share a set of principles:

<table>
<thead>
<tr>
<th>Principle</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Customer involvement</td>
<td>Customers should be closely involved throughout the development process. Their role is to provide and prioritise new system requirements and to evaluate the iterations of the system.</td>
</tr>
<tr>
<td>Incremental delivery</td>
<td>The software is developed in increments with the customer specifying the requirements to be included in each increment.</td>
</tr>
<tr>
<td>People not process</td>
<td>The skills of the development team should be recognised and exploited. Team members should be left to develop their own ways of working without prescriptive processes.</td>
</tr>
<tr>
<td>Embrace change</td>
<td>Expect the system requirements to change, so design the system to accommodate these changes.</td>
</tr>
<tr>
<td>Maintain simplicity</td>
<td>Focus on simplicity in both the software being developed and in the development process. Wherever possible, actively work to eliminate complexity from the system.</td>
</tr>
</tbody>
</table>
In practice, however, the principles underlying agile methods are sometimes difficult to realize:

1. While the idea of customer involvement in the development process is an attractive one, its success depends on having a customer who is willing and able to spend time with the development team and who can represent all system stakeholders. Frequently, the customer representatives are subject to other pressures and cannot take full part in the software development.

2. Individual team members may not have suitable personalities for the intense involvement that is typical of agile methods. They may therefore not interact well with other team members.

3. Prioritising changes can be extremely difficult, especially in systems where there are many stakeholders. Typically, each stakeholder gives different priorities to different changes.

4. Maintaining simplicity requires extra work. Under pressure from delivery schedules, the team members may not have time to carry out desirable system simplifications.

Another, nontechnical problem, which is a general problem with incremental development and delivery, occurs when the system customer uses an outside organization for system development. The software requirements document is usually part of the contract between the customer and the supplier. Because incremental specification is inherent in agile methods, writing contracts for this type of development may be difficult.

Consequently, agile methods have to rely on contracts where the customer pays for the time required for system development rather than the development of a specific set of requirements. So long as all goes well, this benefits both the customer and the developer. However, if problems arise there may be difficult disputes over who is to blame and who should pay for the extra time and resources required to resolve the problems.

All methods have limits, and agile methods are only suitable for some types of system development. They are best suited to the development of small or medium-sized business systems and personal computer products. They are not well suited to large-scale systems development with the development teams in different places and where there may be complex interactions with other hardware and software systems. Nor
should agile methods be used for critical systems development where a detailed analysis of all of the system requirements is necessary to understand their safety or security implications.

**Extreme programming**

Extreme programming (XP) is perhaps the best known and most widely used of the agile methods. The name was coined by Beck (Beck, 2000) because the approach was developed by pushing recognized good practice, such as iterative development, and customer involvement to 'extreme' levels.

In extreme programming, all requirements are expressed as scenarios (called user stories), which are implemented directly as a series of tasks. Programmers work in pairs and develop tests for each task before writing the code. All tests must be successfully executed when new code is integrated into the system. There is a short time gap between releases of the system. Figure 4.1 illustrates the XP process to produce an increment of the system that is being developed.

![Extreme programming release cycle](image)

Figure 4.1 Extreme programming release cycle

Extreme programming involves a number of practices that fit into the principles of agile methods:

1. Incremental development is supported through small, frequent releases of the system and by an approach to requirements description based on customer stories or scenarios that can be the basis for process planning.
2. Customer involvement is supported through the full-time engagement of the customer in the development team. The customer representative takes part in the development and is responsible for defining acceptance tests for the system.

3. People, not process, are supported through pair programming, collective ownership of the system code, and a sustainable development process that does not involve excessively long working hours.

4. Change is supported through regular system releases, test-first development and continuous integration.

5. Maintaining simplicity is supported through constant refactoring to improve code quality and using simple designs that do not anticipate future changes to the system.

In an XP process, customers are intimately involved in specifying and prioritizing system requirements. The requirements are not specified as lists of required system function,. Rather, the system customer is part of the development team and discusses scenario, with other team members. Together, they develop a 'story card' that encapsulates the customer needs. The development team then aims to implement that scenario in a future release of the software.

Once the story cards have been developed, the development team breaks these down into tasks and estimates the effort and resources required for implementation. The customer then prioritizes the stories for implementation, choosing those stories that can be used immediately to deliver useful business support. Of course, as requirements change, the unimplemented stories change or may be discarded. If changes are required for a system that has already been delivered, new story cards are developed and, again, the customer decides whether these changes should have priority over new functionality. Extreme programming takes an extreme approach to iterative development. New versions of the software may be built several times per day and increments are delivered to customers roughly every two weeks.

When a programmer builds the system to create a new version, he or she must run all existing automated tests as well as the tests for the new functionality. The new build of the software is accepted only if all tests execute successfully.

A fundamental precept of traditional software engineering is that you should design for change. That is, you should anticipate future changes to the software and design it so that these changes can be easily implemented. Extreme programming, however, has discarded this principle on the basis that designing for change is often wasted
effort. The changes anticipated often never materialize and completely different change requests are actually made.

The problem with unanticipated change implementation is that it tends to degrade the software structure, so changes become harder and harder to implement. Extreme programming tackles this problem by suggesting that the software should be constantly refactored. This means that the programming team looks for possible improvements to the software and implements them immediately. Therefore, the software should always be easy to understand and change as new stories are implemented.

**XP Testing**

The creation of unit tests before coding commences is a key element of the XP approach. The unit tests that are created should be implemented using a framework that enables them to be automated (hence, they can be executed easily and repeatedly). This encourages a regression testing strategy whenever code is modified (which is often, given the XP refactoring philosophy).

As the individual unit tests are organized into a “universal testing suite” integration and validation testing of the system can occur on a daily basis. This provides the XP team with a continual indication of progress and also can raise warning flags early if things go awry. Wells states: “Fixing small problems every few hours takes less time than fixing huge problems just before the deadline”.

XP acceptance tests, also called customer tests, are specified by the customer and focus on overall system features and functionality that are visible and reviewable by the customer. Acceptance tests are derived from user stories that have been implemented as part of a software release.

**Pair programming**

Another innovative practice that has been introduced is that programmers work in pairs to develop the software. They actually sit together at the same workstation to develop the software. Development does not always involve the same pair of people working together. Rather, the idea is that pairs are created dynamically so that all team members may work with other members in a programming pair during the development process.

The use of pair programming has a number of advantages:
1. It supports the idea of common ownership and responsibility for the system. This reflects Weinberg's idea of egoless programming where the software is owned by the team as a whole and individuals are not held responsible for problems with the code. Instead, the team has collective responsibility for resolving these problems.

2. It acts as an informal review process because each line of code is looked at by at least two people. Code inspections and reviews are very successful in discovering a high percentage of software errors. However, they are time consuming to organize and, typically, introduce delays into the development process. While pair programming is a less formal process that probably doesn't find so many errors, it is a much cheaper inspection process than formal program inspections.

3. It helps support refactoring, which is a process of software improvement. A principle of XP is that the software should be constantly refactored. That is, parts of the code should be rewritten to improve their clarity or structure. The difficulty of implementing this in a normal development environment is that this is effort that is expended for long-term benefit, and an individual who practices refactoring may be judged less efficient than one who simply carries on developing code. Where pair programming and collective ownership are used, others gain immediately from the refactoring so they are likely to support the process.

You might think that pair programming is less efficient than individual programming and that a pair of developers would produce half as much code as two individuals working alone. Studies of XP developments, however, do not bear this out. Development productivity with pair programming seems to be comparable with that of two people working independently. The reasons for this are that pairs discuss the software before development so probably have fewer false starts and less rework, and that the number of errors avoided by the informal inspection is such that less time is spent repairing bugs discovered during the testing process.

**RAD**

Although agile methods as an approach to iterative development have received a great deal of attention in the last few years, business systems have been developed iteratively for many years using rapid application development techniques. Rapid application development (RAD) techniques evolved from so-called fourth-generation
languages in the 1980s and are used for developing applications that are data intensive. Consequently, they are usually organized as a set of tools that allow data to be created, searched, displayed and presented in reports. Figure 4.2 illustrates a typical organization for a RAD system.

The tools that are included in a RAD environment are:

1. A Database programming language that embeds knowledge of the database structure; and includes fundamental database manipulation operations. SQL is the standard database programming language. The SQL commands may be input directly or generated automatically from forms filled in by an end-user.
2. An interface generator, which is used to create forms for data input and display.
3. Links to office applications such as a spreadsheet for the analysis and manipulation of numeric information or a word processor for report template creation.
4. A report generator, which is used to define and create reports from information in the database.

RAD systems are successful because there is a great deal of commonality across business applications. In essence, these applications are often concerned with updating a database and producing reports from the information in the database. Standard forms are used for input and output. RAD systems are geared towards producing interactive applications that rely on abstracting information from an organizational database, presenting it to end-users on their terminal or workstation, and updating the database with changes made by users.

Figure 4.2: A rapid application development environment
Software project management is an essential part of software engineering. Good management cannot guarantee project success. However, bad management usually results in project failure: The software is delivered late, costs more than originally estimated and fails to meet its requirements.

We need software project management because professional software engineering is always subject to organizational budget and schedule constraints. The software project manager's job is to ensure that the software project meets these constraints and delivers software that contributes to the goals of the company developing the software.

Software managers do the same kind of job as other engineering project managers. However, software engineering is different from other types of engineering in a number of ways. These distinctions make software management particularly difficult. Some of the differences are:

1. **The product is intangible.** The manager of a shipbuilding project or of a civil engineering project can see the product being developed. If a schedule slips, the effect on the product is visible-parts of the structure are obviously unfinished. Software is intangible. It cannot be seen or touched. Software project managers cannot see progress. They rely on others to produce the documentation needed to review progress.

2. **There are no standard software processes.** In engineering disciplines with a long history, the process is tried and tested. The engineering process for some types of system, such as bridges and buildings is well understood. However, software processes vary dramatically from one organization to another. Although our understanding of these processes has developed significantly in the past few years, we still cannot reliably predict when a particular software process is likely
to cause development problems. This is especially true when the software project is part of a wider systems engineering project.

3. **Large software projects are often one-off projects.** Large software projects are usually different in some ways from previous projects. Therefore, even managers who have a large body of previous experience may find it difficult to anticipate problems. Furthermore, rapid technological changes in computers and communications can make a manager's experience obsolete. Lessons learned from previous projects may not be transferable to new projects.

**Management activities**

It is impossible to write a standard job description for a software manager. The job varies tremendously depending on the organization and the software product being developed. However, most managers take responsibility at some stage for some or all of the following activities:

- Proposal writing
- Project planning and scheduling
- Project cost
- Project monitoring and reviews
- Personnel selection and evaluation
- Report writing and presentations

The first stage in a software project may involve **writing a proposal** to win a contract to carry out the work. The proposal describes the objectives of the project and how it will be carried out. It usually includes cost and schedule estimates, and justifies why the project contract should be awarded to a particular organization or team. Proposal writing is a critical task as the existence of many software organizations depends on having enough proposals accepted and contracts awarded. There can be no set guidelines for this task; proposal writing is a skill that you acquire through practice and experience.

**Project planning** is concerned with identifying the activities, milestones and deliverables produced by a project. A plan is drawn up to guide the development towards the project goals.
Cost estimation is a related activity that is concerned with estimating the resources required to accomplish the project plan.

Project monitoring is a continuing project activity. The manager must keep track of the progress of the project and compare actual and planned progress and costs. Although most organizations have formal mechanisms for monitoring, a skilled manager can often form a clear picture of what is going on through informal discussions with project staff.

Informal monitoring can often predict potential project problems by revealing difficulties as they occur. For example, daily discussions with project staff might reveal a particular problem in finding some software fault. Rather than waiting for a schedule slippage to be reported, the software manager might assign some expert to the problem or might decide that it should be programmed around.

During a project, it is normal to have a number of formal project management reviews. They are concerned with reviewing overall progress and technical development of the project and checking whether the project and the goals of the organization paying for the software are still aligned.

The outcome of a review may be a decision to cancel a project. The development time for a large software project may be several years. During that time, organizational objectives are almost certain to change. These changes may mean that the software is no longer required or that the original project requirements are inappropriate.

Management may decide to stop software development or to change the project to accommodate the changes to the organizational objectives.

Project managers usually have to select people to work on their project. Ideally, skilled staff with appropriate experience will be available to work on the project.

However, in most cases, managers have to settle for a less-than-ideal project team.

The reasons for this are:

1. The project budget may not cover the use of highly paid staff. Less experienced, less well-paid staff may have to be used.
2. Staff with the appropriate experience may not be available either within an organization or externally. It may be impossible to recruit new staff to the project.
Within the organization, the best people may already be allocated to other projects.

3. The organization may wish to develop the skills of its employees. Inexperienced staff may be assigned to a project to learn and to gain experience.

The software manager has to work within these constraints when selecting project staff. However, problems are likely unless at least one project member has some experience with the type of system being developed. Without this experience, many simple mistakes are likely to be made.

Project managers are usually responsible for reporting on the project to both the client and contractor organizations. They have to write concise, coherent documents that abstract critical information from detailed project reports. They must be able to present this information during progress reviews. Consequently, if you are a project manager, you have to be able to communicate effectively both orally and in writing.

**Project planning**

Effective management of a software project depends on thoroughly planning the progress of the project. Managers must anticipate problems that might arise and prepare tentative solutions to those problems. A plan, drawn up at the start of a project, should be used as the driver for the project. This initial plan should be the best possible plan given the available information. It evolves as the project progresses and better information becomes available.

The pseudo-code shown in Figure 1 sets out a project planning process for software development. It shows that planning is an iterative process, which is only complete when the project itself is complete. As project information becomes available during the project, the plan should be regularly revised. The goals of the business are an important factor that must be considered when formulating the project plan. As these change, the project's goals also change so changes to the project plan are necessary.
Establish the project constraints
Make initial assessments of the project parameters
Define project milestones and deliverables
while project has not been completed or cancelled loop
  Draw up project schedule
  Initiate activities according to schedule
  Wait (for a while)
  Review project progress
  Revise estimates of project parameters
  Update the project schedule
  Renegotiate project constraints and deliverables
  If (problems arise) then
    Initiate technical review and possible revision
  end If
end loop

Figure 1: Project planning

At the beginning of a planning process, you should assess the constraints (required delivery date, staff available, overall budget, etc.) affecting the project. In conjunction with this, you should estimate project parameters such as its structure, size, and distribution of functions. You next define the progress milestones and deliverables. The process then enters a loop. You draw up an estimated schedule for the project and the activities defined in the schedule are started or given permission to continue. After some time (usually about two to three weeks), you should review progress and note discrepancies from the planned schedule. Because initial estimates of project parameters are tentative, you will always have to modify the original plan. As more information becomes available. You revise your original assumptions about the project and the project schedule. If the project is delayed, you may have to renegotiate the project constraint and deliverables with the customer. If this renegotiation is unsuccessful and the schedule cannot be met, a project technical review may be held. The objective of
this review is to find an alternative approach that falls within the project constraints and meets the schedule.

Of course, you should never assume that everything will always go well. Problems of some description nearly always arise during a project. Your initial assumptions and scheduling should be pessimistic rather than optimistic. There should be sufficient contingency built into your plan so that the project constraints and milestones need not be renegotiated every time round the planning loop.

**The project plan**
The project plan sets out the resources available to the project, the work breakdown and a schedule for carrying out the work.

The details of the project plan vary depending on the type of project and organization. However, most plans should include the following sections:

1. **Introduction** This briefly describes the objectives of the project and sets out the constraints (e.g., budget, time, etc.) that affect the project management.
2. **Project organization** This describes the way in which the development team is organized, the people involved and their roles in the team.
3. **Risk analysis** This describes possible project risks, the likelihood of these risks arising and the risk reduction strategies that are proposed.
4. **Hardware and software resource requirements** This specifies the hardware and the support software required to carry out the development. If hardware has to be bought, estimates of the prices and the delivery schedule may be included.
5. **Work breakdown** This sets out the breakdown of the project into activities and identifies the milestones and deliverables associated with each activity.
6. **Project schedule** This shows the dependencies between activities, the estimated time required to reach each milestone and the allocation of people to activities.
7. **Monitoring and reporting mechanisms** This defines the management reports that should be produced, when these should be produced and the project monitoring mechanisms used.
You should regularly revise the project plan during the project. Some parts, such as the project schedule, will change frequently; other parts will be more stable. To simplify revisions, you should organize the document into separate sections that can be individually replaced as the plan evolves.

Milestones and deliverables
Managers need information to do their job. Because software is intangible, this information can only be provided as reports and documents that describe the state of the software being developed. Without this information, it is impossible to assess how well the work is progressing, and cost estimates and schedules cannot be updated.

When planning a project, you should establish a series of milestones. Where a milestone is a recognizable end-point of a software process activity. At each milestone, there should be a formal output, such as a report, that can be presented to management. Milestone reports need not be large documents. They may simply be a short report of what has been completed. Milestones should represent the end of a distinct, logical stage in the project. Indefinite milestones such as 'Coding 80% complete' that can't be checked are useless for project management. You can't check whether this state has been achieved because the amount of code that still has to be developed is uncertain.

A deliverable is a project result that is delivered to the customer. It is usually delivered at the end of some major project phase such as specification or design. Deliverables are usually milestones, but milestones need not be deliverables.

Milestones may be internal project results that are used by the project manager to check project progress but which are not delivered to the customer. To establish milestones, the software process must be broken down into basic activities with associated outputs. For example, Figure 2 shows possible activities involved in requirements specification when prototyping is used to help validate requirements. The milestones in this case are the completion of the outputs for each activity. The project
deliverables, which are delivered to the customer, are the requirements definition and the requirements specification.
Project Scheduling

Project scheduling is one of the most difficult jobs for a project manager. Managers estimate the time and resources required to complete activities and organize them into a coherent sequence. Unless the project being scheduled is similar to a previous project, previous estimates are an uncertain basis for new project scheduling.

Schedule estimation is further complicated by the fact that different projects may use different design methods and implementation languages. If the project is technically advanced, initial estimates will almost certainly be optimistic even when you try to consider all eventualities. In this respect, software scheduling is no different from scheduling any other type of large advanced project. New aircraft, bridges and even new models of cars are frequently late because of unanticipated problems. Schedules, therefore, must be continually updated as better progress information becomes available.

Project scheduling (Figure 1) involves separating the total work involved in a project into separate activities and judging the time required to complete these activities. Usually, some of these activities are carried out in parallel. You have to coordinate these parallel activities and organize the work so that the workforce is used optimally. It's important to avoid a situation where the whole project is delayed because a critical task is unfinished.

![Figure 1: The project scheduling process](image-url)
Project activities should normally last at least a week. Finer subdivision means that a disproportionate amount of time must be spent on estimating and chart revision. It is also useful to set a maximum amount of time for any activity of about 8 to 10 weeks. If it takes longer than this, it should be subdivided for project planning and scheduling.

When you are estimating schedules, you should not assume that every stage of the project will be problem free. People working on a project may fall ill or may leave, hardware may break down, and essential support software or hardware may be delivered late. If the project is new and technically advanced, certain parts of it may turn out to be more difficult and take longer than originally anticipated.

As well as calendar time, you also have to estimate the resources needed to complete each task. The principal resource is the human effort required. Other resources may be the disk space required on a server, the time required on specialized hardware such as a simulator, and the travel budget required for project staff.

A good rule of thumb is to estimate as if nothing will go wrong, then increase your estimate to cover anticipated problems. A further contingency factor to cover unanticipated problems may also be added to the estimate. This extra contingency factor depends on the type of project, the process parameters (deadline, standards, etc.) and the quality and experience of the software engineers working on the project. ("Add 30% to your original estimate for anticipated problems then another 20% to cover things you hadn't thought of").

Project schedules are usually represented as a set of charts showing the work breakdown, activities dependencies and staff allocations.

Software management tools, such as Microsoft Project, are usually used to automate chart production.

**Bar charts and activity networks**

Bar charts and activity networks are graphical notations that are used to illustrate the project schedule. Bar charts show who is responsible for each activity and when the activity is scheduled to begin and end. Activity networks show the dependencies between the different activities making up a project. Bar charts and activity charts can be generated automatically from a database of project information using a project management tool.
Table 1 shows activities, their duration, and activity interdependencies.

<table>
<thead>
<tr>
<th>Task</th>
<th>Duration (days)</th>
<th>Dependencies</th>
</tr>
</thead>
<tbody>
<tr>
<td>T1</td>
<td>8</td>
<td></td>
</tr>
<tr>
<td>T2</td>
<td>15</td>
<td></td>
</tr>
<tr>
<td>T3</td>
<td>15</td>
<td>T1 (M1)</td>
</tr>
<tr>
<td>T4</td>
<td>10</td>
<td></td>
</tr>
<tr>
<td>T5</td>
<td>10</td>
<td>T2, T4 (M2)</td>
</tr>
<tr>
<td>T6</td>
<td>5</td>
<td>T1, T2 (M3)</td>
</tr>
<tr>
<td>T7</td>
<td>20</td>
<td>T1 (M1)</td>
</tr>
<tr>
<td>T8</td>
<td>25</td>
<td>T4 (M5)</td>
</tr>
<tr>
<td>T9</td>
<td>15</td>
<td>T3, T6 (M4)</td>
</tr>
<tr>
<td>T10</td>
<td>15</td>
<td>T5, T7 (M7)</td>
</tr>
<tr>
<td>T11</td>
<td>7</td>
<td>T9 (M6)</td>
</tr>
<tr>
<td>T12</td>
<td>10</td>
<td>T11 (M8)</td>
</tr>
</tbody>
</table>

From Table 1, you can see that Activity T3 is dependent on Activity T1. This means that T1 must be completed before T3 starts. For example, T1 might be the preparation of a component design and T3, the implementation of that design. Before implementation starts, the design should be complete.

Given the dependencies and estimated duration of activities, an activity chart that shows activity sequences may be generated (Figure 2). This shows which activities can be carried out in parallel and which must be executed in sequence because of a dependency on an earlier activity. Activities are represented as rectangles; milestones and project deliverables are shown with rounded corners. Dates in this diagram show the start date of the activity and are written in British style, where the day precedes the month. You should read the chart from left to right and from top to bottom.
In the project management tool used to produce this chart, all activities must end in milestones. An activity may start when its preceding milestone (which may depend on several activities) has been reached. Therefore, in the third column in Table 1 shows the corresponding milestone (e.g., M5) that is reached when the tasks finish (see Figure 2).

Before progress can be made from one milestone to another, all paths leading to it must be complete. For example, when activities T3 and T6 are finished, then activity T9, shown in Figure 2, can start.

The minimum time required to finish the project can be estimated by considering the longest path in the activity graph (the critical path). In this case, it is 11 weeks of elapsed time or 55 working days. In Figure 2, the critical path is shown as a sequence of emboldened boxes. The critical path is the sequence of dependent activities that defines the time required to complete the project. The overall schedule of the project depends on the critical path. Any slippage in the completion in any critical activity causes project delays because the following activities cannot start until the delayed activity has been completed.
However, delays in activities that do not lie on the critical path do not necessarily cause an overall schedule slippage. So long as these delays do not extend these activities so much that the total time for that activity plus future dependent activities does not exceed the critical path, the project schedule will not be affected. For example, if T8 is delayed by two weeks, it will not affect the final completion date of the project because it does not lie on the critical path. Most project management tools compute the allowed delays, as shown in the project bar chart.

Managers also use activity charts when allocating project work. They can provide insights into activity dependencies that are not intuitively obvious. It may be possible to modify the system design so that the critical path is shortened. The project schedule may be shortened because of the reduced amount of time spent waiting for activities to finish.

Inevitably, initial project schedules will be incorrect. As a project develops, estimates should be compared with actual elapsed time. This comparison can be used as a basis for revising the schedule for later parts of the project. When actual figures are known, the activity chart should be reviewed. Later project activities may then be reorganized to reduce the length of the critical path.

Figure 3 is a complementary way of representing project schedule information.
It is a bar chart showing a project calendar and the start and finish dates of activities. Sometimes these are called *Gantt charts*, after their inventor. Reading from left to right, the bar chart clearly shows when activities start and end.

Some of the activities shown in the bar chart in Figure 3 are followed by a shaded bar whose length is computed by the scheduling tool. This highlights the flexibility in the completion date of these activities. If an activity does not complete on time, the critical path will not be affected until the end of the period marked by the shaded bar. Activities that lie on the critical path have no margin of error and can be identified because they have no associated shaded bar.

In addition to considering schedules, as a project manager you must also consider resource allocation and, in particular, the allocation of staff to project activities. This allocation can also be input to project management tools and a bar chart generated that shows when staff are employed on the project (Figure 4). People don't have to be assigned to a project at all times. During intervening periods they may be on holiday, working on other projects, attending training courses or engaging in some other activity.

![Figure 4: Staff allocation vs. time chart](image)

Large organizations usually employ a number of specialists who work on a project when needed. In Figure 4, you can see that Mary and Jim are specialists who work on only a single task in the project. This can cause scheduling problems. If one project is delayed while a specialist is working on it, this may have a knock-on effect on other projects. They may also be delayed because the specialist is not available.
Analysis Concepts and Principles

Software Requirements Engineering
Software requirements engineering is a process of discovery, refinement, modeling, and specification.
Both the software engineer and customer take an active role in software requirements engineering—a set of activities that is often referred to as analysis. The customer attempts to reformulate a sometimes nebulous system-level description of data, function, and behavior into concrete detail. The developer acts as interrogator, consultant, problem solver, and negotiator.

1.1 Requirements Analysis
Requirements analysis is a software engineering task that bridges the gap between system level requirements engineering and software design (Figure 1).

![Figure 1 Analysis as a bridge between system engineering and software design](image)

Requirements engineering activities result in:
1. The specification of software’s operational characteristics (function, data, and behavior).
2. Indicate software's interface with other system elements
3. Establish constraints that software must meet. Requirements analysis allows the software engineer (analyst) to refine the software allocation and build models of the data, functional, and behavioral domains that will be treated by software. Requirements analysis provides the software designer with a representation of information, function, and behavior that can be translated to data, architectural, interface, and component-level designs. Finally, the requirements specification provides the developer and the customer with the means to assess quality once software is built.

Software requirements analysis may be divided into five areas of effort:

1. Problem recognition,
2. Evaluation and synthesis
3. Modeling,
4. Specification
5. Review.

1.2 Requirements Elicitation for Software

Before requirements can be analyzed, modeled, or specified they must be gathered through an elicitation process. A customer has a problem that may be amenable to a computer-based solution. A developer responds to the customer's request for help. Communication has begun.

1.2.1 Initiating the Process

The most commonly used requirements elicitation technique is to conduct a meeting or interview. During the meeting Gause and Weinberg suggest that the analyst start by asking context-free questions. That is, a set of questions that will lead to a basic understanding of the problem, the people who want a solution, the nature of the solution that is desired, and the effectiveness of the first encounter itself. The first set of context-free questions focuses on the customer, the overall goals, and the benefits.

The next set of questions enables the analyst to gain a better understanding of the problem and the customer to voice his or her perceptions about a solution.

The final set of questions focuses on the effectiveness of the meeting. Gause and Weinberg call these meta-questions.
The Q&A session should be used for the first encounter only and then replaced by a meeting format that combines elements of problem solving, negotiation, and specification.

### 1.2.2 Facilitated Application Specification Techniques

A number of independent investigators have developed a team-oriented approach to requirements gathering that is applied during early stages of analysis and specification. Called *facilitated application specification techniques* (FAST), this approach encourages the creation of a joint team of customers and developers who work together to identify the problem, propose elements of the solution, negotiate different approaches and specify a preliminary set of solution requirements.

### 1.2.3 Quality Function Deployment

*Quality function deployment* (QFD) is a quality management technique that translates the needs of the customer into technical requirements for software. QFD “concentrates on maximizing customer satisfaction from the software engineering process.” To accomplish this, QFD emphasizes an understanding of what is valuable to the customer and then deploys these values throughout the engineering process. QFD identifies three types of requirements:

- **Normal requirements.** The objectives and goals that are stated for a product or system during meetings with the customer. If these requirements are present, the customer is satisfied. Examples of normal requirements might be requested types of graphical displays and specific system functions.

- **Expected requirements.** These requirements are implicit to the product or system and may be so fundamental that the customer does not explicitly state them. Their absence will be a cause for significant dissatisfaction. Examples of expected requirements are: ease of human/machine interaction, overall operational correctness and reliability, and ease of software installation.

- **Exciting requirements.** These features go beyond the customer’s expectations and prove to be very satisfying when present. For example, word processing software is requested with standard features. The delivered product contains a number of page layout capabilities that are quite pleasing and unexpected.
In meetings with the customer, *function deployment* is used to determine the value of each function that is required for the system. *Information deployment* identifies both the data objects and events that the system must consume and produce. These are tied to the functions. Finally, *task deployment* examines the behavior of the system or product within the context of its environment. *Value analysis* is conducted to determine the relative priority of requirements determined during each of the three deployments.

QFD uses customer interviews and observation, surveys, and examination of historical data (e.g., problem reports) as raw data for the requirements gathering activity. These data are then translated into a table of requirements—called the *customer voice table*—that is reviewed with the customer. A variety of diagrams, matrices, and evaluation methods are then used to extract expected requirements and to attempt to derive exciting requirements.

### 1.2.4 Use-Cases

As requirements are gathered as part of informal meetings, FAST, or QFD, the software engineer (analyst) can create a set of scenarios that identify a thread of usage for the system to be constructed. The scenarios, often called *use-case*, provide a description of how the system will be used.

To create a use-case, the analyst must first identify the different types of people (or devices) that use the system or product. These *actors* actually represent roles that people (or devices) play as the system operates. Defined somewhat more formally, an actor is anything that communicates with the system or product and that is external to the system itself.

As an example, consider a microprocessor-based home security system called *SafeHome* will be built to protect against and/or recognize a variety of undesirable "situations" such as illegal entry, fire, flooding, and others. This product (*SafeHome*) will use appropriate sensors to detect each situation, can be programmed by the homeowner, and will automatically telephone a monitoring agency when a situation is detected. For this product we can define three actors: the homeowner (the user), sensors (devices attached to the system), and the monitoring and response subsystem (the central station that monitors *SafeHome*).

Once actors have been identified, use-cases can be developed. The use-case describes the manner in which an actor interacts with the system.
1.3 Analysis Principles

Over the past two decades, a large number of analysis modeling methods have been developed. Investigators have identified analysis problems and their causes and have developed a variety of modeling notations and corresponding sets of heuristics to overcome them. Each analysis method has a unique point of view. However, all analysis methods are related by a set of operational principles:

1. The information domain of a problem must be represented and understood.
2. The functions that the software is to perform must be defined.
3. The behavior of the software (as a consequence of external events) must be represented.
4. The models that depict information, function and behavior must be partitioned in a manner that uncovers detail in a layered (or hierarchical) fashion.
5. The analysis process should move from essential information toward implementation detail.

By applying these principles, the analyst approaches a problem systematically.

1.3.1-The Information Domain

All software applications can be collectively called data processing. This term contains a key to our understanding of software requirements. Software is built to process data, to transform data from one form to another; that is, to accept input, manipulate it in some way, and produce output.

Software also processes events. An event represents some aspect of system control and is really nothing more than Boolean data—it is either on or off, true or false, there or not there. For example, a pressure sensor detects that pressure exceeds a safe value and sends an alarm signal to monitoring software. The alarm signal is an event that controls the behavior of the system. Therefore, data (numbers, text, images, sounds, video, etc.) and control (events) both reside within the information domain of a problem.

The information domain contains three different views of the data and control as each is processed by a computer program:

1. Information content and relationships (the data model).
2. Information flow.
3. Information structure.
To fully understand the information domain, each of these views should be considered.

*Information content* represents the individual data and control objects that constitute some larger collection of information transformed by the software.

**Example:** the data object, *paycheck*, is a composite of a number of important pieces of data: the payee's name, the net amount to be paid, the gross pay, deductions, and so forth. Therefore, the content of *paycheck* is defined by the attributes that are needed to create it.

Similarly, the content of a control object called *system status* might be defined by a string of bits. Each bit represents a separate item of information that indicates whether or not a particular device is on- or off-line.

Data and control objects can be related to other data and control objects.

*Information flow* represents the manner in which data and control change as each moves through a system. Referring to Figure 4, input objects are transformed to intermediate information (data and/or control), which is further transformed to output.

![Figure 4: Information flow and transformation](image)

Along this transformation path (or paths), additional information may be introduced from an existing data store (e.g., a disk file or memory buffer). The transformations applied to the data are functions or subfunctions that a program must perform.

*Information structure* represents the internal organization of various data and control items. Are data or control items to be organized as an *n*-dimensional table or as a hierarchical tree structure? Within the context of the structure, what information is related to other information? Is all information contained within a single structure or
are distinct structures to be used? How does information in one information structure relate to information in another structure? These questions and others are answered by an assessment of information structure.

1.3.2 Modeling
We create functional models to gain a better understanding of the actual entity to be built. When the entity is a physical thing (a building, a plane, a machine), we can build a model that is identical in form and shape but smaller in scale. However, when the entity to be built is software, our model must take a different form. It must be capable of representing the information that software transforms, the functions (and subfunctions) that enable the transformation to occur, and the behavior of the system as the transformation is taking place.

The second and third operational analysis principles require that we build models of function and behavior.

Functional models. Software transforms information, and in order to accomplish this, it must perform at least three generic functions: input, processing, and output. When functional models of an application are created, the software engineer focuses on problem specific functions. The functional model begins with a single context level model (i.e., the name of the software to be built). Over a series of iterations, more and more functional detail is provided, until a thorough delineation of all system functionality is represented.

Behavioral models. Most software responds to events from the outside world. This stimulus/response characteristic forms the basis of the behavioral model. A computer program always exists in some state—an externally observable mode of behavior (e.g., waiting, computing, printing) that is changed only when some event occurs. For example, software will remain in the wait state until (1) an internal clock indicates that some time interval has passed, (2) an external event (e.g., a mouse movement) causes an interrupt, or (3) an external system signals the software to act in some manner. A behavioral model creates a representation of the states of the software and the events that cause a software to change state.
Models created during requirements analysis serve a number of important roles:

- The model aids the analyst in understanding the information, function, and behavior of a system, thereby making the requirements analysis task easier and more systematic.
- The model becomes the focal point for review and, therefore, the key to a determination of completeness, consistency, and accuracy of the specifications.
- The model becomes the foundation for design, providing the designer with an essential representation of software that can be "mapped" into an implementation context.

1.3.3 Partitioning

Problems are often too large and complex to be understood as a whole. For this reason, we tend to partition (divide) such problems into parts that can be easily understood and establish interfaces between the parts so that overall function can be accomplished. The fourth operational analysis principle suggests that the information, functional, and behavioral domains of software can be partitioned.

In essence, partitioning decomposes a problem into its constituent parts. Conceptually, we establish a hierarchical representation of function or information and then partition the uppermost element by

1. Exposing increasing detail by moving vertically in the hierarchy
2. Functionally decomposing the problem by moving horizontally in the hierarchy.
1.3.4 Essential and Implementation Views (Logical and Physical Views)

An essential view of software requirements presents the functions to be accomplished and information to be processed without regard to implementation details. For example, the essential view of the SafeHome function read sensor status does not concern itself with the physical form of the data or the type of sensor that is used. In fact, it could be argued that read status would be a more appropriate name for this function, since it disregards details about the input mechanism altogether. Similarly, an essential data model of the data item phone number (implied by the function dial phone number) can be represented at this stage without regard to the underlying data structure (if any) used to implement the data item. By focusing attention on the essence of the problem at early stages of requirements engineering, we leave our options open to specify implementation details during later stages of requirements specification and software design.

The implementation view of software requirements presents the real world manifestation of processing functions and information structures. A SafeHome input device is a perimeter sensor (not a watch dog, a human guard, or a booby trap). The sensor detects illegal entry by sensing a break in an electronic circuit. The general characteristics of the sensor should be noted as part of a software requirements specification. The analyst must recognize the constraints imposed by predefined system elements (the sensor) and consider the implementation view of function and information when such a view is appropriate.

Note:
Software requirements engineering should focus on what the software is to accomplish, rather than on how processing will be implemented.
1 Software Prototyping

Analysis should be conducted regardless of the software engineering paradigm that is applied. However, the form that analysis takes will vary. In some cases it is possible to apply operational analysis principles and derive a model of software from which a design can be developed. In other situations, requirements elicitation (via FAST, QFD, use-cases, or other) is conducted, analysis principles are applied, and a model of the software to be built, called a *prototype*, is constructed for customer and developer assessment. Finally, some circumstances require the construction of a prototype at the beginning of analysis, since the model is the only means through which requirements can be effectively derived. The model then evolves into production software. The prototyping paradigm can be either close-ended or open-ended.

The close-ended approach is often called *throwaway prototyping*. Using this approach, a prototype serves solely as a rough demonstration of requirements. It is then discarded, and the software is engineered using a different paradigm. An open-ended approach, called *evolutionary prototyping*, uses the prototype as the first part of an analysis activity that will be continued into design and construction. The prototype of the software is the first evolution of the finished system.

A number of prototyping candidacy factors can be defined: to determine whether the system to be built is amenable to prototyping:

- Application area.
- Application complexity
- Customer characteristics
- Project characteristics.

In general, any application that creates dynamic visual displays, interacts heavily with a user, or demands algorithms or combinatorial processing that must be developed in an evolutionary fashion is a candidate for prototyping. However, these application areas must be weighed against application complexity. If a candidate application (one that has the characteristics noted) will require the development of tens of thousands of
lines of code before any demonstrable function can be performed, it is likely to be too complex for prototyping. If, however, the complexity can be partitioned, it may still be possible to prototype portions of the software.

Because the customer must interact with the prototype in later steps, it is essential that (1) customer resources be committed to the evaluation and refinement of the prototype and (2) the customer is capable of making requirements decisions in a timely fashion.

Finally, the nature of the development project will have a strong bearing on the efficacy of prototyping. Is project management willing and able to work with the prototyping method? Are prototyping tools available? Do developers have experience with prototyping methods?

1.1 Prototyping Methods and Tools

For software prototyping to be effective, a prototype must be developed rapidly so that the customer may assess results and recommend changes. To conduct rapid prototyping, three generic classes of methods and tools are available:

- **Fourth generation techniques.** Fourth generation techniques (4GT) encompass a broad array of database query and reporting languages, program and application generators, and other very high-level nonprocedural languages. Because 4GT enable the software engineer to generate executable code quickly, they are ideal for rapid prototyping.

- **Reusable software components.** Another approach to rapid prototyping is to assemble, rather than build, the prototype by using a set of existing software components. Melding prototyping and program component reuse will work only if a library system is developed so that components that do exist can be cataloged and then retrieved. It should be noted that an existing software product can be used as a prototype for a "new, improved" competitive product. In a way, this is a form of reusability for software prototyping.

- **Formal specification and prototyping environments.** Over the past two decades, a number of formal specification languages and tools have been developed as a replacement for natural language specification techniques. Today, developers of these formal languages are in the process of developing interactive environments that (1) enable an analyst to interactively create
language-based specifications of a system or software, (2) invoke automated tools that translate the language-based specifications into executable code, and (3) enable the customer to use the prototype executable code to refine formal requirements.

2 Specification
There is no doubt that the mode of specification has much to do with the quality of solution. Software engineers who have been forced to work with incomplete, inconsistent, or misleading specifications have experienced the frustration and confusion that invariably results. The quality, timeliness, and completeness of the software suffer as a consequence.

2.1 Representation
If requirements are committed to paper or an electronic presentation medium (and they almost always should be!) a simple set of guidelines is well worth following:

- **Representation format and content should be relevant to the problem.**
  A general outline for the contents of a *Software Requirements Specification* can be developed. However, the representation forms contained within the specification are likely to vary with the application area. For example, a specification for a manufacturing automation system might use different symbology, diagrams and language than the specification for a programming language compiler.

- **Information contained within the specification should be nested.**
  Representations should reveal layers of information so that a reader can move to the level of detail required. Paragraph and diagram numbering schemes should indicate the level of detail that is being presented. It is sometimes worthwhile to present the same information at different levels of abstraction to aid in understanding.

- **Diagrams and other notational forms should be restricted in number and consistent in use.** Confusing or inconsistent notation, whether graphical or symbolic, degrades understanding and fosters errors.
• **Representations should be revisable.** The content of a specification will change. Ideally, CASE tools should be available to update all representations that are affected by each change.

Investigators have conducted numerous studies on human factors associated with specification. There appears to be little doubt that symbology and arrangement affect understanding. However, software engineers appear to have individual preferences for specific symbolic and diagrammatic forms. Familiarity often lies at the root of a person's preference, but other more tangible factors such as spatial arrangement, easily recognizable patterns, and degree of formality often dictate an individual's choice.

### 2.2 The Software Requirements Specification

The *Software Requirements Specification* is produced at the culmination of the analysis task. The function and performance allocated to software as part of system engineering are refined by establishing a complete information description, a detailed functional description, a representation of system behavior, an indication of performance requirements and design constraints, appropriate validation criteria, and other information pertinent to requirements. The National Bureau of Standards, IEEE (Standard No. 830-1984), and the U.S. Department of Defense have all proposed candidate formats for software requirements specifications (as well as other software engineering documentation).

The *Introduction* of the software requirements specification states the goals and objectives of the software, describing it in the context of the computer-based system. Actually, the Introduction may be nothing more than the software scope of the planning document.

The *Information Description* provides a detailed description of the problem that the software must solve. Information content, flow, and structure are documented. Hardware, software, and human interfaces are described for external system elements and internal software functions.

A description of each function required to solve the problem is presented in the *Functional Description*. A processing narrative is provided for each function, design constraints are stated and justified, performance characteristics are stated, and one or
more diagrams are included to graphically represent the overall structure of the software and interplay among software functions and other system elements. The Behavioral Description section of the specification examines the operation of the software as a consequence of external events and internally generated control characteristics.

Validation Criteria is probably the most important and, ironically, the most often neglected section of the Software Requirements Specification. How do we recognize a successful implementation? What classes of tests must be conducted to validate function, performance, and constraints? We neglect this section because completing it demands a thorough understanding of software requirements—something that we often do not have at this stage. Yet, specification of validation criteria acts as an implicit review of all other requirements. It is essential that time and attention be given to this section.

Finally, the specification includes a Bibliography and Appendix. The bibliography contains references to all documents that relate to the software. These include other software engineering documentation, technical references, vendor literature, and standards. The appendix contains information that supplements the specifications. Tabular data, detailed description of algorithms, charts, graphs, and other material are presented as appendixes.

In many cases the Software Requirements Specification may be accompanied by an executable prototype (which in some cases may replace the specification), a paper prototype or a Preliminary User's Manual. The Preliminary User's Manual presents the software as a black box. That is, heavy emphasis is placed on user input and the resultant output. The manual can serve as a valuable tool for uncovering problems at the human/machine interface.

4 Specification Review

A review of the Software Requirements Specification (and/or prototype) is conducted both the software developer and the customer. Because the specification forms the foundation of the development phase, extreme care should be taken in conducting the review.

The review is first conducted at a macroscopic level; that is, reviewers attempt to ensure that the specification is complete, consistent, and accurate when the overall information, functional, and behavioral domains are considered. However, to fully
explore each of these domains, the review becomes more detailed, examining not only broad descriptions but the way in which requirements are worded. For example, when specifications contain “vague terms” (e.g., some, sometimes, often, usually, ordinarily, most, or mostly), the reviewer should flag the statements for further clarification.

Once the review is complete, the Software Requirements Specification is "signed off" by both the customer and the developer. The specification becomes a "contract" for software development. Requests for changes in requirements after the specification is finalized will not be eliminated. But the customer should note that each after the-fact change is an extension of software scope and therefore can increase cost and/or protract the schedule.

Even with the best review procedures in place, a number of common specification problems persist. The specification is difficult to "test" in any meaningful way, and therefore inconsistency or omissions may pass unnoticed. During the review, changes to the specification may be recommended. It can be extremely difficult to assess the global impact of a change; that is, how a change in one function affects requirements for other functions.
Analysis Modeling

At a technical level, software engineering begins with a series of modeling tasks that lead to a complete specification of requirements and a comprehensive design representation for the software to be built. The analysis model, actually a set of models, is the first technical representation of a system. Over the years many methods have been proposed for analysis modeling. However, two now dominate. The first, structured analysis, is a classical modeling method. The other approach, object oriented analysis. Structured analysis is a model building activity. Applying the operational analysis principles we create and partition data, functional, and behavioral models that depict the essence of what must built.

Note:
The written word is a wonderful vehicle for communication, but it is not necessarily the best way to represent the requirements for computer software. Analysis modeling uses a combination of text and diagrammatic forms to depict requirements for data, function, and behavior in a way that is relatively easy to understand, and more important, straightforward to review for correctness, completeness, and consistency.

1 The Elements of the Analysis Model
The analysis model must achieve three primary objectives:

1. To describe what the customer requires.
2. To establish a basis for the creation of a software design.
3. To define a set of requirements that can be validated once the software is built.

To accomplish these objectives, the analysis model derived during structured analysis takes the form illustrated in Figure 1.
At the core of the model lies the *data dictionary*—a repository that contains descriptions of all data objects consumed or produced by the software. Three different diagrams surround the core. The *entity relation diagram* (ERD) depicts relationships between data objects. The ERD is the notation that is used to conduct the data modeling activity. The attributes of each data object noted in the ERD can be described using a data object description. The *data flow diagram* (DFD) serves two purposes:

1. To provide an indication of how data are transformed as they move through the system.
2. To depict the functions (and subfunctions) that transform the data flow.

The DFD provides additional information that is used during the analysis of the information domain and serves as a basis for the modeling of function. A description of each function presented in the DFD is contained in a *process specification* (PSPEC).
The state transition diagram (STD) indicates how the system behaves as a consequence of external events. To accomplish this, the STD represents the various modes of behavior (called states) of the system and the manner in which transitions are made from state to state. The STD serves as the basis for behavioral modeling.

Additional information about the control aspects of the software is contained in the control specification (CSPEC).

The analysis model encompasses each of the diagrams, specifications, descriptions, and the dictionary noted in Figure 1.

2 Data Modeling

Data modeling methods make use of the entity relationship diagram. The ERD enables a software engineer to identify data objects and their relationships using a graphical notation.

In the context of structured analysis, the ERD defines all data that are entered, stored, transformed, and produced within an application.

The entity relationship diagram focuses solely on data (and therefore satisfies the first operational analysis principles), representing a "data network" that exists for a given system. The ERD is especially useful for applications in which data and the relationships that govern data are complex. Unlike the data flow diagram, data modeling considers data independent of the processing that transforms the data.

2.1 Data Objects, Attributes, and Relationships

The data model consists of three interrelated pieces of information: the data object, the attributes that describe the data object, and the relationships that connect data objects to one another.

Data objects. A data object is a representation of almost any composite information that must be understood by software. By composite information, we mean something that has a number of different properties or attributes. Therefore, width (a single value) would not be a valid data object, but dimensions (incorporating height, width, and depth) could be defined as an object.

A data object can be an external entity (e.g., anything that produces or consumes information), a thing (e.g., a report or a display), an occurrence (e.g., a telephone call)
or event (e.g., an alarm), a role (e.g., salesperson), an organizational unit (e.g., accounting department), a place (e.g., a warehouse), or a structure (e.g., a file).

**EX:**

A person or a car (Figure 2) can be viewed as a data object in the sense that either can be defined in terms of a set of attributes. The data object description incorporates the data object and all of its attributes.

Data objects (represented in bold) are related to one another. For example, **person** can **own car**, where the relationship **own** connotes a specific "connection" between **person** and **car**. The relationships are always defined by the context of the problem that is being analyzed.

A data object encapsulates data only—there is no reference within a data object to operations that act on the data (This distinction separates the data object from the **class** or **object** defined as part of the object-oriented paradigm). Therefore, the data object can be represented as a table as shown in Figure 3. The headings in the table reflect attributes of the object. In this case, a car is defined in terms of make, model, ID number, body type color and owner. The body of the table represents specific instances of the data object.

For example, a Chevy Corvette is an instance of the data object **car**.

**Figure 2** Data objects, attributes and relationships
Attributes. Attributes define the properties of a data object and take on one of three different characteristics. They can be used to (1) name an instance of the data object, (2) describe the instance, or (3) make reference to another instance in another table. In addition, one or more of the attributes must be defined as an identifier—that is, the identifier attribute becomes a "key" when we want to find an instance of the data object. In some cases, values for the identifier(s) are unique, although this is not a requirement. Referring to the data object car, a reasonable identifier might be the ID number. The set of attributes that is appropriate for a given data object is determined through an understanding of the problem context.

Relationships. Data objects are connected to one another in different ways. Consider two data objects, book and bookstore. These objects can be represented using the simple notation illustrated in Figure 4a. A connection is established between book and bookstore because the two objects are related. But what are the relationships? To determine the answer, we must understand the role of books and bookstores within the context of the software to be built. We can define a set of object/relationship pairs that define the relevant relationships. For example,

- A bookstore orders books.
- A bookstore displays books.
- A bookstore stocks books.
- A bookstore sells books.
- A bookstore returns books.
The relationships orders, displays, stocks, sells, and returns define the relevant connections between book and bookstore. Figure 12.4b illustrates these object/relationship pairs graphically.

![Diagram of object/relationship pairs](image)

(a) A basic connection between objects

(b) Relationships between objects

**Figure 4 Relationships**

**Note:**
It is important to note that object/relationship pairs are bidirectional. That is, they can be read in either direction. A bookstore orders books or books are ordered by a bookstore.

2.2 Cardinality and Modality
The elements of data modeling—data objects, attributes, and relationships—provide the basis for understanding the information domain of a problem. However, additional information related to these basic elements must also be understood.

We have defined a set of objects and represented the object/relationship pairs that bind them. But a simple pair that states: **object X relates to object Y** does not provide enough information for software engineering purposes. We must understand how
many occurrences of **object X** are related to how many occurrences of **object Y**. This leads to a data modeling concept called **cardinality**.

**Cardinality.** The data model must be capable of representing the number of occurrences objects in a given relationship. Tillmann defines the **cardinality** of an object/relationship pair in the following manner:

Cardinality is the specification of the number of occurrences of one [object] that can be related to the number of occurrences of another [object].

Cardinality is usually expressed as simply 'one' or 'many.'

Taking into consideration all combinations of 'one' and 'many,' two [objects] can be related as:

- **One-to-one (1:1)**—An occurrence of [object] 'A' can relate to one and only one occurrence of [object] 'B,' and an occurrence of 'B' can relate to only one occurrence of 'A.'
- **One-to-many (1:N)**—One occurrence of [object] 'A' can relate to one or many occurrences of [object] 'B,' but an occurrence of 'B' can relate to only one occurrence of 'A.'
- **Many-to-many (M:N)**—An occurrence of [object] 'A' can relate to one or more occurrences of 'B,' while an occurrence of 'B' can relate to one or more occurrences of 'A.'

Cardinality defines “the maximum number of objects that can participate in a relationship”. It does not, however, provide an indication of whether or not a particular data object must participate in the relationship. To specify this information, the data model adds modality to the object/relationship pair.

**Modality.** The **modality** of a relationship is 0 if there is no explicit need for the relationship to occur or the relationship is optional. The modality is 1 if an occurrence of the relationship is mandatory. To illustrate, consider software that is used by a local telephone company to process requests for field service. A customer indicates that there is a problem. If the problem is diagnosed as relatively simple, a single repair action occurs. However, if the problem is complex, multiple repair actions may be required. Figure 5 illustrates the relationship, cardinality, and modality between the data objects **customer** and **repair action**.
Referring to the figure, a one to many cardinality relationship is established. That is, a single customer can be provided with zero or many repair actions. The symbols on the relationship connection closest to the data object rectangles indicate cardinality. The vertical bar indicates one and the three-pronged fork indicates many. Modality is indicated by the symbols that are further away from the data object rectangles. The second vertical bar on the left indicates that there must be a customer for a repair action to occur. The circle on the right indicates that there may be no repair action required for the type of problem reported by the customer.

2.3 Entity/Relationship Diagrams

The object/relationship pair (discussed in Section 2.1) is the cornerstone of the data model. These pairs can be represented graphically using the entity/relationship diagram. The ERD was originally proposed by Peter Chen for the design of relational database systems and has been extended by others. A set of primary components are identified for the ERD: data objects, attributes, relationships, and various type indicators. The primary purpose of the ERD is to represent data objects and their relationships.

Rudimentary ERD notation has already been introduced in Section 2. Data objects are represented by a labeled rectangle. Relationships are indicated with a labeled line connecting objects. In some variations of the ERD, the connecting line contains a diamond that is labeled with the relationship. Connections between data objects and
relationships are established using a variety of special symbols that indicate cardinality and modality (Section 2.2).

The relationship between the data objects car and manufacturer would be represented as shown in Figure 6. One manufacturer builds one or many cars. Given the context implied by the ERD, the specification of the data object car (data object table in Figure 6) would be radically different from the earlier specification (Figure 3). By examining the symbols at the end of the connection line between objects, it can be seen that the modality of both occurrences is mandatory (the vertical lines).

![A simple ERD and data object table](image)

**FIGURE 12.6** A simple ERD and data object table (Note: In this ERD the relationship builds is indicated by a diamond)
Functional Modeling and Information Flow

Information is transformed as it flows through a computer-based system. The system accepts input in a variety of forms; applies hardware, software, and human elements to transform it; and produces output in a variety of forms. Input may be a control signal transmitted by a transducer, a series of numbers typed by a human operator, a packet of information transmitted on a network link, or a voluminous data file retrieved from secondary storage. The transform(s) may comprise a single logical comparison, a complex numerical algorithm. Output may light a single LED or produce a 200-page report. In effect, we can create a flow model for any computer-based system, regardless of size and complexity.

Structured analysis began as an information flow modeling technique. A computer-based system is represented as an information transform as shown in Figure 1.

![Figure 1: Information flow model](image-url)
A rectangle is used to represent an external entity; that is, a system element (e.g., hardware, a person, another program) or another system that produces information for transformation by the software or receives information produced by the software. A circle (sometimes called a bubble) represents a process or transform that is applied to data (or control) and changes it in some way. An arrow represents one or more data items (data objects). All arrows on a data flow diagram should be labeled. The double line represents a data store—stored information that is used by the software. The simplicity of DFD notation is one reason why structured analysis techniques are widely used.

**Note:**
The DFD is not procedural. That is, do not try to represent conditional processing or loops with this diagrammatic form. Simply show the flow of data.

### Data Flow Diagrams

As information moves through software, it is modified by a series of transformations. A data flow diagram is a graphical representation that depicts information flow and the transforms that are applied as data move from input to output. The basic form of a data flow diagram, also known as a data flow graph or a bubble chart, is illustrated in Figure 1.

The data flow diagram may be used to represent a system or software at any level of abstraction. In fact, DFDs may be partitioned into levels that represent increasing information flow and functional detail. Therefore, the DFD provides a mechanism for functional modeling as well as information flow modeling. In so doing, it satisfies the second operational analysis principle (i.e., creating a functional model).

A level 0 DFD, also called a fundamental system model or a context model, represents the entire software element as a single bubble with input and output data indicated by incoming and outgoing arrows, respectively. Additional processes (bubbles) and information flow paths are represented as the level 0 DFD is partitioned to reveal more detail. For example, a level 1 DFD might contain five or six bubbles with interconnecting arrows. Each of the processes represented at level 1 is a subfunction of the overall system depicted in the context model.

As has been noted earlier, each of the bubbles may be refined or layered to depict more detail. Figure 2 illustrates this concept.
A fundamental model for system $F$ indicates the primary input is $A$ and ultimate output is $B$. We refine the $F$ model into transforms $f1$ to $f7$. Note that information flow continuity must be maintained; that is, input and output to each refinement must remain the same. This concept, sometimes called balancing, is essential for the development of consistent models. Further refinement of $f4$ depicts detail in the form of transforms $f41$ to $f45$. Again, the input ($X, Y$) and output $Z$ remain unchanged.

DFD graphical notation must be augmented with descriptive text. A process specification (PSPEC) can be used to specify the processing details implied by a bubble within a DFD. The process specification describes the input to a function, the algorithm that is applied to transform the input, and the output that is produced. In addition, the PSPEC indicates restrictions and limitations imposed on the process (function), performance characteristics that are relevant to the process, and design constraints that may influence the way in which the process will be implemented.

The process specification (PSPEC) is used to describe all flow model processes that appear at the final level of refinement. The content of the process specification can include narrative text, a program design language (PDL) description of the process algorithm, mathematical equations, tables, diagrams, or charts. By providing a PSPEC to accompany each bubble in the flow model, the software engineer creates a "mini-
spec" that can serve as a first step in the creation of the Software Requirements Specification and as a guide for design of the software component that will implement the process.

Creating a Data Flow Model

The data flow diagram enables the software engineer to develop models of the information domain and functional domain at the same time. As the DFD is refined into greater levels of detail, the analyst performs an implicit functional decomposition of the system, thereby accomplishing the fourth operational analysis principle for function. At the same time, the DFD refinement results in a corresponding refinement of data as it moves through the processes that embody the application.

A few simple guidelines can aid immeasurably during derivation of a data flow diagram:

1. The level 0 data flow diagram should depict the software/system as a single bubble.
2. Primary input and output should be carefully noted.
3. Refinement should begin by isolating candidate processes, data objects, and stores to be represented at the next level.
4. All arrows and bubbles should be labeled with meaningful names.
5. Information flow continuity must be maintained from level to level.
6. One bubble at a time should be refined.

Extensions for Real-Time Systems

Many software applications are time dependent and process as much or more control-oriented information as data. A real-time system must interact with the real world in a time frame dictated by the real world. manufacturing process control, and industrial instrumentation are but a few of hundreds of real-time software applications.

To accommodate the analysis of real-time software, a number of extensions to the basic notation for structured analysis have been defined. These extensions, developed by Ward and Mellor and Hatley and Pirbhai, enable the analyst to represent control flow and control processing as well as data flow and processing.
1-Ward and Mellor Extensions
In a significant percentage of real-time applications, the system must monitor time continuous information generated by some real-world process. For example, a real-time test monitoring system for gas turbine engines might be required to monitor turbine speed, combustor temperature, and a variety of pressure probes on a continuous basis. Conventional data flow notation does not make a distinction between discrete data and time-continuous data. One extension to basic structured analysis notation, shown in Figure 3, provides a mechanism for representing time-continuous data flow.

The double headed arrow is used to represent time-continuous flow while a single headed arrow is used to indicate discrete data flow. In the figure, monitored temperature is measured continuously while a single value for temperature set point is also provided. The process shown in the figure produces a time-continuous output, corrected value.

![Figure 3: Time continuous data flow](image)

Continuing the convention established for data flow diagrams, data flow is represented using a solid arrow. Control flow, however, is represented using a dashed or shaded arrow. A process that handles only control flows, called a control process, is similarly represented using a dashed bubble.
Control flow can be input directly to a conventional process or into a control process. Figure 4 illustrates control flow and processing as it would be represented using Ward and Mellor notation.

The figure illustrates a top-level view of a data and control flow for a manufacturing cell. As components to be assembled by a robot are placed on fixtures, a status bit is set within a **parts status buffer** (a control store) that indicates the presence or absence of each component. Event information contained within the **parts status buffer** is passed as a bit string to a process, **monitor fixture and operator interface**. The process will read **operator commands** only when the control information, bit string, indicates that all fixtures contain components. An event flag, **start/stop flag**, is sent to **robot initiation control**, a control process that enables further command processing. Other data flows occur as a consequence of the **process activate** event that is sent to **process robot commands**.

**Figure 4**: Data and control flows using Ward and Mellor notation
2- Hatley and Pirbhai Extensions

The Hatley and Pirbhai extensions to basic structured analysis notation focus less on the creation of additional graphical symbols and more on the representation and specification of the control-oriented aspects of the software. The dashed arrow is once again used to represent control or event flow. Unlike Ward and Mellor, Hatley and Pirbhai suggest that dashed and solid notation be represented separately. Therefore, a control flow diagram is defined. The CFD contains the same processes as the DFD, but shows control flow, rather than data flow. Instead of representing control processes directly within the flow model, a notational reference (a solid bar) to a control specification (CSPEC) is used. In essence, the solid bar can be viewed as a "window" into an "executive" (the CSPEC) that controls the processes (functions) represented in the DFD based on the event that is passed through the window. The CSPEC is used to indicate

1. How the software behaves when an event or control signal is sensed.
2. Which processes are invoked as a consequence of the occurrence of the event.

A process specification is used to describe the inner workings of a process represented in a flow diagram.

Using the notation described in Figures 3 and 4 (these figures are in lecture 7), along with additional information contained in PSPECs and CSPECs, Hatley and Pirbhai create a model of a real-time system. Data flow diagrams are used to represent data and the processes that manipulate it. Control flow diagrams show how events flow among processes and illustrate those external events that cause various processes to be activated. The interrelationship between the process and control models is shown schematically in Figure 5.
The process model is "connected" to the control model through data conditions. The control model is "connected" to the process model through process activation information contained in the CSPEC.

A data condition occurs whenever data input to a process result in control output. This situation is illustrated in Figure 6, part of a flow model for an automated monitoring and control system for pressure vessels in an oil refinery.

The process check and convert pressure implements the algorithm described in the PSPEC pseudocode shown. When the absolute tank pressure is greater than an allowable maximum, an above pressure event is generated. Note that when Hatley and Pirbhai notation is used, the data flow is shown as part of a DFD, while the control flow is noted separately as part of a control flow diagram. As we noted earlier, the vertical solid bar into which the above pressure event flows is a pointer to the CSPEC.
Therefore, to determine what happens when this event occurs, we must check the CSPEC.

The control specification (CSPEC) contains a number of important modeling tools. A process activation table is used to indicate which processes are activated by a given event. For example, a process activation table (PAT) for Figure 6 might indicate that the above pressure event would cause a process reduce tank pressure (not shown) to be invoked. In addition to the PAT, the CSPEC may contain a state transition diagram. The STD is a behavioral model that relies on the definition of a set of system states.
Behavioral Modeling

Behavioral modeling is an operational principle for all requirements analysis methods. Yet, only extended versions of structured analysis provide a notation for this type of modeling. The state transition diagram represents the behavior of a system by depicting its states and the events that cause the system to change state. In addition, the STD indicates what actions (e.g., process activation) are taken as a consequence of a particular event.

A state is any observable mode of behavior. A state transition diagram indicates how the system moves from state to state.

To illustrate the use of the Hatley and Pirbhai control and behavioral extensions, consider software embedded within an office photocopying machine. A simplified representation of the control flow for the photocopier software is shown in Figure 7.

Figure 7: Level 1 CFD for photocopier software
Data flow arrows have been lightly shaded for illustrative purposes, but in reality they are not shown as part of a control flow diagram. Control flows are shown entering and exiting individual processes and the vertical bar representing the CSPEC "window."

For example, the **paper feed status** and **start/stop** events flow into the CSPEC bar. This implies that each of these events will cause some process represented in the CFD to be activated. If we were to examine the CSPEC internals, the **start/stop** event would be shown to activate/deactivate the **manage copying** process. Similarly, the **jammed** event (part of **paper feed status**) would activate **perform problem diagnosis**. It should be noted that all vertical bars within the CFD refer to the same CSPEC. An event flow can be input directly into a process as shown with **repro fault**. However, this flow does not activate the process but rather provides control information for the process algorithm.

A simplified state transition diagram for the photocopier software is shown in Figure 8.

---

**Figure 8:** State transition diagram for photocopier software
The rectangles represent system states and the arrows represent transitions between states. Each arrow is labeled with a ruled expression. The top value indicates the event(s) that cause the transition to occur. The bottom value indicates the action that occurs as a consequence of the event. Therefore, when the paper tray is full and the start button is pressed, the system moves from the reading commands state to the making copies state. Note that states do not necessarily correspond to processes on a one-to-one basis. For example, the state making copies would encompass both the manage copying and produce user displays processes shown in Figure 7.

The Data Dictionary
The analysis model encompasses representations of data objects, function, and control. In each representation data objects and/or control items play a role. Therefore, it is necessary to provide an organized approach for representing the characteristics of each data object and control item. This is accomplished with the data dictionary.

The data dictionary has been proposed as a quasi-formal grammar for describing the content of objects defined during structured analysis. This important modeling notation has been defined in the following manner:
The data dictionary is an organized listing of all data elements that are pertinent to the system, with precise, rigorous definitions so that both user and system analyst will have a common understanding of inputs, outputs, components of stores and [even] intermediate calculations.

Today, the data dictionary is always implemented as part of a CASE "structured analysis and design tool." Although the format of dictionaries varies from tool to tool, most contain the following information:

• Name—the primary name of the data or control item, the data store or an external entity.

• Alias—other names used for the first entry.

• Where-used/how-used—a listing of the processes that use the data or control item and how it is used (e.g., input to the process, output from the process, as a store, as an external entity.

• Content description—a notation for representing content.
• Supplementary information—other information about data types, preset values (if known), restrictions or limitations, and so forth.

Once a data object or control item name and its aliases are entered into the data dictionary, consistency in naming can be enforced. That is, if an analysis team member decides to name a newly derived data item \( xyz \), but \( xyz \) is already in the dictionary, the CASE tool supporting the dictionary posts a warning to indicate duplicate names. This improves the consistency of the analysis model and helps to reduce errors.

“Where-used/how-used” information is recorded automatically from the flow models. A dictionary entry is created, the CASE tool scans DFDs and CFDs to determine which processes use the data or control information and how it is used. Although this may appear unimportant, it is actually one of the most important benefits of the dictionary. During analysis there is an almost continuous stream of changes. For large projects, it is often quite difficult to determine the impact of a change. Many a software engineer has asked, "Where is this data object used? What else will have to change if we modify it? What will the overall impact of the change be?" Because the data dictionary can be treated as a database, the analyst can ask "where used/how used" questions, and get answers to these queries.

The notation used to develop a content description is noted in the following table:

<table>
<thead>
<tr>
<th>Data Construct</th>
<th>Notation</th>
<th>Meaning</th>
</tr>
</thead>
<tbody>
<tr>
<td>Sequence</td>
<td>+</td>
<td>and</td>
</tr>
<tr>
<td>Selection</td>
<td>[</td>
<td>]</td>
</tr>
<tr>
<td>Repetition</td>
<td>{</td>
<td>}</td>
</tr>
<tr>
<td></td>
<td>(</td>
<td>optional data</td>
</tr>
<tr>
<td></td>
<td>* ... *</td>
<td>delimits comments</td>
</tr>
</tbody>
</table>

The notation enables a software engineer to represent composite data in one of the three fundamental ways that it can be constructed:

1. As a sequence of data items.
2. As a selection from among a set of data items.

3. As a repeated grouping of data items. Each data item entry that is represented as part of a sequence, selection, or repetition may itself be another composite data item that needs further refinement within the dictionary.

To illustrate the use of the data dictionary, we consider the level 2 DFD for the *monitor system* process for *SafeHome*, shown in Figure 9.

Referring to the figure, the data item *telephone number* is specified as input. But what exactly is a telephone number? It could be a 7-digit local number, or a 4-digit extension. The data dictionary provides us with a precise definition of *telephone number* for the DFD in question. In addition it indicates where and how this data item is used and any supplementary information that is relevant to it. The data dictionary entry begins as follows:

![Figure 9: Level 2 DFD that refines the monitor sensors process](image-url)
name: telephone number
aliases: none
where used/how used: assess against set-up (output)
dial phone (input)
description:

telephone number = [local number | long distance number]
local number = prefix + access number
long distance number = 1 + area code + local number
area code = [800 | 888 | 561]
prefix = *a three digit number that never starts with 0 or 1*
access number = * any four number string *

The content description is expanded until all composite data items have been represented as elementary items (items that require no further expansion) or until all composite items are represented in terms that would be well-known and unambiguous to all readers. It is also important to note that a specification of elementary data often restricts a system. For example, the definition of area code indicates that only three area codes (two toll-free and one in South Florida) are valid for this system.

The data dictionary defines information items unambiguously

For large computer-based systems, the data dictionary grows rapidly in size and complexity. In fact, it is extremely difficult to maintain a dictionary manually. For this reason, CASE tools should be used.