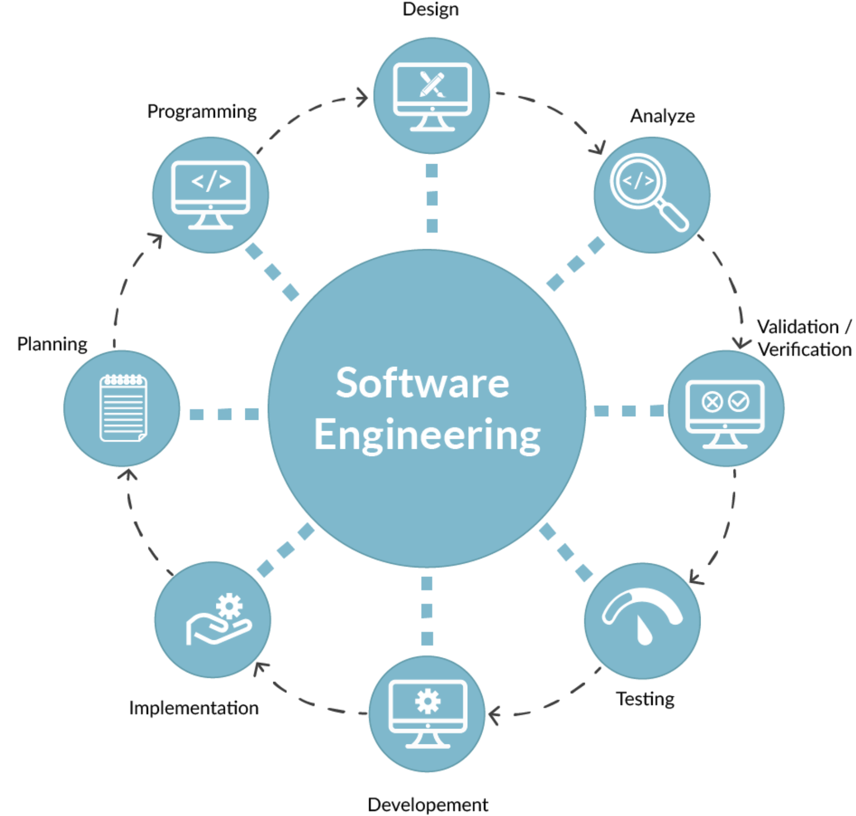
جامعة الموصل

كلية التربية للعلوم الصرفة

قسم علوم الحاسوب



هندسة برامجيات

إعداد : م.م. حسنين علي طالب

المرحلة الثالثة

1.INTRODUCTION TO SOFTWARE ENGINEERING

1.1 Software Definition

What is it?

Computer software

is the product that software engineers design and build.

It **encompasses programs** that execute within a computer of any size and architecture,

documents that encompass hard-copy and virtual forms, and **data** that combine numbers and text but also includes representations of pictorial, video, and a udio information.

Who does it? Software engineers build it, and virtually everyone in the industrialized world uses it either directly or indirectly.

Why is it important? Because it simulates every aspect of our lives and has become widespread in our commerce, our culture, and our everyday activities.

What are the steps? You build computer software like you build any successful product, by applying a process that leads to a high-quality result that meets the needs of the people who will use the product. You apply a software engineering approach.

What is the work product?

-From the **point of view of a software engineer**, the work product is the programs, documents, and data that are computer software.

-But from the **user's viewpoint**, the work product is the resultant information that somehow makes the user's world better.

software is an information transformer producing, managing, acquiring, modifying, displaying, or

transmitting information that can be as simple as a single bit or as complex as a multimedia presentation and we can say also that Software is:

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

1.2 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:

1.2.1 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.

Software costs are concentrated in engineering. This means that software projects cannot be managed as if they were manufacturing projects.

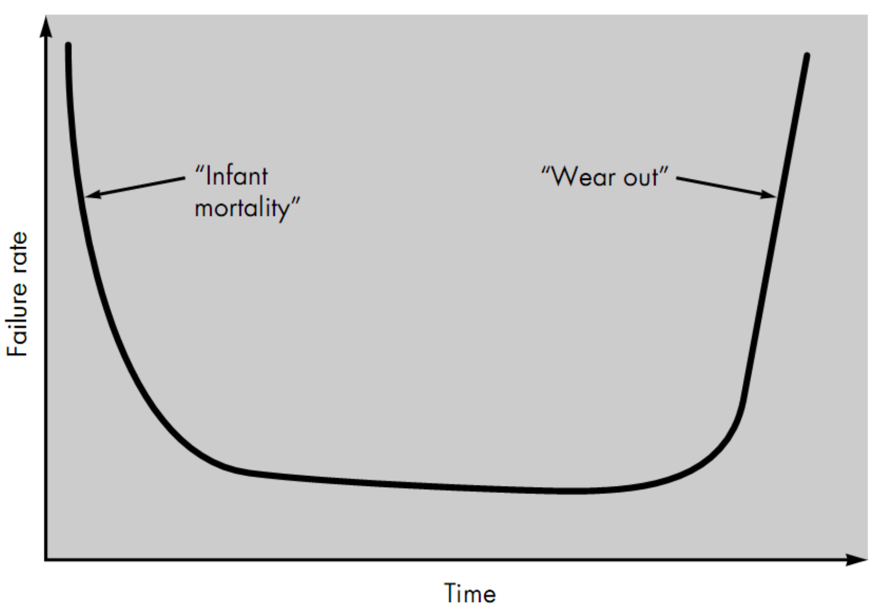


FIGURE Failure curve for hardware

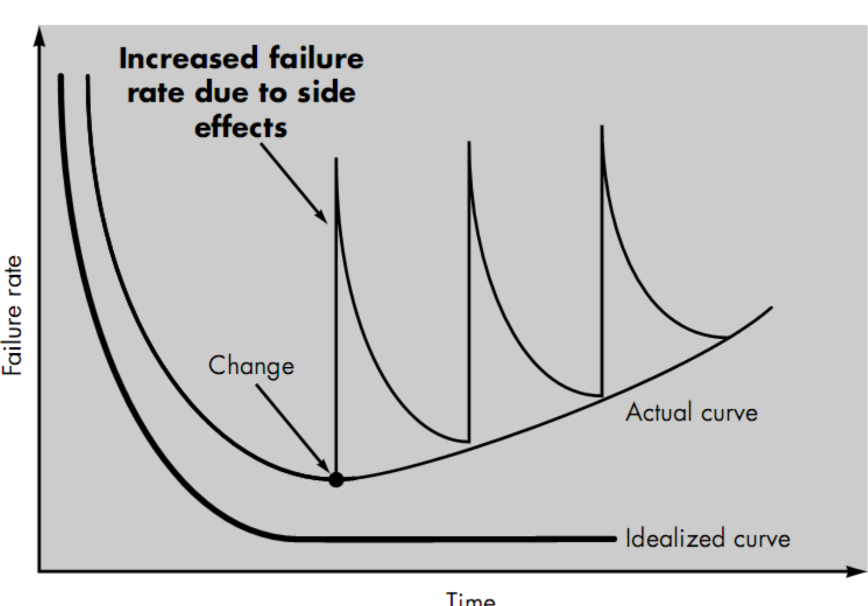
1.2.2. Software doesn't "wear out."

Figure 1.1 depicts failure rate as a function of time for hardware. The relationship, often called the "bathtub curve," indicates that hardware shows 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, temperature, and many other environmental maladies. 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 1.2. 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. 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(يتلف) !

This seeming contradiction (كلام متناقض) can best be explained by considering the “actual curve” shown in Figure 1.2. During its life, software will undergo(يخضع للتغيير) change (maintenance).

FIGURE 1.2 Idealized and actual failure curves for software



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 1.2. 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.

- Consider the manner (أسلوب) in which the control hardware for a computer-based product is designed and built.
- The design engineer draws a simple schematic (مخطط) 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 some-thing 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, we built scientific subroutine libraries that were reusable in a broad array of engineering and scientific applications. These subroutine libraries reused well-defined algorithms in an effective manner but had a limited domain of application. Today, we have extended our view of reuse 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.

1.3 Software Applications

Software may be applied in any situation for which a prespecified set of procedural steps (i.e., an algorithm) has been defined (notable exceptions to this rule are expert system software and neural network software). Information content and determinacy are important factors in determining the nature of a software application. Content refers to the meaning and form of incoming and outgoing information. For example, many business applications use highly structured input data (a database) and produce formatted "reports." Software that controls an automated machine (e.g., a numerical control) accepts discrete data items with limited structure and produces individual machine commands in rapid succession.

Information determinacy refers to the predictability of the order and timing of information. An engineering analysis program accepts data that have a predefined order, executes the analysis algorithm(s) without interruption, and produces resultant data in report or graphical format. Such applications are determinate. A multiuser operating system, on the

other hand, accepts inputs that have varied content and arbitrary timing, executes algorithms that can be interrupted by external conditions, and produces output that varies as a function of environment and time. Applications with these characteristics are indeterminate.

It is somewhat difficult to develop meaningful generic categories for software applications. As software complexity grows, neat compartmentalization disappears. The following software areas indicate the breadth of potential applications:

System software. System software is a collection of programs written to service other programs. Some system software (e.g., compilers, editors, and file management utilities) process complex, but determinate, information structures. Other systems applications (e.g., operating system components, drivers, telecommunications processors) process largely indeterminate data. In either case, the system software area is characterized by heavy interaction with computer hardware; heavy usage by multiple users; concurrent operation that requires scheduling, resource sharing, and sophisticated process management; complex data structures; and multiple external interfaces.Real-time software. Software that monitors/analyzes/controls real-world events as they occur is called real time. Elements of real-time software include a data gath- ering component that collects and formats information from an

external environ-ment, an analysis component that transforms information as required by the application, a control/output component that responds to the external environment, and a monitoring component that coordinates all other components so that real-time response (typically ranging from 1 millisecond to 1 second) can be maintained.

Business software. Business information processing is the largest single software application area. Discrete "systems" (e.g., payroll, accounts receivable/payable, inventory) have evolved into management information system (MIS) software that accesses one or more large databases containing business information. Applications in this area restructure existing data in a way that facilitates business operations or management decision making. In addition to conventional data processing application, business software applications also encompass interactive computing (e.g., point-of-sale transaction processing).

Engineering and scientific software. Engineering and scientific software have been characterized by "number crunching" algorithms. Applications range from astronomy to volcanology, from automotive stress analysis to space shuttle orbital dynamics, and from molecular biology to automated manufacturing. However, modern applications within the engineering/scientific area are moving away from

conventional numerical algorithms. Computer-aided design, system simulation, and other interactive applications have begun to take on real-time and even system software characteristics.

Embedded software. Intelligent products have become commonplace in nearly every consumer and industrial market. Embedded software resides in read-only memory and is used to control products and systems for the consumer and industrial markets. Embedded software can perform very limited and esoteric functions (e.g., keypad control for a microwave oven) or provide significant function and control capability (e.g., digital functions in an automobile such as fuel control, dashboard displays, and braking systems).Personal computer software. The personal computer software market has burgeoned over the past two decades. Word processing, spreadsheets, computer graphics, multimedia, entertainment, database management, personal and business financial applications, external network, and database access are only a few of hundreds of applications.

Web-based software. The Web pages retrieved by a browser are software that incorporates يدمج executable instructions (e.g., CGI, HTML, Perl, or Java), and data (e.g.,hypertext and a variety of visual and audio formats). In essence, the network becomes a massive computer providing an almost unlimited software resource that can be accessed by

anyone with a modem. Artificial intelligence software. Artificial intelligence (AI) software makes use of nonnumerical algorithms to solve complex problems that are not amenable سهلة الانقياد to computation or straightforward analysis. Expert systems, also called knowledge-based systems, pattern recognition (image and voice), artificial neural networks, theorem proving, and game playing are representative of applications within this category.

1.4 SOFTWARE CRISIS

The word crisis is defined in Dictionary as “a turning point in the course of anything; decisive time الوقت الحاسم, stage or event.” Yet, in terms of overall software quality and the speed with which computer-based systems and products are developed, there has been no "turning point," no "decisive time," only slow, evolutionary change, punctuated تتخللها by explosive انفجاري technological changes in disciplines التخصصات associated with software.

The word crisis has another definition: "the turning point in the course of a disease, when it becomes clear whether the patient will live or die." This definition may give us a clue دليل about the real nature of the problems that have plagued ينتشر فيه مرض software development.

What we really have might be better characterized as a chronic affliction الاصابة المزمنة.2 The word affliction is defined as "anything causing pain or distress محنة مضايقة." But the definition of the adjective chronic is the key to our argument برهان: "lasting a long time or recurring often المتكررة في كثير من الأحيان; continuing indefinitely استمرار إلى أجل غير مسمى." It is far more accurate to describe the problems we have endured تحملت in the software business as a chronic affliction than a crisis.

Regardless of what we call it, the set of problems that are encountered in the development of computer software is not limited to software that "doesn't function properly." Rather, the affliction encompasses problems associated with how we develop software, how we support a growing volume of existing software, and how we can expect to keep pace with a growing demand for more software.

We live with this affliction to this day—in fact, the industry prospers in spite رغم of it.

And yet, things would be much better if we could find and broadly apply a cureشفاء .

1.5 Characteristics of a Well-engineered Software

To define a well-engineered software, one takes a look at specific characteristics that the software exhibits. Some of them are enumerated below:

•Usability

It is the characteristic of the software that exhibits ease with which the user communicates with the system.

•Portability

It is the capability of the software to execute in different platforms and architecture.

•Reusability

It is the ability of the software to transfer from one system to another.

•Maintainability

It is the ability of the software to evolve تتطور and adapt تكيف to changes over time. It is characterized by the ease of upgrading and maintaining.

•Dependability

It is the characteristic of the software to be reliable موثوق, secure and safe.

•Efficiency

It is the capability of the software to use resources efficiently.

1.6 SOFTWARE ENGINEERING DEFINITION

Definition of *Software engineer*: A person who designs and programs system-level software, such as operating systems, database management systems (DBMSs) and embedded systems. The title is often used for programmers in the software industry who create commercial software packages, whether they be system level or application level.

While *Software engineering* is an engineering discipline (فرع)that is concerned معني with all aspect of software production from the early stages of system specification through 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 (مناسب) . However, 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 (محددات) 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. It also includes activities such as software project management and the development of tools, methods, and theories to support software production.

1.7 The goals of software engineering.

1. Costs: software engineering must make cost of software development less.

2. Efficiency. The degree to which the software makes optimal use of system resources as indicated by the following sub attributes: time behavior, resource behavior. And this software must work in a way without harming Hardwar like CPU or Memory.

3.Portability. Effort required to transfer the program from one hardware and/or software system environment to another.

4. Maintainability. Ability to locate and fix an error in a program and we can say it is The ease with which repair may be made to the software without needing additional unnecessary cost.

5. Reliability. The amount of time that the software is available for use.

6. Delivery on Time: by Using software engineering concepts, Software must delivered in exact time that the software engineer assigned during software designing.

7.The software should offer appropriate ملائم user interface.

2. SOFTWARE DEVELOPMENT MODELS:

2.1 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 2.1) 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).

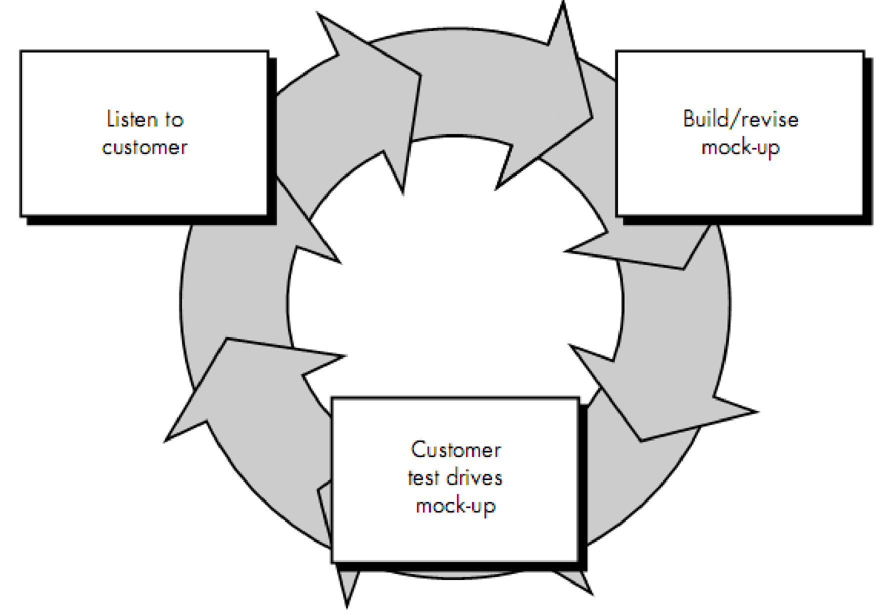


FIGURE 2.1
The prototyping paradigm

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, i.e. either you built Prototype and throw it away "if it's not satisfy customer needs" and plan in advance to build a throwaway, or give this throwaway to customers".

Ideally, the prototype serves as a mechanism for identifying software requirements.

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 throwaway. 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 no one has considered over all 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 demands that "a few fixes" be applied to make the prototype a working product. Too often, software development management relents يرجع في قراره.

2. The developer often makes implementation compromises حلول وسيطة، تسوية in order to get a prototype working quickly. An inappropriate غير ملائم 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 جزءا لايتجزأ من النظام.

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 التخلص منها (at least in part) and the actual software is engineered with an eye toward quality and maintainability.

2.2 THE LINEAR SEQUENTIAL MODEL

Sometimes called the classic life cycle or the waterfall model, the linear sequential model suggests a systematic , نظامي sequential approach to software development that begins at the system level and progresses through analysis, design, coding, testing, and support. Figure 2.2 illustrates the linear sequential model for software engineering.

FIGURE 2.1

FIGURE 2.4
The linear sequential model

System/information engineering

Analysis → Design → Code → Test

Modeled after a conventional تقليدي engineering cycle, 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. Information engineering encompasses requirements gathering at the strategic business level and at the business area level .

Software requirements analysis: The requirements gathering process is intensified يشدد 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 تعاين وتفحص 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 (من غير شك، يقينا) 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 انتقاد of the paradigm has caused even active supporters to question its efficacy [HAN95]. 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, 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 يفوق 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.

2.3 THE INCREMENTAL MODEL

The incremental model combines elements of the linear sequential model (applied repetitively) with the iterative philosophy of prototyping.

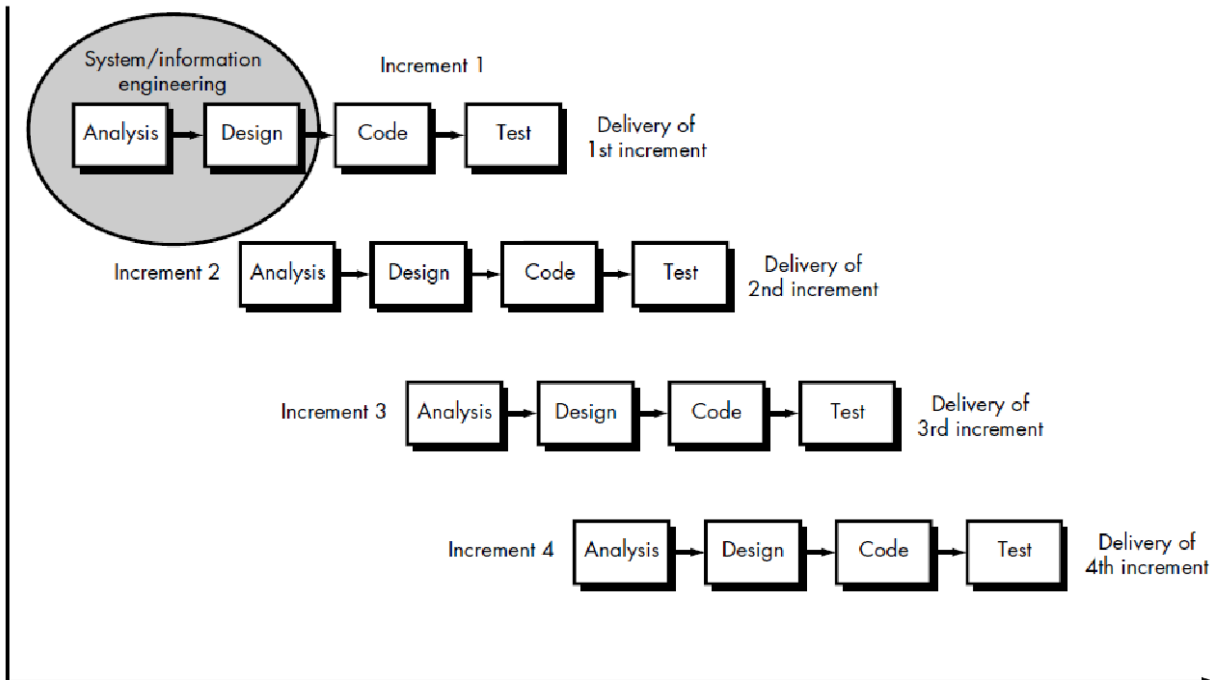
Referring to Figure below, the incremental model applies linear sequences in a staggered fashion as calendar time progresses.

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.

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.

2.4 THE SPIRAL MODEL

The spiral model, 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.*6 Typically, there are between three and six task regions. Figure 2.8 depicts a spiral model that contains six task regions:

Customer communication—tasks required to establish effective communication between developer and customer.
• **Planning**—tasks required to define resources, timelines, and other projectrelated 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.

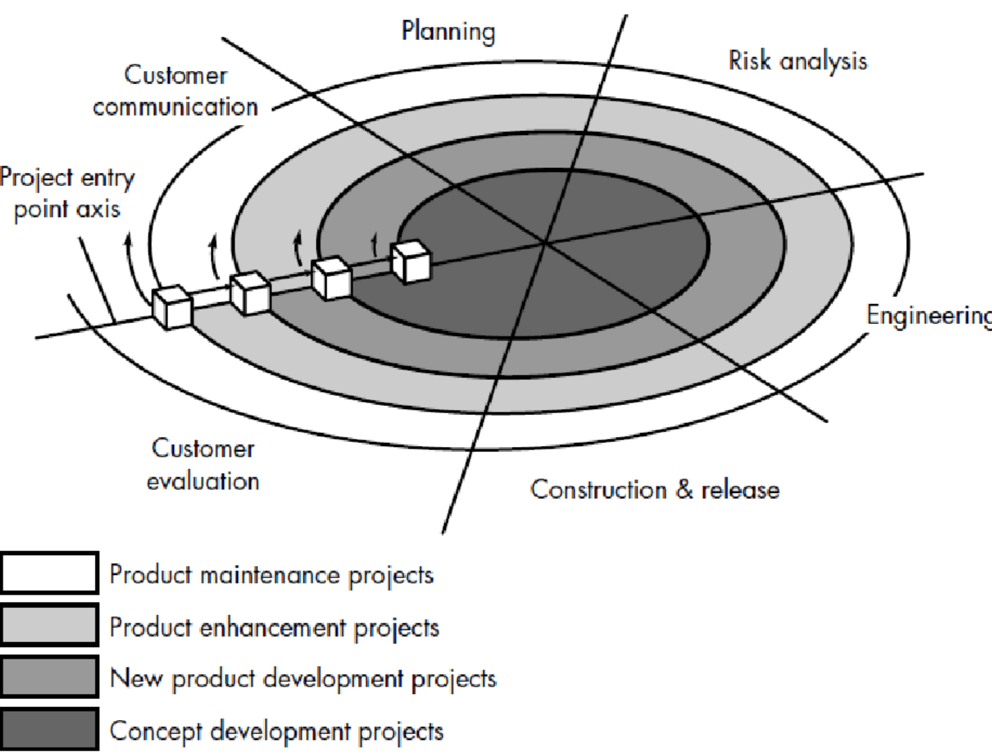


FIGURE 2.8
A typical spiral model

Each of the regions is populated by a set of work tasks, called a *task set,* that are adapted to the characteristics of the project to be undertaken. For small projects, the number of work tasks and their formality is low. For larger, more critical projects, each task region contains more work tasks that are defined to achieve a higher level of formality.

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. An alternative view of the spiral model can be considered by examining the project entry point axis, also shown in The Figure Each cube placed along the axis can be used to represent the starting point for different types of projects. A “concept development project” starts at the core of the spiral and will continue (multiple iterations occur along the spiral path that bounds the central shaded

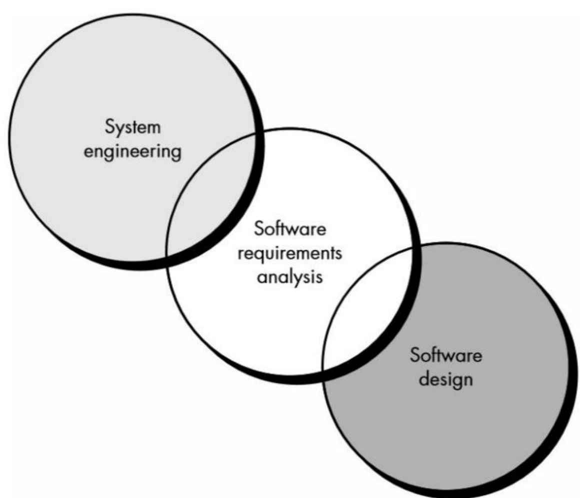
region) until concept development is complete. If the concept is to be developed into an actual product, the process
proceeds through the next cube (new product development project entry point) and a “new development project” is initiated. The new product will evolve through a number of iterations around the spiral, following the path that bounds the region that has somewhat lighter shading than the core. In essence, the spiral, when characterized in this way, remains operative until the software is retired. There are times when the process is dormant, but whenever a change is initiated, the process starts at the appropriate entry point (e.g., product enhancement).

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.

3.1 REQUIREMENTS ANALYSIS

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



Requirements engineering activities result in the specification of software’s operational characteristics (function, data, and behavior), indicate software's interface with other system elements, and establish constraints that software must meet.

Requirements analysis allows the software engineer (sometimes called analyst in this role) 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, and (5) review.

Analysis principles:

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.

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.

Analysis Modeling Objectives

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, and (3) to define a set of requirements that can be validated once the software is built.

3.2 STRUCTURED ANALYSIS

Is a model building activity, using notation that satisfied the operational analysis principles, (that suggest that the information, functional, and behavioral domains of software can be partitioned). we create models that depict information (data and control) content and flow, we partition the system functionality and behaviorally, and we depict the essence of what must be built.

THE ELEMENTS OF THE ANALYSIS MODEL

1-Data Dictionary

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.

2-The entity relation diagram (ERD)

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.

3-The data flow diagram (DFD)

The data flow diagram (DFD) serves two purposes: (1) to provide an indication of how data are transformed as they move through the system and (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).

4-The state transition diagram (STD)

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).

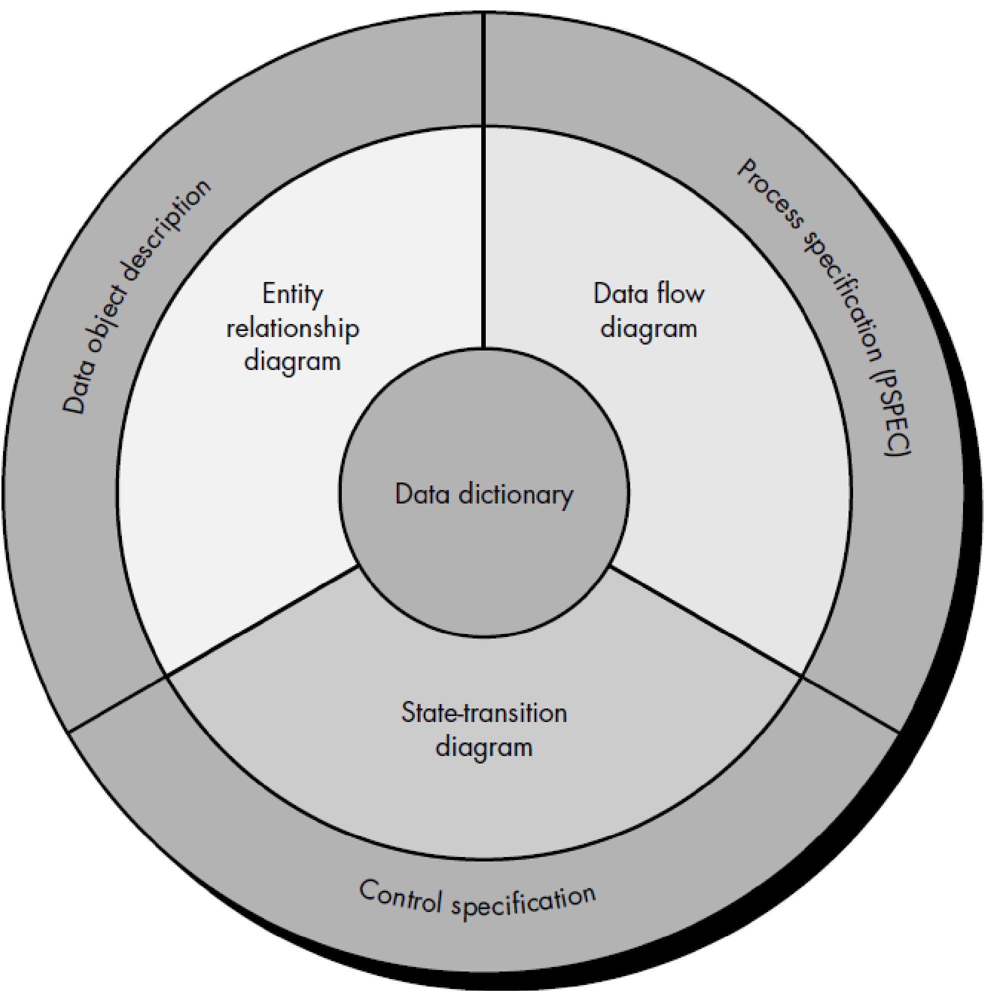


FIGURE The structure of the analysis model

3.3 DATA MODELING

Data modeling answers a set of specific questions that are relevant to any data processing application.
What are the primary data objects to be processed by the system?
What is the composition of each data object and what attributes describe the object? Where do the the objects currently reside? What are the relationships between each object and other objects?
What are the relationships between the objects and the processes that transform them?

Data Modeling
To answer these questions, 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 (discussed in Section 12.4 and used to represent how data are transformed), data modeling considers data independent of the processing that transforms the data.

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 object:

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). For example, a person or a car (Figure 12.2) 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.

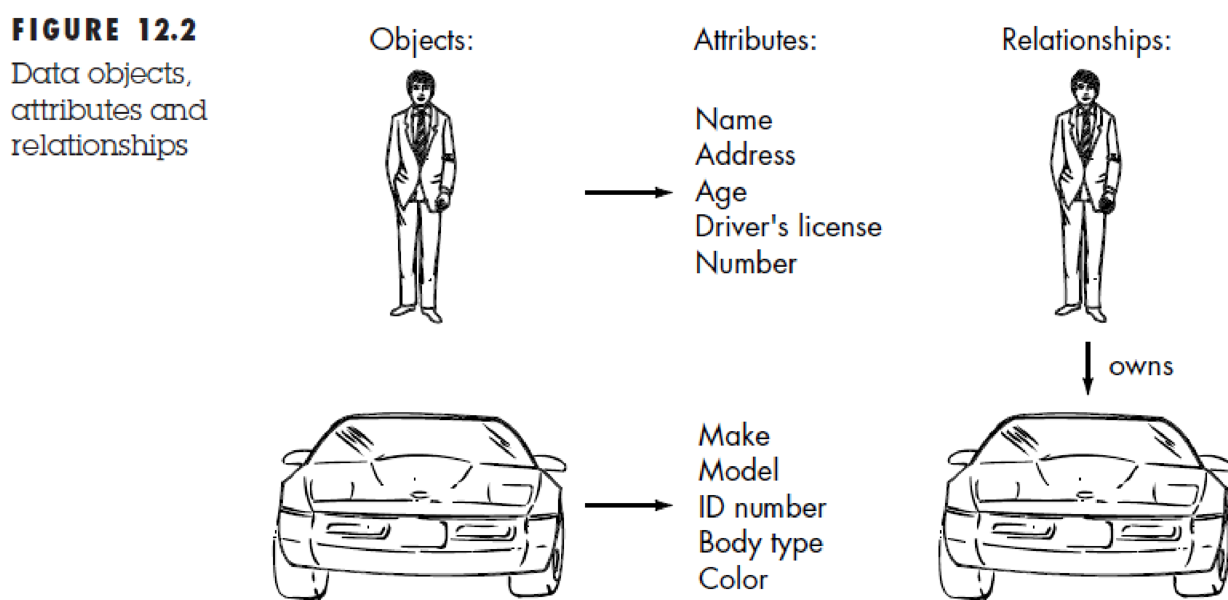
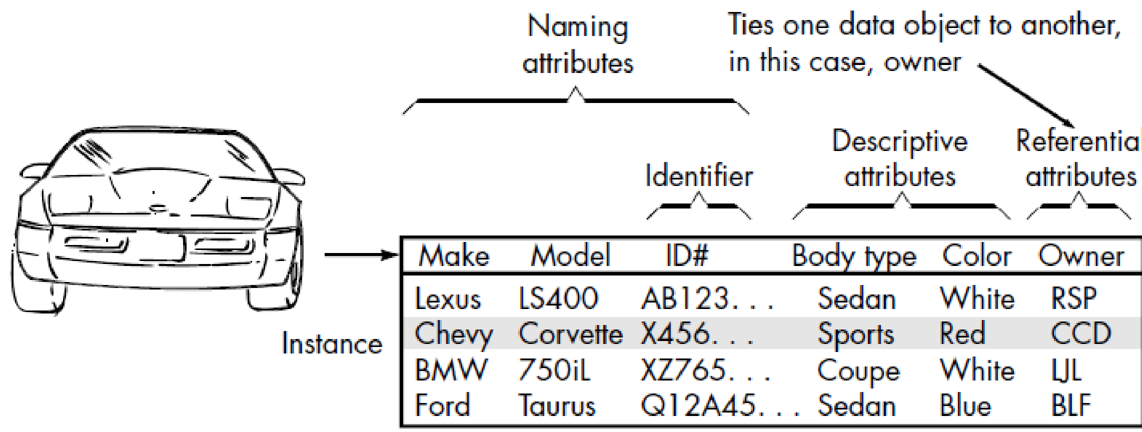


FIGURE 12.2 Data objects, attributes and relationships

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.



Relationships:

Data objects are connected to one another by relationships in different ways. And this relationship can be represented using the simple notation and establishing connection between two objects.

Relationships:

There are two main points in relationships that must be taken into consideration:

Cardinality: The number of times items appear in a relationship.

Modality: Is the relationship obligatory or optional?

Cardinality: Represents the number of times an object appears in a particular relationship and takes two values, either one or many, for example:

One to one (1:1)

One to many (1:m)

Many to many (m:n)

One to one (1:1)

One to many (1:m)

Many to many (m:n)

Modality: it is zero in case the relationship (optional) is not needed and it is one if the relationship is mandatory (necessary).

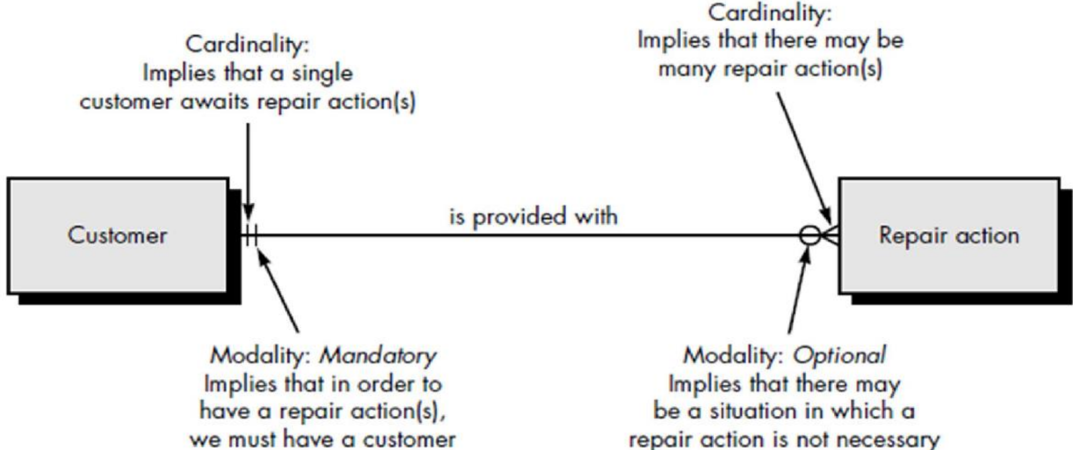
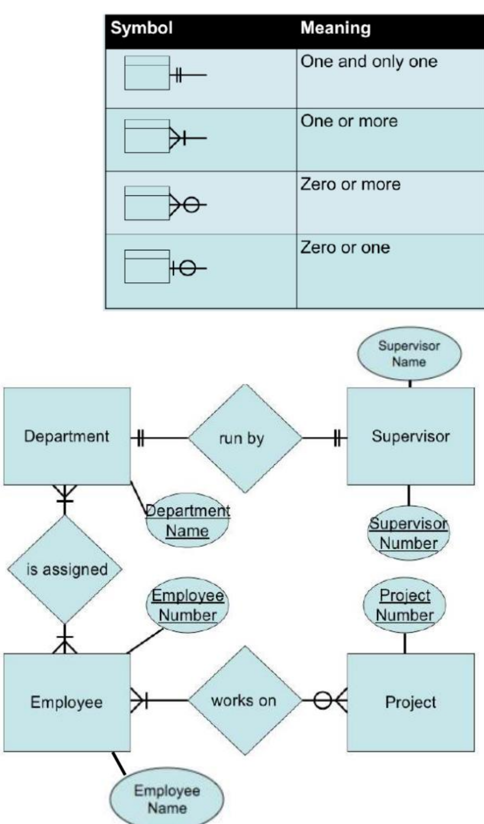


FIGURE Cardinality and Modality

Entity-Relationship Diagrams

A company has several departments. Each department has a supervisor and at least one employee. Employees must be assigned to at least one, but possibly more departments. At least one employee is assigned to a project, but an employee may be on vacation and not assigned to any projects. The important data fields are the names of the departments, projects, supervisors and employees, as well as the supervisor and employee number and a unique project number.



Symbol	Meaning
	One and only one
	One or more
	Zero or more
	Zero or one

Supervisor:

Each department has one supervisor.

Department:

Each supervisor has one department.

Each employee can belong to one or more departments

Employee:

Each department must have one or more employees

Each project must have one or more employees

Project:

Each employee can have 0 or more projects.

Creating an Entity Relationship Diagram (ERD)

Like most elements of the analysis model, the ERD is constructed in an iterative manner. The following approach is taken:

مثل معظم عناصر نموذج التحليل ، يتم إنشاء ERD بطريقة تكرارية. وذلك باتباع النهج التالي:

1. During requirements elicitation, customers are asked to list the “things” that the application or business process addresses. These “things” evolve into a list of input and output data objects as well as external entities that produce or consume information.

1.أثناء مرحلة توضيح المتطلبات ، يُطلب من الزبائن سرد الأشياء التي يعالجها التطبيق أو المعالجة الخاصة بعمل معين. تتطور هذه "الأشياء" إلى قائمة كائنات بيانات المدخلات والمخرجات وكذلك الكيانات الخارجية التي تنتج أو تستهلك المعلومات.

2. Taking the objects one at a time, the analyst and customer define whether or not a connection (unnamed at this stage) exists between the data object and other objects.

2- يتم أخذ الكائنات واحدة في الوقت الواحد ، ويحدد المحلل والعميل ما إذا كان هناك اتصال (غير مسمى في هذه المرحلة) بين كائن البيانات والكائنات الأخرى أم لا.

3. Wherever a connection exists, the analyst and the customer create one or more object/relationship pairs.

3- حيثما يوجد اتصال، يقوم المحلل والزبون بإنشاء زوج أو أكثر من الأزواج بين الكيانين.

4. For each object/relationship pair, cardinality and modality are explored.

4- بالنسبة لكل زوج من العلاقة ، يتم استكشاف وتحديد نوع العلاقة cardinality وmodality

5. Steps 2 through 4 are continued iteratively until all object/relationships have been defined. It is common to discover omissions as this process continues. New objects and relationships will invariably be added as the number of iterations grows.

5- تستمر الخطوات من 2 إلى 4 بشكل متكرر حتى يتم تحديد كافة العلاقات الموجودة بين الكيانات. من الشائع حدوث سهو مع استمرار هذه العملية. ستتم دائمًا إضافة كائنات وعلاقات جديدة كلما زادت عدد التكرارات.

6. The attributes of each entity are defined.

6- يتم تحديد ووضع خصائص لكل كيان.

7. An entity relationship diagram is formalized and reviewed.

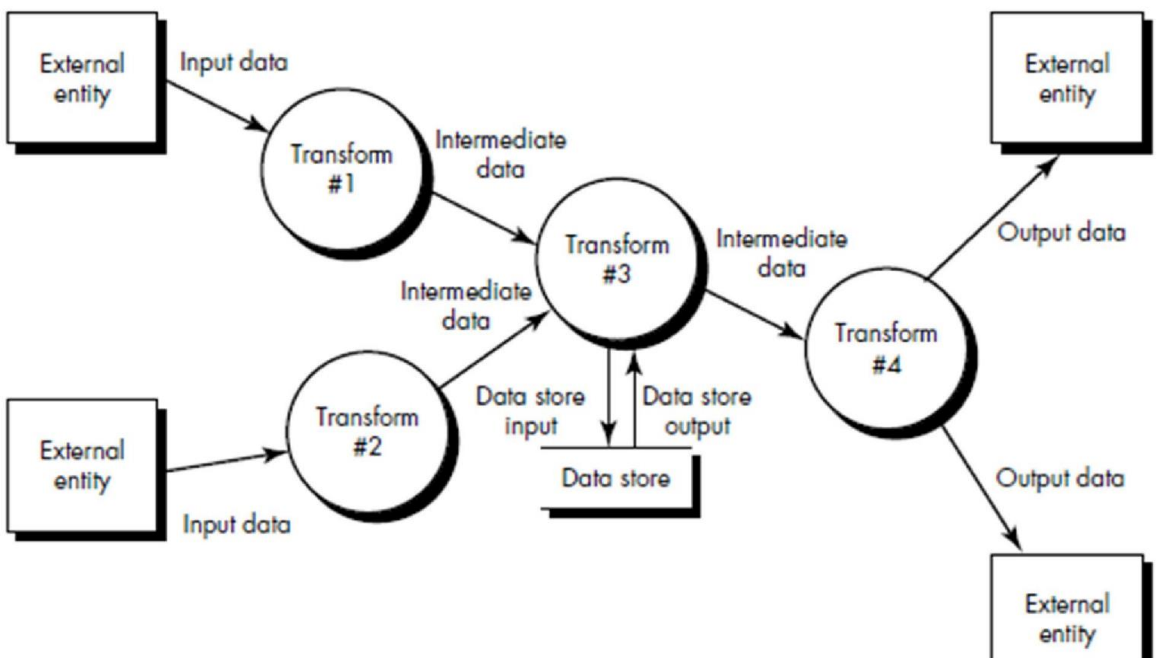
7- رسم مخطط علاقة الكيان ومراجعته.

8. Steps 1 through 7 are repeated until data modeling is complete.

8- يتم تكرار الخطوات من 1 إلى 7 حتى اكتمال نمذجة البيانات.

Greating DFD

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 below.



Information Flow Model

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).

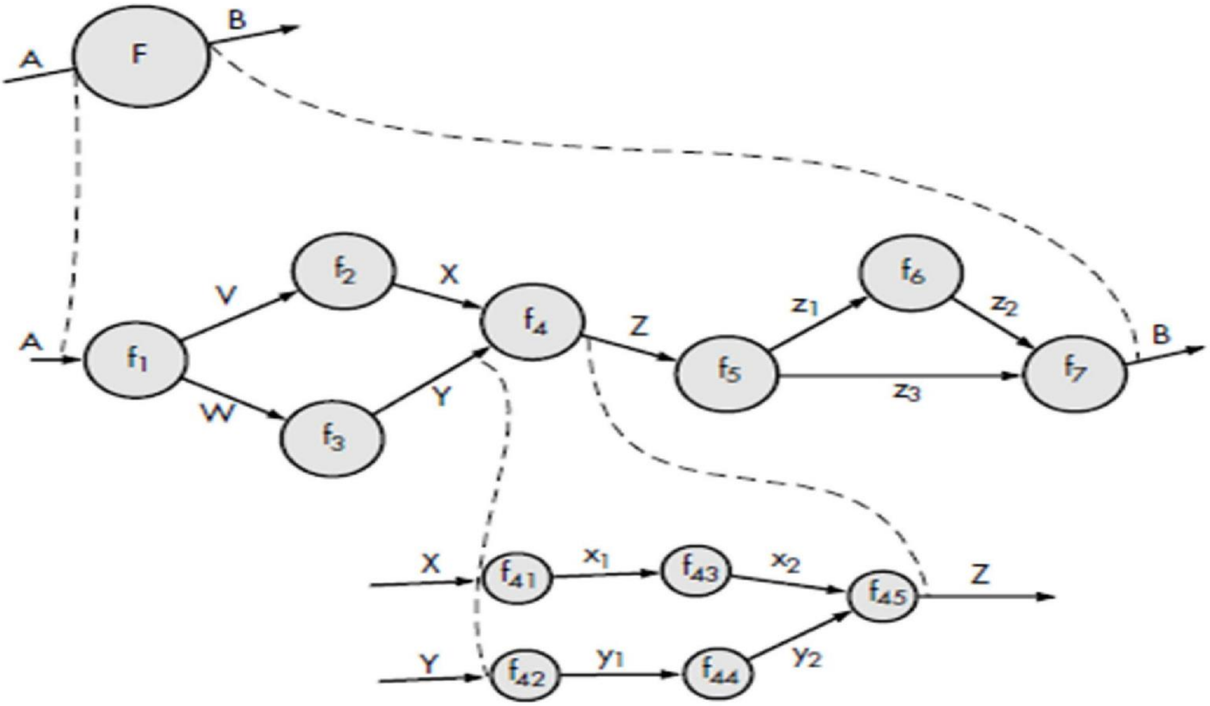
A level 0 DFD, also called a fundamental system model or a context model, represents the entire software element as a single bubble with

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.

DFD graphical notation must be augmented with descriptive text.



Information flow refinements تحسينات

يجب تعزيز التدوين الرسومي DFD بوصف نصي.

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.

Creating a Data Flow Diagram:

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

Software design

Design engineering:

Design engineering encompasses the set of principles, concepts, and practices that lead to the development of a high-quality system or product. The goal of design is to create a model of software that will implement all customer requirements correctly and bring delight to those who use it.Design engineering for computer software changes continually as new methods, better analysis, and broader understanding evolve.

Software design:

Software design sits at the technical kernel of software engineering and is applied regardless of the software process model that is used. Beginning once software requirements have been analyzed and specified, software design is the first of three technical activities—design, code generation, and test—that are required to build and verify the software. Each activity transforms information in a manner that ultimately results in validated computer software.

During design, progressive refinements of data structure, architecture, interfaces, and

Data design :

Data design translates the data objects defined in the analysis model into data structures that reside within the software. The attributes that describe the object, the relationships between data objects and their use within the program all influence the choice of data structures.

Like other software engineering activities, data design (sometimes referred to as data architecting) creates a model of data and/or information that is represented at a high level of abstraction (the customer/user's view of data).

The data objects defined during software requirements analysis are modeled using entity/relationship diagrams and the data dictionary. The data design activity translates these elements of the requirements model into data structures at the software component level and, when

necessary, a database architecture or a data warehouse at the application level.

(2) Architectural design:

The architectural design defines the relationship between major structural elements of the software. It depicts the structure and organization of software components, their properties, and the connections between them.

Software components include program modules and the various data representations that are manipulated by the program. Therefore, data design is an integral part of the derivation of the software architecture.

The primary objective of architectural design is to develop a modular program structure and represent the control relationship between modules, in addition, architectural design melds program structure and data structure, defining interface that enables data flow throughout the program.

The architectural design for software is the equivalent to the floor plan of a house. The floor plan depicts the overall layout of the rooms, their size, shape, and relationship to one another, and the doors and windows that allow movement into and out of the rooms. The floor plan gives us an overall view of the house. Architectural design elements give us an overall view of the software.

(3) component-level design:

component-level design, also called procedural design, occurs after data, architectural, and interface designs have been established.

The component-level design transforms structural elements of the software architecture into a procedural description of software components. Information obtained from the PSPEC, CSPEC, and STD serve as the basis for component design.

Component-level design depicts the software at a level of abstraction that is very close to code.At the component level, the software engineer must represent data structures, interfaces, and algorithms in sufficient detail to guide in the generation of programming language source code.

To accomplish this, the designer uses one of a number of design notations that represent component-level detail in either graphical, tabular, or text- based formats.

Structured programming is a procedural design philosophy that constrains the number and type of logical constructs used to represent algorithmic detail. The intent of structured programming is to assist the designer in defining algorithms that are less complex and therefore easier to read, test, and maintain.

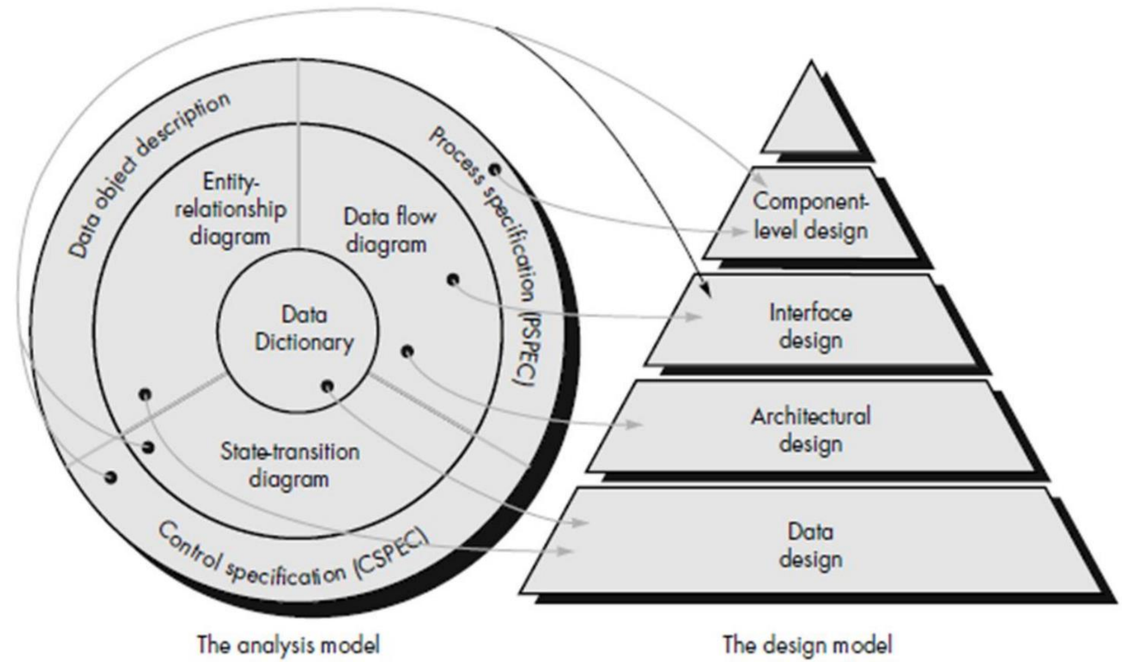


Figure : Translating the analysis model into a software design

Component-level design techniques: Component-level design depicts the software at a level of abstraction that is very close to code.

At the component level, the software engineer must represent data structures, interfaces, and algorithms in sufficient detail to guide in the generation of programming language source code.

To accomplish this, the designer uses one of a number of design notations that represent component-level detail, the following are some of these techniques: a. Structured programming: Structured programming is a design technique that constrains logic flow to a three constructs: sequence, condition, and repetition, used to represent algorithmic detail.

The intent of structured programming is to assist the designer to limit the procedural design of software to a small number of predictable operations, defining algorithms that are less complex and therefore easier to read, test, and maintain.

b. Graphical design notation: The activity diagram allows a designer to represent sequence, condition, and repetition-all elements of structured programming-by using a flowchart. "A picture is worth a thousand words," but it's rather important to know which picture and which 1000 words.

There is no question that graphical tools, such as the flowchart, , provide useful pictorial patterns that readily depict procedural detail. However, if graphical tools are misused, the wrong picture may lead to the wrong software.

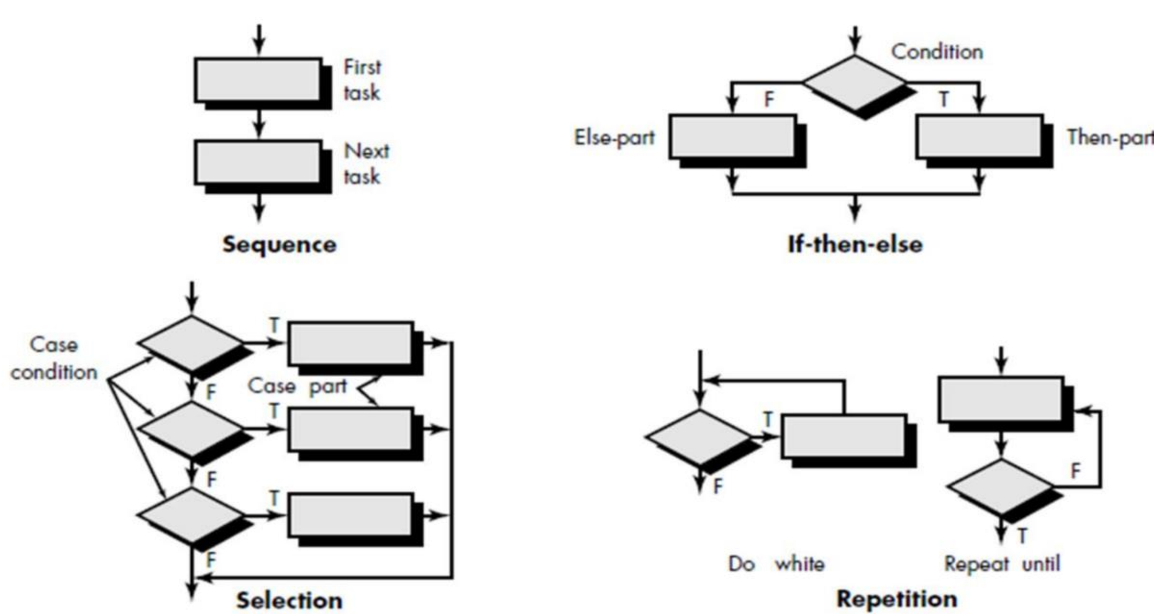


Figure Flowchart constructs

c. Tabular design notation: In many software applications, a module may be required to evaluate a complex combination of conditions and select appropriate actions based on these conditions.

Decision tables provide a notation that translates actions and conditions (described in a processing narrative) into a tabular form.

The table is difficult to misinterpret and may even be used as a machine readable input to a table driven algorithm.

Decision table organization is illustrated in Figure below Referring to the figure, the table is divided into four sections. The upper left-hand quadrant contains a list of all conditions.

The lower left-hand quadrant contains a list of all actions that are possible based on combinations of conditions.

The right-hand quadrants form a matrix that indicates condition combinations and the corresponding actions that will occur for a specific combination. Therefore, each column of the matrix may be interpreted as a processing rule.

	Rules								
Conditions	**1**	**2**	**3**	**4**					n
Condition #1	✓			✓	✓				
Condition #2		✓		✓					
Condition #3			✓		✓				
Actions									
Action #1	✓			✓	✓				
Action #2		✓		✓					
Action #3			✓						
Action #4			✓	✓	✓				
Action #5	✓	✓			✓				

Figure Resultant decision table

Effective Modular Design

Modularity has become an accepted approach in all engineering disciplines. A modular design reduces complexity, facilitates change (a critical aspect of software maintainability), and results in easier implementation by encouraging parallel development of different parts of a system.

1. Functional Independence

The concept of functional independence is a direct outgrowth of modularity and the concepts of abstraction and information hiding.

functional independence is a key to good design, and design is the key to software quality.

2. Cohesion

A cohesive module performs a single task within a software procedure, requiring little interaction with procedures being performed in other parts of a program. Stated simply, a cohesive module should (ideally) do just one thing.

3. Coupling

Coupling is a measure of interconnection among modules in a software structure. Coupling depends on the interface complexity between modules, the point at which entry or reference is made to a module, and what data pass across the interface. Coupling is a qualitative indication of the degree to which a module is connected to other modules and to the outside world.

Program Design Language (PDL)

Program design language (PDL), also called structured English or pseudocode, is "a pidgin language in that it uses the vocabulary of one language (i.e., English) and the overall syntax of another (i.e., a structured programming language)".

At first glance PDL looks like a modern programming language. The difference between PDL and a real programming language lies in the use of narrative text (e.g., English) embedded directly within PDL statements.

Given the use of narrative text embedded directly into a syntactical structure, PDL cannot be compiled (at least not yet). However, PDL tools currently exist to translate PDL into a programming language "skeleton" and/or a graphical representation (e.g., a flowchart) of design. These tools also produce nesting maps, a design operation index, cross-reference tables, and a variety of other information.

A program design language may be a simple transposition of a language such as Ada or C. Alternatively, it may be a product purchased specifically for procedural design.

PDL Example

To illustrate the use of PDL, we present an example of a procedural design for the SafeHome security system software. The system monitors alarms for fire, smoke, burglar, water, and temperature (e.g., furnace breaks while homeowner is away during winter) and produces an alarm bell and calls a monitoring service, generating a voice-synthesized message. In the PDL that follows, we illustrate some of the important constructs noted in earlier sections.

Recall that PDL is not a programming language. The designer can adapt as required without worry of syntax errors. However, the design for the monitoring software would have to be reviewed (do you see any problems?) and further refined before code could be written. The following PDL defines an elaboration of the procedural design for the security monitor component.

```
PROCEDURE security.monitor;
INTERFACE RETURNS system.status;
TYPE signal IS STRUCTURE DEFINED
        name IS STRING LENGTH VAR;
        address IS HEX device location;
```

```
        bound.value IS upper bound SCALAR;
        message IS STRING LENGTH VAR;
END signal TYPE;

TYPE system.status IS BIT (4);
TYPE alarm.type DEFINED
smoke.alarm IS INSTANCE OF signal;
fire.alarm IS INSTANCE OF signal;
water.alarm IS INSTANCE OF signal;
temp.alarm IS INSTANCE OF signal;
burglar.alarm IS INSTANCE OF signal;
TYPE phone.number IS area code + 7-digit number;
•
•
•
initialize all system ports and reset all hardware;
CASE OF control.panel.switches (cps):
        WHEN cps = "test" SELECT
         CALL alarm PROCEDURE WITH "on" for test.time in seconds;
         WHEN cps = "alarm-off" SELECT
         CALL alarm PROCEDURE WITH "off";
         WHEN cps = "new.bound.temp" SELECT
          CALL keypad.input PROCEDURE;
WHEN cps = "burglar.alarm.off" SELECT deactivate signal [burglar.alarm];
•
•
•
DEFAULT none;
ENDCASE
REPEAT UNTIL activate.switch is turned off
reset all signal.values and switches;
```

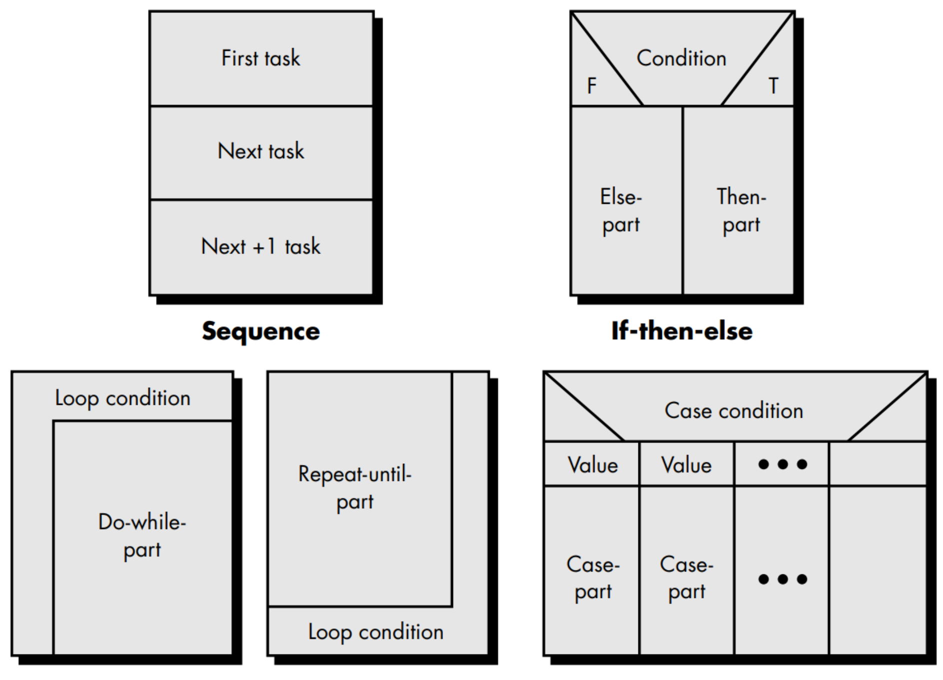
```
DO FOR alarm.type = smoke, fire, water, temp, burglar;
     READ address [alarm.type] signal.value;
     IF signal.value > bound [alarm.type]
        THEN phone.message = message [alarm.type];
         set alarm.bell to "on" for alarm.timeseconds;


          PARBEGIN
              CALL alarm PROCEDURE WITH "on", alarm.time in seconds;
               CALL phone PROCEDURE WITH message [alarm.type],
          phone.number;
          ENDPAR
      ELSE skip
ENDIF
ENDFOR
ENDREP
END
```

Box Diagram (Nassi-Shneiderman charts, N-S charts)

It’s a graphical design tool, evolved from a desire to develop a procedural design representation that would not allow violation of the structured constructs. Developed by Nassi and Shneiderman and extended by Chapin, the diagrams (also called Nassi-Shneiderman charts, N-S charts, or Chapin charts) have the following characteristics: (1) functional domain (that is, the scope of repetition or if-then-else) is well defined and clearly visible as a pictorial representation, (2) arbitrary transfer of control is impossible, (3) the scope of local and/or global data can be easily determined, (4) recursion is easy to represent.

The graphical representation of structured constructs using the box diagram is illustrated in Figure below. The fundamental element of the diagram is a box. To represent sequence, two boxes are connected bottom to top. To represent if-then-else, a condition box is followed by a then-part and else-part box. Repetition is depicted with a bounding pattern that encloses the process (do-while part or repeat-until part) to be repeated. Finally, selection is represented using the graphical form shown at the bottom of the figure. Like flowcharts, a box diagram is layered on multiple pages as processing elements of a module are refined. A "call" to a subordinate module can be represented within a box by specifying the module name enclosed by an oval.


First task
Next task
Next +1 task
Sequence
Condition
F
T
Else-part
Then-part
If-then-else
Loop condition
Do-while-part
Repeat-until-part
Loop condition
Repetition
Case condition
Value
Value
Case-part
Case-part
Selection

SOFTWARE TESTING

Software Testing Once source code has been generated, software must be tested to uncover (and correct) as many errors as possible before delivery to the customer. Software testing is a critical element of software quality assurance and represents the ultimate review of specification, design, and code generation.

Testing Objectives

There is a number of rules that can serve well as testing objectives: Testing is a process of executing a program with the intent of finding an error.

1. A good test case is one that has a high probability of finding an as-yet- undiscovered error.

3. A successful test is one that uncovers an as-yet-undiscovered error..

Testing Goals

The software testing process has two distinct goals:

1. To demonstrate to the developer and the customer that the software meets its requirements. This means that there should be at least one test for every requirement in the user and system requirements documents.
2. To discover faults or defects in the software where the behavior of the software is incorrect, undesirable or does not conform to its specification.

Testing Principles

Before applying methods to design effective test cases, a software engineer must understand the basic principles that guide software testing. A set of testing principles that have been adapted for use:

.

• **All tests should be traceable to customer requirements**. As we have seen, the objective of software testing is to uncover errors. It follows that the most severe defects (from the customer's point of view) are those that cause the program to fail to meet its requirements.

• **Tests should be planned long before testing begins**. Test planning can begin as soon as the requirements model is complete. Detailed definition of test cases can begin as soon as the design model has been solidified. Therefore, all tests can be planned and designed before any code has been generated.

• **The Pareto principle applies to software testing**. Stated simply, the Pareto principle implies that 80 percent of all errors uncovered during testing will likely be traceable to 20 percent of all program components. The problem, of course, is to isolate these suspect components and to thoroughly test them.

• **Testing should begin “in the small” and progress toward testing “in the large**.” The first tests planned and executed generally focus on individual components. As testing progresses, focus shifts in an attempt to find errors in integrated clusters of components and ultimately in the entire system.

•**Exhaustive testing is not possible**. The number of path permutations for even a moderately sized program is exceptionally large. For this reason, it is impossible to execute every combination of paths during testing. It is possible, however, to adequately cover program logic and to ensure that all conditions in the component-level design have been exercised.