جامعة بغداد
كلية التربية للعلوم الصرفة ـ ابن الهيثم
قسم علوم الحاسوب

المـــادة : هندسة البرامجيات
مدرس المادة : د. علي يحيى

المرحــلة : الثالثـــــة
الدراسة: الصباحية / المسائية
العام الدراسي : ٢٠١٨ـ٢٠١٩

Chapter one

Introduction

Definition

Computer software is the product that software professionals build and then support over the long term. It includes the following:

(1) Instructions (computer programs) that when executed provide desired function and performance.

(2) Data structures that enable the programs to adequately manipulate information.

(3) Documents that describe the operation and use of the programs.

Software Types

There are two fundamentals 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. **Customized (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.

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.

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

2. Software doesn't "wear out"

  As time passes, the failure rate rises as hardware components suffer from the cumulative affects 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.

 However, the implication is clear—software doesn't wear out. But it does deteriorate!

 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 construction, most software continues to be custom built

Consider 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 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 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.

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 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.

Software Applications

There are *seven* categories of computer software:

1. 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).

2. Application Software

Application software consists of standalone programs that solve a specific business need. Application in this area process business or technical data in a way that facilitates business operations or management / technical decision-making. In addition to conventional data processing applications, application software is used to control business functions in real-time (e.g., point-of-sale transaction processing, real-time manufacturing process control).

3. Engineering / 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.

4. Embedded Software

Embedded software resides within a product or system and is used to implement and control features and functions for the end-user and for the system itself. Embedded software can perform 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, etc.).

5. Product-Line Software

Designed to provide a specific capability for use by many different customers, product-line software can focus on a limited marketplace (e.g., inventory control products) or address mass consumer markets (e.g., word processing, computer graphics, multimedia, entertainment, database management, personal and business financial applications).

6. Web-Applications

The Web pages retrieved by a browser are software that incorporates 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.

7. Artificial intelligence software

Artificial intelligence (AI) software makes use of non numerical algorithms to solve complex problems that are not amenable to computation or straightforward analysis. Applications within area include robotics, expert systems, pattern recognition (image and voice), artificial neural networks, Theorem proving, and game playing.

Software Crisis

The software crisis resulted directly from the introduction of third generation computer hardware which based on integrated circuits. These

machines were orders of magnitude more powerful than the second generation machines and their power made unrealizable computer applications a feasible proposition.

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 predicated, was unreliable, was difficult to maintain and performed poorly. Software development was in crisis. Hardware costs were tumbling while software costs were rising rapidly. New techniques and methods were needed to control the complexity inherent in large software systems.

Many industry observers have characterized the problems associated with software development as a "crisis".

 In 1968 a conference was held to discuss the "software crisis". During this conference, the notion of "*software engineering"* was first proposed.

The crisis manifested itself in several ways:

- Projects running over-budget.
- Projects running over-time.
- Projects were unmanageable and code difficult to maintain.
- Software was very inefficient.
- Software was of low quality.
- Software often did not meet requirements.
- Software was never delivered.

Software Engineering

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 appropriate, but they use the 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 constrains, so they look for solutions within these constrains.

 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, this is often the most effective way to produce high-quality software.

The Characteristics Of Well-Designed Software System

The following attributes which any well-designed software system should possess:

1. **Maintainability:** Software should be written in such a way that it may evolve to meet the changing needs of customers.

2. Dependability: 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.

3. Efficiency: Software should not make wasteful use of system resources such as memory and processor cycles. Efficiency therefore includes responsiveness, processing time, memory utilization, etc.

4. Usability: 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.

The goals of software engineering

Software engineering aims to achieve the following goals:

1. Costs: It should reduce the cost of the development operation and maintenance of the software.

2. Efficiency: The software produced in time expected within the limits of available resources, the software that is produced runs within the time expected for various computation to be completed, and produces the correct output.

3. portability: The software system can be ported to other computers or systems without major rewriting of the software. The software needs only to be re-compiled in order to have probably working on the new machine is considered to be very portable.

4. Maintenance: The software should maintainable because the software is subject to regular change as a long-lifetime. It is important that the

software is written and documented such a way that changes can be made without undue costs.

5. Reliability: The reliability of a system is the probability, over a given period of time, that the system will correctly deliver services as expected by the user. An appropriate level of reliability is essential if a software is to be of any use (in multi-user systems, the system performs it's function even with other on the system).

6. Delivery on time: The high quality software should be produced within predicated date.

7. The software should offer appropriate user interface: It is clear that much software is not used to its full potential simply because the interface which it offers make it difficult to use. The user interface designs take into account the capabilities and background of the intended system users and should be tailored accordingly.

CHAPTER TWO

SW Process Models

Introduction:

To solve actual problems in an industry setting, a software engineer or a team of engineers must incorporate a development strategy that encompasses the process, methods, and tools. This strategy is often referred to as a *process model* or a *software engineering paradigm.* All software processes include:

- **Software Specification:** the functionality of the software and constraints on operation must be defined.
- **Software Development (design and implementation):** the software to meet the specification must be produced.
- **Software Validation:** the software must be validated to ensure that it does what the customer wants.
- **Software Evolution:** the software must evolve to meet changing customer needs.

A process model for software engineering is chosen based on the nature of the project and application, the methods and tools to be used, and the controls and deliverables that are required.

There are **three** generic process models that widely used in current software engineering practice:

1. **The Waterfall Model**: This takes the fundamental process activities of specification, development, validation and evolution and represent them as separate process phases such as requirements specification, software design, implementation, testing and so on.

2. **Evolutionary development:** This approach interleaves the activities of specification, development and validation. An initial system is rapidly

developed from abstract specifications. This is then refined with customer input to produce a system that satisfies the customer needs. Evolutionary models are iterative. A prototyping and spiral model, are examples of evolutionary models.

3. Component-based software engineering: This approach is based on the existence of a significant number of reusable components. The system development process focuses on integrating these components into a system rather than developing them from scratch.

The Waterfall Model

The waterfall model, sometimes called the *classic life cycle,* is the oldest and simplest model of software development process that views the stages as successors to one another. It is called waterfall model because the development stages proceed as a flow, with the diagram of phases.

looking a little like a waterfall. Figure (2.1) shows the waterfall model stages. The following stage should not start until the previous stage has finished. In practice, these stages overlap and feed information to each other. The software process is not a simple linear model but involves a sequence of iterations of the development activities.

The waterfall model should only be used when the requirements are well understood and unlikely to change radically during system development.

Waterfall Model Stages

The principal stages of the model are:

1. **Requirements Analysis and Definition:** The system's services, constrains and goals are established by consultation with system users. They are then defined in detail and serve as a system specification.

2. **System and Software Design:** The system design process partitions the requirements to either hardware or software systems. It establishes overall system architecture. Software design involves identifying and describing the fundamental software system abstractions and their relationships.

3. **Implementation and Unit Testing:** During this stage, the software design is realized as a set of programs or program units. Unit testing involves verifying that each unit meets its specification.

4. **Integration and System Testing:** The individual program units or programs are integrated and tested as a complete system to ensure that the software requirements have been met. After testing, the software system is delivered to the customer.

5. **Operation and Maintenance:** Normally this is the longest life-cycle phase. The system is install and put into practical use. Maintenance involves correcting errors which were not discovered in earlier stages of the life cycle, improving the implementation of system units and enhancing the system's services as new requirements are discovered.

Advantages

The advantages of the waterfall model are that documentation is produced at each phase and that it fits with other engineering process models.
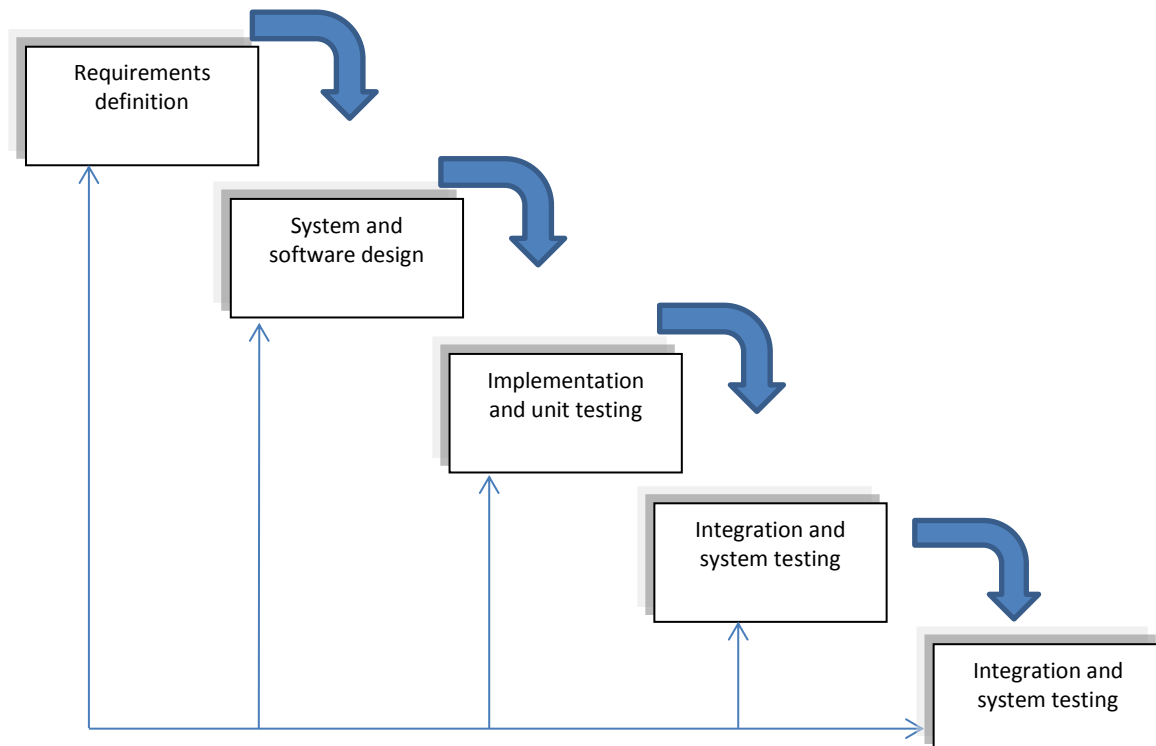

Figure (2.1): The waterfall model of software development stages

Disadvantages (problems) of Waterfall Model

Among the problems that are sometimes encountered when the waterfall 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 waterfall 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.

 4. 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.

 5. Today, software work is fast-paced and subject to a never-ending stream of changes (to features, functions, and information content). The waterfall model is often inappropriate for such work.

Prototyping Model

The prototyping model is an iterative development process which an example of *evolutionary development model*. It may offers the best approach for the following cases:

  (a) Concentrates on experimenting with the customer requirements that are poorly understood, and hence, develop a better requirements definition for the system.

  (b) 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.

When implementing a prototype model, you first develop the parts of the system you understand least, then you begin by developing the parts of the system you understand best. See figure (2.2).

Prototyping Model Stages

1. **Communication:** The software engineer and customer meet and define the overall objectives for the software, identify whatever requirements are known, and outline areas where further definition is mandatory. The customer and developer must agree that the prototype is built to serve as a mechanism for defining requirements.

2. **Quick design:** The quick design focuses on a representation

of those aspects of the software that will be visible to the customer/end-user (e.g., human interface layout or output display formats).

3. **Construction of prototype:** The quick design leads to the construction of a prototype, i.e., the prototype is deployed.

4. **Prototype evaluation:** The prototype is evaluated by the customer/user.

5. **Requirements refinement:** Feedback used to refine requirements for the software. 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.

6. **System engineering:** Ideally, the prototype serves as a mechanism for identifying software requirements. After all requirements have been identified, then the next stages are started (design, construction, testing,
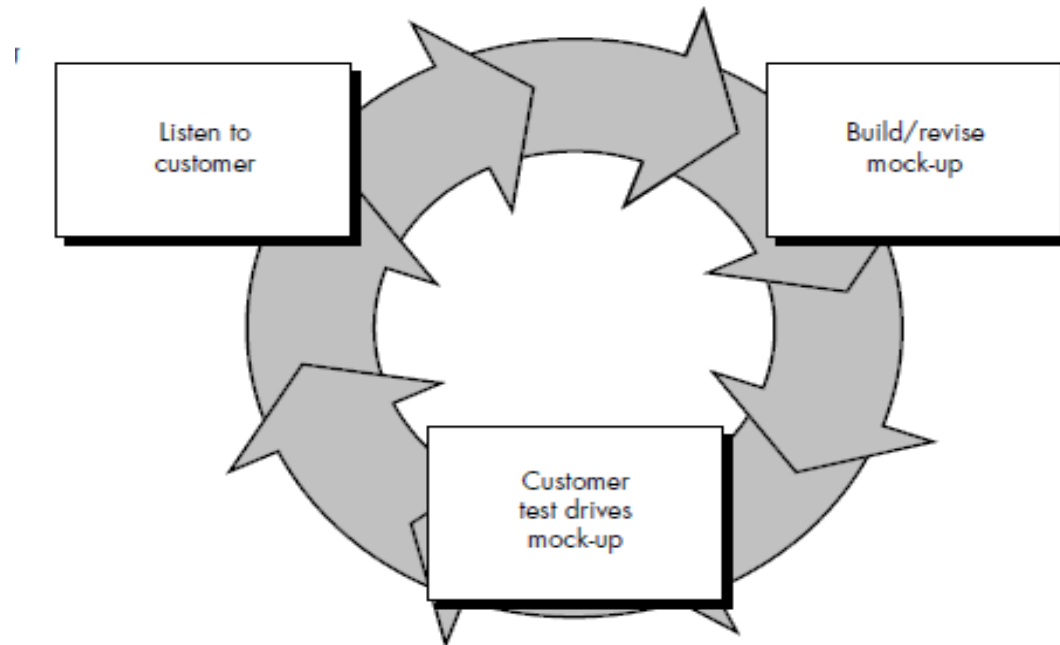
Figure (2.2): Prototyping Model

Advantages of Using Prototype Model

In (1995), a study of (39) prototyping projects found that the benefits (advantages) of using prototyping were:

1. Improved system usability.

2. A closer match of the system to users' needs.

3. Improved design quality.

4. Improved maintainability.

5. Reduce development effort

Disadvantages of Using Prototype Model

Prototyping model can be problematic for the following reasons:

1. The customer sees what appears to be a working version of the software, unaware that in the rush to get it working no one has considered overall software quality or long-term maintainability.

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.

3. It may be impossible to tune the prototype to meet some important requirements that were ignored during prototype development, such as performance, security, robustness and reliability.

4. Rapid change during development inevitably means that prototype is undocumented. Only the design specification is the prototype code. This is not good enough for long-term maintenance.

The Incremental Model

The *incremental model* combines elements of the waterfall model applied in an iterative fashion. Referring to Figure (2.3), the incremental model applies linear sequences in a staggered fashion as calendar time progresses. Each linear sequence produces a deliverable "increment" of the software.

In an incremental development process, customers identify, in outline, the services to provided by the system. They identify which of the services

are most important and which are least important to them. A number of delivery increments are then defined, with each increment providing a sub-set of the system functionality.

Once an increment is completed and delivered, customers can put it into service. As new increments are completed, they are integrated with existing increments so that the system functionality improves with each delivered increment.

For example, word-processing software developed using the incremental

 model 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 model.
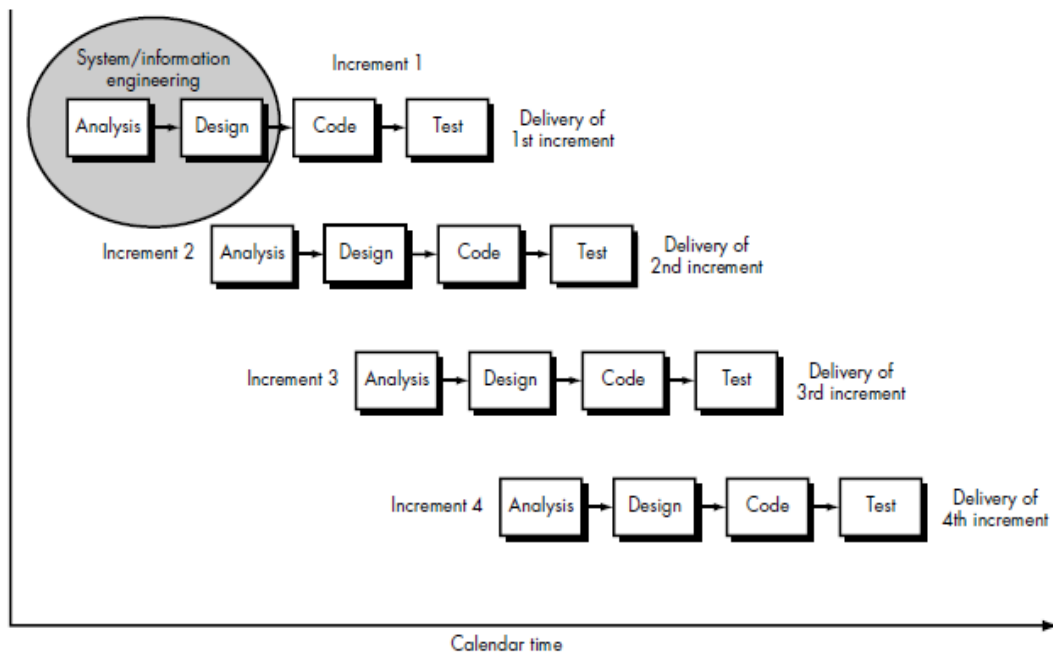

Figure (2.3): The incremental model

Advantages of Using Incremental Model

The incremental development process has a number of advantages:

1. Customers do not have to wait until the entire system is delivered before they can gain value from it. The first increment satisfies their most critical requirements so they can use the software immediately.

2. Customers can use the early increments as prototypes and gain experience that informs their requirements for later system increments.

3. There is a lower risk overall project failure. Although problems may be encountered in some increments, it is likely that will be successfully

delivered to the customer.

4. It is 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, then additional staff (if required) can be added to implement the next increment.

The Spiral Model

The *spiral model,* originally proposed by Boehm (1988), is an evolutionary software process model that couples the iterative nature of prototyping with the controlled and systematic aspects of the waterfall model. It provides the potential for rapid development of incremental versions of the software. Rather than represent the software process as a sequence of activities with some backtracking from one activity to another, the process is represented as a spiral. As this process begins, the software engineering team moves around the spiral in a clockwise direction, beginning at the centre. Each loop in the spiral represent a

phase of software process. Thus, the innermost loop might be concerned with system feasibility, the next loop with requirements definition, the next loop with system design and so on.
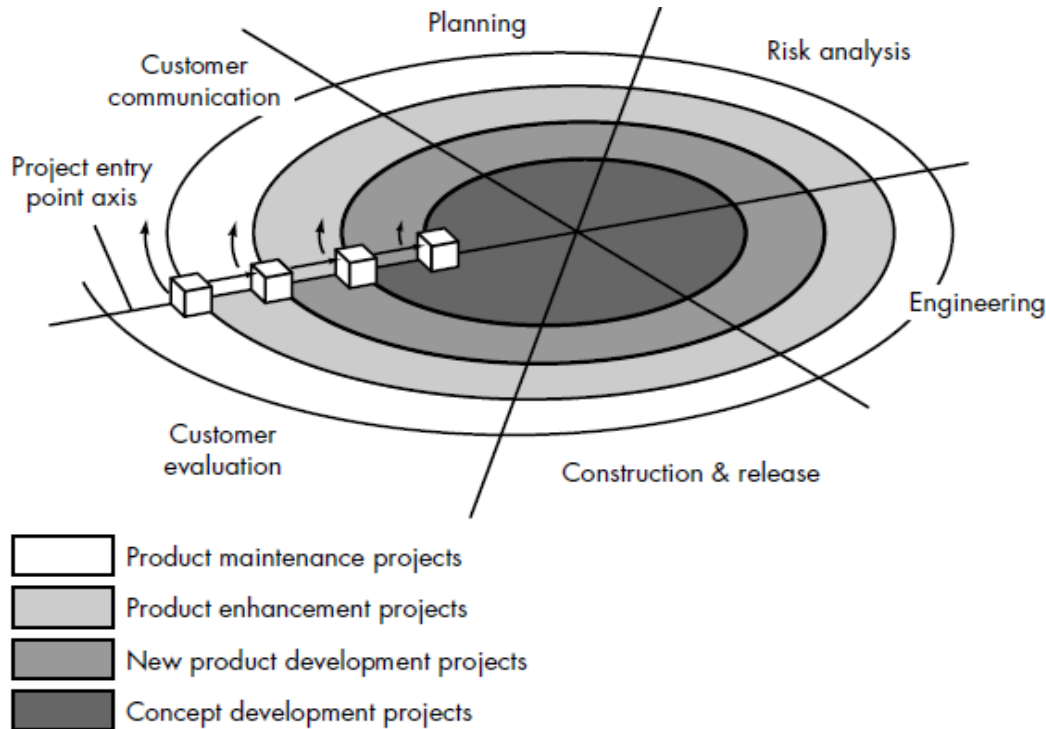
Figure (2.4: The Spiral Model

The Spiral Model Task Regions

Figure (2.4) 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 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.

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

Spiral Model Advantages

1. Focuses attention on reuse options.
2. Focuses attention on early error elimination.
3. Puts quality objectives up front.
4. Integrates development and maintenance.
5. Provides a framework for hardware/software development.

Spiral Model Problems

Like other paradigms, the spiral model is not a panacea, for many reasons:

1-Contractual development often specifies process model and deliverables in advance.

2-Requires risk assessment expertise and relies on this expertise for success.

3- It may be difficult to convince customers (particularly in contract situations) that the evolutionary approach is controllable.

4-The model has not been used as widely as the prototyping paradigms.

CHAPTER THREE

Software Requirements Definitions

And

Analysis

Requirements Definitions

The *requirements* for a system are the descriptions of the services provided by the system and its operational constraints. These requirements reflect the needs of customers for a system that helps solve some problem such as controlling a device, placing an order or finding information.

The process of finding out, analyzing, documented and checking these services and constraints is called *requirements engineering.*

We need to write requirements at different levels of detail because different types of readers use them in different ways. To distinguish between them we can use the term *user requirements* to mean the high-level abstract requirements and *system requirements* to mean the detailed description of what the system should do.

User Requirements

Are statements, in natural language with simple tables and diagrams (that are easily understood), of what services the system is expected to provide. Thus, they should be expressed in such a way that they are understandable by non-specialist staff. They are intended for use by people involved in using and procuring the system. The user requirements for a system should describe the functional and non-functional requirements so that they are understandable by system users without detailed technical knowledge. They should avoid, as far as possible, system design characteristics.

System Requirements

System requirements are expanded versions of the user requirements that are used by software engineers as the starting point for the system design. They set out the system's functions, services and operational constraints in detail, they may be written in a structured form of natural language (to reduce ambiguity). The system requirements document (sometimes called a *functional specification*) should be precise. They should define exactly what is to be implemented. They may be used as part of the contract for the implementation between the system buyer and the software developers. Thus, they should be a complete and consistent specification of the whole system.

Functional Requirement

These are statements of services the system should provide, how the system should react to particular inputs and how the system should behave in particular situations. In other words, functional requirements for a system describe what the system should do.

Non-Functional Requirements

As the name suggests, they are requirements that are not directly concerned with specific functions delivered by the system. These are constraints on the services or functions offered by the system. They include timing constraints, constraints on the development process. Non-functional requirements often apply to the system as a whole. They do not usually just apply to individual system features or services.

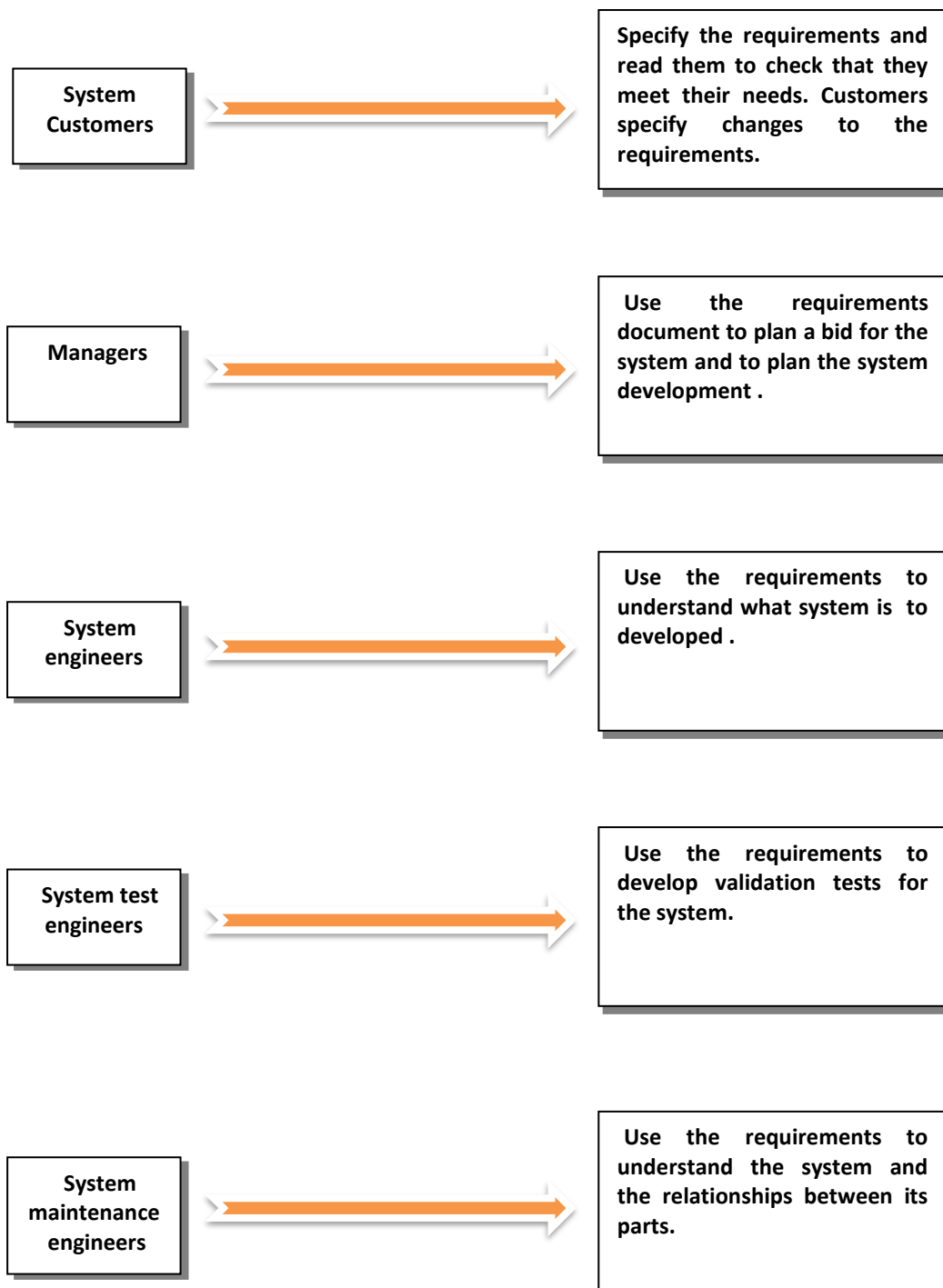## Requirements Engineering (Software Specification) Processes

They are the processes of understanding and defining what services are required from the system and identifying the constraints on the system's operation and development. It is necessary to understand requirements before design and construction of a computer-based system can begin. To accomplish this, a set of requirements engineering processes (tasks) are conducted. Requirements engineering process includes a feasibility study, requirements definition and analysis, requirements specification, requirements validation and requirements management. The goal of requirements engineering process is to create and maintain a *system requirements document.*

The Software Requirements Document

The software requirements document (sometimes called the software requirements specification or **SRS**) is the official statement of what the system developers should implement. It should include both the user the user requirements for a system and a detailed specification of the system requirements. In some cases, the user and system requirements may be integrated into a single description. In other cases, the user requirements are defined in an introduction to the system requirements specification. The software requirements document is the agreed statement of the system requirements. It should be organized so that both system customers and software developers can use it. The diversity of possible users means that the requirements document has to compromise between communicating the requirements to customers, defining the requirements in precise detail for developers and testers, and including information about possible system evolution. Information on anticipated changes can help system designers avoid restrictive design decisions and help system

maintenance engineers who have to adapt the system to new requirements.

The following diagram illustrates possible users of the requirements document and how they use it:

System Customers	**Specify the requirements and read them to check that they meet their needs. Customers specify changes to the requirements.**
Managers	**Use the requirements document to plan a bid for the system and to plan the system development .**
System engineers	**Use the requirements to understand what system is to developed .**
System test engineers	**Use the requirements to develop validation tests for the system.**
System maintenance engineers	**Use the requirements to understand the system and the relationships between its parts.**

The structure for a requirement document is as follows:

1.Preface

This should define the expected readership of the document and describe its history, including a rationale for the creation of a new version and a summary of the changes made in each version.

2.Introduction

This should describe the need for the system, Briefly describe its functions and explain how it will work with other systems. It should describe how the system fits into the overall business or strategic objectives of the organization using the software.

3.Glossary

This should define the technical terms used in the document.

4.User   Requirements   Definition

This section should describe the services provided for the user and the non-functional system requirements. This description may use natural language, diagrams and other notations that are understandable by customers.

5.System   Architecture

This should present a high-level overview of the anticipated system architecture showing the distribution of functions across system modules.

6.System   Requirements   Specification

This should describe the functional and non-functional requirements in more detail.

7.System-models

This should set out one or more system models showing the relationships between the system components and the system and its environment. These may be object models, data-flow models, data models.

8.System-Evolution

This should describe the fundamental assumptions on which the system is based and anticipated changes due to hardware evolution, changing user needs, etc.

9.Appendices

These should provide details, specific information which is related to the application which is being developed. Examples of appendices that may be included are hardware and database descriptions (hardware requirements, database requirements).

10.Index

Several indexes to the document may be included. As well as a normal alphabetic index, there may be an index of diagrams, an index of functions, etc.

Requirements Analysis

Requirements analysis is the first technical step in the software process. *Requirements analysis* is a software engineering task that bridges the gap between system level requirements engineering and software design (Figure3.1).Requirements analysis allows the software engineer (sometimes called *analyst* or *modeler* 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.

Software requirements analysis may be divided into <u>five areas of effort</u>: (1) Problem recognition.

(2) Evaluation.
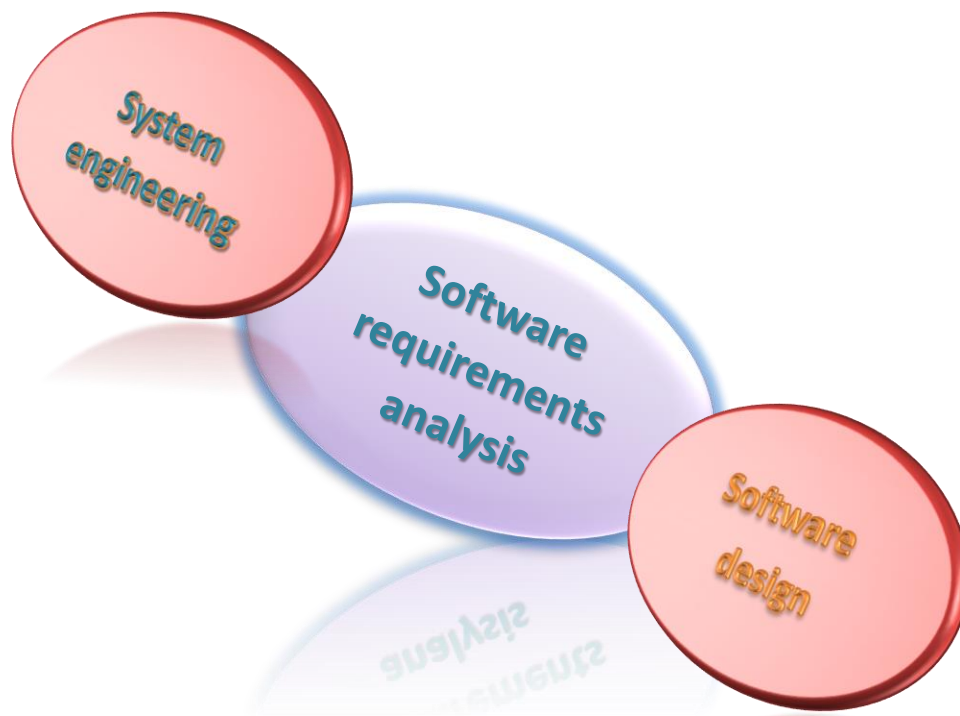
(3) Modeling.

(4) Specification.

(5) Review.

Figure (3.1): Analysis as a Bridge Between System Engineering and Software Design

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 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.

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 <u>three</u> 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.

Analysis Modeling Approaches

Over the years many methods have been proposed for analysis modeling.

However, <u>two</u> now dominate**:**

1. **Structure Analysis:** This approach, is a widely used for analysis modeling, considers data and the processes that transform the data as separate entities. Data objects are modeled in a way that defines their attributes and relationships. Processes that manipulate data objects are modeled in a manner that shows how they transform data as data objects flow through the system.

2. **Object-Oriented Analysis:** This approach to analysis modeling focuses on the definition of classes and the manner in which they collaborate with one another to effect user requirements.

Structured Analysis

Is a model building activity, using notation that satisfied the operational analysis principles, 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 Structured Analysis

The elements of structured analysis take the form as illustrated in (figure 3.2):

(1) Data dictionary (D.D.): Is a repository that contains descriptions of all data objects consumed or produced by the software.

It 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.

The data dictionary has been proposed as a quasi-formal grammar for describing the content of objects defined during structured analysis.

Most *data dictionaries* 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.
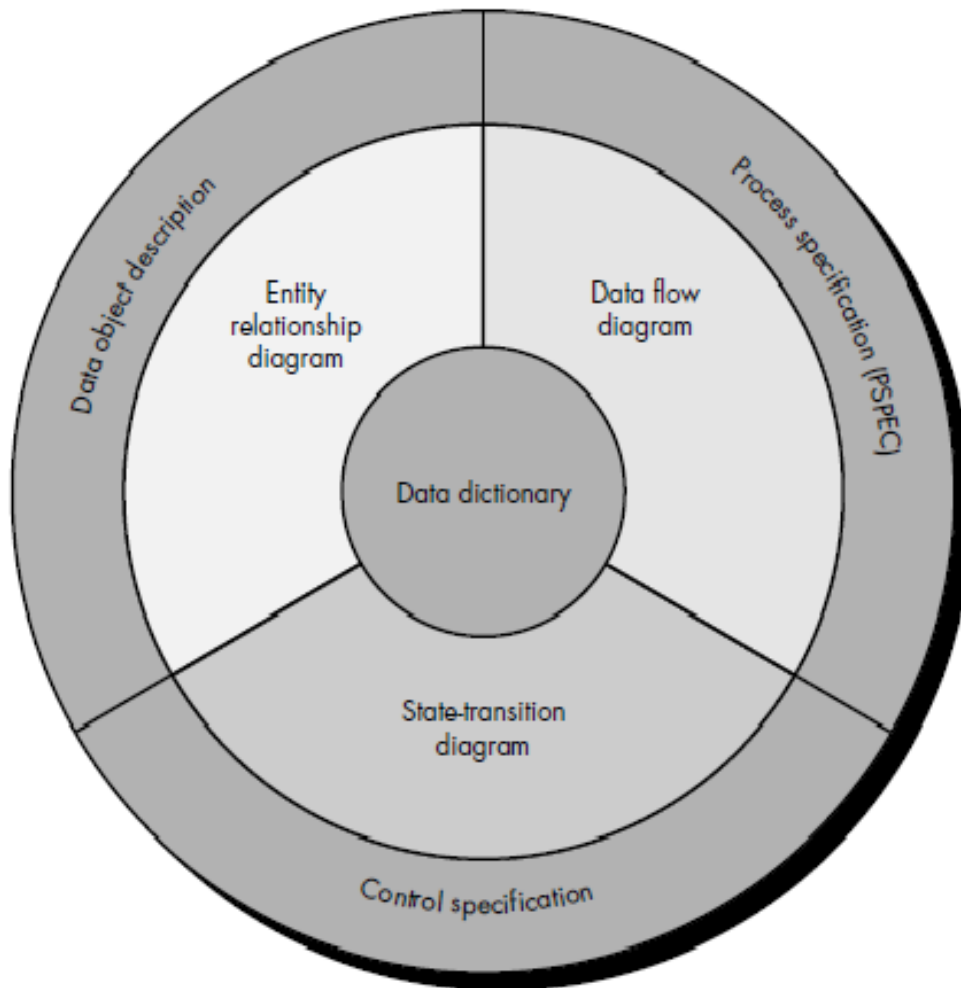
Figure (3.2): The Elements of Analysis Model Using Structured Analysis

(2) The Entity Relation Diagram (ERD): The ERD is the notation that is used to conduct the data modeling activity. It depicts relationships

between data objects. The primary purpose of the ERD is to represent entities (data objects) and their relationships with one another. The attributes of each data object noted in the ERD can be described using a data object description.

(3) Data Object Description: The attributes of each data object noted in the ERD can be described using a data object description.

(4) The Data Flow Diagram (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. A data flow diagram, also known as a *data flow graph* or a *bubble chart*. The data flow diagram may be used to represent a system or software at any level of abstraction.

 The 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 sub functions) 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).

 (5) The Process Specification (PSPEC): 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.

(6) The <u>S</u>tate <u>T</u>ransition <u>D</u>iagram (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. The STD indicates what actions (e.g., process activation) are taken as a consequence of a particular event.

(7) The <u>C</u>ontrol <u>S</u>pecification (CSPEC): It contains additional information about the control aspects of the software. The CSPEC describes the behavior of the system, but it gives us no information

about the inner working of the processes that are activated as a result of this behavior. This information is provided by the process specification (*PSPEC).* The control specification (CSPEC) represents the behavior of the system (at the level from which it has been referenced) in two different ways,   (1) how the software behaves when an event or control signal is sensed and (2) which processes are invoked as a consequence of the occurrence of the event. The control specification (CSPEC) contains a number of important modeling tools, for example, a *process activation table* is used to indicate which processes are activated by a given event.

Data Modeling Concepts

Analysis modeling often begins with *data modeling*. The software engineer or analyst defines all data objects that are processed within the system, the relationships between the data objects, and other information that is pertinent to the relationships. Most large software systems make use of a large database of information, it is for the system being developed. An important part of systems modeling is defining the data processed by the system. These are sometimes called *data models.*

The most widely used data modeling technique is ***Entity-Relationship-Diagram*** method (ERD). 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 (ERD) focuses solely on data (and therefore satisfies the first operational analysis principles), representing a "data network" that exists for a given system. This approach to modeling was first proposed in the mid-1970s by Chen, several variants have been developed since then, all with the same basic form.

The data model consists of <u>three</u> 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.
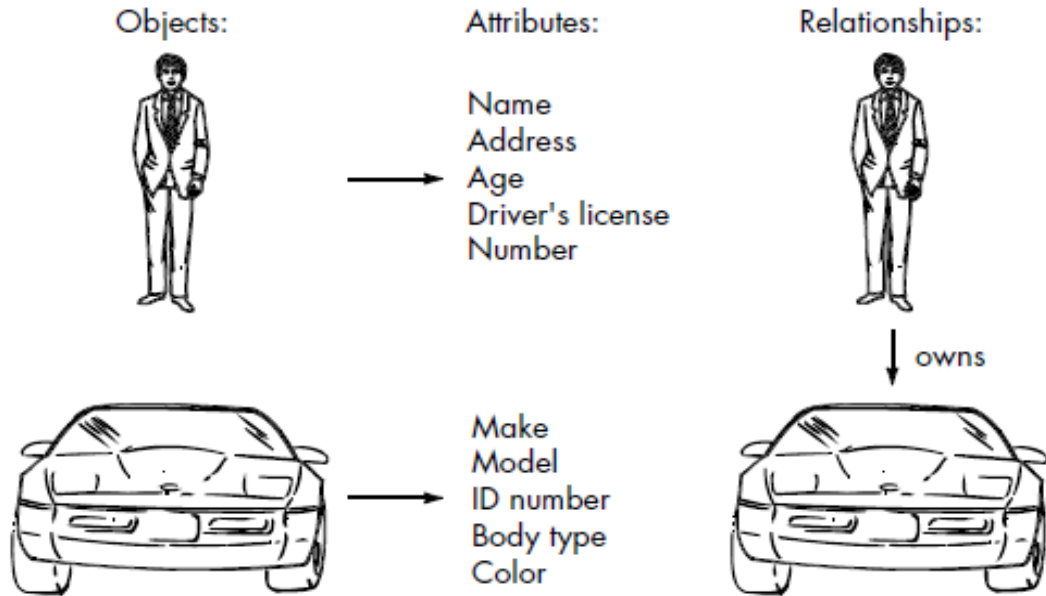
Therefore, width (a single value) would not be a valid data object, but **dimensions** (incorporating height, width, and depth) could be defined as an object.

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). For example, a person or a car (Figure 3.3) 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 approach.

Objects:

Attributes:

Name
Address
Age
Driver's license
Number

Relationships:

owns

Make
Model
ID number
Body type
Color

Therefore, the data object can be represented as a table as shown in (Figure 3. 4). 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**

Naming attributes

Ties one data object to another, in this case, owner

Descriptive attributes   Referential attributes

Identifier

Make	Model	ID#	Body type	Color	Owner
Lexus	LS400	AB123...	Sedan	White	RSP
Chevy	Corvette	X456...	Sports	Red	CCD
BMW	750iL	XZ765...	Coupe	White	LJL
Ford	Taurus	Q12A45...	Sedan	Blue	BLF

Instance

Figure (3.4): Tabular Representation of Data Objects

Data Attributes

Data 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. The attributes for **car** might serve well for an application that would be used by a Department of Motor Vehicles, but these attributes would be useless for an automobile company that needs manufacturing control software. In the latter case, the attributes for **car** might also include ID number, body type and color, but many additional attributes (e.g., interior code, drive train type, trim package designator, transmission type) would have to be added to make car a meaningful object in the manufacturing control context.

Relationships

Data objects are connected to one another in different ways. Consider

two data objects, **person** and **car**. These objects can be represented using

the simple notation illustrated in Figure 3.5a. A connection is established

between **person** and **car** because the two objects are related. But what are

the relationships? To determine the answer, we must understand the role of people (owners, in this case) and cars 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 person owns a car.

• A person is insured to drive a car.

(a) A basic connection between data objects

(b) Relationships between data objects

Figure (3.5): Relationships Between Data Objects

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

Cardinality is the maximum number of objects that can participate in a relationship.

Cardinality is usually expressed as simply 'one' or 'many.' For example:

• One-to-one (l:l)—An occurrence of [object] 'A' can relate to only one occurrence of [object] 'B,' and an occurrence of 'B' can relate to only one occurrence of 'A.'

• One-to-many (l: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.'

 For example, a mother can have many children, but a child can have only one mother.

• 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.' For example, an uncle can have many nephews, while a nephew can have many uncles.

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 (3.6) illustrates the relationship, cardinality, and modality between the data objects **customer** and **repair action.**
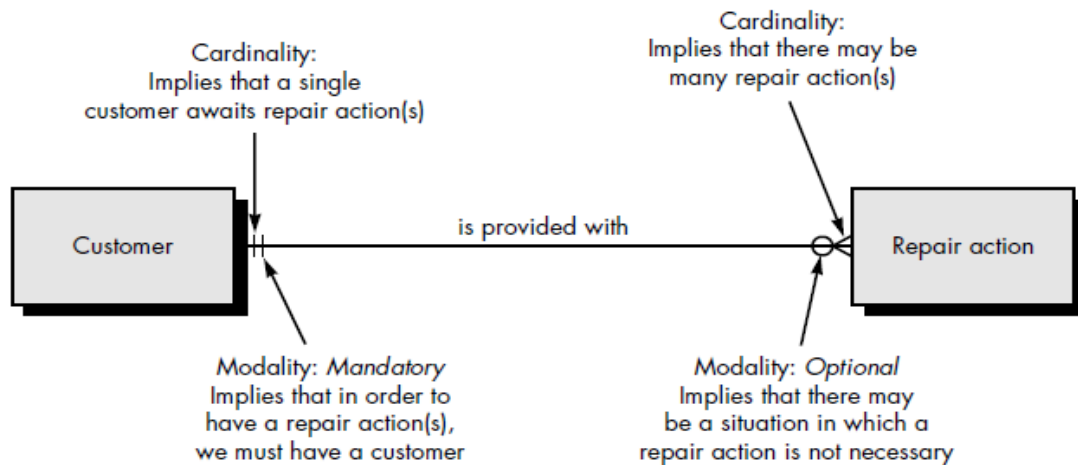

Figure (3.6): Cardinality and Modality

Referring to the figure (3.6), 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.

Entity-Relationship Diagrams

The object/relationship pair   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. 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.The relationship between the data objects **car** and **manufacturer** would be represented as shown in Figure(3.7). 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 3.7) would be radically different from the earlier specification (Figure 3.4). 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).

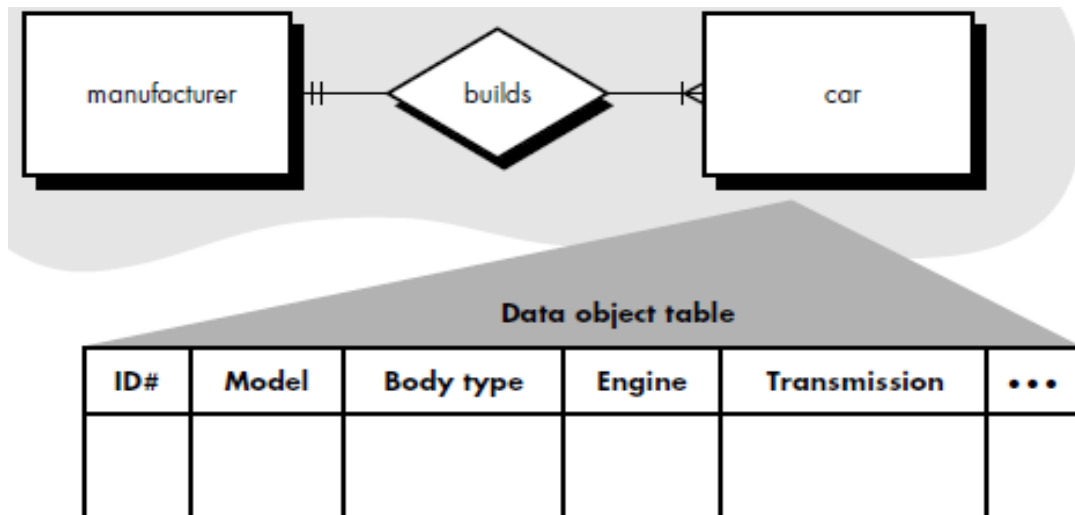| ID#	Model	Body type	Engine	Transmission	•••

Figure (3.7): A Simple ERD and Data Object Table

Expanding the model, we represent a grossly oversimplified ERD (Figure 3.8) of the distribution element of the automobile business. New data objects, **shipper** and **dealership,** are introduced. In addition, new relationships—*transports, contracts, licenses,* and *stocks*—indicate how the data objects shown in the figure associate with one another. Tables for each of the data objects contained in the ERD would have to be developed.

Figure (3.8): An Expanded ERD.

Creating an Entity/Relationship Diagram

The entity/relationship diagram enables a software engineer to fully specify the data objects that are input and output from a system, the attributes that define the properties of these objects, and their relationships. Like most elements of the analysis model, the ERD is constructed in an iterative manner. The following approach is taken:

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.

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.

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

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

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.

6. The attributes of each entity are defined.

7. An entity relationship diagram is formalized and reviewed.

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

Data Flow Diagram

Structured analysis began as an information flow modeling technique. A computer-based system is represented as an information transform as shown in Figure (3.9). 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.

Figure (3.9): Information Flow Model.

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 (3.9).

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 sub function of the overall system depicted in the context model.

As we noted earlier, each of the bubbles may be refined or layered to depict more detail. Figure (3.10) 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.
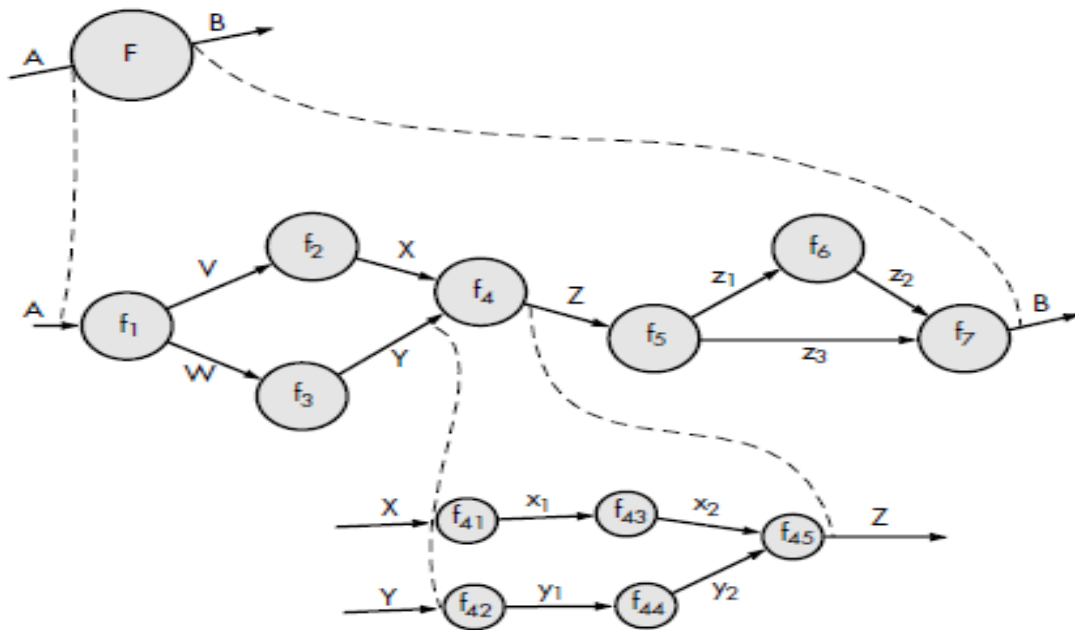
Figure (3.10): Information Flow Refinements.

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.

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;

(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.

CHAPTER FOUR

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 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 procedural detail of software

components are developed, reviewed, and documented. Design results in representations of software that can be assessed for quality. Software design is an iterative process through which requirements are translated into a "blueprint" for constructing the software.

The Design Models

Each of the elements of the analysis model provides information that is necessary to create the ***four design models*** required for a complete

specification of design. The flow of information during software design is illustrated in Figure (4.1).

Software requirements, manifested by the data, functional, and behavioral models, feed the design task. Using one of a number of design methods the design task produces a data design, an architectural design, an interface design, and a component design.

(1) 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) Interface Design

The *interface design* describes how the software communicates within itself, with systems that interoperate with it, and with humans who use it. An interface implies a flow of information (e.g., data and/or control) and a specific type of behavior. Therefore, data and control flow diagrams provide much of the information required for interface design. *Interface design* focuses on three areas of concern:

(a) the design of interfaces between software components,

(b) the design of interfaces between the software and other nonhuman producers and consumers of information (i.e., other external entities),

(c) the design of the interface between a human (i.e., the user) and the computer.

Three important principles guide the design of effective user interfaces: (a) place the user in control,

(b) reduce the user's memory load, and

(c) make the interface consistent.

User interface design begins with the identification of user, task, and environmental requirements. Task analysis is a design activity that defines user tasks and actions.

(4) 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.
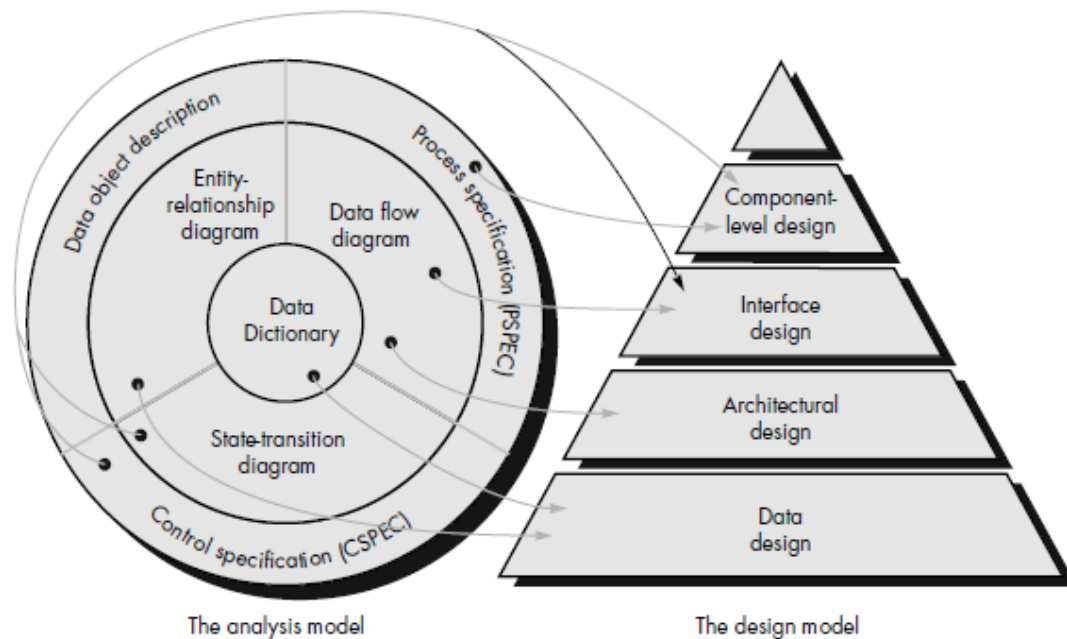

Figure (4.1): Translating the Analysis Model into a Software Design

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.

A *flowchart* is quite simple pictorially. A *box* is used to indicate a *processing step*. A *diamond* represents a *logical condition*, and *arrows* show the *flow of control*. Figure (4.2) illustrates three structured constructs. The *sequence* is represented as two processing boxes connected by an line (arrow) of control. *Condition,* also called *if- then-*

else, is depicted as a decision diamond that if true, causes *then-part* processing to occur, and if false, invokes *else-part* processing. *Repetition* is represented using two slightly different forms. The *do while* tests a

condition and executes a loop task repetitively as long as the condition holds true. A *repeat until* executes the loop task first, then tests a condition and repeats the task until the condition fails.
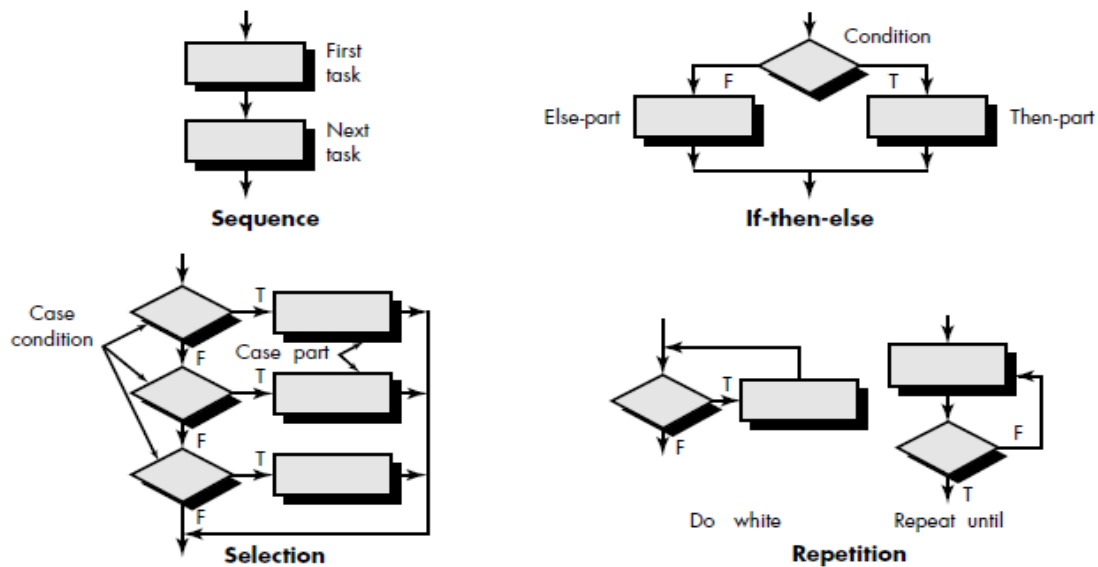

Figure (4.2): Flowchart constructs

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 (4.3) 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.*

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

Figure (4.3): Resultant Decision Table

d. 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 security.monitor

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.

Object-Oriented Design

Object-oriented design is an approach to software design where the fundamental components in the design represent objects with their own private state as well as represent operations rather than functions.

In design models above, we introduced the concept of a design pyramid for conventional software.

Four design layers—data, architectural, interface, and component level—were defined and discussed. For object-oriented systems, we can also define a design pyramid, but the layers are a bit different. Referring to Figure (4.4), the four layers of the OO design pyramid are:

The subsystem layer contains a representation of each of the subsystems

that enable the software to achieve its customer-defined requirements and to implement the technical infrastructure that supports customer requirements.

The class and object layer contains the class hierarchies that enable the

system to be created using generalizations and increasingly more targeted

specializations. This layer also contains representations of each object.

The message layer contains the design details that enable each object to

communicate with its collaborators. This layer establishes the external and internal interfaces for the system.

The responsibilities layer contains the data structure and algorithmic

design for all attributes and operations for each object. management facilities. Object design focuses on the internal detail of individual classes, defining attributes, operations, and message detail.

What is the work product? An OO design model encompasses software architecture, user interface description, data management components, task management facilities, and detailed descriptions of each class to be used in the system. >> How do I ensure that I've done it right? At each stage, the elements of the object-oriented design model are reviewed for clarity, correctness, completeness, and consistency with customer requirements   and   with   one   another.

Figure (4.4): The OO Design Pyramid

Real-Time Design Concepts

Computers are used to control a wide range of systems from simple domestic machine to entire manufacturing plants. These computers interact directly with hardware devices. The software in these systems is *embedded real-time software* that must react to events generated by the hardware and issue control signals in response to these events. It is embedded in some larger hardware system and must respond, in real time, to events from the system's environment.

Real-time embedded systems are different from other types of software systems. Their correct functioning is dependent on the system responding to events within a short time interval.

A real-time system is a software system that must respond to events in real time. Its correctness does not just depend on the results it produces but also on the time when these results are produced.

Part of the system design process involves deciding which system capabilities are to be implemented in software and which in hardware. For many real-time systems embedded in consumer products, such as the systems in cell phones, the costs and power consumption of the hardware are critical. Specific processors designed to support embedded systems may be used and, for some systems, special-purpose hardware may have to be designed and built.

This means that a top-down design process – where the design starts with an abstract model that is decomposed and developed in a series of stages-is impractical for most real-time systems. Low –level decisions on hardware, support software and system timing must be considered early in the process. These limit the flexibility of system designers and may

mean that additional software functionality, such as battery and power management, is required.

Top-Down Design Method

In the *top-down* model an overview of the system is formulated, without going into detail for any part of it. Each part of the system is then refined by designing it in more detail. Each new part may then be refined again, defining it in yet more detail until the entire specification is detailed enough to begin development.

Top-down approaches emphasis planning, and a complete understanding of the system. It is inherent that no coding can begin until a sufficient level of detail has been reached on at least some part of the system.

Top-down design is a strategy that supports the iterative development. It also supports the concepts of abstraction and refinement.

Bottom-Up Design Method

In *bottom-up* design, individual parts of the system are specified in detail, and may even be coded. The parts are then linked together to form larger components, which are in turn linked until a complete system is arrived.

This is a designed methodology in which the lowest level portion of design is completed first. Only after the low-level building blocks are completed then the higher-level hierarchical blocks in the design will be finished.
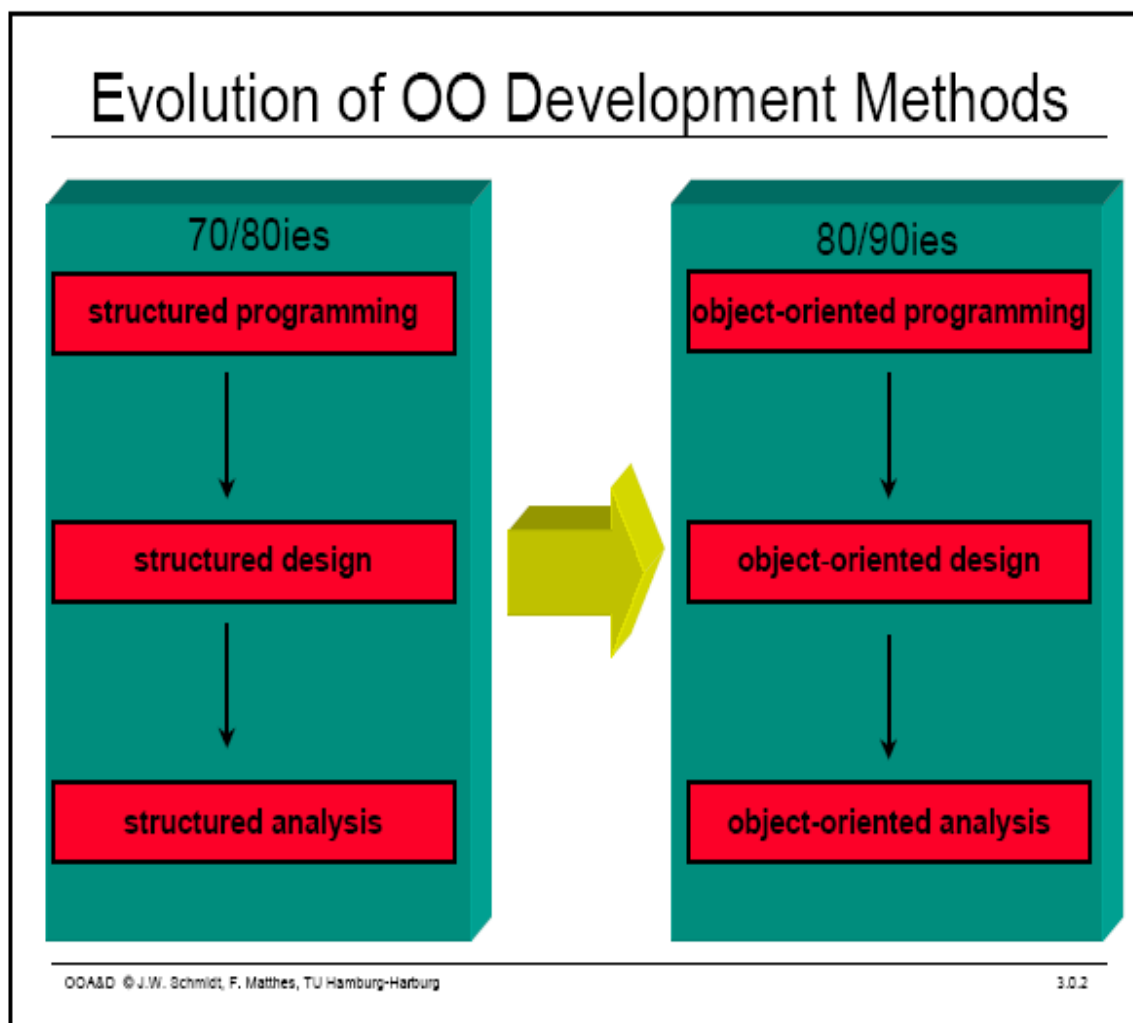
CHAPTER FIVE

Unified Modeling Language (UML)

Modeling

- Describing a system at a high level of abstraction

  - A model of the system

  - Used for requirements and specifications

  - Is it necessary to model software systems?
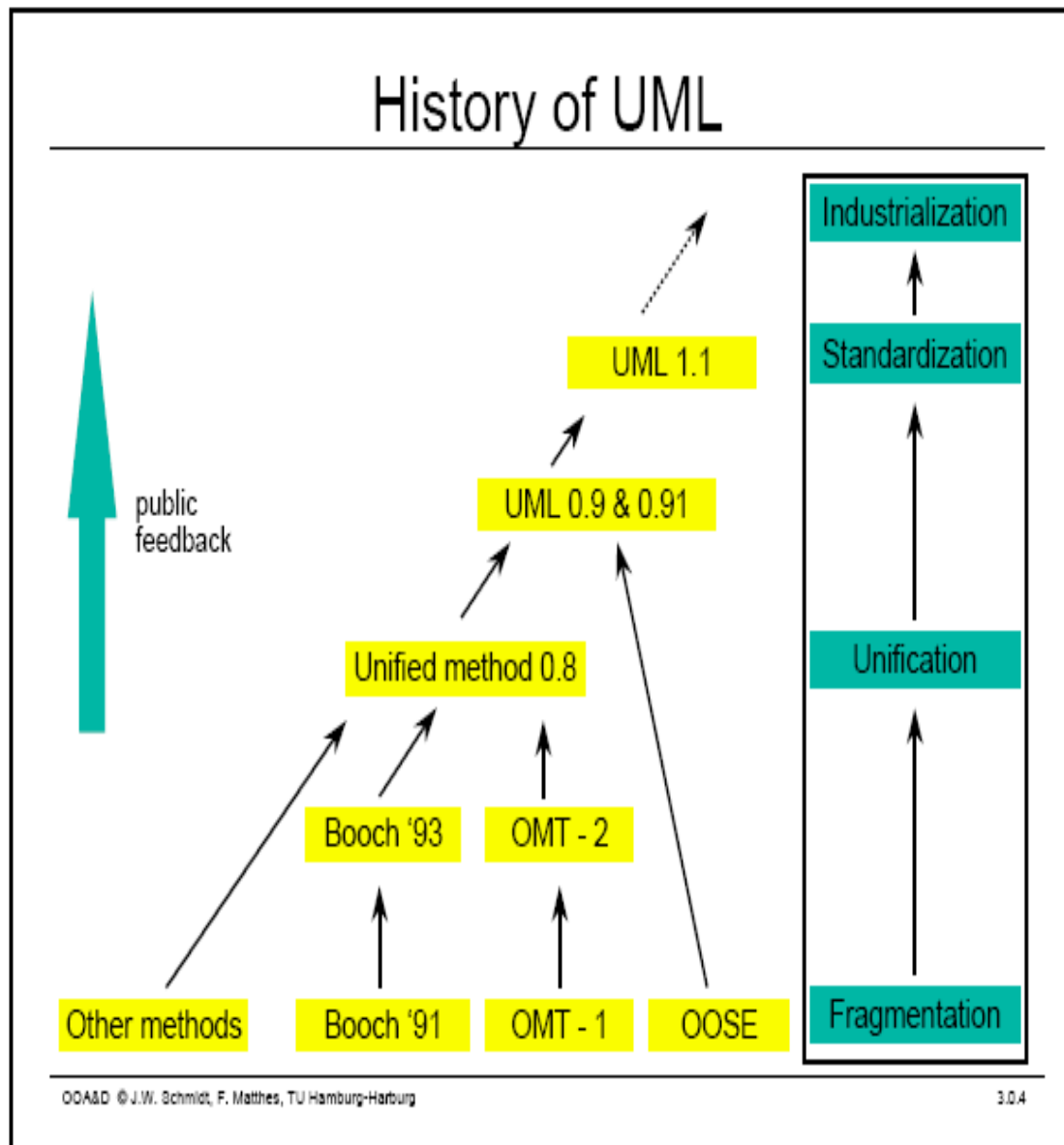
Object Oriented Modeling

What is UML?

UML stands for "Unified Modeling Language"

- It is a industry-standard graphical language for specifying, visualizing, constructing, and documenting the artifacts of software systems

- The UML uses mostly graphical notations to express the OO analysis and design of software projects.

- Simplifies the complex process of software design

Why UML for Modeling

- Use graphical notation to communicate more clearly than natural language (imprecise) and code(too detailed).

- Help acquire an overall view of a system.

- UML is *not* dependent on any one language or technology.

- UML moves us from fragmentation to standardization.

History of UML

History of UML

public feedback

Industrialization

Standardization

UML 1.1

UML 0.9 & 0.91

Unified method 0.8

Unification

Booch '93   OMT - 2

Other methods   Booch '91   OMT - 1   OOSE   Fragmentation

ODA&D © J.W. Schmidt, F. Matthes, TU Hamburg-Harburg   3.0.4

Types of UML Diagrams

1. Use Case Diagram

2. Class Diagram

3. Sequence Diagram

4. Collaboration Diagram

5. State Diagram

1. Use Case Diagram

Used for describing a set of user scenarios

Mainly used for capturing user requirements

Work like a contract between end user and software developers

Use Case Diagram (core components)

Actors: A role that a user plays with respect to the system, including human users and other systems. e.g., inanimate physical objects (e.g. robot); an external system that needs some information from the current system.

Use case: A set of scenarios that describing an interaction between a user and a system, including alternatives.

Actor   Use Case

System boundary: rectangle diagram representing the boundary between the actors and the system.

Use Case Diagram(core relationship)

Association: communication between an actor and a use case; Represented by a solid line. ─────────────

Generalization: relationship between one general use case and a special use case (used for defining special alternatives)

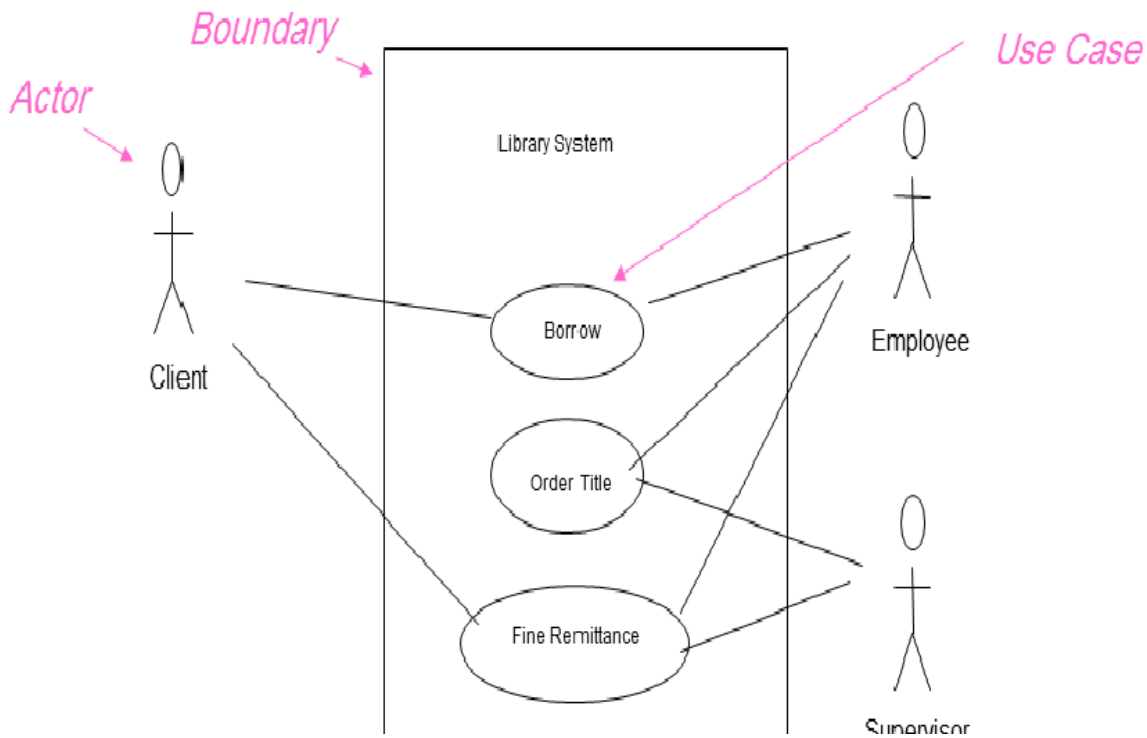Represented by a line with a triangular arrow head toward the parent use case. ──────────▷

Include: a dotted line labeled <<include>> beginning at base use case and ending with an arrows pointing to the include use case. The include relationship occurs when a chunk of behavior is similar across more than one use case. Use "include" instead of copying the description of that behavior.
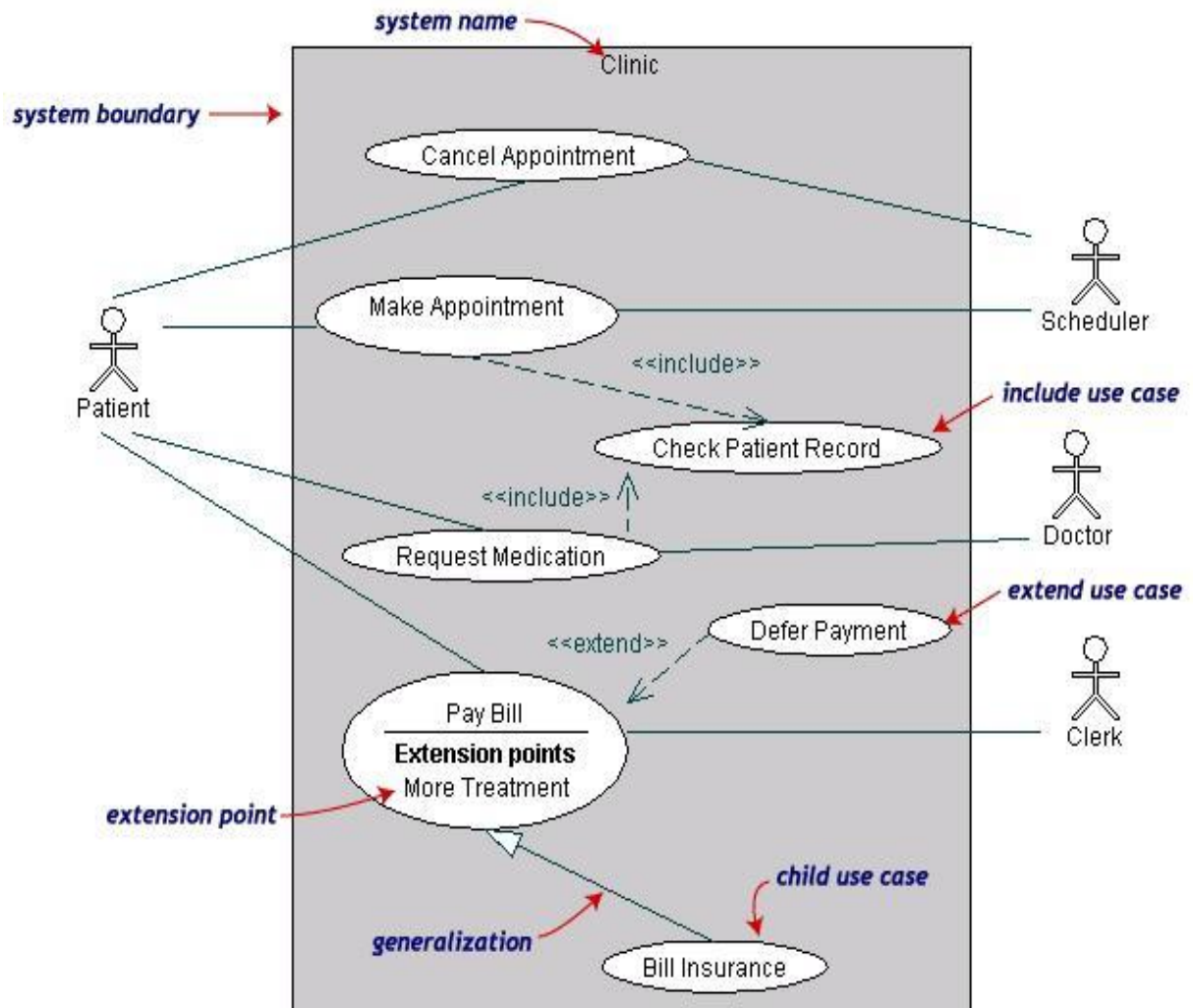
<<include>>
------------------▶

Extend: a dotted line labeled <<extend>> with an arrow toward the base case. The extending use case may add behavior to the base use case. The base class declares "extension points".

<<extend>>
--------------------------▶

-
-
-
-

Use Case Diagrams

- A generalized description of how a system will be used.
- Provides an overview of the intended functionality of the system.

- **Pay Bill** is a parent use case and **Bill Insurance** is the child use case. (generalization)
- Both **Make Appointment** and **Request Medication** include **Check Patient Record** as a subtask.(include)
- The **extension point** is written inside the base case

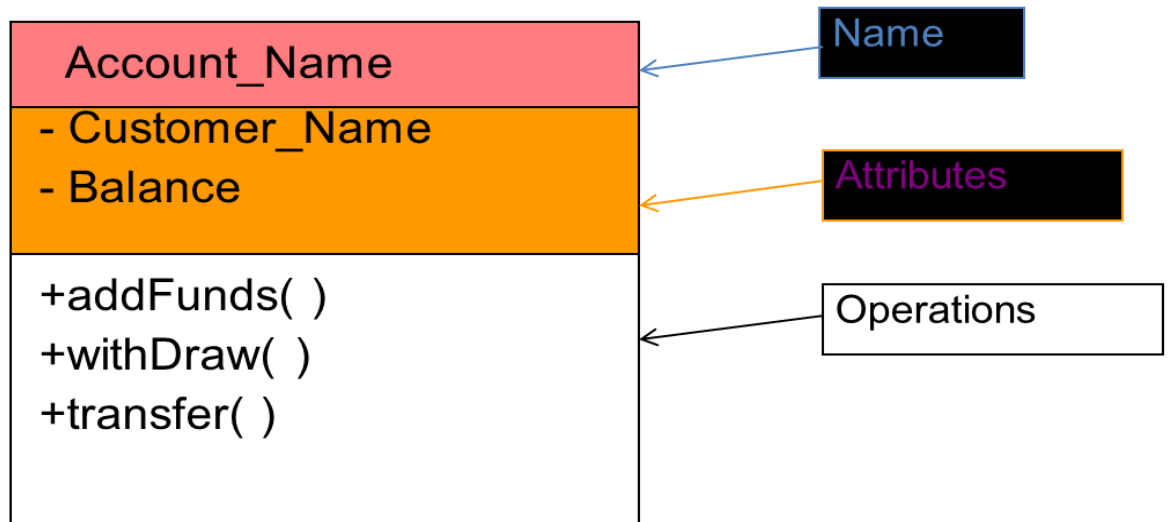**Pay bill**; the extending class **Defer payment** adds the behavior of this extension point. (extend).

2. Class Diagram

- Used for describing structure and behavior in the use cases
- Provide a conceptual model of the system in terms of entities and their relationships
- Used for requirement capture, end-user interaction
- Detailed class diagrams are used for developers

Class Representation

- Each class is represented by a rectangle subdivided into three compartments
– Name
– Attributes
– Operations
- Modifiers are used to indicate visibility of attributes and operations.
– '+' is used to denote *Public* visibility (everyone)
– '#' is used to denote *Protected* visibility (friends and derived)
– '-' is used to denote *Private* visibility (no one)
- By default, attributes are hidden and operations are visible.

An   example   of   Class
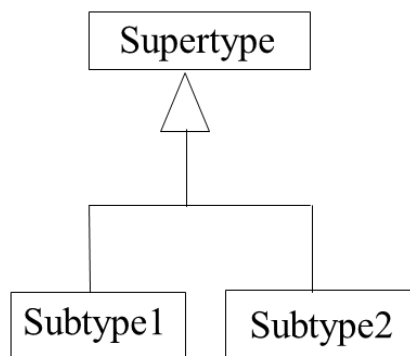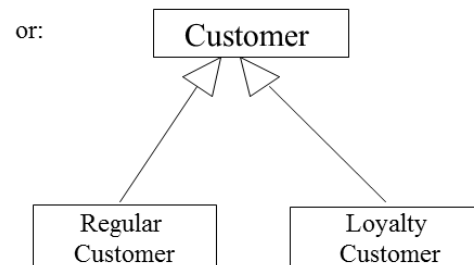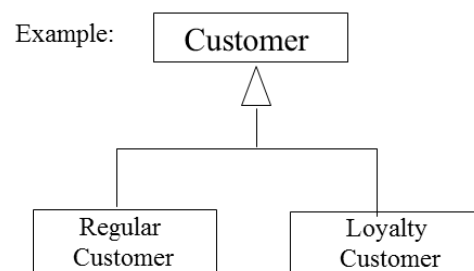
An example of Class

OO Relationships

- There are two kinds of Relationships
    - Generalization (parent-child relationship)
    - Association (student enrolls in course)
- Associations can be further classified as
    - Aggregation
    - Composition

OO Relationships: **Generalization**

- Generalization expresses a parent/child relationship among related classes.

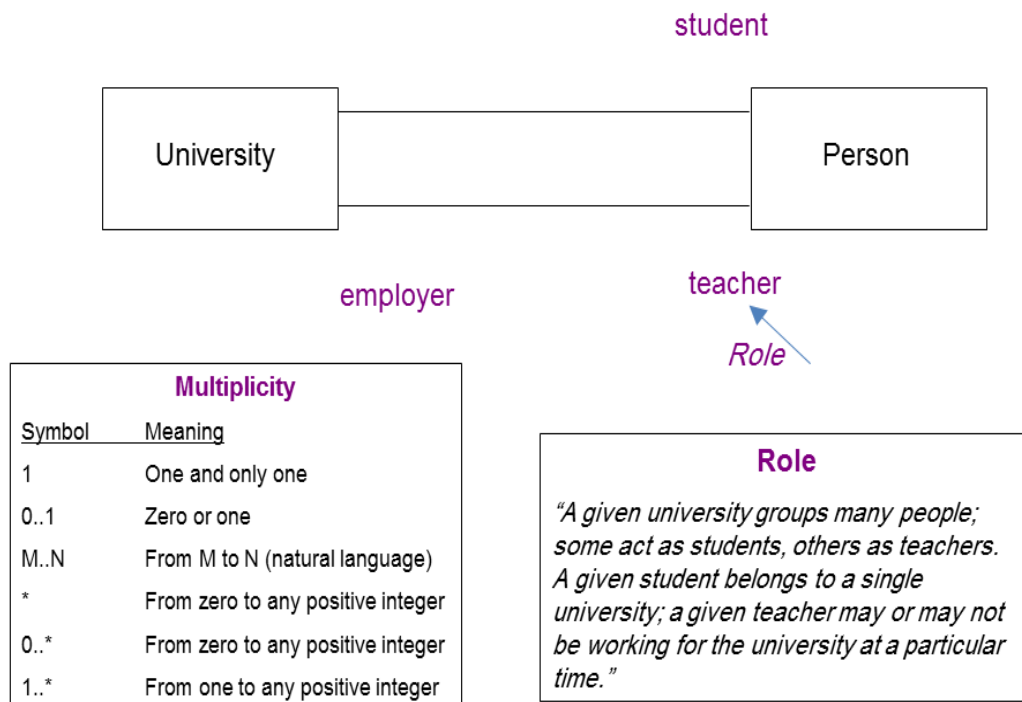- Used for abstracting details in several layers

OO Relationships: **Association**

- Represent relationship between instances of classes
    - Student enrolls in a course
    - Courses have students
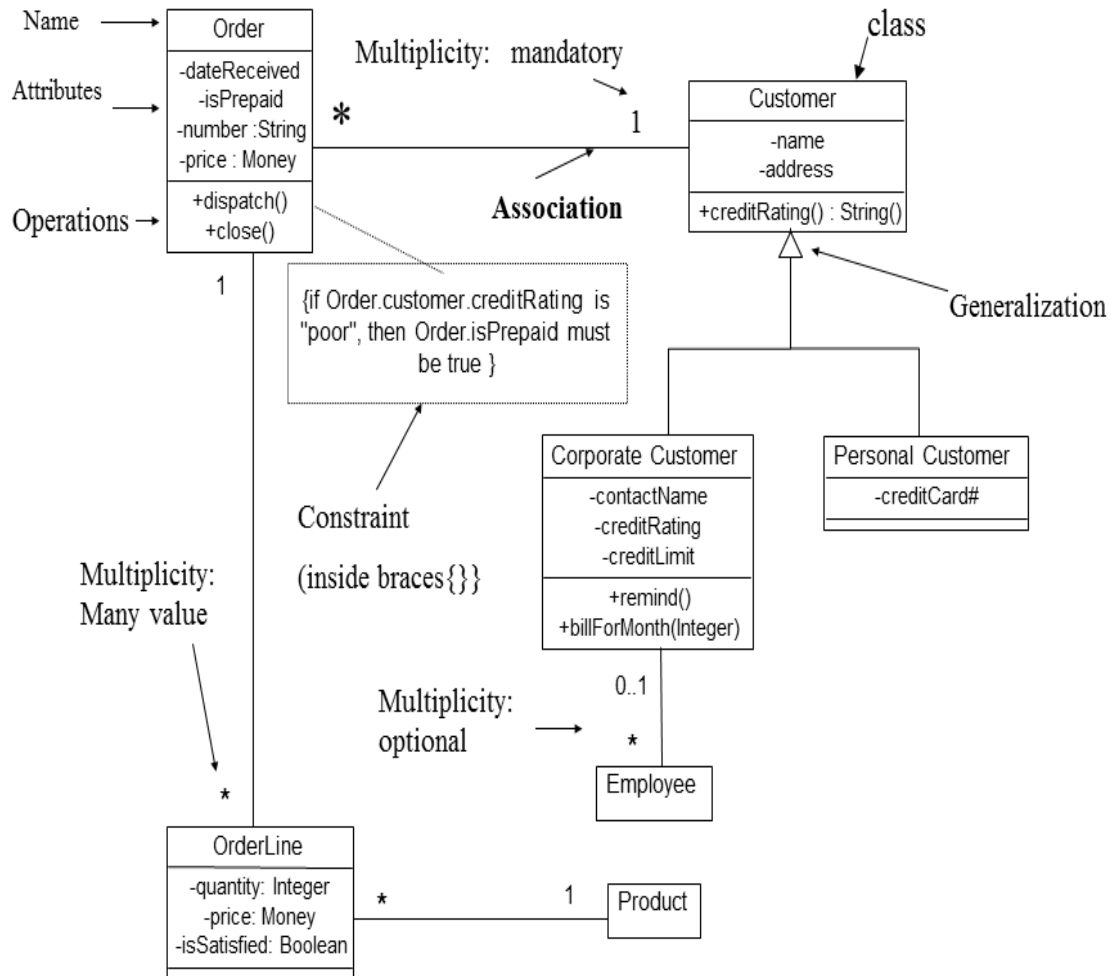   - Courses have exams
    - Etc.

- Association has two ends

    - Role names (e.g. enrolls)

    - Multiplicity (e.g. One course can have many students)

    - Navigability (unidirectional, bidirectional)

Association: Multiplicity and Roles

Association: Multiplicity and Roles

student

| University | Person |

employer   teacher

                            Role

Multiplicity

Symbol	Meaning
1	One and only one
0..1	Zero or one
M..N	From M to N (natural language)
*	From zero to any positive integer
0..*	From zero to any positive integer
1..*	From one to any positive integer

Role

"A given university groups many people; some act as students, others as teachers. A given student belongs to a single university; a given teacher may or may not be working for the university at a particular time."

Class Diagram

Name → Order

Attributes →
- -dateReceived
- -isPrepaid
- -number :String
- -price : Money

Operations →
- +dispatch()
- +close()

Multiplicity: mandatory

*   1

Association

class

Customer
- -name
- -address

+creditRating() : String()

Generalization

{if Order.customer.creditRating is "poor", then Order.isPrepaid must be true }

Constraint

(inside braces{}}

1

Multiplicity: Many value

Corporate Customer
- -contactName
- -creditRating
- -creditLimit

+remind()
+billForMonth(Integer)

Personal Customer
- -creditCard#

Multiplicity: optional

0..1

*

Employee

*

OrderLine
- -quantity: Integer
- -price: Money
- -isSatisfied: Boolean

*   1   Product

[from *UML Distilled   Third Edition*]

Association: Model to Implementation

```
            *                    4
Student ──────────────── Course
        has        enrolls
```

Class Student {
   Course enrolls[4];
}

Class Course {
    Student have[];
}

OO Relationships: **Composition**

Whole Class

```
      Class W
        ◆
    ┌───┴───┐
 Class P₁   Class P₂
```

Part Classes

Example

```
   Automobile
       ◆
   ┌───┴───┐
 Engine  Transmission
```

Composition: expresses a relationship among instances of related classes. It is a specific kind of Whole-Part relationship.

It expresses a relationship where an instance of the Whole-class has the responsibility to create and initialize instances of each Part-class.

It may also be used to express a relationship where instances of the Part-classes have privileged access or visibility to certain attributes and/or behaviors defined by the Whole-class.

Composition should also be used to express relationship where instances of the Whole-class have exclusive access to and control of instances of the Part-classes.

Composition should be used to express a relationship where the behavior of Part instances is undefined without being related to an instance of the Whole. And, conversely, the behavior of the Whole is ill-defined or incomplete if one or more of the Part instances are undefined.
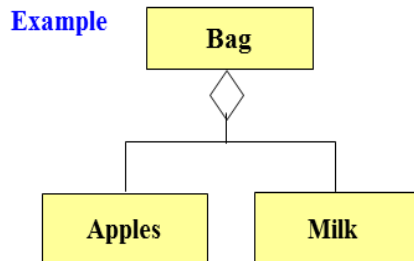
OO Relationships: **Aggregation**

Container Class

Class C

AGGREGATION ◇

Class E₁   **Class E₂**

Containee Classes

Aggregation: expresses a relationship among instances of related classes. It is a specific kind of Container-Containee relationship.

It expresses a relationship where an instance of the Container-class has the responsibility to hold and maintain instances of each Containee-class that have been created outside the auspices of the Container-class.

Aggregation should be used to express a more informal relationship than composition expresses. That is, it is an appropriate relationship where the Container and its Containees

Aggregation is appropriate when Container and Containees have no special access privileges to each other.

Example   **Bag**

◇

Apples   **Milk**

Aggregation vs. Composition

- **Composition** is really a strong form of **aggregation**

  - components have only one owner

  - components cannot exist independent of their owner

  - components live or die with their owner

e.g. Each car has an engine that can not be shared with other cars.

- **Aggregations** may form "part of" the aggregate, but may not be essential to it. They may also exist independent of the aggregate.
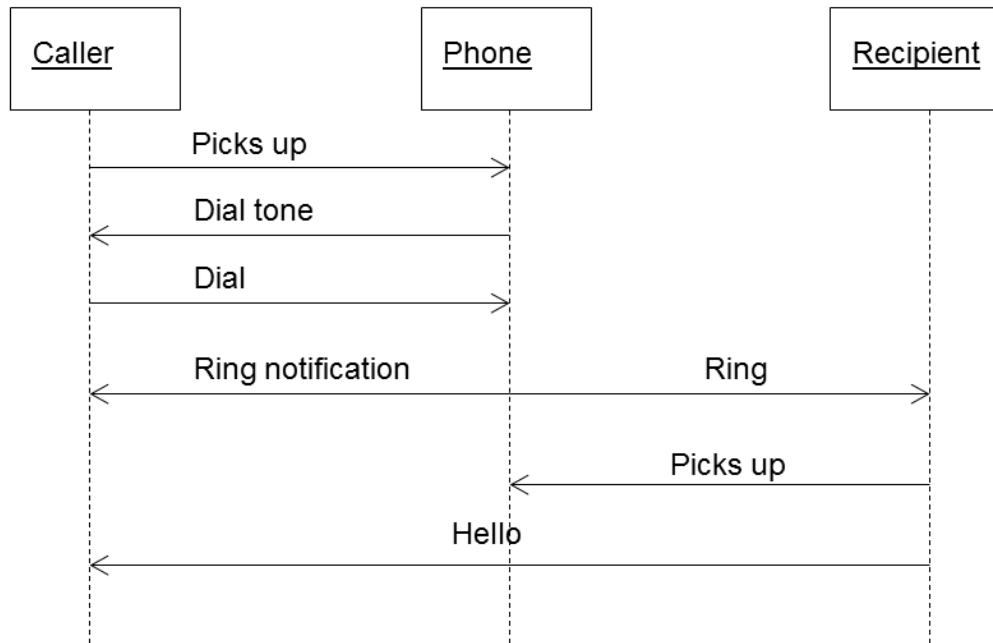
e.g. Apples may exist independent of the bag.

Good Practice: CRC Card
(Class Responsibility Collaborator)

Benefits: It is easy to describe how classes work by moving cards around; allows to quickly consider alternatives.

Class	Collaborators
Reservations	• Catalog • User session
Responsibility • Keep list of reserved titles • Handle reservation	

Sequence Diagram(make a phone call)

Caller	Phone	Recipient

Picks up

Dial tone

Dial

Ring notification   Ring

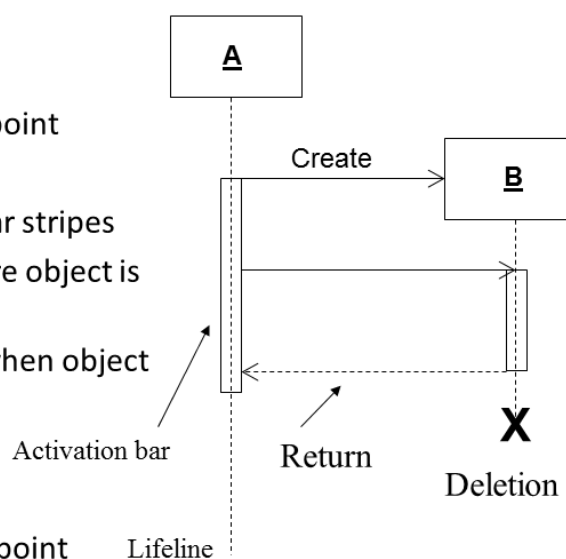Picks up

Hello

Sequence Diagram:Object interaction

Self-Call: A message that an Object sends to itself.

Condition: indicates when a message is sent. The message is sent only if the condition is true.

Synchronous

Asynchronous

Transmission delayed

[condition] remove()

Condition

*[for each] remove()
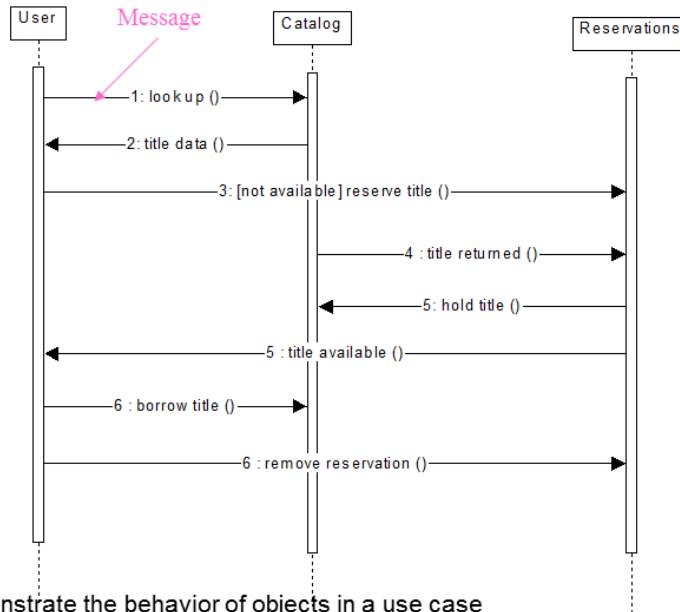
Iteration

Self-Call

A   B

Sequence Diagrams – Object Life Spans

- Creation
  - Create message
  - Object life starts at that point
- Activation
  - Symbolized by rectangular stripes
  - Place on the lifeline where object is activated.
  - Rectangle also denotes when object is deactivated.
- Deletion
  - Placing an 'X' on lifeline
  - Object's life ends at that point

A

Create

B

Activation bar   Return   X
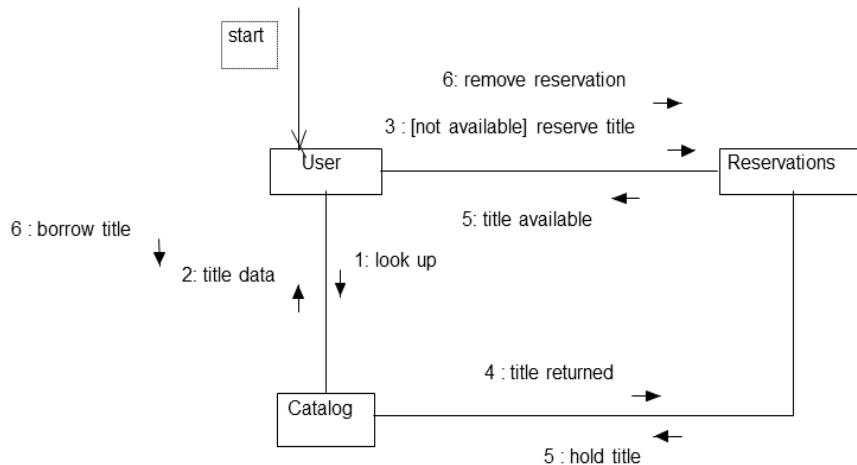
Deletion

Lifeline

Sequence Diagram

- Sequence diagrams demonstrate the behavior of objects in a use case by describing the objects and the messages they pass.
- The horizontal dimension shows the objects participating in the interaction.
- The vertical arrangement of messages indicates their order.
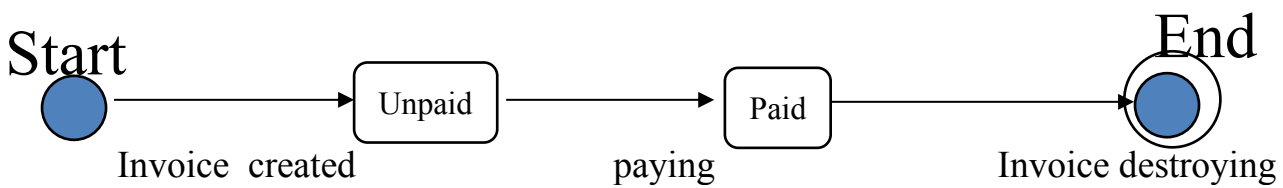- The labels may contain the seq. # to indicate concurrency.

Interaction Diagrams: Collaboration diagrams

- Shows the relationship between objects and the order of messages passed between them. between them.
- The objects are listed as rectangles and arrows indicate the messages being passed
- The numbers next to the messages are called sequence numbers. They show the sequence of the messages as they are passed between the objects.
- convey the same information as sequence diagrams, but focus on object roles instead of the time sequence.
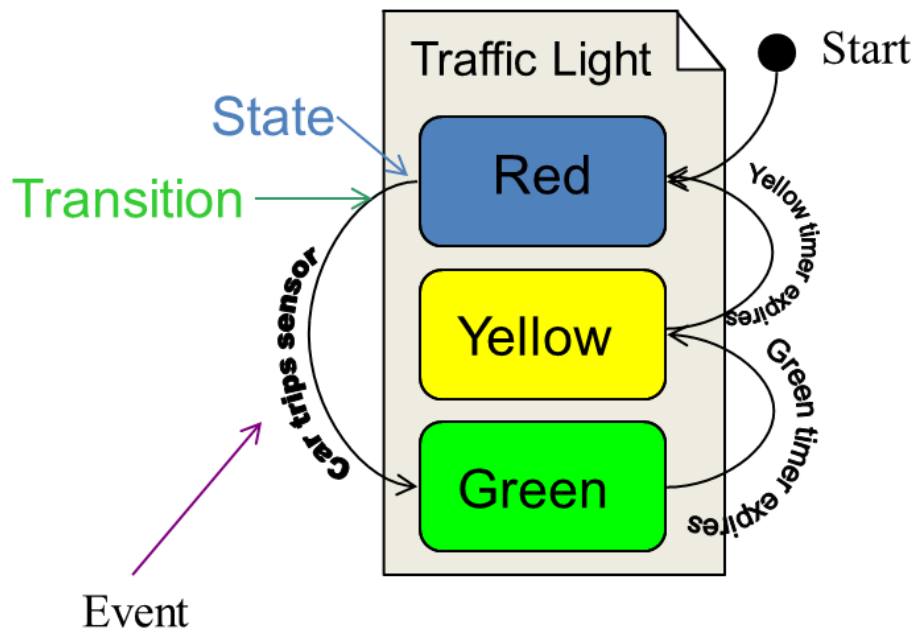
State Diagrams (Billing Example)

State Diagrams show the sequences of states an object goes through during its life cycle in response to stimuli, together with its responses and actions; an abstraction of all possible behaviors.

State Diagrams
(Traffic light example)

UML Modeling Tools

- Rational Rose   (www.rational.com) by IBM

- TogetherSoft   Control   Center,   Borland (http://www.borland.com/together/index.html)

- **ArgoUML** (free software) (http://argouml.tigris.org/)

   OpenSource; written in java

- Others ([http://www.objectsbydesign.com/tools/umltools_byComp](http://www.objectsbydesign.com/tools/umltools_byCompany.html) [any.html](http://www.objectsbydesign.com/tools/umltools_byCompany.html))

CHAPTER SIX

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:

1. Testing is a process of executing a program with the intent of finding   an error.

2. 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.
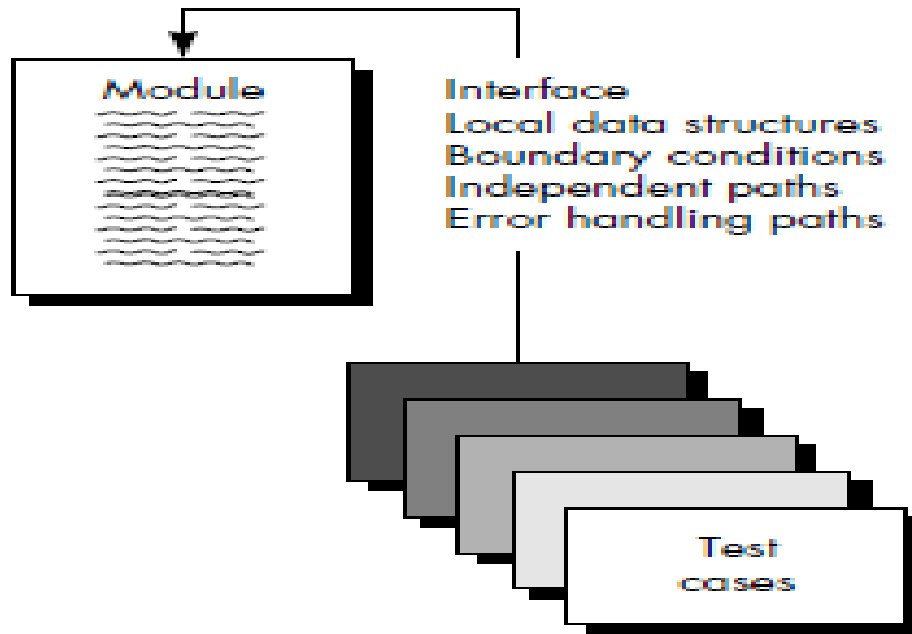
Steps (Stages) in Testing Process

Unit Testing

Unit testing focuses verification effort on the smallest unit of software design—the software component or module to ensure that it operates correctly. Each component or module is tested independently, without other system components. Using the component-level design description as a guide, important control paths are tested to uncover errors within the boundary of the module. Components may be simple entities such as functions or object classes.

Unit Testing Considerations

The tests that occur as part of unit tests are illustrated schematically in Figure (5.1). The module interface is tested to ensure that information properly flows into and out of the program unit under test. The local data structure is examined to ensure that data stored temporarily maintains its integrity during all steps in an algorithm's execution. Boundary conditions are tested to ensure that the module operates properly at boundaries established to limit or restrict processing. All independent paths (basis paths) through the control structure are exercised to ensure

that all statements in a module have been executed at least once. And finally, all error handling paths are tested.

Integration Testing

Integration testing is a systematic technique for constructing the program structure while at the same time conducting tests to uncover errors associated with interfacing. The objective is to take unit tested components and build a program structure that has been dictated by design. All components are combined in advance. The entire program is tested as a whole.

Validation Testing

Validation testing begins at the culmination of integration testing, when individual components have been exercised, the software is completely assembled as a package, and interfacing errors have been uncovered and corrected, and a final series of software tests—*validation testing*—may begin. Validation can be defined in many ways, but a simple definition is that validation succeeds when software functions in a manner that can be reasonably expected by the customer. Software validation is achieved through a series of tests that demonstrate conformity with requirements.

System Testing

As we know, software is only one element of a larger computer-based system. Ultimately, software is incorporated with other system elements (e.g., hardware, people, information), and a series of system integration

and validation tests are conducted. These tests fall outside the scope of the software process and are not conducted solely by software engineers. This test is also concerned with validating that the system meets its functional and non-functional requirements and testing the emergent system properties. However, steps taken during software design and testing can greatly improve the probability of successful software integration in the larger system.

System testing is actually a series of different tests whose primary purpose is to fully exercise the computer-based system. Although each test has a different purpose, all work to verify that system elements have been properly integrated and perform allocated functions.

The types of system tests are:

- Recovery testing.

- Security testing.

- Stress testing.

- Performance testing.

Acceptance Testing

This is the final stage in the testing process before the system is accepted for operational use. The system is tested with data supplied by the system customer rather than with simulated test data. Acceptance testing may reveal errors and omissions in the system requirements definition because the real data exercise the system in different ways from the test data. Acceptance testing may also reveal requirements problems where the system's facilities do not really meet the user's needs or the system performance is unacceptable.
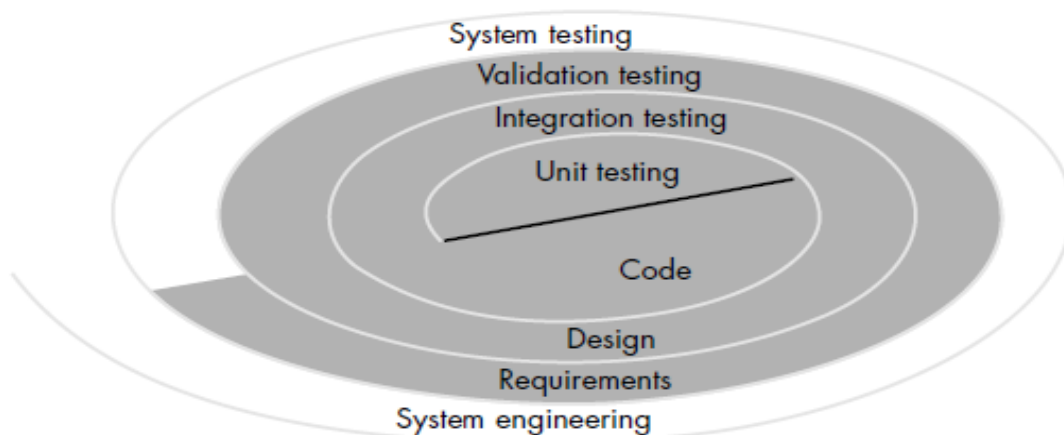
As the acceptance testing is concerned, there are *two* testing processes:

- Alpha Testing.
- Beta Testing.

Testing Strategy

Strategy for software testing may also be viewed in the context of the spiral Figure (5.2). *Unit testing* begins at the vortex of the spiral and concentrates on each unit (i.e., component) of the software as implemented in source code. Testing progresses by moving outward along the spiral to *integration testing,* where the focus is on design

and the construction of the software architecture. Taking another turn outward on the spiral, we encounter *validation testing,* where requirements established as part of software requirements analysis are validated against the software that has been constructed. Finally, we arrive at *system testing,* where the software and other system elements are tested as a whole. To test computer software, we spiral out along streamlines that broaden the scope of testing with each turn

Figure (6.2): Testing Strategy

Test Cases Design Methods

The primary objective for test case design is to derive a set of tests that can be used for uncovering errors in the software. To accomplish this objective, two different categories of test case design techniques are used: *black-box testing* and *white-box testing*.

Black-Box Testing

Black-box testing, also called *behavioral testing,* focuses on the functional requirements of the software. That is, black-box testing enables the software engineer to derive sets of input conditions that will fully exercise all functional requirements for a program. Black-box testing is not an alternative to white-box techniques. Rather, it is a complementary approach that is likely to uncover a different class of errors than white-box methods.

Black-box tests are designed to validate functional requirements without regard to the internal workings of a program.

Unlike white-box testing, which is performed early in the testing process, black-box testing tends to be applied during later stages of testing.

Black-box testing attempts to find errors in the following categories:

(1) incorrect or missing functions,

(2) interface errors,

(3) errors in data structures or external data base access,

(4) behavior or performance errors,

(5) initialization and termination errors.

White-Box Testing

White-box testing, sometimes called *glass-box testing,* is a test case design method that uses the control structure of the procedural design to derive test cases.

White-box tests focus on the program control structure. Test cases are derived to ensure that all statements in the program have been executed at least once during testing and that all logical conditions have been exercised.

Using white-box testing methods, the software engineer can derive test cases that:

(1) guarantee that all independent paths within a module have been exercised at least once,

(2) exercise all logical decisions on their true and false sides,

(3) execute all loops at their boundaries and within their operational bounds,

(4) exercise internal data structures to ensure their validity.

Alpha and Beta Testing

It is virtually impossible for a software developer to foresee how the customer will really use a program. Instructions for use may be misinterpreted; strange combinations of data may be regularly used; output that seemed clear to the tester may be unintelligible to a user in the field.

When custom software is built for one customer, a series of *acceptance tests* are conducted to enable the customer to validate all requirements. Conducted by the end user rather than software engineers, an acceptance test can range from an informal "test drive" to a planned and systematically executed series of tests. In fact, acceptance testing can be conducted over a period of weeks or months, thereby uncovering

cumulative errors that might degrade the system over time.

If software is developed as a product to be used by many customers, it is impractical to perform formal acceptance tests with each one. Most software product builders use a process called **alpha and beta testing** to uncover errors that only the end-user seems able to find.

Alpha Test

Custom systems are developed for a single client. The *alpha test* is conducted at the developer's site by a customer. The software is used in a natural setting with the developer "looking over the shoulder" of the user

and recording errors and usage problems. Alpha tests are conducted in a controlled environment.

The alpha testing process continues until the system developer and the client agree that the delivered system is an acceptable implementation of the system requirements.

 Beta Test

When a system is to marked as a software product, beta testing is used. The *beta test* is conducted at one or more customer sites by the end-user of the software. Unlike alpha testing, the developer is generally not

present. Therefore, the beta test is a "live" application of the software in an environment that cannot be controlled by the developer. The customer records all problems (real or imagined) that are encountered during beta testing and reports these to the developer at regular intervals. As a result of problems reported during beta tests, software engineers make modifications and then prepare for release of the software product to the entire customer base.

CHAPTER SEVEN

Software Project Management And Quality Assurance

Software Project Management

Software project management is an essential part of software engineering. Good software project management can be achieved if software engineering projects are to be developed on schedule and within budget.

Software project management is an umbrella activity within software engineering. It begins before any technical activity is initiated and continues throughout the definition, development, and support of computer software.

Four P's have a substantial influence on software project management—*people, product, process*, and *project*. People must be organized into effective teams, motivated to do high-quality software work, and coordinated to achieve effective communication.

The product requirements must be communicated from customer to

developer, partitioned (decomposed) into their constituent parts, and positioned for work by the software team. The process must be adapted to the people and the problem. A common process framework is selected, an appropriate software engineering paradigm is applied, and a set of work tasks is chosen to get the job done. Finally, the project must be organized in a manner that enables the software team to succeed.

Software project management is a huge topics, it includes many management activities: project planning, project scheduling, risk management, managing people, software cost estimation and quality management.

Types of Plan

There are many types of plan used to manage software projects, some of these are:

1. *Quality plan:* Describes the quality procedures and standards that will be used in the project.

2. *Validation plan:* Describes the approach, resources and schedule used for system validation.

3. *Configuration management plan:* Describes the configuration management procedures and structures to be used.

4. *Maintenance plan:* Predicts the maintenance requirements of the system, maintenance costs and effort required.

5. *Staff development plan:* Describes how the skills and experience of the project team members will be developed.

Project Planning

The objective of software project planning is to provide a framework that enables the manager to make reasonable estimates of resources, cost, and schedule. These estimates are made within a limited time frame at the beginning of a software project and should be updated regularly as the project progresses. In addition, estimates should attempt to define best-case and worst-case scenarios so that project outcomes can be bounded.

The plan must be adapted and updated as the project proceeds.

Software Planning Activities

1. Establish project scope: The first activity in software project planning is the determination of software scope. *Software scope* describes the data and control to be processed, function, performance, constraints, interfaces, and reliability. Functions described in the statement of scope are evaluated and in some cases refined to provide more detail prior to the beginning of estimation.

Performance considerations encompass processing and response time requirements. Constraints identify limits placed on the software by external hardware, available memory, or other existing systems.

2. Determine feasibility: Once scope has been identified (with the concurrence of the customer), it is reasonable to ask: "Can we build software to meet this scope? Is the project feasible?"

3. Analyze risks: Risk analysis can absorb a significant amount of project planning effort. Identification, projection, assessment, management, and monitoring all take time. But the effort is worth it. For the software project manager, the enemy is risk.

4. Define required resources: There are *three* major categories of software engineering resources:

   a. Define human resources required: The planner begins by evaluating scope and selecting the skills required to complete development. Both organizational position (e.g., manager, senior software engineer) and specialty (e.g., telecommunications, database, client/server) are specified. The number of people required for a software project can be

determined only after an estimate of development effort (e.g., person-months) is made.

b. Define reusable software resources: Component-based software engineering emphasizes reusability—that is, the creation and reuse of software building blocks. Such building blocks, often called *components,* must be cataloged for easy reference, standardized for easy application, and validated for easy integration.

c. Identify environment resources: The environment that supports the software project, often called the *software engineering environment*

(SEE), incorporates hardware and software. Hardware provides a platform that supports the tools (software) required to produce the work products that are an outcome of good software engineering practice.

5. Estimate cost and effort: Software cost and effort estimation will never be an exact science. Too many variables—human, technical, environmental, political—can affect the ultimate cost of software and effort applied to develop it. However, software project estimation can be transformed as a series of systematic steps that provide estimates with acceptable risk. It includes:

a. Decompose the problem: Software project estimation is a form of problem solving, and in most cases, the problem to be solved (i.e., developing a cost and effort estimate for a software project) is too complex to be considered in one piece. For this reason, we decompose

the problem, re-characterizing it as a set of smaller (and hopefully, more manageable) problems.

b. Develop two or more estimates using size, function points, process tasks, or use-cases.

c. Reconcile the estimates.

6. Develop a project schedule.

a. Establish a meaningful task set.

b. Define a task network.

c. Use scheduling tools to develop a timeline chart.

d. Define schedule tracking mechanisms.

Software Quality Assurance (SQA)

Quality assurance is an essential activity for any business that produces products to be used by others.

The history of quality assurance in software development parallels the history of quality in hardware manufacturing. During the early days of computing (1950s and 1960s), quality was the sole responsibility of the programmer. Standards for quality assurance for software were introduced in military contract software development during the 1970s and have spread rapidly into software development in the commercial

world. Software quality assurance is an umbrella activity that is applied at each step in the software process.

SQA encompasses procedures for the effective application of methods and tools, formal technical reviews, testing strategies and techniques, procedures for change control, procedures for assuring compliance to standards, and measurement and reporting mechanisms.

Quality assurance consists of the <u>auditing</u> and <u>reporting</u> functions of management.

The goal of quality assurance is to provide management with the data necessary to be informed about product quality, thereby gaining insight and confidence that product quality is meeting its goals. Of course, if the data provided through quality assurance identify problems, it is management's responsibility to address the problems and apply the necessary resources to resolve quality issues.

SQA Activities

Software quality assurance is composed of a variety of tasks associated with two different constituencies :

❖ The software engineers who do technical work.
❖ An SQA group that has responsibility for quality assurance planning, oversight, record keeping, analysis, and reporting.

REFEREENCES

1. Software Engineering, Ian Summervill.
2. Software Engineering a practitioner Approach, Pressman, 6th Edition.
3. UML2 and Unified Process, Arlow & Neustadt, 2nd Edition.
4. **UML Distilled:** A Brief Guide to the Standard Object Modeling Language
   Martin Fowler, Kendall Scott
5. IBM Rational:
   http://www-306.ibm.com/software/rational/uml/
6. Practical UML --- A Hands-On Introduction for Developers:
   http://www.togethersoft.com/services/practical_guides/um lonlinecourse/
7. Software Engineering Principles and Practice. Second Edition; Hans van Vliet.
8. http://www-inst.eecs.berkeley.edu/~cs169/