Jaipur Engineering College & Research Centre, Jaipur

Notes
Software Engineering
[3CS4 - 07]

Prepared By:
Manju Vyas
Abhishek Jain
Geerija Lavania

VISION AND MISSION OF INSTITUTE

VISION

To become renowned centre of outcome based learning and work towards academic, professional, cultural and social enrichments of the lives of individual and communities”

MISSION

M1. Focus on evaluation of learning outcomes and motivate students to inculcate research aptitude by project based learning.

M2. Identify areas of focus and provide platform to gain knowledge and solutions based on informed perception of Indian, regional and global needs.

M3. Offer opportunities for interaction between academia and industry.

M4. Develop human potential to its fullest extent so that intellectually capable and imaginatively gifted leaders can emerge in a range of professions.

VISION AND MISSION OF DEPARTMENT

VISION

To become renowned Centre of excellence in computer science and engineering and make competent engineers & professionals with high ethical values prepared for lifelong learning.

MISSION

M1: To impart outcome based education for emerging technologies in the field of computer science and engineering.

M2: To provide opportunities for interaction between academia and industry.

M3: To provide platform for lifelong learning by accepting the change in technologies

M4: To develop aptitude of fulfilling social responsibilities.

COURSE OUTCOMES

CO1) understand the purpose of designing a system and evaluate the various models suitable as per its requirement analysis

CO2) understand and apply software project management, effort estimation and project scheduling.

CO3) formulate requirement analysis, process behaviour and software designing.

CO4) Implement the concept of object oriented analysis modelling with the reference of UML and advance SE tools

Program Outcomes (PO)

1. **Engineering knowledge:** Apply the knowledge of mathematics, science, engineering fundamentals, and an engineering specialization to the solution of complex engineering problems.
2. **Problem analysis**: Identify, formulate, research literature, and analyze complex engineering problems reaching substantiated conclusions using first principles of mathematics, natural sciences, and engineering sciences.
3. **Design/development of solutions**: Design solutions for complex engineering problemsand design system components or processes that meet thespecified needs with appropriate consideration for the public health and safety, andthe cultural, societal, and environmental considerations.
4. **Conduct investigations of complex problems**: Use research-based knowledge and research methods including design of experiments, analysis and interpretation of data, and synthesis of the information to provide valid conclusions.
5. **Modern tool usage**: Create, select, and apply appropriate techniques, resources, and modern engineering and IT tools including prediction and modeling to complex engineering activities with an understanding of the limitations.
6. **The engineer and society**: Apply reasoning informed by the contextual knowledge to assess societal, health, safety, legal and cultural issuesand the consequent responsibilities relevant to the professional engineering practice.
7. **Environment and sustainability**: Understand the impact of the professional engineering solutions in societal and environmental contexts, and demonstrate the knowledge of, and need for sustainable development.
8. **Ethics**: Apply ethical principles and commit to professional ethics and responsibilities and norms of the engineering practice.
9. **Individual and team work**: Function effectively as an individual, and as a member or leader in diverse teams, and in multidisciplinary settings.
10. **Communication**: Communicate effectively on complex engineering activities with the engineering community and with society at large, such as, being able to comprehend and write effective reports and design documentation, make effective presentations, and give and receive clear instructions.
11. **Project management and finance**: Demonstrate knowledge and understanding of the engineering and management principles and apply these to one's own work, as a member and leader in a team, to manage projects and in multidisciplinary environments.
12. **Life-long learning**: Recognize the need for, and have the preparation and ability to engage in independent and life-long learning in the broadest context of technological change.

Program Educational Objectives (PEO)

1.To provide students with the fundamentals of Engineering Sciences with more emphasis in Computer Science & Engineering by way of analyzing and exploiting engineering challenge

2. To train students with good scientific and engineering knowledge so as to comprehend, analyze, design, and create novel products and solutions for the real life problems.

3. To inculcate professional and ethical attitude, effective communication skills, teamwork skills, multidisciplinary approach, entrepreneurial thinking and an ability to relate engineering issues with social issues.

4. To provide students with an academic environment aware of excellence, leadership, written ethical codes and guidelines, and the self-motivated life-long learning needed for a successful professional career.

5. To prepare students to excel in Industry and Higher education by Educating Students along with High moral values and Knowledge.

MAPPING CO-PO

Cos/POs	PO1	PO2	PO3	PO4	PO5	PO6	PO7	PO8	PO9	PO10	PO11	PO12
CO1	3	3	3	3	3	2	1	2	1	1	2	3
CO2	3	3	3	3	2	2	1	2	2	2	3	3
CO3	3	3	3	2	2	2	1	2	1	2	2	3
CO4	3	3	3	3	3	1	0	1	1	2	2	3

PSO

PSO1: Ability to interpret and analyze network specific and cyber security issues, automation in real word environment.

PSO2: Ability to Design and Develop Mobile and Web-based applications under realistic constraints.

SYLLABUS

UNIT 1: Introduction, software life-cycle models, software requirements specification, formal requirements specification, verification and validation.

UNIT 2: Software Project Management: Objectives, Resources and their estimation, LOC and FP estimation, effort estimation, COCOMO estimation model, risk analysis, software project scheduling.

UNIT 3: Requirement Analysis: Requirement analysis tasks, Analysis principles. Software prototyping and specification data dictionary, Finite State Machine (FSM) models. **Structured Analysis**: Data and control flow diagrams, control and process specification behavioral modeling

UNIT 4: Software Design: Design fundamentals, Effective modular design: Data architectural and procedural design, design documentation.

UNIT 5: Object Oriented Analysis: Object oriented Analysis Modeling, Data modeling. **Object Oriented Design**: OOD concepts, Class and object relationships, object modularization, Introduction to Unified Modeling Language

UNIT -3 REQUIREMENT ANALYSIS

REQUIREMENT ANALYSIS-

Requirements analysis is a software engineering task that bridges the gap between system level requirements engineering and software design . Requirements engineering activities result in the specification of software’s operational characteristics (function, data, and behavior), indicate software's interface with other system elements, and establish constraints that software must meet. Requirements analysis allows the software engineer (sometimes called analyst in this role) to refine the software allocation and build models of the data, functional, and behavioral domains that will be treated by software.

Requirements analysis provides the software designer with a representation of information, function, and behavior that can be translated to data, architectural, interface, and component-level designs. Finally, the requirements specification provides the developer and the customer with the means to assess quality once software is built.

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

(1) problem recognition,

(2) evaluation and synthesis,

(3) modeling,

(4) specification

(5) review.

Initially, the analyst studies the System Specification (if one exists) and the Software Project Plan. It is important to understand software in a system context and to review the software scope that was used to generate planning estimates. Next, communication for analysis must be established so that problem recognition is ensured. The goal is recognition of the basic problem elements as perceived by the customer/users.

Problem evaluation and solution synthesis is the next major area of effort for analysis.The analyst must define all externally observable data objects, evaluate the flow and content of information, define and elaborate all software functions, understand software behavior in the context of events that affect the system, establish system interface characteristics, and uncover additional design constraints. Each of these tasks serves to describe the problem so that an overall approach or solution may be synthesized.

ANALYSIS PRINCIPLES-

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

1. The information domain of a problem must be represented and understood.

2. The functions that the software is to perform must be defined.

3. The behavior of the software (as a consequence of external events) must be

represented.

4. The models that depict information, function, and behavior must be partitioned

in a manner that uncovers detail in a layered (or hierarchical) fashion.

5. The analysis process should move from essential information toward implementation detail.

By applying these principles, the analyst approaches a problem systematically. The information domain is examined so that function may be understood more completely.

Models are used so that the characteristics of function and behavior can be

communicated in a compact fashion. Partitioning is applied to reduce complexity.

Essential and implementation views of the software are necessary to accommodate the logical constraints imposed by processing requirements and the physical constraints imposed by other system elements.

Davis suggests a set of guiding principles for requirements engineering:

• ***Understand the problem before you begin to create the analysis model.*** There is a tendency to rush to a solution, even before the problem is understood. This often leads to elegant software that solves the wrong problem!

• ***Develop prototypes that enable a user to understand how human/machine interaction will occur.***

Since the perception of the quality of software is often based on the perception of the "friendliness" of the interface, prototyping (and the iteration that results) are highly recommended.

• ***Record the origin of and the reason for every requirement.***

This is the first step in establishing traceability back to the customer.

• ***Use multiple views of requirements***.

Building data, functional, and behavioral models provide the software engineer with three different views. This reduces the likelihood that something will be missed and increases the likelihood that inconsistency will be recognized.

• ***Rank requirements.***

Tight deadlines may preclude the implementation of every software requirement. If an incremental process model is applied, those requirements to be delivered in the first increment must be identified.

• ***Work to eliminate ambiguity.***

Because most requirements are described in a natural language, the opportunity for ambiguity abounds. The use of formal technical reviews is one way to uncover and eliminate ambiguity. A software engineer who takes these principles to heart is more likely to develop a software specification that will provide an excellent foundation for design.

The Information Domain

All software applications can be collectively called *data processing.* Interestingly, this term contains a key to our understanding of software requirements. Software is built to process data, to transform data from one form to another; that is, to accept input, manipulate it in some way, and produce output. This fundamental statement of objective is true whether we build batch software for a payroll system or real-time embedded software to control fuel flow to an automobile engine.

The first operational analysis principle requires an examination of the information

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

(1) information content and relationships (the data model)

(2) information flow, and

(3) information structure.

To fully understand the information domain, each of these

views should be considered.

Information content represents the individual data and control objects that constitute some larger collection of information transformed by the software. For example, the data object, **paycheck,** is a composite of a number of important pieces of data: the payee's name, the net amount to be paid, the gross pay, deductions, and so forth. Therefore, the content of **paycheck** is defined by the attributes that are needed to create it. Similarly, the content of a control object called **system status** might be defined by a string of bits. Each bit represents a separate item of information that indicates whether or not a particular device is on- or off-line.

.***Information flow*** represents the manner in which data and control change as each moves through a system. Referring to Figure 11.3, input objects are transformed to intermediate information (data and/or control), which is further transformed to output. Along this transformation path (or paths), additional information may be introduced from an existing data store (e.g., a disk file or memory buffer). The transformations applied to the data are functions or subfunctions that a program must perform. Data and control that move between two transformations (functions) define the interface for each function.

Information structure represents the internal organization of various data and control items. Are data or control items to be organized as an *n*-dimensional table or as a hierarchical tree structure? Within the context of the structure, what information is related to other information?

Modelling

We create functional models to gain a better understanding of the actual entity to be built. When the entity is a physical thing (a building, a plane, a machine), we can build a model that is identical in form and shape but smaller in scale. However, when the entity to be built is software, our model must take a different form.

The second and third operational analysis principles require that we build models

of function and behavior.

Functional models. Software transforms information, and in order to

accomplish this, it must perform at least three generic functions: input, processing, and output. When functional models of an application are created,

the software engineer focuses on problem specific functions. The functional

model begins with a single context level model (i.e., the name of the software to be built). Over a series of iterations, more and more functional detail is

provided, until a thorough delineation of all system functionality is represented.

Behavioral models.

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

Partitioning

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

In essence, *partitioning* decomposes a problem into its constituent parts.

Conceptually, we establish a hierarchical representation of function or information and then partition the uppermost element by

(1) exposing increasing detail by moving vertically in the hierarchy or

(2) functionally decomposing the problem by moving horizontally in the hierarchy

Essential and Implementation Views

An ***essential view*** of software requirements presents the functions to be accomplished and information to be processed without regard to implementation details. For example, the essential view of the *SafeHome* function *read sensor status* does not concern itself with the physical form of the data or the type of sensor that is used. In fact, it could be argued that read status would be a more appropriate name for this function, since it disregards details about the input mechanism altogether.

Similarly, an essential data model of the data item **phone number** (implied by the function *dial phone number*) can be represented at this stage without regard to the underlying data structure (if any) used to implement the data item. By focusing attention on the essence of the problem at early stages of requirements engineering, we leave our options open to specify implementation details during later stages of requirements specification and software design.

The ***implementation view*** of software requirements presents the real world manifestation of processing functions and information structures. In some cases, a physical representation is developed as the first step in software design. However most computer-based systems are specified in a manner that dictates accommodation of certain implementation details. A *SafeHome* input device is a perimeter sensor (not a watch dog, a human guard, or a booby trap). The sensor detects illegal entry by sensing a break in an electronic circuit. The general characteristics of the sensor should be noted as part of a software requirements specification. The analyst must recognize the constraints imposed by predefined system elements (the sensor) and consider the implementation view of function and information when such a view is appropriate.

SOFTWARE PROTOTYPING

Analysis should be conducted regardless of the software engineering paradigm that is applied. However, the form that analysis takes will vary. In some cases it is possible to apply operational analysis principles and derive a model of software from which a design can be developed. In other situations, requirements elicitation (via FAST, QFD, use-cases, or other "brainstorming" techniques [JOR89]) is conducted, analysis principles are applied, and a model of the software to be built, called a *prototype,* is constructed for customer and developer assessment. Finally, some circumstances require the construction of a prototype at the beginning of analysis, since the model is the only means through which requirements can be effectively derived. The model then evolves into production software.

Selecting the Prototyping Approach

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

Before a close-ended or open-ended approach can be chosen, it is necessary to

determine whether the system to be built is amenable to prototyping. A number of prototyping candidacy factors [BOA84] can be defined: application area, application complexity, customer characteristics, and project characteristics

Because the customer must interact with the prototype in later steps, it is essential that

(1) customer resources be committed to the evaluation and refinement of the

prototype

(2) the customer is capable of making requirements decisions in a

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

Prototyping Methods and Tools

For software prototyping to be effective, a prototype must be developed rapidly so that the customer may assess results and recommend changes. To conduct rapid prototyping, three generic classes of methods and tools (e.g., [AND92], [TAN89]) are available:

Fourth generation techniques.

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

Reusable software components.

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

Formal specification and prototyping environments.

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

SOFTWARE REQUIREMENT SPECIFICATION

The *Software Requirements Specification* is produced at the culmination of the analysis task. The function and performance allocated to software as part of system engineering are refined by establishing a complete information description, a detailed functional description, a representation of system behavior, an indication of performance requirements and design constraints, appropriate validation criteria, and other information pertinent to requirements.

The National Bureau of Standards, IEEE (Standard No. 830-1984), and the U.S. Department of Defense have all proposed candidate formats for software requirements specifications (as well as other software engineering

documentation).

The ***Introduction*** of the software requirements specification states the goals and

objectives of the software, describing it in the context of the computer-based system.Actually, the Introduction may be nothing more than the software scope of the planning document.

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

A description of each function required to solve the problem is presented in the

Functional Description. A processing narrative is provided for each function, design constraints are stated and justified, performance characteristics are stated, and one or more diagrams are included to graphically represent the overall structure of the software and interplay among software functions and other system elements.

The ***Behavioral Description*** section of the specification examines the operation of the software as a consequence of external events and internally generated control characteristics.

Validation Criteria is probably the most important and, ironically, the most often

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

Finally, the specification includes a ***Bibliography and Appendix***. The bibliography contains references to all documents that relate to the software. These include other software engineering documentation, technical references, vendor literature, and standards. The appendix contains information that supplements the specifications. Tabular data, detailed description of algorithms, charts, In many cases the *Software Requirements Specification* may be accompanied by an executable prototype (which in some cases may replace the specification), a paper prototype or a *Preliminary User's Manual.*

The ***Preliminary User's Manual*** presents the software as a black box. That is, heavy emphasis is placed on user input and the resultant output. The manual can serve as a valuable tool for uncovering problems at the human/machine interface.

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

The review is first conducted at a macroscopic level; that is, reviewers attempt to

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

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

REPRESENTATION

We have already seen that software requirements may be specified in a variety of ways. However, if requirements are committed to paper or an electronic presentation medium (and they almost always should be!) a simple set of guidelines is well worth following:

Representation format and content should be relevant to the problem.

A general outline for the contents of a *Software Requirements Specification*

can be developed. However, the representation forms contained within

the specification are likely to vary with the application area. For example, a

specification for a manufacturing automation system might use different

symbology, diagrams and language than the specification for a programming

language compiler.

Information contained within the specification should be nested.

Representations should reveal layers of information so that a reader can move to

the level of detail required. Paragraph and diagram numbering schemes

should indicate the level of detail that is being presented. It is sometimes worthwhile to present the same information at different levels of abstraction

to aid in understanding.

Diagrams and other notational forms should be restricted in number

and consistent in use.

Confusing or inconsistent notation, whether graphical

or symbolic, degrades understanding and fosters errors.

Representations should be revisable.

The content of a specification will change. Ideally, CASE tools should be available to update all representations that are affected by each change.

Investigators have conducted numerous studies (e.g., [HOL95], [CUR85]) on human factors associated with specification. There appears to be little doubt that symbology and arrangement affect understanding. However, software engineers appear to have individual preferences for specific symbolic and diagrammatic forms. Familiarity often lies at the root of a person's preference, but other more tangible factors such as spatial arrangement, easily recognizable patterns, and degree of formality often dictate an individual's choice.

SPECIFICATION PRINCIPLES

Specification, regardless of the mode through which we accomplish it, may be viewed as a representation process. Requirements are represented in a manner that ultimately leads to successful software implementation. A number of specification principles,

adapted from the work of Balzer and Goodman [BAL86], can be proposed:

1. Separate functionality from implementation.

2. Develop a model of the desired behavior of a system that encompasses data and the functional responses of a system to various stimuli from the environment.

3. Establish the context in which software operates by specifying the manner in which other system components interact with software.

4. Define the environment in which the system operates and indicate how "a highly intertwined collection of agents react to stimuli in the environment(changes to objects) produced by those agents" .

5. Create a cognitive model rather than a design or implementation model. The cognitive model describes a system as perceived by its user community.

6. Recognize that "the specifications must be tolerant of incompleteness and augmentable." A specification is always a model—an abstraction—of some real (or envisioned) situation that is normally quite complex. Hence, it will be incomplete and will exist at many levels of detail.

7. Establish the content and structure of a specification in a way that will enable it to be amenable to change. This list of basic specification principles provides a basis for representing software requirements. However, principles must be translated into realization. In the next section we examine a set of guidelines for creating a specification of requirements.

THE ELEMENTS OF ANALYSIS MODEL

The analysis model must achieve three primary objectives:

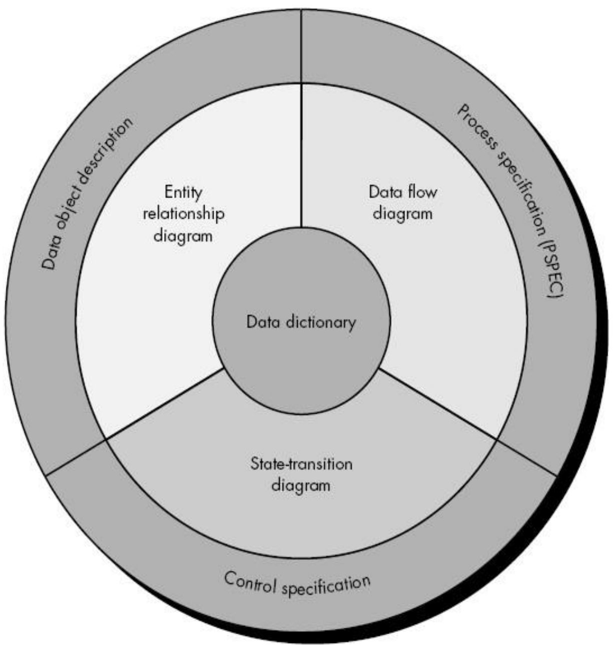
(1) to describe what the customer requires

(2) to establish a basis for the creation of a software design, and

(3) to define a set of requirements that can be validated once the software is built. To accomplish these objectives, the analysis model derived during structured analysis takes the form illustrated in Figure 1.1.

At the core of the model lies the ***data dictionary***—a repository that contains descriptions of all data objects consumed or produced by the software. Three different diagrams surround the the core.

The ***entity relation diagram*** (ERD) depicts relationships between data objects. The ERD is the notation that is used to conduct the data modeling activity. The attributes of each data object noted in the ERD can be described using a data object description.



2.1
re of
is

The ***data flow diagram*** (DFD) serves two purposes: (1) to provide an indication of how data are transformed as they move through the system and (2) to depict the functions (and subfunctions) that transform the data flow. The DFD provides additional information that is used during the analysis of the information domain and serves as a basis for the modeling of function. A description of each function presented in the DFD is contained in a *process specification* (PSPEC).

The ***state transition diagram*** (STD) indicates how the system behaves as a consequence of external events. To accomplish this, the STD represents the various modes of behavior (called *states*) of the system and the manner in which transitions are made from state to state. The STD serves as the basis for behavioral modeling. Additional information about the control aspects of the software is contained in the *control specification* (CSPEC).

DATA MODELLING

Data modeling answers a set of specific questions that are relevant to any data processing application. What are the primary data objects to be processed by the system?

What is the composition of each data object and what attributes describe the

object? Where do the the objects currently reside? What are the relationships between each object and other objects? What are the relationships between the objects and the processes that transform them?

Data Objects, Attributes, and Relationships

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

Data objects

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

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.

Attributes

Attributes define the properties of a data object and take on one of three

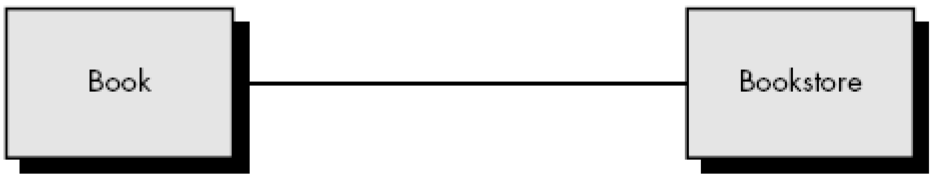
different characteristics. They can be used to

(1) name an instance of the data object,

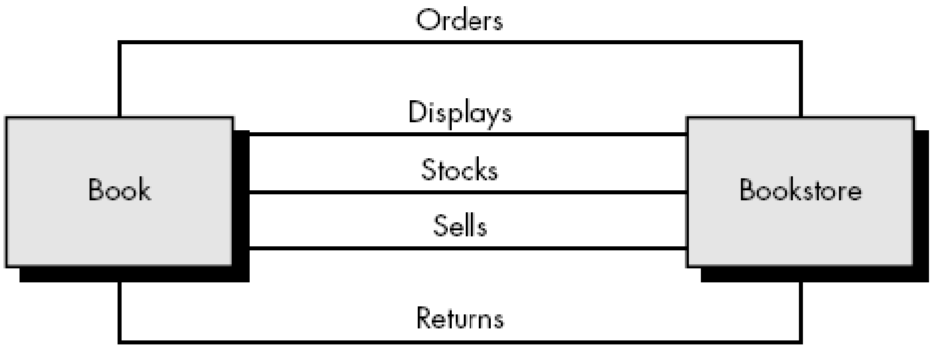
(2) describe the instance, or

(3) make reference to another instance in another table.

In addition, one or more of the attributes must be defined as an *identifier*—that is, the identifier attribute becomes a "key" when we want to find an instance of the data object. In some cases, values for the identifier(s) are unique, although this is not a requirement



(a) A basic connection between objects



(b) Relationships between objects

Relationships

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

two data objects, **book** and **bookstore**. These objects can be represented using

the simple notation illustrated in Figure 2.a. A connection is established between **book** and **bookstore** because the two objects are related. But what are the relationships?

To determine the answer, we must understand the role of books and bookstores

within the context of the software to be built. We can define a set of

object/relationship pairs that define the relevant relationships. For example,

• A bookstore orders books.

• A bookstore displays books.

• A bookstore stocks books.

• A bookstore sells books.

• A bookstore returns books.

<u>Cardinality and Modality</u>

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 is the specification of the number of occurrences of one [object] that can be related to the number of occurrences of another [object]. Cardinality is usually expressed as simply 'one' or 'many.' For example, a husband can have only one wife (in most cultures), while a parent can have many children. Taking into consideration all combinations of 'one' and 'many,' two [objects] can be related as

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

• One-to-many (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.

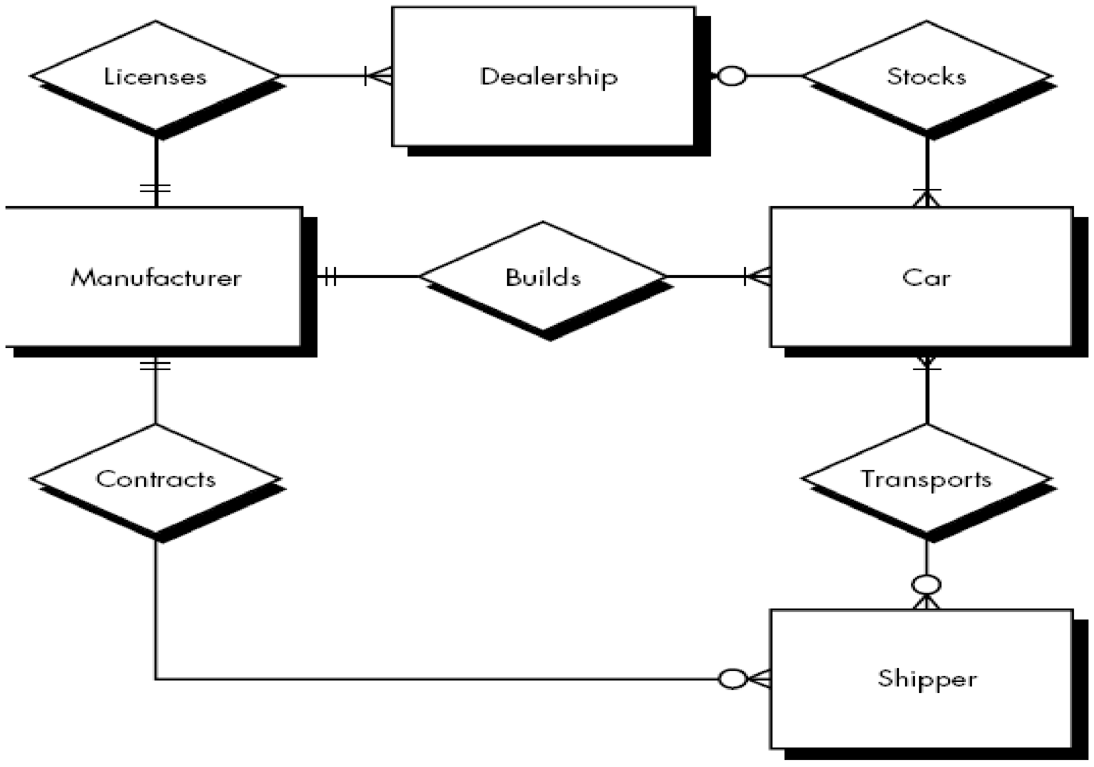
Cardinality defines “the maximum number of objects that can participate in a relationship” It does not, however, provide an indication of whether or not a particular data object must participate in the relationship. To specify this information, the data model adds modality to the object/relationship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

Modality

The *modality* of a relationship is 0 if there is no explicit need for the relationship

to occur or the relationship is optional. The modality is 1 if an occurrence of

the relationship is mandatory. To illustrate, consider software that is used by a local telephone company to process requests for field service. A customer indicates that there is a problem. If the problem is diagnosed as relatively simple, a single repair action occurs. However, if the problem is complex, multiple repair actions may be required. Figure 12.5 illustrates the relationship, cardinality, and modality between the data objects **customer** and **repair action.**



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

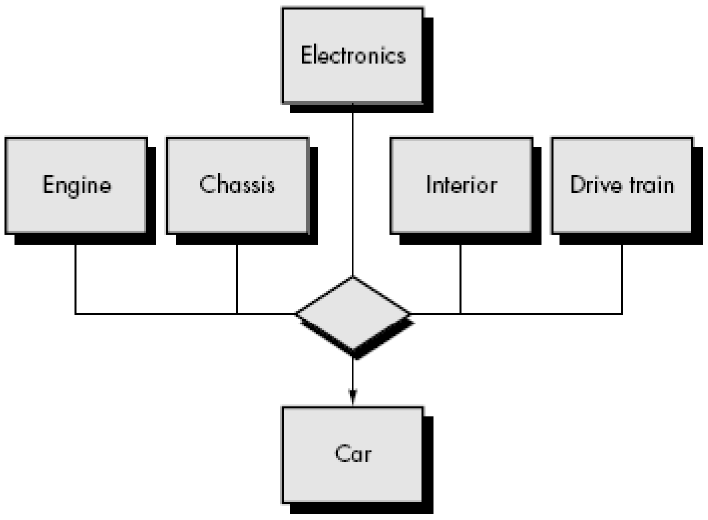


Fig 4

The relationship between the data objects **car** and **manufacturer** would be represented as shown in Figure 3. One manufacturer builds one or many cars. Given the context implied by the ERD, the specification of the data object **car.**

ERD notation also provides a mechanism that represents the associativity between objects. In the figure, each of the data objects that model the individual subsystems is associated with the data object **car.**

An *associative data object* is represented as shown in Figure 4.

Data Flow Diagrams

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

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

A level 0 DFD, also called a *fundamental system model* or a *context model,* represents the entire software element as a single bubble with input and output data indicated by incoming and outgoing arrows, respectively.

Additional processes (bubbles) and information flow paths are represented as the level 0 DFD is partitioned to reveal more detail. For example, a level 1 DFD might contain five or six bubbles with interconnecting arrows. Each of the processes represented at level 1 is a subfunction of the overall system depicted in the context model.

BEHAVIORAL MODELLING

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

A state is any observable mode of behavior. For example, states for a monitoring

and control system for pressure vessels described might be *monitoring*

state, alarm state, pressure release state, and so on. Each of these states represents a mode of behavior of the system. A state transition diagram indicates how the system moves from state to state.

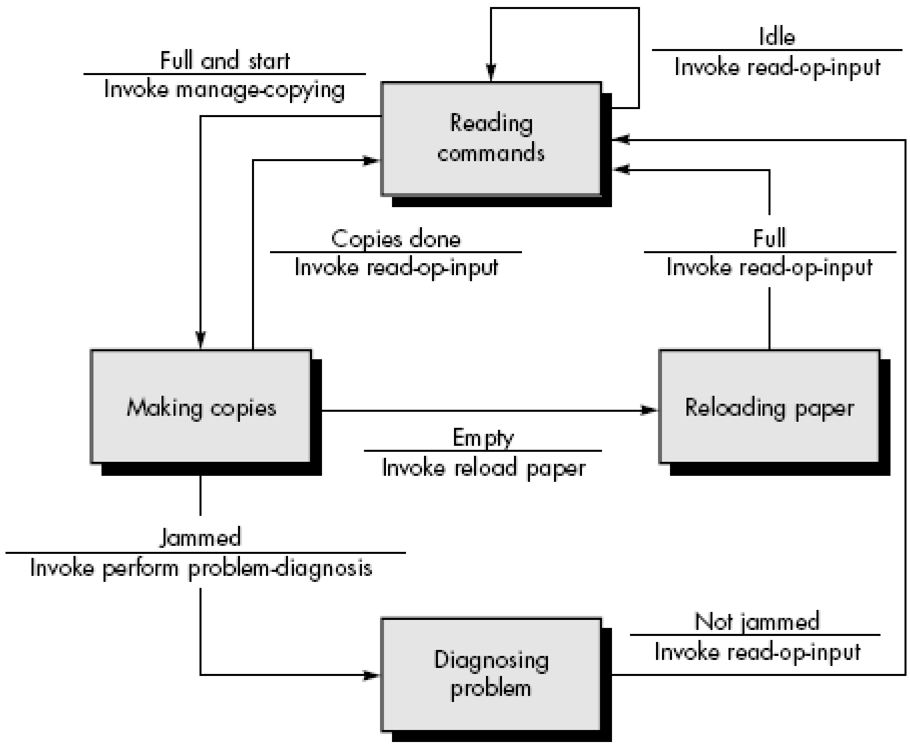
Control flows are shown entering and exiting individual processes and the vertical bar representing the control specification(CSPEC) "window." For example, the **paper feed status** and **start/stop** events flow into the CSPEC bar. This implies that each of these events will cause some process represented in the CFD to be activated. If we were to examine the CSPEC internals, the **start/stop** event would be shown to activate/deactivate the *manage copying* process. Similarly, the **jammed** event (part of **paper feed status**) would activate *perform problem diagnosis.* It should be noted that all vertical bars within the CFD refer to the same CSPEC. An event flow can be input directly into

a process as shown with **repro fault.** However, this flow does not activate the process but rather provides control information for the process algorithm.

The Hatley and Pirbhai extensions to basic structured analysis notation focus

less on the creation of additional graphical symbols and more on the representation and specification of the control-oriented aspects of the software. The dashed arrow is once again used to represent control or event flow. Unlike Ward and Mellor, Hatley and Pirbhai suggest that dashed and solid notation be represented separately. Therefore, a *control flow diagram* is defined. The CFD contains the same processes as the DFD, but shows control flow, rather than data flow.

Instead of representing control processes directly within the flow model, a notational reference (a solid bar) to a *control specification* (CSPEC) is used. In essence, the solid bar can be viewed as a "window" into an "executive" (the CSPEC) that controls the processes (functions) represented in the DFD based on the event that is passed through the window. A process specification is used to describe the inner workings of a process represented in a flow diagram.



A simplified state transition diagram for the photocopier software is shown in Figure 5. The rectangles represent system states and the arrows represent transitions between states. Each arrow is labeled with a ruled expression. The top value indicates the event(s) that cause the transition to occur. The bottom value indicates the action that occurs as a consequence of the event. Therefore, when the paper tray is **full** and the **start** button is pressed, the system moves from the *reading commands* state to the *making copies* state. Note that states do not necessarily correspond to processes on a one-to-one basis. For example, the state *making copies* would encompass both the *manage copying* and *produce user displays* processes.

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.

Creating a Data Flow Model

The data flow diagram enables the software engineer to develop models of the information domain and functional domain at the same time. As the DFD is refined into greater levels of detail, the analyst performs an implicit functional decomposition of the system, thereby accomplishing the fourth operational analysis principle for function.

At the same time, the DFD refinement results in a corresponding refinement of

data as it moves through the processes that embody the application.

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

diagram:

(1) the level 0 data flow diagram should depict the software/system as a

single bubble.

(2) primary input and output should be carefully noted.

(3) refinement should begin by isolating candidate processes, data objects, and stores to be represented at the next level.

(4) all arrows and bubbles should be labeled with meaningful names.

(5) information flow continuity must be maintained from level to level, and

(6) one bubble at a time should be refined. There is a natural tendency to overcomplicate the data flow diagram.

This occurs when the analyst attempts to show too much detail too early or represents procedural aspects of the software in lieu of information flow.

THE DATA DICTIONARY

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

The data dictionary has been proposed as a quasi-formal grammar for describing

the content of objects defined during structured analysis. This important modeling notation has been defined in the following manner :

“ The *data dictionary* is an organized listing of all data elements that are pertinent to the system, with precise, rigorous definitions so that both user and system analyst will have a common understanding of inputs, outputs, components of stores and [even] intermediate calculations “.

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

• *Name*—the primary name of the data or control item, the data store or an

external entity.

• *Alias*—other names used for the first entry.

• *Where-used/how-used*—a listing of the processes that use the data or control

item and how it is used (e.g., input to the process, output from the process,

as a store, as an external entity.

• *Content description*—a notation for representing content.

• *Supplementary information*—other information about data types, preset values (if known), restrictions or limitations, and so forth.

Once a data object or control item name and its aliases are entered into the data

dictionary, consistency in naming can be enforced. That is, if an analysis team member decides to name a newly derived data item **xyz,** but **xyz** is already in the dictionary, the CASE tool supporting the dictionary posts a warning to indicate duplicate names. This improves the consistency of the analysis model and helps to reduce errors.

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

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

Data Construct	**Notation**	**Meaning**
	=	is composed of
Sequence	+	and
Selection	[|]	either-or
Repetition	{ }*n*	n repetitions of
	( )	optional data
	* ... *	delimits comments

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

1. As a sequence of data items.

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

3. As a repeated grouping of data items. Each data item entry that is represented

as part of a sequence, selection, or repetition may itself be another composite data item that needs further refinement within the dictionary.

The data dictionary provides us with a precise definition of **telephone number** for the DFD in question. In addition it indicates where and how this data item is used and any supplementary information that is relevant to it.

The data dictionary entry begins as follows:

name: telephone number

aliases: none

where used/how used: assess against set-up (output)

dial phone (input)

description:

telephone number = [local number|long distance number] local number = prefix + access number

long distance number = 1 + area code + local

number area code = [800 | 888 | 561]

prefix = *a three digit number that never starts with 0

or 1* access number = * any four number string *

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

The data dictionary defines information items unambiguously. Although we might

assume that the telephone number represented by the DFD in Figure 12.22 could accommodate a 25-digit long distance carrier access number, the data dictionary content description tells us that such numbers are not part of the data that may be used.