LAB MANUAL

SOFTWARE ENGINEERING

Software engineering is the application of a systematic, disciplined, quantifiable approach to the development, operation, and maintenance of software, and the study of these approaches; that is, the application of engineering to software.[1]

The term software engineering first appeared in the 1968 NATO Software Engineering Conference and was meant to provoke thought regarding the current "software crisis" at the time. Since then, it has continued as a profession and field of study dedicated to creating software that is of higher quality, more affordable, maintainable, and quicker to build. Since the field is still relatively young compared to its sister fields of engineering, there is still much debate around what software engineering actually is, and if it conforms to the classical definition of engineering. It has grown organically out of the limitations of viewing software as just computer programming.

A **software development process** is a structure imposed on the development of a software product. Synonyms include software life cycle and *software process*. There are several models for such processes, each describing approaches to a variety of tasks or activities that take place during the process.

What is a software life cycle model?
A software life cycle model is either a **descriptive** or **prescriptive** characterization of how software is or should be developed. A descriptive model describes the history of how a particular software system was developed. Descriptive models may be used as the basis for understanding and improving software development processes, or for building empirically grounded prescriptive models.

A prescriptive model prescribes how a new software system should be developed. Prescriptive models are used as guidelines or frameworks to organize and structure how software development activities should be performed, and in what order. Typically, it is easier and more common to articulate a prescriptive life cycle model for how software systems should be developed. This is possible since most such models are intuitive or well reasoned. This means that many idiosyncratic details that describe how a software systems is built in practice can be ignored, generalized, or deferred for later consideration. This, of course, should raise concern for the relative validity and robustness of such life cycle models when developing different kinds of application systems, in different kinds of development settings, using different programming languages, with differentially skilled staff, etc. However, prescriptive models are also used to package the development tasks and techniques for using a given set of software engineering tools or environment during a development project. Descriptive life cycle models, on the other hand, characterize how particular software systems are actually developed in specific settings. As such, they are less common and more difficult to articulate for an obvious reason: one must observe or collect data throughout the life cycle of a software system, a period of elapsed time often measured in years. Also, descriptive models are specific to the systems observed and only generalizable through systematic comparative analysis.

Therefore, this suggests the prescriptive software life cycle models will dominate attention until a sufficient base of observational data is available to articulate empirically grounded descriptive life cycle models.

These two characterizations suggest that there are a variety of purposes for articulating software life cycle models. These characterizations serve as a

- Guideline to organize, plan, staff, budget, schedule and manage software project work over organizational time, space, and computing environments.
- Prescriptive outline for what documents to produce for delivery to client.
- Basis for determining what software engineering tools and methodologies will be most appropriate to support different life cycle activities.
- Framework for analyzing or estimating patterns of resource allocation and consumption during the software life cycle (Boehm 1981)
- Basis for conducting empirical studies to determine what affects software productivity, cost, and overall quality.

What is a software process model?

In contrast to software life cycle models, software process models often represent a networked sequence of activities, objects, transformations, and events that embody strategies for accomplishing software evolution. Such models can be used to develop more precise and formalized descriptions of software life cycle activities. Their power emerges from their utilization of a sufficiently rich notation, syntax, or semantics, often suitable for computational processing. Software process networks can be viewed as representing multiple interconnected task chains (Kling 1982, Garg 1989). Task chains represent a non-linear sequence of actions that structure and transform available computational objects (resources) into intermediate or finished products. Non-linearity implies that the sequence of actions may be non-deterministic, iterative, accommodate multiple/parallel alternatives, as well as partially ordered to account for incremental progress. Task actions in turn can be viewed a non-linear sequences of primitive actions which denote atomic units of computing work, such as a user's selection of a command or menu entry using a mouse or keyboard. Winograd and others have referred to these units of cooperative work between people and computers as "structured discourses of work" (Winograd 1986), while task chains have become popularized under the name of "workflow" (Bolcer 1998). Task chains can be employed to characterize either prescriptive or descriptive action sequences. Prescriptive task chains are idealized plans of what actions should be accomplished, and in what order. For example, a task chain for the activity of object-oriented software design might include the following task actions:

- Develop an informal narrative specification of the system.
- Identify the objects and their attributes.
- Identify the operations on the objects.
- Identify the interfaces between objects, attributes, or operations.
- Implement the operations.

Clearly, this sequence of actions could entail multiple iterations and non-procedural primitive action invocations in the course of incrementally progressing toward an object-oriented software design. Task chains join or split into other task chains resulting in an overall production network or web (Kling 1982). The production web represents the "organizational production system" that transforms raw computational, cognitive, and other organizational resources into assembled, integrated and usable software systems. The production lattice therefore structures how a software system is developed, used, and maintained. However, prescriptive task chains and actions cannot be formally guaranteed to anticipate all possible circumstances or idiosyncratic

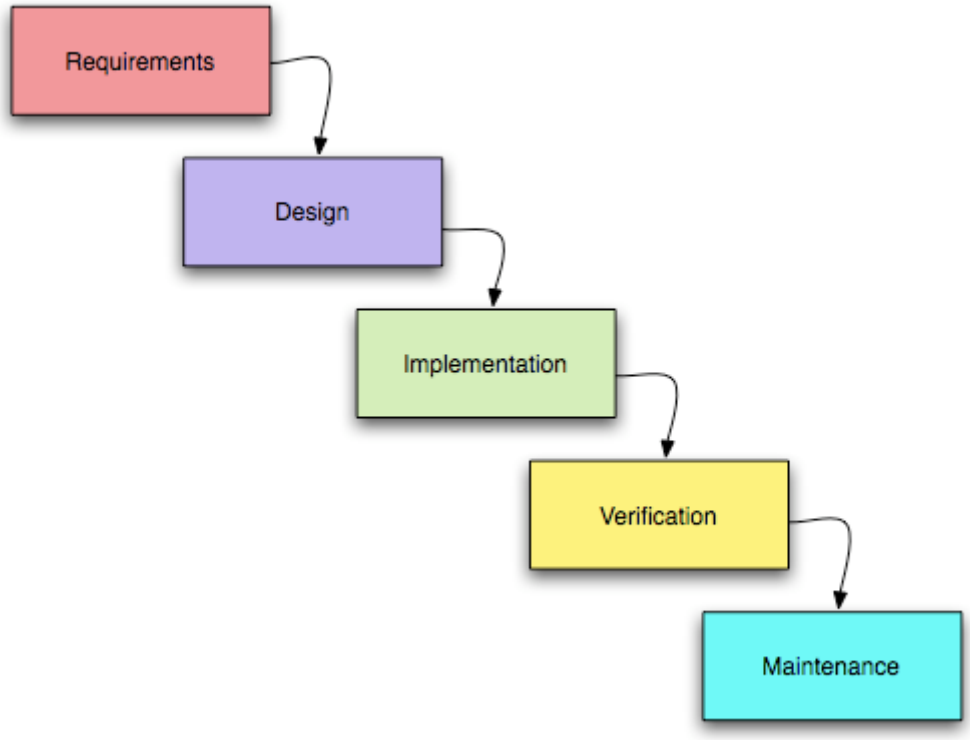
Software development activities

Planning

The important task in creating a software product is extracting the requirements or requirements analysis. Customers typically have an abstract idea of what they want as an end result, but not what software should do. Incomplete, ambiguous, or even contradictory requirements are recognized by skilled and experienced software engineers at this point. Frequently demonstrating live code may help reduce the risk that the requirements are incorrect.

Once the general requirements are gleaned from the client, an analysis of the scope of the development should be determined and clearly stated. This is often called a scope document.

Certain functionality may be out of scope of the project as a function of cost or as a result of unclear requirements at the start of development. If the development is done externally, this document can be considered a legal document so that if there are ever disputes, any ambiguity of what was promised to the client can be clarified.



Implementation, testing and documenting

Implementation is the part of the process where software engineers actually program the code for the project. Software testing is an integral and important part of the software development process. This part of the process ensures that bugs are recognized as early as possible.

Documenting the internal design of software for the purpose of future maintenance and enhancement is done throughout development. This may also include the authoring of an API, be it external or internal.

Deployment and maintenance

Deployment starts after the code is appropriately tested, is approved for release and sold or otherwise distributed into a production environment.

Software Training and Support is important because a large percentage of software projects fail because the developers fail to realize that it doesn't matter how much time and planning a development team puts into creating software if nobody in an organization ends up using it. People are often resistant to change and avoid venturing into an unfamiliar area, so as a part of the deployment phase, it is very important to have training classes for new clients of your software.

Maintenance and enhancing software to cope with newly discovered problems or new requirements can take far more time than the initial development of the software. It may be necessary to add code that does not fit the original design to correct an unforeseen problem or it may be that a customer is requesting more functionality and code can be added to accommodate their requests. It is during this phase that customer calls come in and you see whether your testing was extensive enough to uncover the problems before customers do. If the labor cost of the maintenance phase exceeds 25% of the prior-phases' labor cost, then it is likely that the overall quality, of at least one prior phase, is poor. In that case, management should consider the option of rebuilding the system (or portions) before maintenance cost is out of control.

Bug Tracking System tools are often deployed at this stage of the process to allow development teams to interface with customer/field teams testing the software to identify any real or perceived issues. These software tools, both open source and commercially licensed, provide a customizable process to acquire, review, acknowledge, and respond to reported issues.

Algorithms

In mathematics, computing, and related subjects, an algorithm is an effective method for solving a problem using a finite sequence of instructions. Algorithms are used for calculation, data processing, and many other fields.

Each algorithm is a list of well-defined instructions for completing a task. Starting from an initial state, the instructions describe a computation that proceeds through a well-defined series of successive states, eventually terminating in a final ending state. The transition from one state to the next is not necessarily deterministic; some algorithms, known as randomized algorithms, incorporate randomness.

Why algorithms are necessary: an informal definition

For a detailed presentation of the various points of view around the definition of "algorithm" see Algorithm characterizations. For examples of simple addition algorithms specified in the detailed manner described in Algorithm characterizations, see Algorithm examples.

While there is no generally accepted *formal* definition of "algorithm", an informal definition could be "a process that performs some sequence of operations." For some people, a program is only an algorithm if it stops eventually. For others, a program is only an algorithm if it stops before a given number of calculation steps. A prototypical example of an algorithm is Euclid's algorithm to determine the maximum common divisor of two integers.

We can derive clues to the issues involved and an informal meaning of the word from the following quotation from Boolos & Jeffrey:

No human being can write fast enough, or long enough, or small enough† (†"smaller and smaller without limit ...you'd be trying to write on molecules, on atoms, on electrons") to list all members of an enumerably infinite set by writing out their names, one after another, in some notation. But humans can do something equally useful, in the case of certain enumerably infinite sets: They can give explicit instructions for determining the *n*th member of the set, for arbitrary finite *n*. Such instructions are to be given quite explicitly, in a form in which they could be followed by a computing machine, or by a human who is capable of carrying out only very elementary operations on symbols.

The term "enumerably infinite" means "countable using integers perhaps extending to infinity." Thus Boolos and Jeffrey are saying that an algorithm *implies* instructions for a process that "creates" output integers from an *arbitrary* "input" integer or integers that, in theory, can be chosen from 0 to infinity. Thus we might expect an algorithm to be an algebraic equation such as y = m + n — two arbitrary "input variables" m and n that produce an output y. As we see in Algorithm characterizations — the word algorithm implies much more than this, something on the order of (for our addition example):

Precise instructions (in language understood by "the computer") for a "fast, efficient, good" process that specifies the "moves" of "the computer" (machine or human, equipped with the necessary internally-contained information and capabilities) to find, decode, and then munch arbitrary input integers/symbols m and n, symbols + and = ... and (reliably, correctly, "effectively") produce, in a "reasonable" time, output-integer y at a specified place and in a specified format.

The concept of algorithm is also used to define the notion of decidability. That notion is central for explaining how formal systems come into being starting from a small set of axioms and rules. In logic, the time that an algorithm requires to complete cannot be measured, as it is not apparently related with our customary physical dimension. From such uncertainties, that characterize ongoing work, stems the unavailability of a definition of *algorithm* that suits both concrete (in some sense) and abstract usage of the term.

Algorithms can be expressed in many kinds of notation, including natural languages, pseudocode, flowcharts, programming languages or control tables (processed by interpreters). Natural language expressions of algorithms tend to be verbose and ambiguous, and are rarely used for complex or technical algorithms. Pseudocode, flowcharts and control tables are structured ways to express algorithms that avoid many of the ambiguities common in natural language statements, while remaining independent of a particular implementation language. Programming languages are primarily intended for expressing algorithms in a form that can be executed by a computer, but are often used as a way to define or document algorithms.

There is a wide variety of representations possible and one can express a given Turing machine program as a sequence of machine tables (see more at finite state machine and state transition table), as flowcharts (see more at state diagram), or as a form of rudimentary machine code or assembly code called "sets of quadruples". Sometimes it is

helpful in the description of an algorithm to supplement small "flow charts" (state diagrams) with natural-language and/or arithmetic expressions written inside "block diagrams" to summarize what the "flow charts" are accomplishing.

In computer systems, an algorithm is basically an instance of logic written in software by software developers to be effective for the intended "target" computer(s), in order for the software on the target machines to *do something*. For instance, if a person is writing software that is supposed to print out a PDF document located at the operating system folder "/My Documents" at computer drive "D:" every Friday at 10PM, they will write an algorithm that specifies the following actions: "If today's date (computer time) is 'Friday,' open the document at 'D:/My Documents' and call the 'print' function". While this simple algorithm does not look into whether the printer has enough paper or whether the document has been moved into a different location, one can make this algorithm more robust and anticipate these problems by rewriting it as a formal CASE statement[14] or as a (carefully crafted) sequence of IF-THEN-ELSE statements. For example the CASE statement might appear as follows:

CASE 1: IF today's date is NOT Friday THEN *exit this CASE instruction* ELSE
CASE 2: IF today's date is Friday AND the document is located at 'D:/My Documents' AND there is paper in the printer THEN print the document (and *exit this CASE instruction*) ELSE
CASE 3: IF today's date is Friday AND the document is NOT located at 'D:/My Documents' THEN display 'document not found' error message (and *exit this CASE instruction*) ELSE
CASE 4: IF today's date is Friday AND the document is located at 'D:/My Documents' AND there is NO paper in the printer THEN (i) display 'out of paper' error message and (ii) *exit*.

Note that CASE 3 includes two possibilities: (i) the document is NOT located at 'D:/My Documents' AND there's paper in the printer OR (ii) the document is NOT located at 'D:/My Documents' AND there's NO paper in the printer.

The sequence of IF-THEN-ELSE tests might look like this:

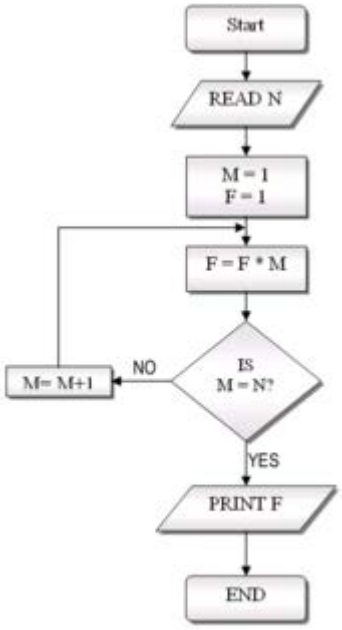
TEST 1: IF today's date is NOT Friday THEN *done* ELSE TEST 2:
TEST 2: IF the document is NOT located at 'D:/My Documents' THEN display 'document not found' error message ELSE TEST 3:
TEST 3: IF there is NO paper in the printer THEN display 'out of paper' error message ELSE print the document.

These examples' logic grants precedence to the instance of "NO document at 'D:/My Documents' ". Also observe that in a well-crafted CASE statement or sequence of IF-THEN-ELSE statements the number of distinct actions—4 in these examples: do nothing, print the document, display 'document not found', display 'out of paper' -- equals the number of cases. Given unlimited memory, a computational machine with the ability to execute either a set of CASE statements or a sequence of IF-THEN-ELSE statements is

Turing complete. Therefore, anything that is computable can be computed by this machine. This form of algorithm is fundamental to computer programming in all its forms.

Flowchart

A flowchart is a diagrammatic representation of a step-by-step solution to a given problem. It is a common type of diagram, that represents an algorithm or process, showing the steps as boxes of various kinds, and their order by connecting these with arrows. Data is represented in these boxes, and arrows connecting them represent flow / direction of flow of data. Flowcharts are used in analyzing, designing, documenting or managing a process or program in various fields.[1



A simple flowchart for computing factorial N (N!)

A flowchart for computing factorial N (N!) where N! = (1 * 2 * 3 * ... * N), see image. This flowchart represents a "loop and a half" — a situation discussed in introductory programming textbooks that requires either a duplication of a component (to be both inside and outside the loop) or the component to be put inside a branch in the loop.

Symbols

A typical flowchart from older Computer Science textbooks may have the following kinds of symbols:

Start and end symbols

Represented as circles, ovals or rounded rectangles, usually containing the word "Start" or "End", or another phrase signaling the start or end of a process, such as "submit enquiry" or "receive product".

Arrows

Showing what's called "flow of control" in computer science. An arrow coming from one symbol and ending at another symbol represents that control passes to the symbol the arrow points to.

Processing steps

Represented as rectangles. Examples: "Add 1 to X"; "replace identified part"; "save changes" or similar.

Input/Output

Represented as a parallelogram. Examples: Get X from the user; display X.

Conditional or decision

Represented as a diamond (rhombus). These typically contain a Yes/No question or True/False test. This symbol is unique in that it has two arrows coming out of it, usually from the bottom point and right point, one corresponding to Yes or True, and one corresponding to No or False. The arrows should always be labeled. More than two arrows can be used, but this is normally a clear indicator that a complex decision is being taken, in which case it may need to be broken-down further, or replaced with the "pre-defined process" symbol.

A number of other symbols that have less universal currency, such as:

- A *Document* represented as a rectangle with a wavy base;
- A *Manual input* represented by parallelogram, with the top irregularly sloping up from left to right. An example would be to signify data-entry from a form;
- A *Manual operation* represented by a trapezoid with the longest parallel side at the top, to represent an operation or adjustment to process that can only be made manually.
- A *Data File* represented by a cylinder.

Flowcharts may contain other symbols, such as connectors, usually represented as circles, to represent converging paths in the flowchart. Circles will have more than one arrow coming into them but only one going out. Some flowcharts may just have an arrow point to another arrow instead. These are useful to represent an iterative process (what in Computer Science is called a loop). A loop may, for example, consist of a connector where control first enters, processing steps, a conditional with one arrow exiting the loop, and one going back to the connector. Off-page connectors are often used to signify a connection to a (part of another) process held on another sheet or screen. It is important to remember to keep these connections logical in order. All processes should flow from top to bottom and left to right.

Types of flowcharts

Sterneckert (2003) suggested that flowcharts can be modelled from the perspective of different user groups (such as managers, system analysts and clerks) and that there are four general types:

- *Document flowcharts*, showing controls over a document-flow through a system
- *Data flowcharts*, showing controls over a data flows in a system
- *System flowcharts* showing controls at a physical or resource level
- *Program flowchart*, showing the controls in a program within a system

Notice that every type of flowchart focusses on some kind of control, rather than on the particular flow itself

However there are several of these classifications. For example Andrew Veronis (1978) named three basic types of flowcharts: the *system flowchart*, the *general flowchart*, and the *detailed flowchart* That same year Marilyn Bohl (1978) stated "in practice, two kinds of flowcharts are used in solution planning: *system flowcharts* and *program flowcharts*..." More recently Mark A. Fryman (2001) stated that there are more differences: "Decision flowcharts, logic flowcharts, systems flowcharts, product flowcharts, and process flowcharts are just a few of the different types of flowcharts that are used in business and government".

In addition, many diagram techniques exist that are similar to flowcharts but carry a different name, such as UML activity diagrams

Data Flow Diagram

A data-flow diagram (DFD) is a graphical representation of the "flow" of data through an information system. DFDs can also be used for the visualization of data processing (structured design). On a DFD, data items flow *from* an external data source or an internal data store *to* an internal data store or an external data sink, *via* an internal *process*.

A DFD provides no information about the timing or ordering of processes, or about whether processes will operate in sequence or in parallel. It is therefore quite different from a flowchart, which shows the *flow of control* through an algorithm, allowing a reader to determine what operations will be performed, in what order, and under what circumstances, but *not* what kinds of data will be input to and output from the system, *nor* where the data will come from and go to, *nor* where the data will be stored (all of which are shown on a DFD).

It is common practice to draw a context-level data flow diagram first, which shows the interaction between the system and external agents which act as data sources and data sinks. On the context diagram (also known as the *Level 0 DFD*) the system's interactions with the outside world are modelled purely in terms of data flows across the *system*

boundary. The context diagram shows the entire system as a single process, and gives no clues as to its internal organization.

This context-level DFD is next "exploded", to produce a Level 1 DFD that shows some of the detail of the system being modeled. The Level 1 DFD shows how the system is divided into sub-systems (processes), each of which deals with one or more of the data flows to or from an external agent, and which together provide all of the functionality of the system as a whole. It also identifies internal data stores that must be present in order for the system to do its job, and shows the flow of data between the various parts of the system.

Data-flow diagrams were invented by Larry Constantine, the original developer of structured design,[2] based on Martin and Estrin's "data-flow graph" model of computation. Data-flow diagrams (DFDs) are one of the three essential perspectives of the structured-systems analysis and design method SSADM. The sponsor of a project and the end users will need to be briefed and consulted throughout all stages of a system's evolution. With a data-flow diagram, users are able to visualize how the system will operate, what the system will accomplish, and how the system will be implemented. The old system's dataflow diagrams can be drawn up and compared with the new system's data-flow diagrams to draw comparisons to implement a more efficient system. Data-flow diagrams can be used to provide the end user with a physical idea of where the data they input ultimately has an effect upon the structure of the whole system from order to dispatch to report. How any system is developed can be determined through a data-flow diagram.

Data flow diagrams can be used to provide a clear representation of any business function. The technique starts with an overall picture of the business and continues by analyzing each of the functional areas of interest. This analysis can be carried out to precisely the level of detail required. The technique exploits a method called top-down expansion to conduct the analysis in a targeted way. The result is a series of diagrams that represent the business activities in a way that is clear and easy to communicate. A business model comprises one or more data flow diagrams (also known as business process diagrams). Initially a context diagram is drawn, which is a simple representation of the entire system under investigation. This is followed by a level 1 diagram; which provides an overview of the major functional areas of the business. Don't worry about the symbols at this stage, these are explained shortly. Using the context diagram together with additional information from the area of interest, the level 1 diagram can then be drawn.

The level 1 diagram identifies the major business processes at a high level and any of these processes can then be analyzed further - giving rise to a corresponding level 2 business process diagram. This process of more detailed analysis can then continue – through level 3, 4 and so on. However, most investigations will stop at level 2 and it is very unusual to go beyond a level 3 diagram. Identifying the existing business processes, using a technique like data flow diagrams, is an essential precursor to business process re-engineering, migration to new technology, or refinement of an existing business

process. However, the level of detail required will depend on the type of change being considered.

Data Flow Diagrams – Diagram Notation

There are only five symbols that are used in the drawing of business process diagrams (data flow diagrams). These are now explained, together with the rules that apply to them. This diagram represents a banking process, which maintains customer accounts. In this example, customers can withdraw or deposit cash, request information about their account or update their account details. The five different symbols used in this example represent the full set of symbols required to draw any business process diagram.

External Entity

An external entity is a source or destination of a data flow which is outside the area of study. Only those entities which originate or receive data are represented on a business process diagram. The symbol used is an oval containing a meaningful and unique identifier.

Process

A process shows a transformation or manipulation of data flows within the system. The symbol used is a rectangular box which contains 3 descriptive elements: Firstly an identification number appears in the upper left hand corner. This is allocated arbitrarily at the top level and serves as a unique reference. Secondly, a location appears to the right of the identifier and describes where in the system the process takes place. This may, for example, be a department or a piece of hardware. Finally, a descriptive title is placed in the centre of the box. This should be a simple imperative sentence with a specific verb, for example 'maintain customer records' or 'find driver'.

Data Flow

A data flow shows the flow of information from its source to its destination. A data flow is represented by a line, with arrowheads showing the direction of flow. Information always flows to or from a process and may be written, verbal or electronic. Each data flow may be referenced by the processes or data stores at its head and tail, or by a description of its contents.

Data Store

A data store is a holding place for information within the system:

It is represented by an open ended narrow rectangle. Data stores may be long-term files such as sales ledgers, or may be short-term accumulations: for example batches of documents that are waiting to be processed. Each data store should be given a reference followed by an arbitrary number.

Resource Flow

A resource flow shows the flow of any physical material from its source to its destination. For this reason they are sometimes referred to as physical flows. The physical material in question should be given a meaningful name. Resource flows are usually restricted to early, high-level diagrams and are used when a description of the physical flow of materials is considered to be important to help the analysis.

Data Flow Diagrams – The Rules

It is normal for all the information represented within a system to have been obtained from, and/or to be passed onto, an external source or recipient. These external entities may be duplicated on a diagram, to avoid crossing data flow lines. Where they are duplicated a stripe is drawn across the left hand corner, like this. The addition of a lowercase letter to each entity on the diagram is a good way to uniquely identify them.

Processes

When naming processes, avoid glossing over them, without really understanding their role. Indications that this has been done are the use of vague terms in the descriptive title area - like 'process' or 'update'. The most important thing to remember is that the description must be meaningful to whoever will be using the diagram.

Data Flows

Double headed arrows can be used (to show two-way flows) on all but bottom level diagrams. Furthermore, in common with most of the other symbols used, a data flow at a particular level of a diagram may be decomposed to multiple data flows at lower levels.

Data Stores

Each store should be given a reference letter, followed by an arbitrary number. These reference letters are allocated as follows:

'D' - indicates a permanent computer file

'M' - indicates a manual file

'T' - indicates a transient store, one that is deleted after processing.

In order to avoid complex flows, the same data store may be drawn several times on a diagram. Multiple instances of the same data store are indicated by a double vertical bar on their left hand edge.

Data Flow Diagrams – Relationship Grid

There are rules governing various aspects of the diagram components and how they can relate to one another.

Data Flows

For data flows the rules are as follows: Data flows and resource flows are allowed between external entities and processes. Data flows are also allowed between different external entities. However, data flows and resource flows are not allowed between external entities and data stores.

Processes

For processes the data flow rules are as follows: Data flows and resource flows are allowed between processes and external entities and between processes and data stores. They are also allowed between different processes. In other words processes can communicate with all other areas of the business process diagram.

Data Stores

For data stores the data flow rules are as follows: Data flows and resource flows are allowed between data stores and processes. However, these flows are not allowed between data stores and external entities or between one data store and another. In practice this means that data stores cannot initiate a communication of information, they require a process to do this.

Data Flow Diagrams – Context Diagrams

The context diagram represents the entire system under investigation. This diagram should be drawn first, and used to clarify and agree the scope of the investigation. The components of a context diagram are clearly shown on this screen. The system under investigation is represented as a single process, connected to external entities by data flows and resource flows.

The context diagram clearly shows the interfaces between the system under investigation and the external entities with which it communicates. Therefore, whilst it is often conceptually trivial, a context diagram serves to focus attention on the system boundary and can help in clarifying the precise scope of the analysis. The context diagram shown on this screen represents a book lending library. The library receives details of books, and orders books from one or more book suppliers.

Books may be reserved and borrowed by members of the public, who are required to give a borrower number. The library will notify borrowers when a reserved book becomes available or when a borrowed book becomes overdue. In addition to supplying books, a book supplier will furnish details of specific books in response to library enquiries.

Note, that communications involving external entities are only included where they involve the 'system' process. Whilst a book supplier would communicate with various agencies, for example, publishers and other suppliers - these data flow are remote from the 'system' process and so this is not included on the context diagram.

Data Flow Diagrams – Context Diagram Guidelines

Firstly, draw and name a single process box that represents the entire system. Next, identify and add the external entities that communicate directly with the process box. Do this by considering origin and destination of the resource flows and data flows. Finally, add the resource flows and data flows to the diagram. In drawing the context diagram you should only be concerned with the most important information flows. These will be concerned with issues such as: how orders are received and checked, with providing good customer service and with the paying of invoices. Remember that no business process diagram is the definitive solution - there is no absolute right or wrong.

Data Flow Diagrams – Level 1 Diagrams

The level 1 diagram shows the main functional areas of the system under investigation. As with the context diagram, any system under investigation should be represented by only one level 1 diagram. There is no formula that can be applied in deciding what is, and what is not, a level 1 process. Level 1 processes should describe only the main functional areas of the system, and you should avoid the temptation of including lower level processes on this diagram. As a general rule no business process diagram should contain more than 12 process boxes.

The level 1 diagram is surrounded by the outline of a process box that represents the boundaries of the system. Because the level 1 diagram depicts the whole of the system under investigation, it can be difficult to know where to start. There are three different methods, which provide a practical way to start the analysis. These are explained in the following section and any one of them, or a combination, may prove to be the most helpful in any given investigation. There are three different methods, which provide a practical way to start the analysis. These are introduced below and any one of them, or a combination, may prove to be the most helpful in any given investigation:

Data Flow Diagrams – Resource Flow Analysis

Resource flow analysis may be a useful method for starting the analysis if the current system consists largely of the flow of goods, as this approach concentrates on following the flow of physical objects. Resource flow analysis may be a useful method for developing diagrams if the current system consists largely of the flow of goods. Physical resources are traced from when they arrive within the boundaries of the system, through the points at which some action occurs, to their exit from the system. The rationale behind this method is that information will normally flow around the same paths as the physical objects.

Data Flow Diagrams – Organizational Structure Analysis

The organizational structure approach starts from an analysis of the main roles that exist within the organization, rather than the goods or information that is flowing around the system. Identification of the key processes results from looking at the organizational structure and deciding which functional areas are relevant to the current investigation. By looking at these areas in more detail, and analyzing what staff actually do, discrete processes can be identified. Starting with these processes, the information flows between them and between these processes and external entities are then identified and added to the diagram.

Data Flow Diagrams – Document Flow Analysis

The document flow analysis approach is appropriate if the part of the business under investigation consists principally of flows of information in the form of documents or computer input and output. Document flow analysis is particularly useful where information flows are of special interest. The first step is to list the major documents and their sources and recipients. This is followed by the identification of other major information flows such as telephone and computer transactions. Once the document flow diagram has been drawn the system boundary should be added.

Data Flow Diagrams – Top Down Expansion

The section explains the process of top down expansion, or leveling. Furthermore, it illustrates that whilst there can only be one context and one level 1 diagram for a given system, these normally give rise to numerous lower level diagrams. Each process within a given business process diagram may be the subject of further analysis. This involves identifying the lower level processes that together constitute the process as it was originally identified. This procedure is known as top-down expansion or leveling. As a business process diagram is decomposed, each process box becomes a boundary for the next, lower level, diagram.

Data Flow Diagrams – Top Down Expansion Illustrated

In order to illustrate the process of top-down expansion, consider the three processes shown within this business process diagram. No detail is shown, only the outline of the process boxes, which have been identified during the drawing of a level 1 diagram. Any area of a level 1 diagram is likely to require further analysis, as the level 1 diagram itself only provides a functional overview of the business system. Therefore, below the level 1 diagram there will be a series of lower level diagrams. These are referred to as level 2, level 3, etcetera. In practice, level 2 is usually sufficient and it is unusual to carry out an analysis beyond level 3.

In this example the process numbered 3, at level 1, will be investigated further thereby giving rise to a level 2 diagram. In the level 2 diagram four processes of interest have

been identified and the numbering of these processes must reflect the parent process. Therefore the level 2 processes are numbered 3.1, 3.2, 3.3 and 3.4

Suppose that of these four level 2 processes, one was of sufficient interest and complexity to justify further analysis. This process, let's say 3.3, could then be further analyzed resulting in a corresponding level 3 diagram. Once again the numbering of these processes must reflect the parent process. Therefore these three level 3 processes are numbered 3.3.1, 3.3.2 and 3.3.3.

Data Flow Diagrams – Numbering Rules

The process boxes on the level 1 diagram should be numbered arbitrarily, so that no priority is implied. Even where data from one process flows directly into another process, this does not necessarily mean that the first one has to finish before the second one can begin. Therefore the processes on a level 1 diagram could be re-numbered without affecting the meaning of the diagram. This is true within any business process diagram - as these diagrams do not imply time, sequence or repetition.

However, as the analysis continues beyond level 1 it is important that a strict numbering convention is followed. The processes on level 2 diagrams must indicate their parent process within the level 1 diagram. This convention should continue through level 3 diagrams, and beyond, should that level of analysis ever be required. The diagram on this screen clearly illustrates how processes on lower level diagrams identify their ancestral path.

Data Flow Diagrams - When to Stop

It is important to know when to stop the process of top-down expansion. Usually this will be at level 2 or level 3. There are 3 useful guidelines to help you to decide when to stop the analysis: Firstly, if a process has a single input data flow or a single output data flow then it should be apparent that there is little point in analyzing it any further. Secondly, when a process can be accurately described by a single active verb with a singular object, this also indicates that the analysis has been carried out to a sufficiently low level. For example, the process named validate enquiry contains a single discrete task.

Finally, ask yourself if anything useful will be gained by further analysis of a process. Would any more detail influence your decisions? If the answer is no, then there is little point in taking the analysis further.

Data Flow Diagrams – Keeping the Diagrams Clear

In this section a variety of simple techniques are introduced to show how a business process diagram can be clarified. The examples used do not relate to any specific scenario but are hypothetical abstracts used for the purpose of illustration. Firstly, where a diagram is considered to contain too many processes, those that are related can often be

combined. As a general rule no business process diagram should contain more than 12 process boxes.

In some examples multiple process boxes can be identified as being related and can be combined into a single process box with a collective description.

Exclude Minor Data Flows

Where information is being retrieved from a data store, it is not necessary to show the selection criteria, or key, that is being used to retrieve it. In the banking example, the customer details are shown being retrieved from the data store but the key used to retrieve this information is not shown. Where a data store is being updated, only the data flow representing the update needs to be shown. The fact that the information must first be retrieved does not need to be shown. Only the most important reports, enquiries, etcetera should be shown on the diagram. Communications that are of less significance can, if necessary, be detailed in support documentation.

Combining External Entities

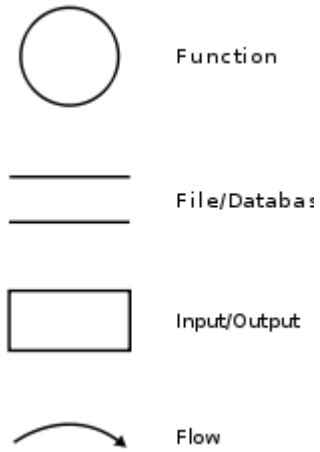
Another way to reduce the complexity of a business process diagram is to combine any related external entities. For example, a business system will often be dealing with different units from within the same external organization, and these can be combined into a single external entity. Where these units are uniquely identified a number should follow the entity identification letter. However, when they are combined the numbers placed after the identifying alphabetic character are not shown.

Combining Data Stores

In a similar way, data stores that are holding related information should be suffixed with a lower case letter. Related data stores can also be combined, and where this is the case the numbers placed after the identifying alphabetic character are not shown. In the course of developing a set of *levelled* data-flow diagrams the analyst/designers is forced to address how the system may be decomposed into component sub-systems, and to identify the transaction data in the data model. There are different notations to draw data-flow diagrams, defining different visual representations for processes, data stores, data flow, and external entities.

Developing a data-flow diagram

data-flow diagram example



data-flow diagram - Yourdon/DeMarco notation

Top-down approach

1. The system designer makes "a context level DFD" or Level 0, which shows the "interaction" (data flows) between "the system" (represented by one process) and "the system environment" (represented by terminators).
2. The system is "decomposed in lower-level DFD (Level 1)" into a set of "processes, data stores, and the data flows between these processes and data stores".
3. Each process is then decomposed into an "even-lower-level diagram containing its subprocesses".
4. This approach "then continues on the subsequent subprocesses", until a necessary and sufficient level of detail is reached which is called the primitive process (aka chewable in one bite).

DFD is also a virtually designable diagram that technically or diagrammatically describes the inflow and outflow of data or information that is provided by the external entity.

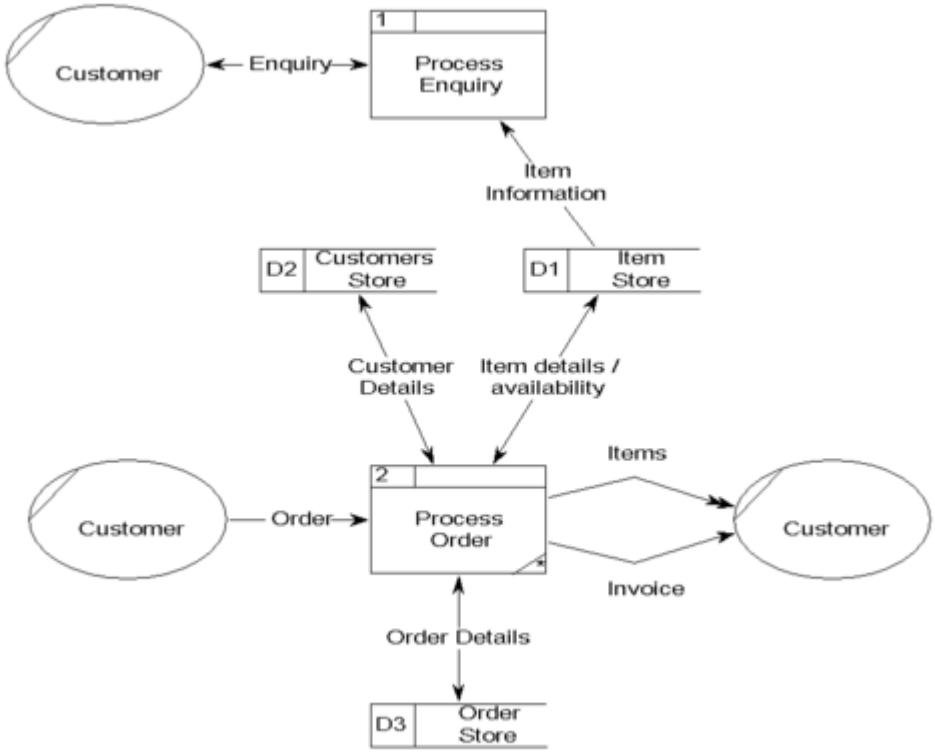
Event partitioning approach

Event partitioning was described by Edward Yourdon in *Just Enough Structured Analysis*.[4]

A context level Data flow diagram created using Select SSADM.

This level shows the overall context of the system and its operating environment and shows the whole system as just one process. It does not usually show data stores, unless they are "owned" by external systems, e.g. are accessed by but not maintained by this system, however, these are often shown as external entities.[5]

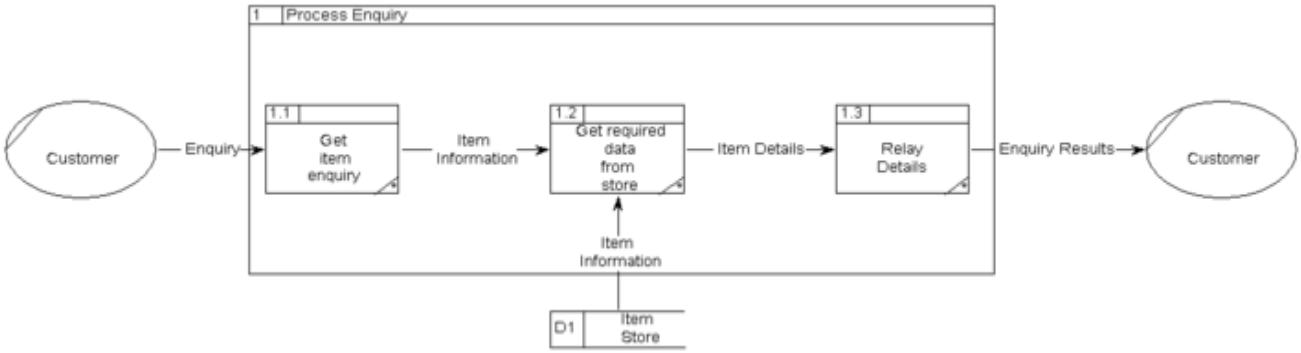
Level 1 (high level diagram)



A Level 1 Data flow diagram for the same system.

This level (level 1) shows all processes at the first level of numbering, data stores, external entities and the data flows between them. The purpose of this level is to show the major high-level processes of the system and their interrelation. A process model will have one, and only one, level-1 diagram. A level-1 diagram must be balanced with its parent context level diagram, i.e. there must be the same external entities and the same data flows, these can be broken down to more detail in the level 1, e.g. the "inquiry" data flow could be split into "inquiry request" and "inquiry results" and still be valid.[5]

Level 2 (low level diagram)



A Level 2 Data flow diagram showing the "Process Enquiry" process for the same system.

This level is a decomposition of a process shown in a level-1 diagram, as such there should be a level-2 diagram for each and every process shown in a level-1 diagram. In this example processes 1.1, 1.2 & 1.3 are all children of process 1, together they wholly and completely describe process 1, and combined must perform the full capacity of this parent process. As before, a level-2 diagram must be balanced with its parent level-1 diagram.[5]

ER Diagram

In software engineering, an entity-relationship model (ERM) is an abstract and conceptual representation of data. Entity-relationship modeling is a database modeling method, used to produce a type of conceptual schema or semantic data model of a system, often a relational database, and its requirements in a top-down fashion. Diagrams created by this process are called entity-relationship diagrams, ER diagrams, or ERDs.

The definitive reference for entity-relationship modelling is Peter Chen's 1976 paper.[1] However, variants of the idea existed previously,[2] and have been devised subsequently.

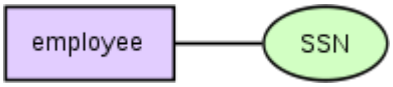
Overview

The first stage of information system design uses these models during the requirements analysis to describe information needs or the type of information that is to be stored in a database. The data modeling technique can be used to describe any ontology (i.e. an overview and classifications of used terms and their relationships) for a certain area of interest. In the case of the design of an information system that is based on a database, the conceptual data model is, at a later stage (usually called logical design), mapped to a logical data model, such as the relational model; this in turn is mapped to a physical model during physical design. Note that sometimes, both of these phases are referred to as "physical design".

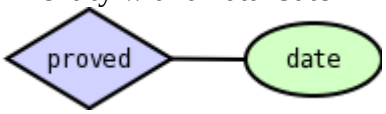
There are a number of conventions for entity-relationship diagrams (ERDs). The classical notation mainly relates to conceptual modeling. There are a range of notations employed in logical and physical database design, such as IDEF1X.

The building blocks: entities, relationships, and attributes

Two related entities



An entity with an attribute



A relationship with an attribute

Primary key

An entity may be defined as a thing which is recognized as being capable of an independent existence and which can be uniquely identified. An entity is an abstraction from the complexities of some domain. When we speak of an entity we normally speak of some aspect of the real world which can be distinguished from other aspects of the real world.[3]

An entity may be a physical object such as a house or a car, an event such as a house sale or a car service, or a concept such as a customer transaction or order. Although the term entity is the one most commonly used, following Chen we should really distinguish between an entity and an entity-type. An entity-type is a category. An entity, strictly speaking, is an instance of a given entity-type. There are usually many instances of an entity-type. Because the term entity-type is somewhat cumbersome, most people tend to use the term entity as a synonym for this term.

Entities can be thought of as nouns. Examples: a computer, an employee, a song, a mathematical theorem. Entities are represented as rectangles.

A relationship captures how two or more entities are related to one another. Relationships can be thought of as verbs, linking two or more nouns. Examples: an *owns* relationship between a company and a computer, a *supervises* relationship between an employee and a department, a *performs* relationship between an artist and a song, a *proved* relationship

between a mathematician and a theorem. Relationships are represented as diamonds, connected by lines to each of the entities in the relationship.

Entities and relationships can both have attributes. Examples: an *employee* entity might have a *Social Security Number* (SSN) attribute; the *proved* relationship may have a *date* attribute. Attributes are represented as ellipses connected to their owning entity sets by a line. Every entity (unless it is a weak entity) must have a minimal set of uniquely identifying attributes, which is called the entity's primary key.

Entity-relationship diagrams don't show single entities or single instances of relations. Rather, they show entity sets and relationship sets. Example: a particular *song* is an entity. The collection of all songs in a database is an entity set. The *eaten* relationship between a child and her lunch is a single relationship. The set of all such child-lunch relationships in a database is a relationship set. In other words, a relationship set corresponds to a relation in mathematics, while a relationship corresponds to a member of the relation.

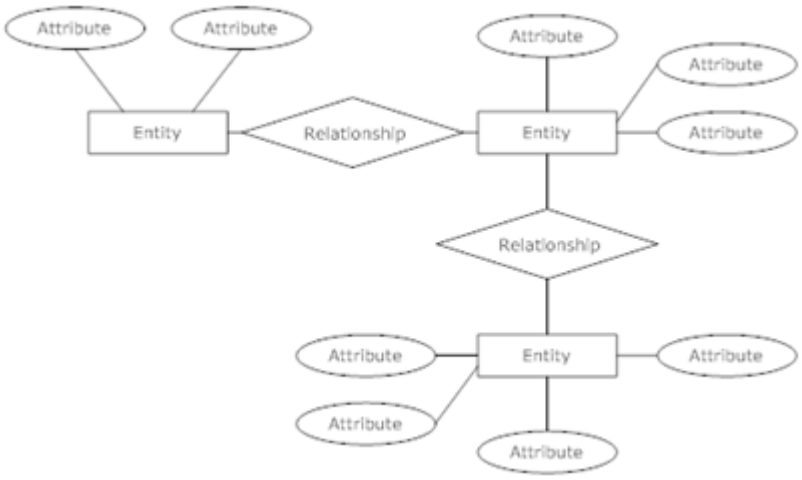
Diagramming conventions

Entity sets are drawn as rectangles, relationship sets as diamonds. If an entity set participates in a relationship set, they are connected with a line. Attributes are drawn as ovals and are connected with a line to exactly one entity or relationship set.

Its components are:

- rectangles representing entity sets.
- ellipses representing attributes.
- diamonds representing relationship sets.
- lines linking attributes to entity sets and entity sets to relationship sets.

Entity Relationship Diagrams (ERDs) illustrate the logical structure of databases.



An ER Diagram

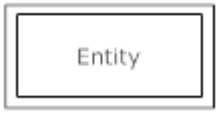
Peter Chen developed ERDs in 1976. Since then Charles Bachman and James Martin have added some sligh refinements to the basic ERD principles.

Entity

An entity is an object or concept about which you want to store information.

Weak Entity

Attributes are the properties or characteristics of an entity.



Key attribute

A key attribute is the unique, distinguishing characteristic of the entity. For example, an employee's social security number might be the employee's key attribute.

Multivalued attribute

A multivalued attribute can have more than one value. For example, an employee entity can have multiple skill values.

Derived attribute

A derived attribute is based on another attribute. For example, an employee's monthly salary is based on the employee's annual salary.

Relationships

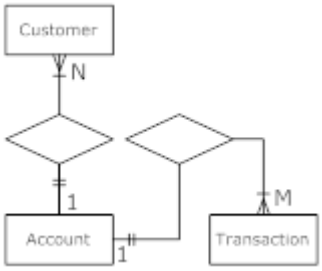
Relationships illustrate how two entities share information in the database structure.

Learn how to draw relationships: First, connect the two entities, then drop the relationship notation on the line.

Cardinality

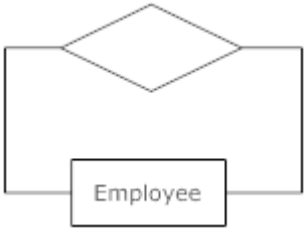
Cardinality specifies how many instances of an entity relate to one instance of another entity.

Ordinality is also closely linked to cardinality. While cardinality specifies the occurences of a relationship, ordinality describes the relationship as either mandatory or optional. In other words, cardinality specifies the maximum number of relationships and ordinality specifies the absolute minimum number of relationships.



Recursive relationship

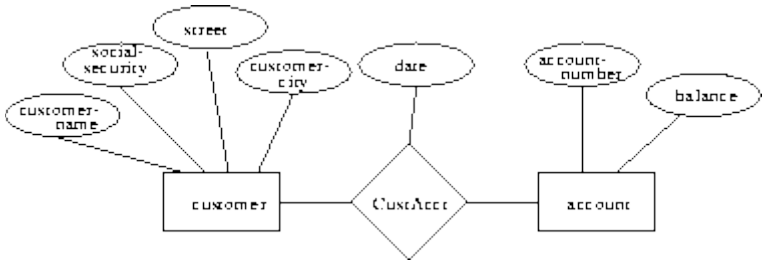
In some cases, entities can be self-linked. For example, employees can supervise other employees.



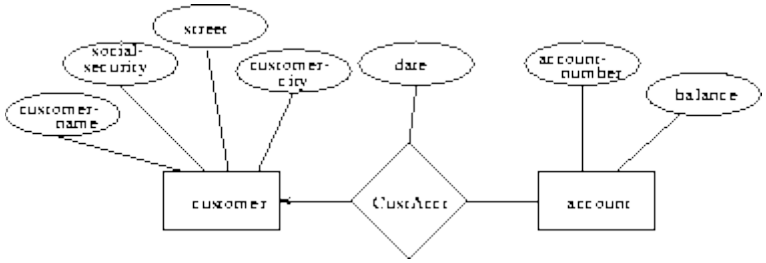
Entity Relationship Models

- Mandatory Relationships
- Optional Relationships
- Many-to-Many Relationships
- One-to-Many Relationships
- One-to-One Relationships

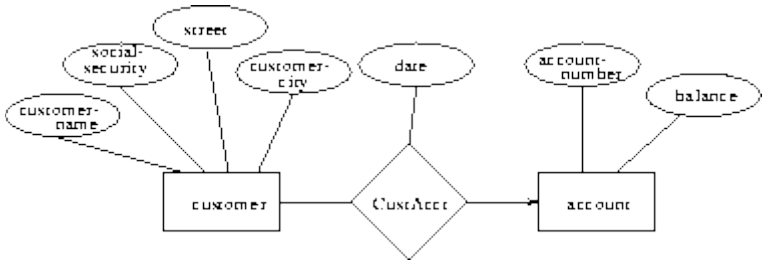
In the text, lines may be directed (have an arrow on the end) to signify mapping cardinalities for relationship sets. Following figures show some examples.



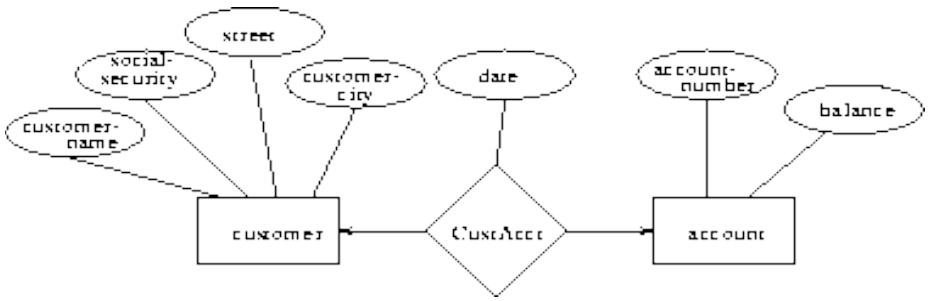
An E-R diagram



One-to-many from *customer* to *account*



Many-to-one from *customer* to *account*



One-to-one from *customer* to *account*

ER Diagram - Entities

Here, we illustrate the concept of an entity, which can be applied to almost anything that is significant to the system being studied. Some examples of information systems and their entities are listed below:

Banking system: Customer, Account, Loan.

Airline system: Aircraft, Passenger, Flight, Airport.

An entity is represented by a box containing the name of that entity.

A precise definition of ‘entity’ is not really possible, as they even vary in nature. For example, in the airline system, whilst an aircraft is a physical object (entities often are) a flight is an event and an airport is a location. However entities are nearly always those things about which data will be stored within the system under investigation. Note that entities are always named in the singular; for example: customer, account and loan, and not customers, accounts and loans. This course uses symbols that are standard in the IT industry. This uses the soft-box symbol shown to represent an entity. If a site uses a different symbol set, this is not a problem, as entity relationship diagramming techniques are the same regardless of the symbols being used.

ER Diagram - Entity Types & Occurrence

Similar entity occurrences are grouped together and collectively termed an entity type. It is entity types that are identified and drawn on the data model. An entity occurrence identifies a specific resource, event, location, notion or (more typically) physical object. In this course the term 'entity' is, by default, referring to entity type. The term entity occurrence will be specifically used where that is relevant. Each entity has a data group associated with it. The elements of the data group are referred to as the 'attributes ' of the entity. The distinction between what is an attribute of an entity and what is an entity in its own right is often unclear. This is illustrated shortly.

ER Diagram - Entity Naming

Entity names are normally single words and the name chosen should be one familiar to the users. The entity name can include a qualifier in order to clarify their meaning. However, if different names are currently used to describe a given entity in different areas of the organization then a new one should be chosen that is original, unique and meaningful to all of the users.

For example, the terms 'signed contract', 'sale' and 'agreement' might be recreated as the entity 'completed'. Conversely an organization may be using a 'catch all' term to describe what the analyst identifies as being a number of separate entities. For example the term 'invoice' may be being used to describe 3 invoice types - each of which is, in fact,

processed in a different manner. In this case prefixing the entity names with qualifiers, is likely to be the best solution.

ER Diagram - Entity Identification

The process of identifying entities is one of the most important steps in developing a data model. It is common practice for an experienced analyst to adopt an intuitive approach to entity identification, in order to produce a shortlist of potential entities. The viability of each of these potential entities can then be considered using a set of entity identification guidelines. This should result in some of the potential entities being confirmed as entities, whilst others will be rejected. In this exercise you will be asked to identify a set of potential entities within a simple business scenario. This should help you to understand and appreciate the entity identification guidelines better. Read the following case study. Study this information carefully and see if you can identify the entities - remember that entities are those things about which data will be stored.Make your own list of those things that you think are likely to be entities, before moving to the next screen.

ER Diagram - Entity Identification Case Study

City Cameras is an independent retailer of cameras, video-cameras and accessories. The owner fulfils the roles of shopkeeper and manager and he purchases a variety of products from a number of different suppliers. The owner can check on different suppliers wholesale and recommended retail prices with reference to their price lists, as shown. During a normal day several customers will enter the shop and a number of them will buy one or more of the products on sale. At some stage the owner may decide that one or more product lines need to be re-ordered, following a visual stock-take. He will then consult the latest suppliers price lists to see who is offering the best deals on given product lines.

Following this, he will ring one or more of the suppliers to order some of their products. At the same time he will also make a written record of the orders that have been placed with each supplier on a separate sheet of paper. These records are then used to verify incoming orders and invoicing details.

ER Diagram - Entity Identification – Exercise#1

With reference to the case study information, make a list of all of those things mentioned in the case study that could be entities - that is the potential entities. Your list should look something like that shown below:

Suppliers Price List, Customer, Product, Order, Invoicing Details & Supplier

There are six potential entities listed. From this initial list we will consider the 'suppliers price list' to be a likely attribute of the entity 'supplier'. Therefore we shall consider this

within the context of the supplier entity. The 'invoicing details' are stated to be attributes of the 'order record' entity, so we shall also discount this as a potential entity at this stage. Remember that entities are described in the singular as they relate to entity types. 'Customer' for example represents the entity type 'customer' which encompasses an infinite number of 'customer' entity occurrences. Taking these four as our list of potential entities, each will be discussed in turn:

ER Diagram - Entity Identification – Exercise#2

In many business systems, information about the customer is of great importance. An insurance company or bank, for example, could not function without a customer database on which comprehensive personal details are stored. This customer database also serves as an essential resource for selling new financial products and services. But how much customer information is likely to be stored by City Cameras? Are they even going to record the name & address of their customers? Interviews with the owner reveal the answer to be that he has no real interest in storing 'information' about his customers. He only records their details onto any necessary warranty documents and then sends these off to the appropriate supplier. Therefore, in the context of this system customer is NOT an entity.

ER Diagram - Entity Identification – Exercise#3

It is a natural assumption that all retail businesses would hold a significant amount of product information. However in this study the only level of product information is that which is held on the suppliers' price lists. Lets look again at the suppliers price list in the case study. This confirms that product information is held within this system and it is apparent from the case study that products are of real interest. So have we identified an entity? At this stage it would be likely that product would be considered to be an entity. However, you will shortly see why the analysis phase needs to be iterative - enabling decisions to be altered later, if necessary.

ER Diagram - Entity Identification – Exercise#4

Once again a natural assumption would be that a retail business would store substantial information about its' suppliers. On requesting to see information about City Cameras' suppliers, the owner once again reaches for the suppliers' price lists. Lets look again at the suppliers' price list in the case study. Each of these lists has the name, address and telephone number of the supplier on the first page. The suppliers' price list is the only place where City Cameras stores information about suppliers.

Whilst the early investigation indicated that 'product' was probably an entity, it now becomes apparent that the unique identification of a product and access to the product information is also only possible after locating the relevant suppliers price list. It has now been established that all of the information that is stored in relation to the two potential

entities 'product' and 'supplier' are held in the same place - the suppliers' price list. This means that the suppliers' price list is an entity and that both product and supplier represent information held within this entity. Both supplier and product are therefore identified as being attributes of the entity 'supplier's price list'.

ER Diagram - Entity Identification – Exercise#5

What about the potential entity: 'Order'. Investigation reveals that the re-ordering process consists of visual stocktaking on an ad-hoc basis, followed by mental recall of those suppliers that stock the identified products. The appropriate suppliers price lists are then referred to for the up-to-date pricing information and contact details and the order is placed over the telephone. The owner keeps a written record of the orders he places, each order on a separate sheet of paper, and these are then filed. Let's look again at the record of an order, as shown in the case study. This written order record is used to check against incoming products, to verify invoicing details and to chase orders that may be overdue. The 'order' is held as stored information and therefore 'order' does represent an entity.

ER Diagram - Entity Identification – Exercise#6

Having started with six potential entities (suppliers price list, customer, product, order, invoicing details and supplier), the analysis has identified that only two of these are in fact entities. We eliminated customer, as no customer information is recorded or stored within this retail outlet. The stored information relating to both a product and a supplier was found to only exist within the suppliers' price list. Therefore Suppliers' Price List was identified as being the only entity amongst these three. Order was confirmed as a system entity and the invoicing details were identified early on as being an attribute of this entity.

Even in this simple scenario it should be apparent that entity identification needs careful consideration. Interestingly, both of the entities that were identified existed as documents within the system. Entities are often synonymous with discrete information stores within a system - whether physical or electronic. The precise definition of what is an entity and what is an attribute will not always be clear. Therefore the process of entity identification should be iterative, enabling the review of decisions made earlier. Remember, entity types are always named in the singular and this name then represents all of the occurrences of that entity type.

ER Diagram - Entity Identification Guidelines

There are a variety of methods that can be employed when trying to identify system entities. There follows a series of entity identification guidelines, which should prove helpful to the inexperienced analyst:

An informal questioning approach can be adopted, in which the analyst asks targeted questions to determine what information is necessary and whether or not that information is recorded within the system. During face to face discussions with users the nouns (or given names of objects) should be recorded - as these often indicate those things that are

entities within a system. The existing documentation often contains clues as to the information that needs to be held and once again the nouns in the text may indicate potential entities.

Every fact that is required to support the business is almost certainly an attribute (or data item). In turn each of these attributes will belong to an entity. If no 'parent' entity can be found for one or more of these low level facts, then this indicates that your entity search is incomplete. However, don't get hung up on the initial analysis. Entity identification can continue once the drawing of the data model diagram has begun. As this diagram is developed and refined further entities may become apparent.

ER Diagram - Attributes

Many different occurrences of a given entity type can usually be identified. In the gift shop example both of the entities 'order' and 'suppliers price list' had numerous occurrences. Each entity type can always be described in terms of attributes, and these attributes will apply to all occurrences of that given entity type. In the camera shop example, all occurrences of the entity 'supplier' could be described by an identifiable set of attributes, including:

The Supplier Name, the Supplier Address, Telephone Number, etcetera.

A given attribute belonging to a given entity occurrence can only have one value. Therefore, if a supplier could have more than one address or telephone number then this should be determined before defining the attributes of that entity type. In this example the defined entity may require two or three address and/or telephone number attributes. It is the maximum practical instances of a given attribute that should be catered for in the entity type definition.

ER Diagram - Entity Keys

An entity is defined by its attributes. Furthermore, each entity occurrence can be uniquely identified, by using an attribute or a combination of attributes as a key. The primary key is the attribute (or group of attributes) that serve to uniquely identify each entity occurrence. Consider the problem that might arise if the name and address of an individual were used as the primary key for identifying the patients within a hospital. Take the example of a patient called David Smith living at 23 Acacia Avenue. He has a son also called David Smith living at the same address.

Name and Address would not necessarily provide a unique identifier and confusion could easily arise, potentially creating a mix up with the patient records. For this reason, in a hospital system patients each have a Patient Number as their primary key. If two or more data items are used as the unique identifier, then this represents a compound key. For example, a compound key used to identify a book could be 'Title' together with 'Author'. There may be occasions of authors using a previously used title but not of an author using the same title for more than one of their own books. Where several possible primary keys

exist they are called candidate keys. For example, a book could be identified, either by 'Title' together with 'Author' or by the widely used unique identifier for books - the ISBN number. Where an attribute of one entity is a candidate key for another entity, it is termed a foreign key.

For example, the attribute 'Author' belonging to the entity Book is a foreign key within the entity Author. You may be able to think of some shortcomings to the use of this attribute as the primary key, for example two authors having the same name. It is worth noting that entity relationships are often indicated by the presence of foreign keys.

ER Diagram - Relationships

The relationship is the association between two entities to which all of the occurrences of those entities must conform. The diagram shown represents the beginnings of a data model where the relationship between a manager and a department needs to be defined. The entities on data models are linked by relationship lines and together these are the only two components that make up a data model diagram. A relationship is an association between two entities to which all of the occurrences of those entities must conform.

Every relationship line shows two reciprocal relationships:

That of the first entity with respect to the second and that of the second entity with respect to the first. In this example a manager is responsible for a department and a department is the responsibility of a manager. Each relationship line has three distinct properties: Firstly the relationship link phrase, secondly the degree or cardinality of the relationship and thirdly the participation or optionality of the relationship. These three properties combine to form the relationship statement.

ER Diagram - Relationship Link Phrase

The first property of the relationship statement is the relationship link phrase. This should be a short description of the nature of the relationship, typically between three and five words long. It is always read clockwise with respect to the entities that it links, so in this example: 'Manager is responsible for department', and 'Department is responsibility of manager'. If the same relationship were to be drawn with department on the left hand side then the positions of the link phrases would have to be reversed.

ER Diagram - Relationship Cardinality

The second property of the relationship statement is the degree, or maximum cardinality, of the relationship. If an entity has a crowsfoot symbol drawn against it, then many occurrences of that entity may relate to the other entity. Conversely if no crowsfoot is drawn against it, at most one occurrence of that entity may relate to the other entity. In this example: Each company employs one or more employees, but Each employee is employed by only one company. This is called a one-to-many relationship. Maximum cardinalities may be combined to give another two relationship types, In this example:

Each manager is responsible for only one department and each department is the responsibility of only one manager. This is called a one-to-one relationship.

And in this example: Each lecturer teaches one or more courses and each course is taught by one or more lecturers. This is called a many-to-many relationship. To recap, three different relationship types have been illustrated, one-to-many, one-to-one and many-to-many.

ER Diagram - Relationship Participation

The third and final property of the relationship statement is the participation or optionality. A solid line shows that an entity occurrence must be associated with each occurrence of the other entity. In this example: Each passenger must possess a ticket, and Each ticket must belong to a passenger. A dotted line shows that an entity occurrence may be associated with each occurrence of the other entity, In this example:Each book may be borrowed by a borrower, and Each borrower may borrow one or more books. Furthermore, these symbols can be combined. In this example: Each book may be recalled by a reservation, but Each reservation must be recalling a book.

Remember, there are only two components to a data model diagram, entities and relationships. A relationship is an association between two entities to which all of the occurrences of those two entities must conform. There are three distinct properties of the relationship; firstly the relationship link phrase, secondly the degree or cardinality of the relationship and thirdly the participation or optionality of the relationship. These three properties are collectively termed the relationship statement.

ER Diagram - Identifying Relationships

There are just two questions that need to be asked, in order to establish the degree of the relationship that exists between any two entities. In order to identify the degree of the relationship between the entities X and Y the following two questions need to be asked.

Question 1

Can an occurrence of X to be associated with more than one occurrence of Y?

Question 2

Can an occurrence of Y to be associated with more than one occurrence of X?

Each of these questions can be answered 'Yes' or 'No' and both questions must be answered. This means that there are four possible outcomes as shown in the table.The nature of the relationship associated with each outcome is as follows:

Option 1, Question1 equals Yes, Question2 equals No.

In this case a one-to-many relationship has been identified, represented by the relationship line shown.

Option 2, Question1 equals No, Question2 equals Yes

As in the first case a one-to-many relationship has been identified, represented by the relationship line shown.

Option 3, Question1 equals Yes, Question2 equals Yes

In this case a many-to-many relationship has been identified.

Many-to-many relationships may be shown in the early 'preliminary' data model in order to aid the clarity of these early diagrams. However, such relationships are invalid and are therefore always re-modeled using 'link entities' in later diagrams. This process is explained later in the course.

Option 4, Question1 equals No, Question2 equals No

In this case a one-to-one link has been identified. Legitimate one-to-one relationships are rare and it is likely that this relationship is one that needs to be rationalized. The methods used to investigate one-to-one relationships and to re-model them where necessary are explained later in the course.

In a one-to-many relationship the entity at the 'one' end is normally referred to as the master, and the entity at the 'many' end referred to as the detail entity. Some analysts adopt the 'no dead crows' rule and avoid drawing crowsfeet pointing upwards. This ensures that detail entities are shown below the master entities to which they belong. This makes the diagram clearer, although congestion may make this rule difficult to enforce throughout the data model.

ER Diagram - Relationship Statements

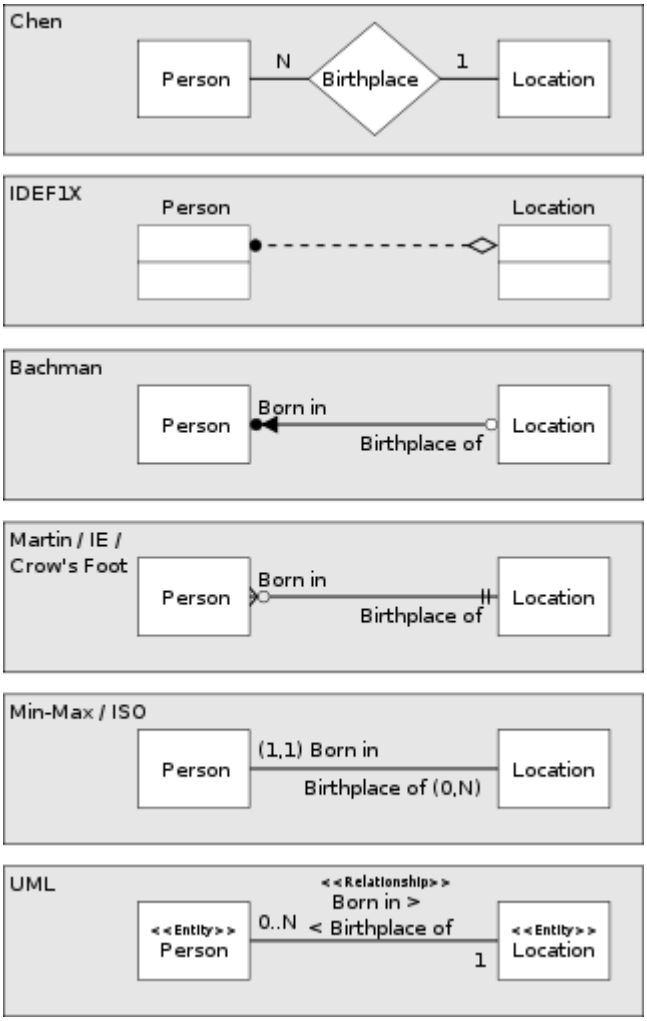
The relationship statement is a formal description that encompasses the three properties of the relationship. The relationship statement encompasses the three properties of the relationship. The first property is the relationship link phrase, the second the degree or cardinality of the relationship and the third the participation or optionality of the relationship.

Cardinality constraints are expressed as follows:

- a double line indicates a *participation constraint*, totality or surjectivity: all entities in the entity set must participate in *at least one* relationship in the relationship set;

- an arrow from entity set to relationship set indicates a key constraint, i.e. injectivity: each entity of the entity set can participate in *at most one* relationship in the relationship set;
- a thick line indicates both, i.e. bijectivity: each entity in the entity set is involved in *exactly one* relationship.
- an underlined name of an attribute indicates that it is a key: two different entities or relationships with this attribute always have different values for this attribute.

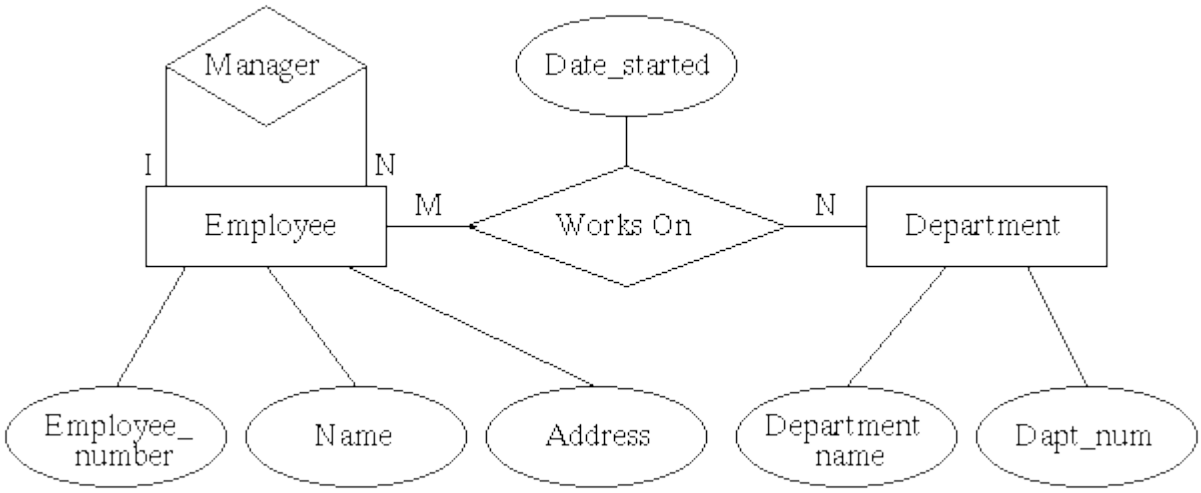
Attributes are often omitted as they can clutter up a diagram; related diagram techniques often list entity attributes within the rectangles drawn for entity sets.



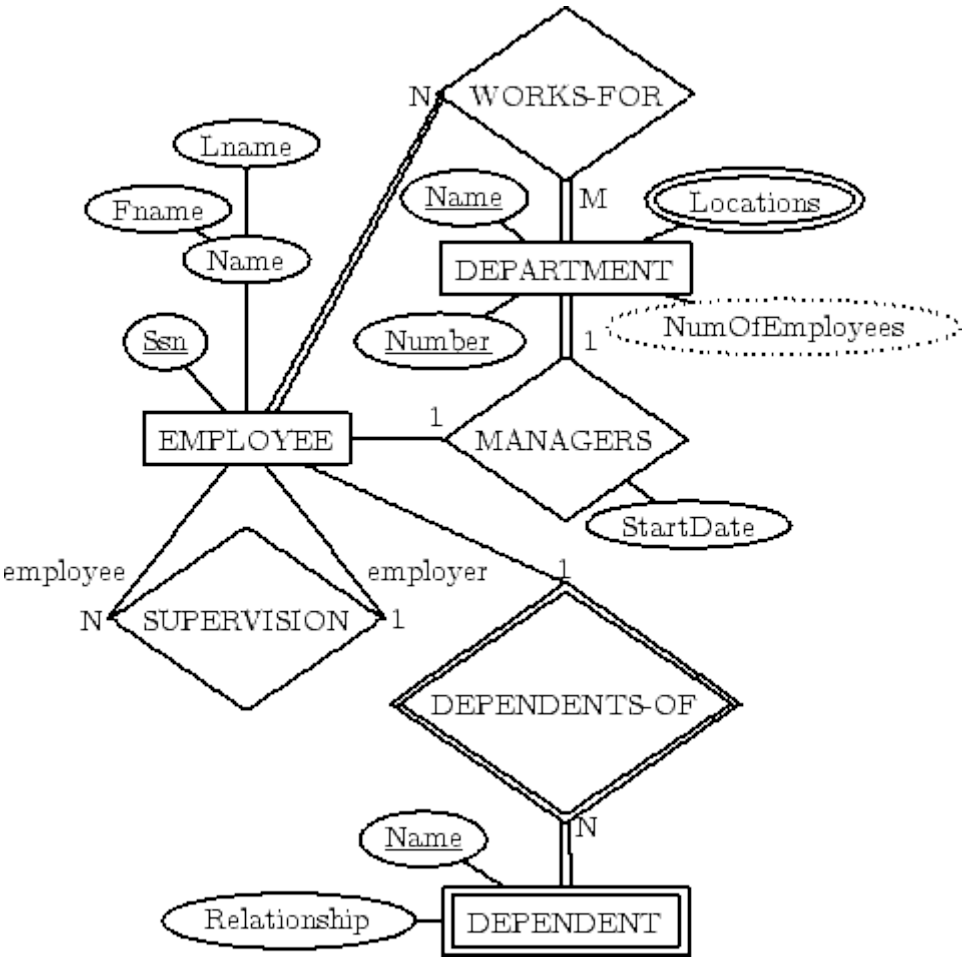
Without understanding the relationship between an employee with his organization we cannot build the payroll system. The below E-R Diagram illustrates the relationship between an employee and an organization, only then we would be able to design the

process that needs to be computerized to build the system. The diagram documents the entities and relationships involved in the employee information and payroll system. It depicts the fundamental relations like recording personnel information, paying salary and getting a loan.

The E-R Diagram for a Employee Payroll system can be simple as well as complex It can be as simple as below:



Or it can be more complex then above:



Software Requirement Specification

A **Software Requirements Specification** (**SRS**) is a complete description of the behavior of the system to be developed. It includes a set of use cases that describe all the interactions the users will have with the software. Use cases are also known as functional requirements. In addition to use cases, the SRS also contains non-functional (or supplementary) requirements. Non-functional requirements are requirements which impose constraints on the design or implementation (such as performance engineering requirements, quality standards, or design constraints).

Requirements analysis in systems engineering and software engineering, encompasses those tasks that go into determining the needs or conditions to meet for a new or altered product, taking account of the possibly conflicting requirements of the various stakeholders, such as beneficiaries or users.

Requirements analysis is critical to the success of a development project. Requirements must be actionable, measurable, testable, related to identified business needs or

opportunities, and defined to a level of detail sufficient for system design. Requirements can be functional and non-functional.

Conceptually, requirements analysis includes three types of activity:

- Eliciting requirements: the task of communicating with customers and users to determine what their requirements are. This is sometimes also called requirements gathering.
- Analyzing requirements: determining whether the stated requirements are unclear, incomplete, ambiguous, or contradictory, and then resolving these issues.
- Recording requirements: Requirements might be documented in various forms, such as natural-language documents, use cases, user stories, or process specifications.

Requirements analysis can be a long and arduous process during which many delicate psychological skills are involved. New systems change the environment and relationships between people, so it is important to identify all the stakeholders, take into account all their needs and ensure they understand the implications of the new systems. Analysts can employ several techniques to elicit the requirements from the customer. Historically, this has included such things as holding interviews, or holding focus groups (more aptly named in this context as requirements workshops) and creating requirements lists. More modern techniques include prototyping, and use cases. Where necessary, the analyst will employ a combination of these methods to establish the exact requirements of the stakeholders, so that a system that meets the business needs is produced.In other words it helps the software engineers to identify the exact problem behind the development of the software and to find the exact requirements for developing the software or the system.

- it helps us to understand , interpret, classify and organise the software requirements in order to assess the completeness,correctness, and consistency of the software. = =

General Outline of a SRS

Software Requirements Specifications (SRS)

Cover Page

Revisions Page

Table of Contents

1 INTRODUCTION

1.1 Product Overview
1.2 Purpose
1.3 Scope
1.4 Reference

Example of SRS for UMS (University Management System):

1. GENERAL DESCRIPTION – UMS is University Management System for managing the records of the alumni's of the university as well as staff, faculty and higher authorities.

1.1 Purpose – The purpose for developing this type of software or introducing this UMS is to facilitate everyone who is concerned with the university.

1.2 Scope – The scope of UMS is global i.e. it should be able to be accessed from anywhere through internet i.e. registered users must be able to login to their accounts by

directly accessing the university's website and then signing in with their username and password anytime and anywhere.

1.3 Abbreviation – UMS University Management System

1.4 Overview – As the ums is able to have a user interface. It should have a drop down boxes and if we drag mouse on any control at our welcome screen information regarding that the control should be displayed. Help menu should be there. As a teacher it should provide them to upload the various assignments and the attendance of the students. As a developer it should make a user interface which is user friendly. He should make the UMS as simple as he can. Backup at the main server should be made.

2. OVERALL DESCRIPTION

2.1 Product Perspective – product i.e. UMS should be able to provide a basic and easy interchange of information i.e. it should be able to remove the communication gaps between a teacher and the student. It should have chat facilities for all the users that are online. It should be compatible with all the operating systems.

2.2 Product Functions - The following are the product functions of the UMS:

- The UMS login box should on the official website of the university.
- The password field should be secured.
- After signing in all updates and new announcements for users should be displayed.
- By clicking on the dropdown box of the options the user should be able to view progress reports, assignments, notes, attendance, placement services and results.
- User should be able to change the passwords.
- Web pages should support pdf, ppt, doc and similar supported formats so that they can be easily downloadable and unloadable.

2.3 User Characteristics – A user can only have his/her registration number as username so if he joins the university then only he can then only he can login. This prevents misuse, unauthorized access and hacking if the product.

2.4 General Constraints – Server capacity is how many users can access or can be online at once. More is the number of users more will be the network traffic and hence the server comes in a down state. Personal firewall and updating is a tough task, it should be such that it should not block the network traffic, making the system slower. Firewall of the UMS should not collide with the firewall of the user system.

2.5 Assumptions and Dependencies – UMS should work even at when the network traffic is high. Server should have a power backup as well as a database backup. The UMS should be compatible with most of the operating systems i.e. previous and latest ones.

3. SPECIFIC REQUIREMENTS

3.1 External Interface Required

3.1.1 User Interfaces – The external users are the students and the teachers of the university. The students can have an access to their accounts for their attendance, assignments etc. The teachers have also an account to access their account for uploading of the students' attendance and the assignments to be submitted by them.

3.1.2 Hardware Interfaces – The external hardware interface used for accessing the UMS is the personal computers of the teachers and the students. The PCs may be laptops with wireless LAN as the internet connections provided will be wireless.

3.1.3 Software Interfaces – The Operating Systems can be any version of Windows, Linux, Unix or Mac which supports TCP/IP protocols.

3.1.4 Communication Interfaces – The communication interface is a local area network through wireless network routers.

3.2 Performance Requirements – The PCs used must be at least Pentium 4 machines so that they can give optimum performance of the product.

3.3 Design Constraints – The constraints at the designing time are that the needs of the university students and the teachers may keep on changing so the designers must keep this in view and design the product in this way that it is easily updatable.

3.4 Attributes – The following are the attributes of the product UMS:

- It should be equipped with current and archive database.
- All records can easily be updated.
- It should have its personal firewall.
- It should facilitate student with updating his/her account, downloading or uploading of assignments from anywhere.
- It should also do the same for teachers they can also have their pay checks online i.e. UMS should be capable of online transaction.

3.5 Other Requirements – The software is such that as the time goes by the need of the university management, students and teachers may keep on changing thus it is made to change from time to time.

UML

Unified Modeling Language (**UML**) is a standardized general-purpose modeling language in the field of software engineering. The standard is managed, and was created by, the Object Management Group.

UML includes a set of graphical notation techniques to create visual models of software-intensive systems.

The Unified Modeling Language (UML) is used to specify, visualize, modify, construct and document the artifacts of an object-oriented software intensive system under development.[1] UML offers a standard way to visualize a system's architectural blueprints, including elements such as:

- actors
- business processes
- (logical) components
- activities
- programming language statements
- database schemas, and
- reusable software components.[2]

UML combines best techniques from data modeling (entity relationship diagrams), business modeling (work flows), object modeling, and component modeling. It can be used with all processes, throughout the software development life cycle, and across different implementation technologies.[3] UML has synthesized the notations of the Booch method, the Object-modeling technique (OMT) and Object-oriented software engineering (OOSE) by fusing them into a single, common and widely usable modeling language. UML aims to be a standard modeling language which can model concurrent and distributed systems. UML is a de facto industry standard, and is evolving under the auspices of the Object Management Group (OMG). OMG initially called for information on object-oriented methodologies that might create a rigorous software modeling language. Many industry leaders have responded in earnest to help create the UML standard.[1]

UML models may be automatically transformed to other representations (e.g. Java) by means of QVT-like transformation languages, supported by the OMG. UML is extensible, offering the following mechanisms for customization: profiles and stereotype. The semantics of *extension by profiles* have been improved with the UML 2.0 major revision.

The important point to note here is that UML is a 'language' for specifying and not a method or procedure. The UML is used to define a software system; to detail the artifacts in the system, to document and construct - it is the language that the blueprint is written in. The UML may be used in a variety of ways to support a software development

methodology (such as the Rational Unified Process) - but in itself it does not specify that methodology or process.

UML defines the notation and semantics for the following domains:

- The User Interaction or Use Case Model - describes the boundary and interaction between the system and users. Corresponds in some respects to a requirements model.

- The Interaction or Communication Model - describes how objects in the system will interact with each other to get work done.

- The State or Dynamic Model - State charts describe the states or conditions that classes assume over time. Activity graphs describe the workflows the system will implement.

- The Logical or Class Model - describes the classes and objects that will make up the system.

- The Physical Component Model - describes the software (and sometimes hardware components) that make up the system.

- The Physical Deployment Model - describes the physical architecture and the deployment of components on that hardware architecture.

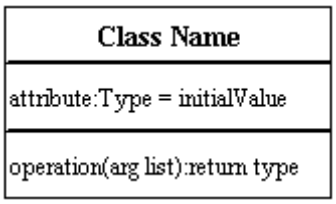
Class Diagrams

Class diagrams are the backbone of almost every object-oriented method including UML. They describe the static structure of a system. The purpose of a class diagram is to depict the classes within a model. In an object oriented application, classes have attributes (member variables), operations (member functions) and relation- ships with other classes. The UML class diagram can depict all these things quite easily. The fundamental element of the class diagram is an icon the represents a class. A class icon is simply a rectangle divided into three compartments. The topmost compartment contains the name of the class. The middle compartment contains a list of attributes (member variables), and the bottom compartment contains a list of operations (member functions).

In many diagrams, the bottom two compartments are omitted. Even when they are present, they typically do not show every attribute and operations. The goal is to show only those attributes and operations that are useful for the particular diagram. This ability to abbreviate an icon is one of the hallmarks of UML. Each diagram has a particular purpose. That purpose may be to highlight on particular part of the system, or it may be to illuminate the system in general. The class icons in such diagrams are abbreviated as necessary. There is typically never a need to show every attribute and operation of a class on any diagram.

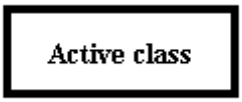
Basic Class Diagrams symbols and notations

Classes represent an abstraction of entities with common characteristics. Associations represent the relationships between classes.Illustrate classes with rectangles divided into compartments. Place the name of the class in the first partition (centered, bolded, and capitalized), list the attributes in the second partition, and write operations into the third.



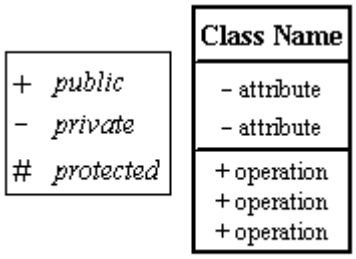
Active Class

Active classes initiate and control the flow of activity, while passive classes store data and serve other classes. Illustrate active classes with a thicker border.



Visibility

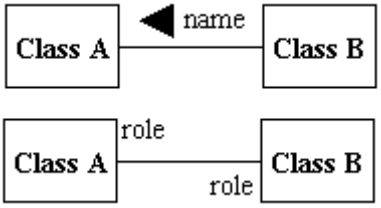
Use visibility markers to signify who can access the information contained within a class. Private visibility hides information from anything outside the class partition. Public visibility allows all other classes to view the marked information. Protected visibility allows child classes to access information they inherited from a parent class.



Associations

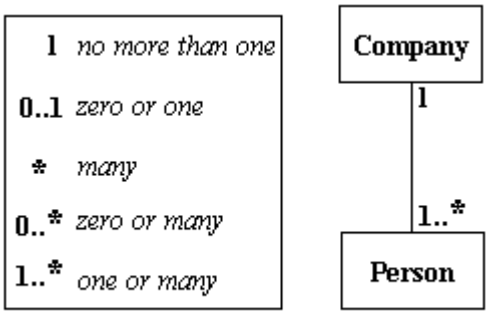
Associations represent static relationships between classes. Place association names above, on, or below the association line. Use a filled arrow to indicate the direction of the relationship. Place roles near the end of an association. Roles represent the way the two

classes see each other.
Note: It's uncommon to name both the association and the class roles.



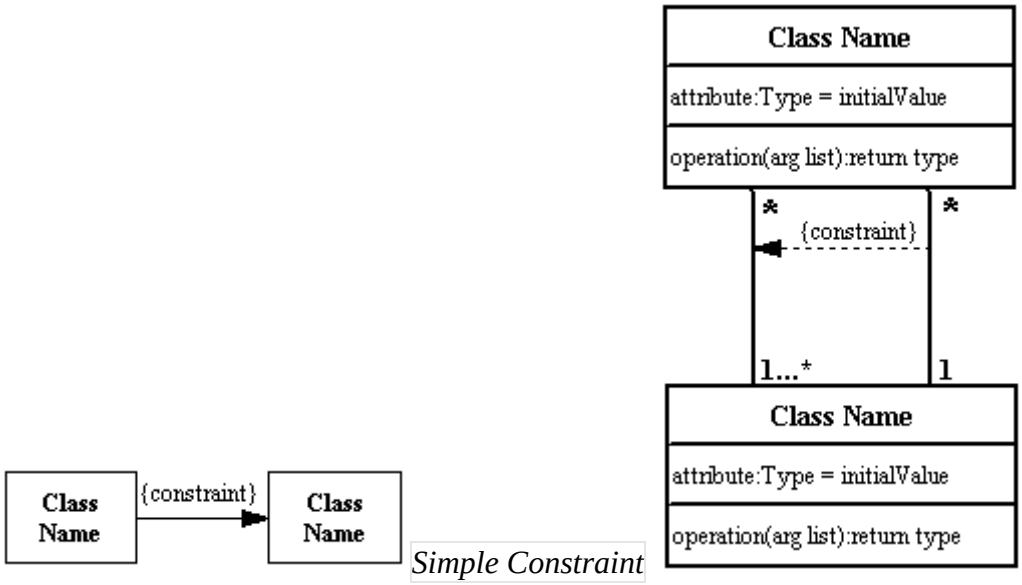
Multiplicity (Cardinality)

Place multiplicity notations near the ends of an association. These symbols indicate the number of instances of one class linked to one instance of the other class. For example, one company will have one or more employees, but each employee works for one company only.



Constraint

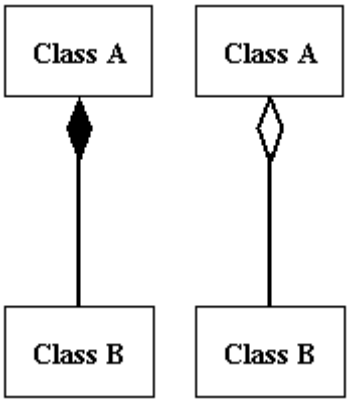
Place constraints inside curly braces {}.



Simple Constraint

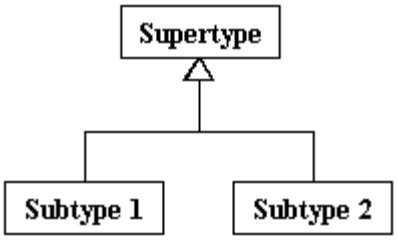
Composition and Aggregation

Composition is a special type of aggregation that denotes a strong ownership between Class A, the whole, and Class B, its part. Illustrate composition with a filled diamond. Use a hollow diamond to represent a simple aggregation relationship, in which the "whole" class plays a more important role than the "part" class, but the two classes are not dependent on each other. The diamond end in both a composition and aggregation relationship points toward the "whole" class or the aggregate.



Generalization

Generalization is another name for inheritance or an "is a" relationship. It refers to a relationship between two classes where one class is a specialized version of another. For example, Honda is a type of car. So the class Honda would have a generalization relationship with the class car.



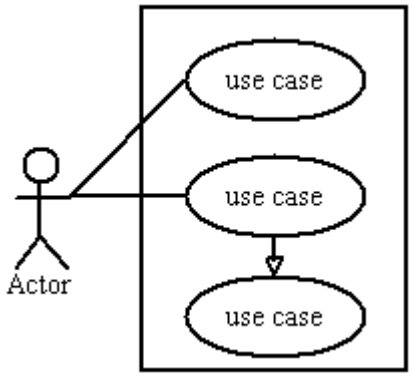
In real life coding examples, the difference between inheritance and aggregation can be confusing. If you have an aggregation relationship, the aggregate (the whole) can access only the PUBLIC functions of the part class. On the other hand, inheritance allows the inheriting class to access both the PUBLIC and PROTECTED functions of the superclass.

Use case diagrams model the functionality of a system using actors and use cases. Use cases are services or functions provided by the system to its users.

Basic Use Case Diagram Symbols and Notations

System

Draw your system's boundaries using a rectangle that contains use cases. Place actors outside the system's boundaries.

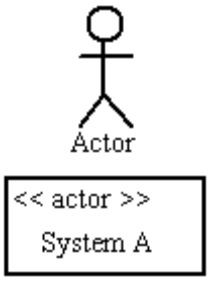


Use Case

Draw use cases using ovals. Label with ovals with verbs that represent the system's functions.

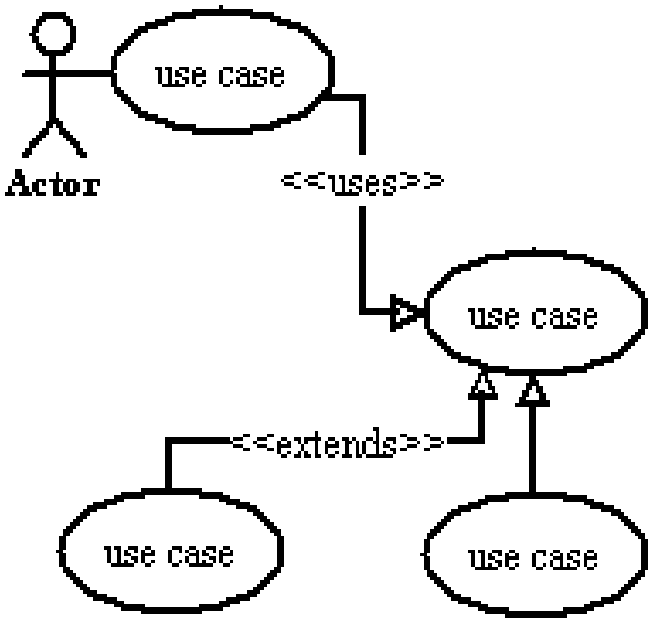
Actors

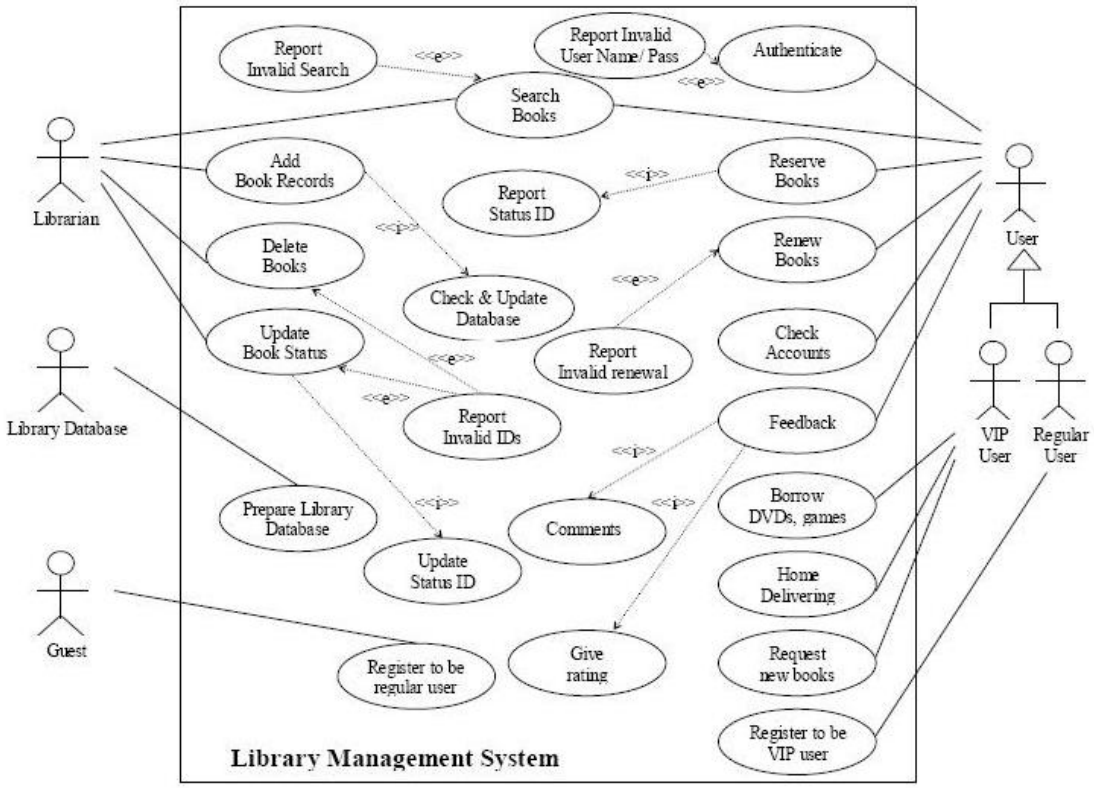
Actors are the users of a system. When one system is the actor of another system, label the actor system with the actor stereotype.



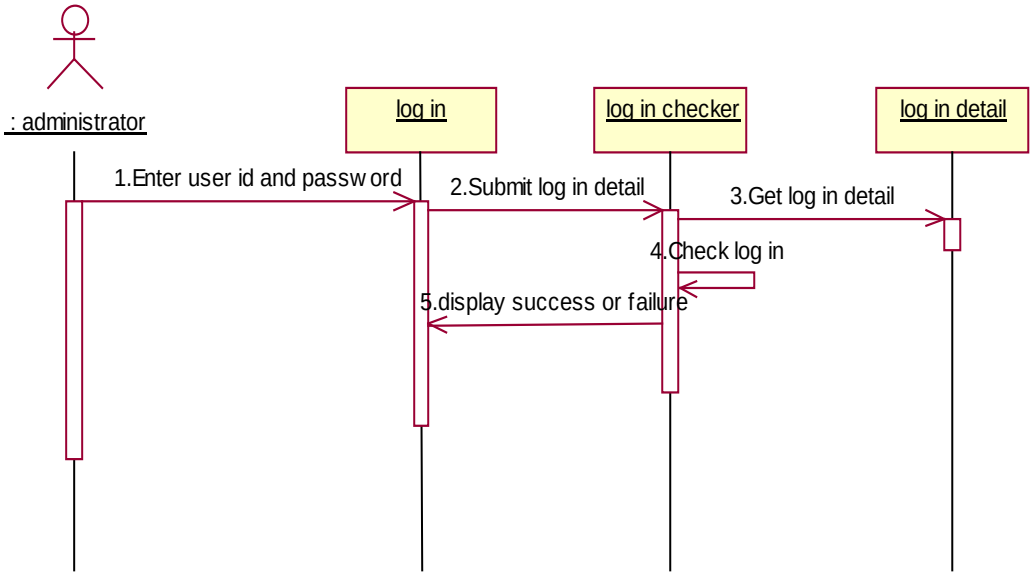
Relationships

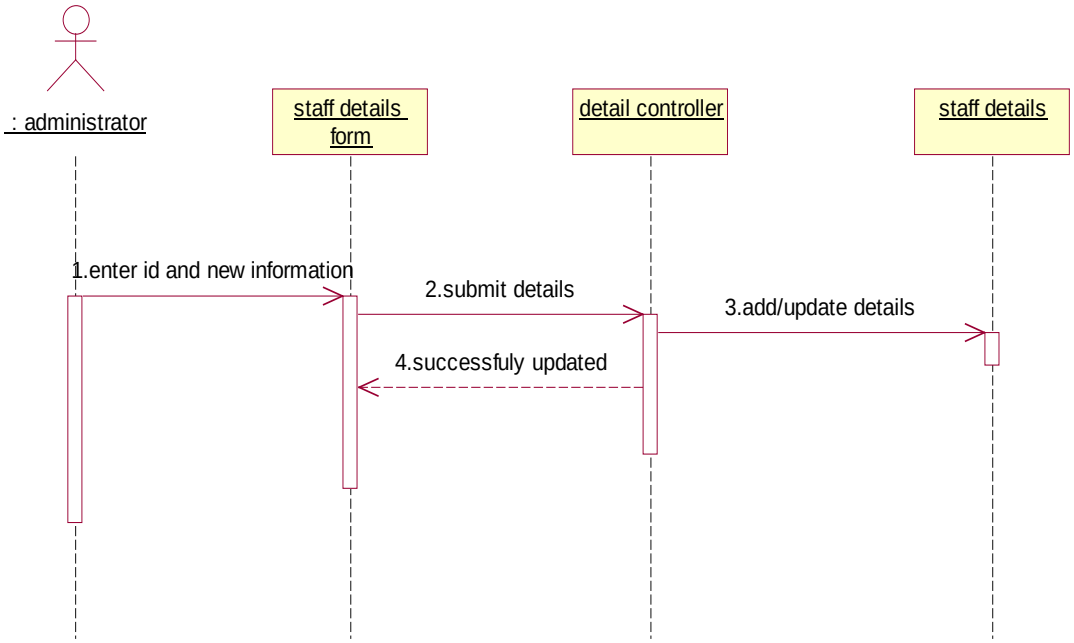
Illustrate relationships between an actor and a use case with a simple line. For relationships among use cases, use arrows labeled either "uses" or "extends." A "uses" relationship indicates that one use case is needed by another in order to perform a task. An "extends" relationship indicates alternative options under a certain use case.

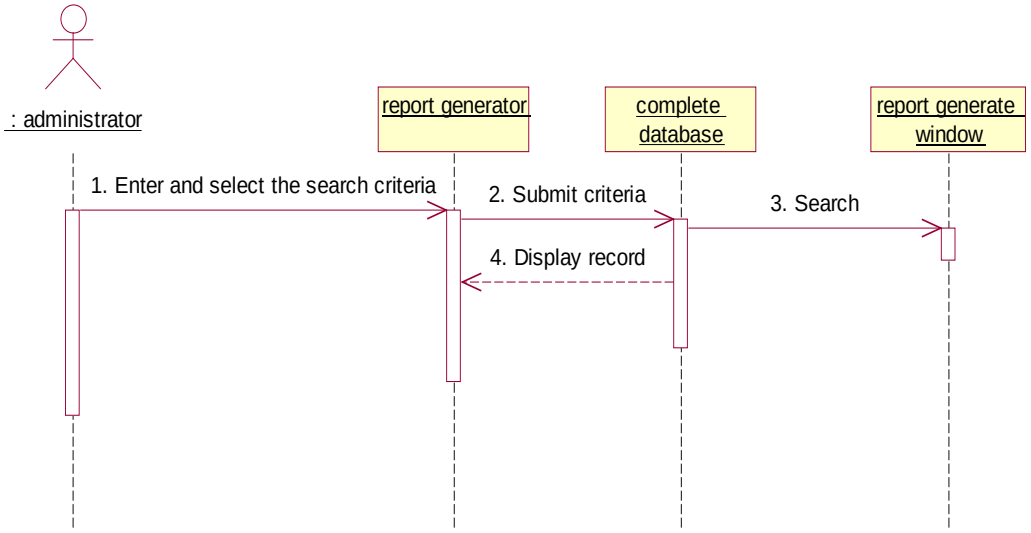


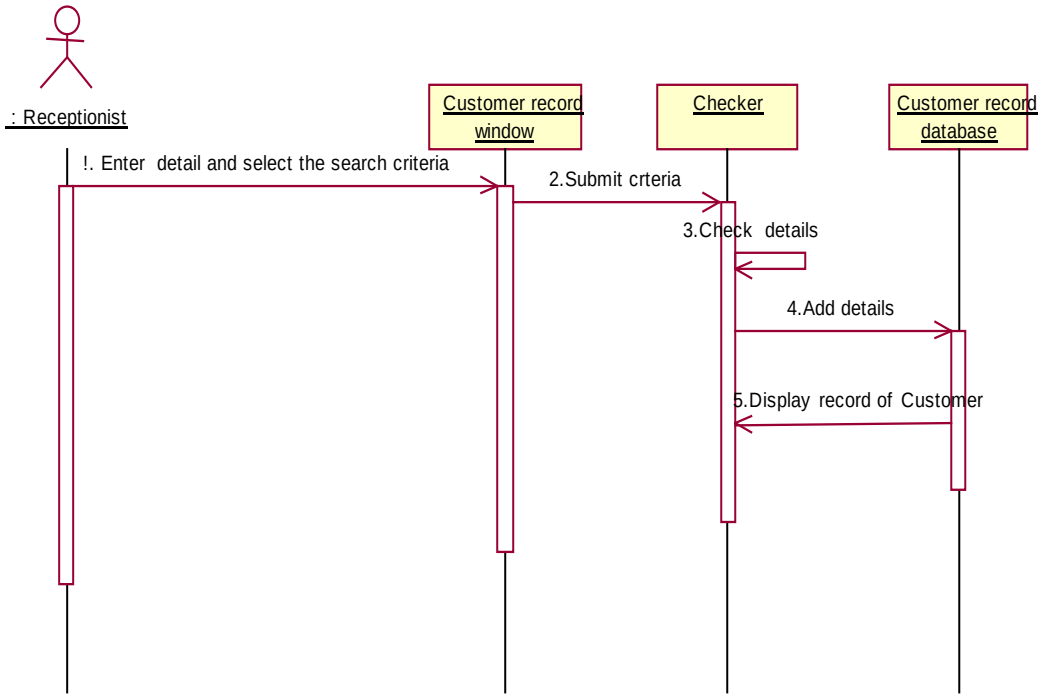


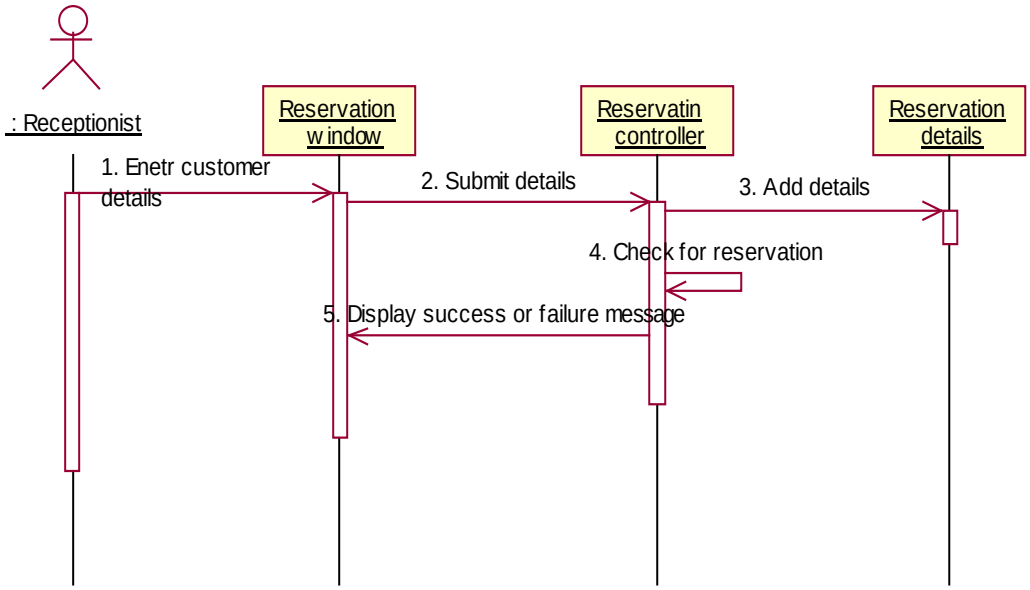
Sequence diagram of Hotel M.S.

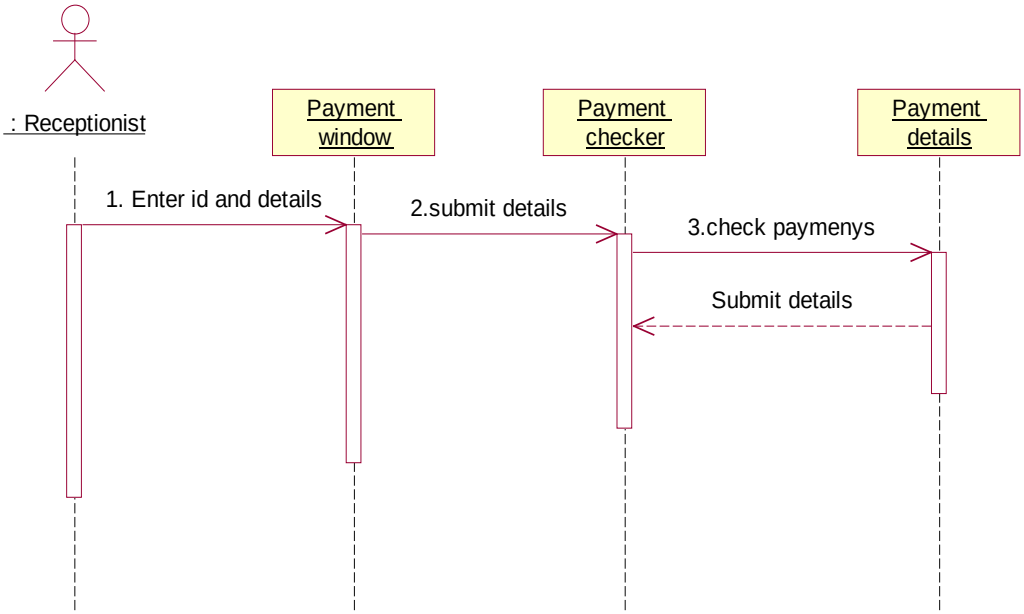


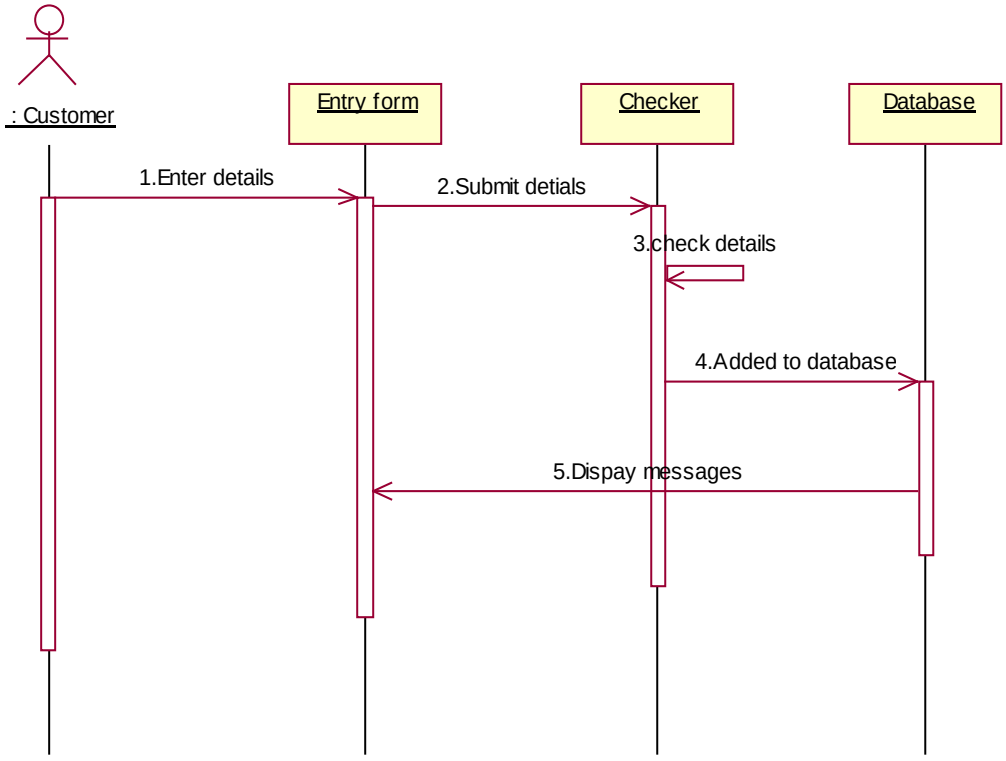


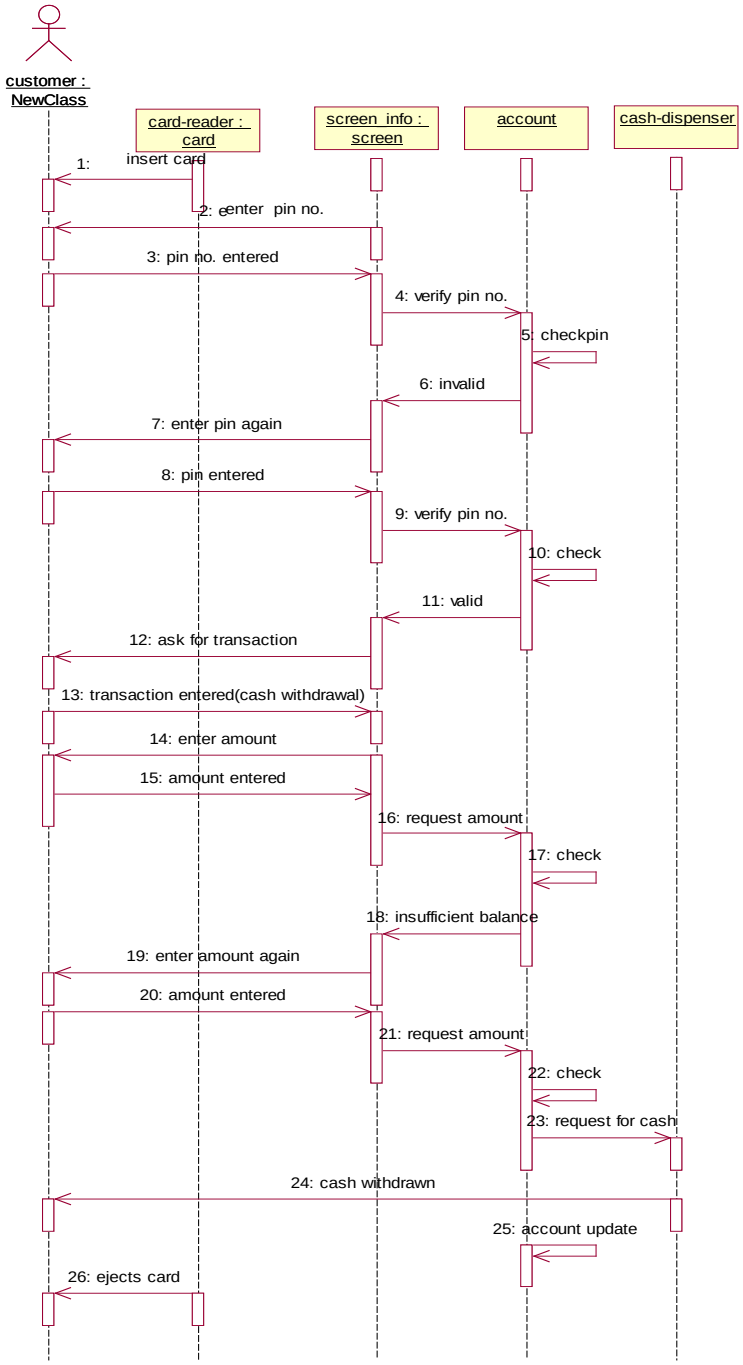












SEQUENCE DIAGRAM OF ATM SYSTEM