

- IEEE defines a requirement as “*A condition of capability needed by a user to solve a problem or achieve an objective*”
- The goal of the requirements phase is to produce the *Software Requirements Specification (SRS)* that describes what the proposed software should do without describing how the software will do it.

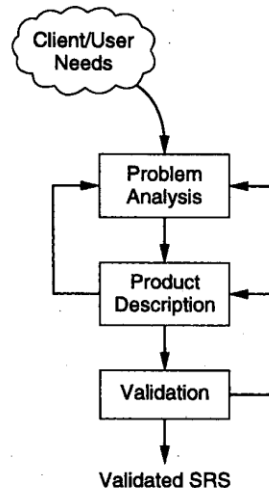
2.1 Value of a Good SRS

- Most of the software systems are originated due to satisfy the needs of clients and customers.
- The three major parties interested in a new system: the client, the developer, and the users.
- The requirements for the system that satisfies the needs of the clients must be communicated to the developer.
- Client usually does not understand software development process, and the developer does not understand the client's problem resulting in communication gap problem.
- A basic purpose of the SRS is to bridge this communication gap.
- Hence, one of the main advantages of a good SRS is:
 1. An SRS establishes the basis for agreement between the client and the supplier on what the software product will do.
 2. An SRS provides a reference for validation of the final product to determine if the software meets the requirements.
 3. A high-quality SRS is a prerequisite to high-quality software.
 4. A high-quality SRS reduces the development cost.

2.2 Requirement Process

- *The requirement process is the sequence of activities that need to be performed in the requirements phase and helps in producing a high-quality document containing the SRS.*
- It consists of three basic tasks: requirement analysis, requirements specification, and requirements validation.
- Problem analysis often starts with a high-level “problem statement.”
- The basic purpose is to have a thorough understanding of what the software needs to provide.
- Such as understanding system behavior, constraints on the system, its inputs and outputs, etc.
- During *requirement analysis*, the analyst will have a series of meetings with the clients and end users.
- In the beginning, the clients and end users will explain to the analyst about their work, their environment, and their needs.
- In these early meetings, the analyst is basically the listener, absorbing the information provided.
- Once the analyst understands the system, he uses the next meetings to get clarifications of the parts he does not understand.
- Finally the analyst explains to the client what he understands the system should do and whether it matches with the objectives of the clients.
- This understanding forms the basis for *requirements specification* and by clearly specifying the requirements in a document.
- Issues such as representation, specification languages, and tools are discussed during this activity.

- **Requirements validation** focuses on making sure that the requirements specified in the SRS are indeed all the requirements of the software.



- The requirements process is not a linear process which helps in building a good quality SRS.

2.3 Requirements Specification

- The final output is the SRS document.
- Formal modeling done during analysis is not treated as an SRS as it focuses on the problem structure, not its external behavior.
- Generally user interfaces, erroneous situations, performance constraints, design constraints, standards compliance, recovery, etc., are specified clearly in the SRS
- SRS helps the designer to properly design the system.
- The SRS is written based on the knowledge acquired during analysis.

2.3.1 Desirable Characteristics of an SRS

To properly satisfy the basic goals, an SRS should have certain properties such as

1. Correct 2. Complete 3. Unambiguous
 4. Verifiable 5. Consistent 6. Ranked for importance and/or stability
- An SRS is **correct** if every requirement specified represents something in the final system.
 - It is **complete** if everything the software is supposed to do and the outputs of the software to all classes of input data are specified in the SRS.
 - It is **unambiguous** if and only if every requirement has one only one interpretation (meaning).
 - An SRS is **verifiable** if and only if every stated requirement is verifiable.
 - It is **consistent** if there is no requirement that conflicts with another (ex: logical)
 - **An SRS is ranked for importance and/or stability** if for each requirement the importance and the stability of the requirement are indicated.
 - Stability of a requirement reflects the chances of it changing in the future.
 - Completeness is the most important and difficult property to establish.
 - The performance, interface requirements & design constraints is called as **nonfunctional requirements**.

2.3.2 Components of an SRS

- Completeness of specifications is difficult to achieve and even more difficult to verify.

- Having guidelines about what different things an SRS should specify will help in completely specifying the requirements.
- The basic issues an SRS must address are:
 1. Functionality
 2. Performance
 3. Design constraints
 4. External interfaces

Functionality

- **Functional requirements** specify the expected behavior of the system—which outputs should be produced from the given inputs.
- They describe the relationship between the input and output of the system.
- For each functional requirement, a detailed description of all the data inputs and their source, the units of measure, and the range of valid inputs must be specified.
- All the operations to be performed on the input data to obtain the output should be specified.
- This includes specifying the parameters affected system behavior in abnormal situations etc.
- For example, formula used for computing the output should be specified.
- It should specify the behavior of the system for invalid inputs and outputs and should specify on how it handles exceptional error situations.

Performance

- All the requirements relating to the **performance** characteristics of the system must be clearly specified.
- There are two types of performance requirements: static and dynamic.
- **Static requirements** are those that do not impose (force) constraint on the execution characteristics of the system and are also called as **capacity requirements**.
- These include requirements like the number of terminals to be supported, the number of simultaneous users to be supported, files that the system has to process and their sizes etc
- **Dynamic requirements** specify constraints on the execution behavior of the system which includes response time and throughput constraints on the system.
- For ex: Response time (It is the expected time for the completion of an operation) and throughput (It is the expected number of operations that can be performed in a given time).
- Different performance parameters as well as acceptable performance for both normal and peak workload conditions should be specified.

Design Constraints

- There are a number of factors in the client's environment that puts constraints on a designer while designing the system.
- An SRS should identify and specify all such constraints. Some examples of these are:
 1. **Standards Compliance:** Specifies the requirements for the standards the system must follow such as the report format and accounting procedures.
 2. **Hardware Limitations:** The software may have to operate on some existing or old hardware, thus forcing restrictions on the design. For ex: terminals used, OS, languages supported, and limits on primary and secondary storage.
 3. **Reliability and Fault Tolerance:** Fault tolerance requirements make the system more complex and expensive. **Recovery requirements** tell what the system should do if some failure occurs.

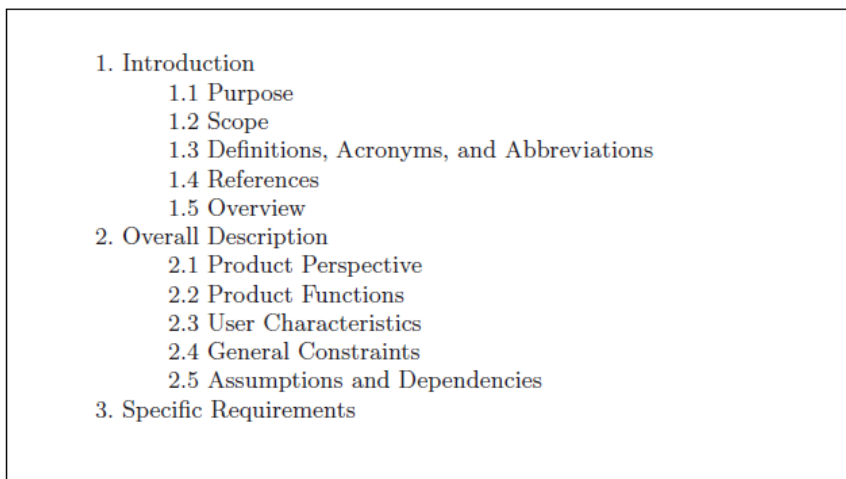
- 4. **Security:** Security requirements place restrictions on the use of certain commands, control access to data, provide different kinds of access requirements for different people, require the use of passwords and cryptography techniques, and maintain a log of activities in the system.

External interface

- In this part, all the interactions of the software with people, hardware, and other software should be clearly specified.
- **User interface:** the characteristics of each user interface of the software product should be specified.
- **Hardware interface requirements:** the SRS should specify the characteristics of each interface between the software product and the hardware components.
- **Software interface requirement:** Specifies the interface with other software the system will use.
- This includes the interface with the operating system and other applications.

2.3 Structure of a Requirements Document (Structure of SRS)

- Requirements have to be specified using some specification language.
- Natural languages are often used for specifying requirements.
- All the requirements should be clear and concise (brief) and therefore must be properly organized.
- Different projects requirements needs to be organized differently as the same one method will not be suitable for all projects.
- It provides different ways of structuring the SRS.
- The general structure of an SRS is given in Figure below.
- The **introduction section** contains the purpose, scope, overview, etc., of the requirements document.
- The key aspect here is to clarify the motivation and business objectives of the project.
- The next section gives an overall perspective of the system—how it fits into the larger system, and an overview of all the requirements of this system.
- Detailed requirements are not mentioned here.



- **Product perspective** is basically the relationship of the product to other products; defining if the product is independent or is a part of a larger product and the interfaces of the product.
- Schematic diagrams provide a general view of different functions and their relationships.
- Similarly, typical characteristics of the end user and general constraints are also specified.
- The **detailed requirements** section describes the details of the requirements that a developer needs to know for designing and developing the system.
- This is the largest and most important part of the document.

```

3. Detailed Requirements
  3.1 External Interface Requirements
    3.1.1 User Interfaces
    3.1.2 Hardware Interfaces
    3.1.3 Software Interfaces
    3.1.4 Communication Interfaces
  3.2 Functional Requirements
    3.2.1 Mode 1
      3.2.1.1 Functional Requirement 1.1
      :
      3.2.1.n Functional Requirement 1.n
    :
    3.2.m Mode m
      3.2.m.1 Functional Requirement m.1
      :
      3.2.m.n Functional Requirement m.n
  3.3 Performance Requirements
  3.4 Design Constraints
  3.5 Attributes
  3.6 Other Requirements

```

- These requirements can be organized by the modes of operation, user class, object or functional hierarchy.
- The **external interface requirements** section specifies all the interfaces of the software: to people, other software, hardware, and other systems.
- **User interfaces** specify each human interface the system plans to have, including screen formats, contents of menus, and command structure.
- In **hardware interfaces**, the logical characteristics of each interface between the software and hardware on which the software can run are specified.
- In **software interfaces**, all other software that is needed for this software to run is specified, along with the interfaces.
- **Communication interfaces** are specified if the software communicates with other entities in other machines.
- In the **functional requirements** section, the functional capabilities of the system are described.
- For each functional requirement, the required inputs, desired outputs, and processing requirements will have to be specified.
- The **performance section** should specify both static and dynamic performance requirements.
- The **attributes section** specifies the overall attributes that the system should have.
- **Design constraints** specify all the constraints imposed on design (e.g., security, fault tolerance, and standards compliance).
- Any requirement not covered under these is listed under **other requirements**.
- When use cases are employed, then the functional requirements section of the SRS is replaced by use case descriptions.

2.4 Functional Specification with Use Cases

- Functional requirements often form the core of a requirements document.
- **Use cases** specify the functionality of a system by specifying the behavior of the system, in the form of interactions of the users with the system.
- Use cases can also be used effectively for analysis.

- Use cases were used as part of the object-oriented modeling approach proposed by Jacobson and viewed as part of an object-oriented approach to software development.

2.4.1 Basics

- A software system may be used by many users, or by other systems.
- In use case terminology, an **actor** is a person or a system which uses the system for achieving some goal.
- Different actors represent groups with different goals.
- A **primary actor** is the main actor that initiates a usecase (UC) for achieving a goal, and whose goal satisfaction is the main objective of the use case.
- The primary actor executes the use case, or some agent can execute it on behalf of the primary actor.
- For example, a Time-driven trigger (report is generated automatically at sometime) of how a use case may be executed on behalf of the primary actor.
- The main goal of a use case is to describe behavior of the system that results in satisfaction of the goals of all the stakeholders and not just the primary actor.
- For example, a use case “Withdraw money from the ATM” has a customer as its primary actor, however the bank is also a stakeholder of the ATM system and its goals may include money is given only if there are sufficient funds in the account.
- A **scenario** describes a set of actions performed to achieve a goal under some specified conditions.
- The set of actions is generally specified as a sequence.
- Each step in a scenario is a logically complete action performed either by the actor or the system.
- Generally, a step is some action by the actor (ex enter information), some logical step that the system performs to progress toward achieving its goals (ex deliver information or update the record).
- A use case always has a **main success scenario**, which describes the interaction if nothing fails and all steps in the scenario succeed.
- But different situations can arise while the system and the actor are interacting which may not allow the system to achieve the goal fully.
- A use case has **extension scenarios** which describe the system behavior if some of the steps in the main scenario do not complete successfully. Sometimes they are also called **exception scenarios**.
- A use case is a collection of all the success and extension scenarios related to the goal.
- Use cases helps in specifying most of the functional requirements of the system.

2.4.2 Use cases examples

- Consider an on-line auction system built for a university, called the University Auction System (UAS)
- Different members of the university can sell and buy goods.
- A separate financial subsystem where the payments are made & each buyer & seller has an account in it.
- In this system, same people might be buying and selling and are treated as separate logical actors.
- The auction system itself is a stakeholder and an actor and the financial system is another stakeholder.
- The main use cases of this system are “put an item for auction,” “make a bid,” and “complete an auction.”
- The use cases are self-explanatory.
- Use cases are natural and story-like which makes them easy to understand by both an analyst and a layman (any user).
- This helps in reducing the communication gap between the developers and other stakeholders.
- The use cases are generally numbered for reference purposes.

- The name of the use case also specifies the goal of the primary actor.

- **UC1: Put an item for auction**

Primary Actor: Seller

Precondition: Seller has logged in

Main Success Scenario:

1. Seller posts an item (its category, description, picture, etc.) for auction
2. System shows past prices of similar items to seller
3. Seller specifies the starting bid price and a date when auction will close
4. System accepts the item and posts it

Exception Scenarios:

- 2 a) There are no past items of this category

- System tells the seller this situation

UC2: Make a bid

Primary Actor: Buyer

Precondition: The buyer has logged in

Main Success Scenario:

1. Buyer searches or browses and selects some item
2. System shows the rating of the seller, the starting bid, the current bids, and the highest bid; asks buyer to make a bid
3. Buyer specifies a bid price
4. System accepts the bid; Blocks funds in bidders account
5. System updates the max bid price, informs other users, and updates the records for the item

Exception Scenarios:

- 3 a) The bid price is lower than the current highest

- System informs the bidder and asks to rebid

- 4 a) The bidder does not have enough funds in his account

- System cancels the bid, asks the user to get more funds

UC3: Complete auction of an item

Primary Actor: Auction System

Precondition: The last date for bidding has been reached

Main Success Scenario:

1. Select highest bidder; send email to selected bidder and seller informing final bid price; send email to other bidders also
2. Debit bidder's account and credit seller's
3. Unblock all other bidders funds
4. Transfer from seller's acct. commission amt. to organization's acct.
5. Remove item from the site; update records

Exception Scenarios: None

- The primary actor can be a person or a system. Ex: UC1 & UC2 are persons but for UC3, it is a system.
- The primary actor can also be software which might request a service.
- Sometimes **preconditions** are checked before a use case is initiated such as “user is logged in,” “input data exists in files or other data structures,” etc.
- The exception situations are failure conditions that can happen anywhere and use cases handles such situations.
- A use case can employ other use cases to perform some of its work. For ex: in UC2 actions like “block the necessary funds” need to be performed for the use case to succeed.
- Financial actions are out of the scope of auction system, hence will not be described in the SRS.
- However, actions like “search” and “browse” are most likely described in the SRS.

- This allows use cases to be hierarchically organized and refinement approach can be used to define a higher-level use case in terms of lower services and then defining the lower services.
- Lower-level use cases are proper use cases with a primary actor, main scenario, etc.
- The primary actor will often be the primary actor of the higher-level use case.
- For example, the primary actor for the use case “find an item” is the buyer. It also implies that while listing the scenarios, new use cases and new actors might emerge.
- In the requirements document, all the use cases should be mentioned.

2.4.3 Extensions

- Besides specifying the actor, goal, success & exception scenarios, a use case can also specify a scope.
- If the system being built has many subsystems, use cases can capture the behavior of some subsystem.
- In such a situation it is better to specify the scope of that use case as the subsystem.
- For example, a use case for a system may be log in. It is a part of the system as well as user interaction with the system and hence forms “login and authentication” subsystem.
- Generally, a business use case has the organization as the scope; a system use case has the system being built as the scope; and a component use case is where the scope is a subsystem.
- These use cases may require many different systems to perform different tasks before the UC can be completed.

– *Use Case 0: Auction an item*
Primary Actor: Auction system
Scope : Auction conducting organization
Precondition: None
Main Success Scenario:

1. Seller performs *Put an item for auction*
2. Various bidders perform *make a bid*
3. On final date perform *Complete the auction of the item*
4. Get feedback from seller; get feedback from buyer; update records

- However, the organization-level UCs provide the context in which the systems operate.
- Hence, sometimes key business processes are described as **summary-level** use cases.
- For example, let us describe the overall use case of performing an auction.
- This use case is a business process, which provides the context for other use cases.
- Use cases may also specify post conditions for the main success scenario, or some minimal guarantees they provide in all conditions.
- For example, in some use cases, atomicity may be a minimal guarantee.
- That is, no matter what exceptions occur, either the entire transaction (goal) will be achieved, or nothing will be done.
- With atomicity, there will be no partial results and any partial changes will be rolled back.

2.4.4 Developing Use Cases

- UCs not only document requirements but also provides a good medium for discussion and brainstorming.
- UCs can be presented at different levels of abstraction. Four natural levels emerge:

1. **Actors and goals:** The actor-goal list sets out the use cases and specifies the actors for each goal. At this level, scope of the system and an overall view is specified. Completeness of functionality can be checked by validating.
2. **Main success scenarios:** The system behavior to achieve main success scenario for each use case is specified. This description ensures that interests of all the stakeholders are met and that the use case is delivering the desired behavior.
3. **Failure conditions:** All the possible failure conditions should be identified. At this level, for each step in the main success scenario, the different ways in which a step can fail form the failure conditions.
4. **Failure handling:** This is perhaps the most tricky and difficult part of writing a use case. People should pay attention to how failures should be handled and determine behavior under different failure conditions.

A step-by-step approach for analysis when employing use cases is:

Step 1. Identify the actors and their goals and get an agreement with the stakeholders. This will clearly define the scope of the system.

Step 2. Understand and specify the main success scenario for each UC. Interaction and discussion is required. The analyst may uncover some use case that are needed, which have not been identified.

Step 3. Failure conditions should be examined for the main success scenario.

Step 4. Finally, specify what should be done for these failure conditions. As details of handling failure scenarios can require a lot of effort and discussion.

- Analyst can go back to earlier steps during some detailed analysis new actors may emerge or new goals and new use cases may be uncovered.

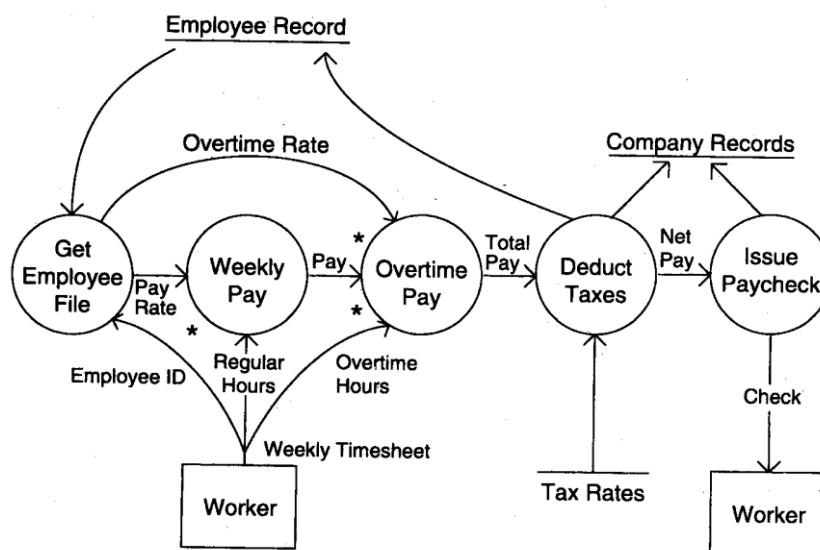
2.5 Other Approaches for Analysis

- The basic aim of problem analysis is to obtain a clear understanding of the needs of the clients and the users.
- Usually the client and the users do not know all their needs.
- The analysts have to ensure that the real needs of the clients and the users are uncovered.
- The basic principle used in analysis is divide and conquer.
- Partition the problem into sub problems and then try to understand each sub problem and its relationship with other sub problems in an effort to understand the total problem.
- The concepts of *state* and *projection* can sometimes also be used effectively in the partitioning process.
- A state of a system represents some conditions about the system.
- When using *state*, a system is first viewed as operating in one of the several possible states, and then a detailed analysis is performed for each state.
- State approach is sometimes used in real-time software or process-control software.
- In *projection*, different viewpoints of the system are defined and the system is then analyzed from these different perspectives.
- Analyzing the system from the different perspectives is often easier, as it limits and focuses the scope of the study.
- The analyst must pick the approach that he feels is best suited to the problem at hand.

2.5.1 Data Flow Diagrams

- Data flow diagrams (also called data flow graphs) are commonly used during problem analysis.

- Data flow diagrams (DFDs) are quite general and are not limited to problem analysis for software requirements specification.
- A DFD shows the flow of data through a system.
- It views a system as a function that transforms the inputs into desired outputs.
- The DFD aims to capture the transformations that take place within a system to the input data so that eventually the output data is produced.
- The agent that performs the transformation of data from one state to another is called a **process** (or a bubble).
- **DFD shows the movement of data through the different transformations or processes in the system.**
- The processes are shown by named circles and data flows are represented by named arrows.
- A rectangle represents a source or sink and is a net originator or consumer of data.
- In this DFD there is one basic input data flow, the weekly timesheet, which originates from the source worker.
- The basic output is the paycheck, the sink for which is also the worker.
- In this system, first the employee's record is retrieved, using the employee ID, which is contained in the timesheet.
- From the employee record, the rate of payment and overtime are obtained.
- These rates and the regular and overtime hours (from the timesheet) are used to compute the pay.
- After the total pay is determined, taxes are deducted. To compute the tax deduction, information from the tax-rate file is used.
- The amount of tax deducted is recorded in the employee and company records. Finally, the paycheck is issued for the net pay.
- The amount paid is also recorded in company records.
- All external files such as employee record, company record, and tax rates are shown as a labeled straight line.
- The need for multiple data flows by a process is represented by a "*" between the data flows.
- This symbol represents the AND relationship.

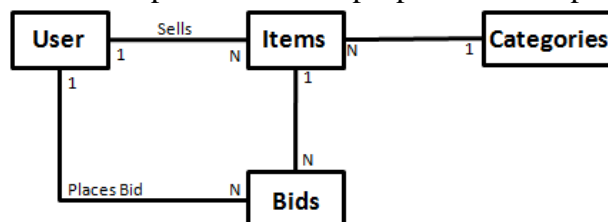


- For example, if there is a "*" between the two input, it means that both inputs are needed for the process.
- In the DFD, for the process "weekly pay" the data flow "hours" and "pay rate" both are needed.

- Similarly, the OR relationship is represented by a “+” between the data flows.
- This DFD is an abstract description of the system for handling payment. For example, what happens if there are errors in the weekly timesheet is not shown in this DFD.
- This is done to avoid too many details and confusion while constructing an overall system DFD.
- DFDs are hierarchically organized, which helps in progressively partitioning and analyzing large systems. Such DFDs together are called a leveled DFD set.
- A leveled DFD set has a starting DFD, which is a very abstract representation of the system, identifying the major inputs and outputs and the major processes in the system.
- Then each process is refined and a DFD is drawn for the process.
- This refinement stops if each process (bubble) is considered to be “atomic,” in that each bubble can be easily specified or understood.
- Data flows are uniquely named and it should convey some meaning about what the data is.
- However, for specifying the structure of data flows, a data dictionary is often used.

2.5.2 ER Diagrams

- Entity relationship diagrams (ERDs) have been used for years for data modeling.
- An ERD can be used to model the data in the system and how the data items relate to each other
- It does not describe how the data is processed or changed in the system.
- It is used often by database designers to represent the structure of the database
- It is a useful tool for analyzing software systems which use databases.
- An ER diagram is complementary to use cases.
- Whereas use cases focus on the nature of interaction and functionality.
- ER diagrams focus on the structure of the entities being used in the use cases.
- Both use cases and ER diagrams can be used while analyzing the requirements of a system and both can be contained in an SRS.
- ER models form the logical database design and can be easily converted into initial table structure for a relational database.
- ER diagrams have two main concepts called as entities and relationships.
- **Entities** are the main information holders in a system and are represented as boxes.
- Entities can be viewed as types which describe all elements of a type which have common properties.
- An entity is equivalent to a table in a database
- Entities may have attributes, which are properties of the concept being represented.
- Relationships between two entities are represented by a line connecting them.
- Notations used to express the nature of relationships are “0”, “1”, or “M” on the two sides of the relationship line to represent the cardinality of the relationship.
- Relationships reflect some properties of the problem domain.



- In ER diagram for the auction system, there are four tables for users, categories, items, and bids.
- As user is related to item by one-to-many.

- Each user can sell many items, so there is a one-to-many relationship “Sell” between the user and items.
- Similarly, there is a one-to-many relationship between items & bids, between users & bids, & between categories and items.
- The item table should have a user-ID field to uniquely identify the user who is selling the item.
- Similarly, the bid table must have a user-ID to identify the user who placed the bid, and an item-ID to identify the item for which the bid has been made.

2.6 Validation

- The analyst has to make sure that the SRS document specifies the client’s requirements correctly.
- It is very important to detect errors in the requirements before the design as the cost for changing the system might be more in the later stages.
- But in requirements specification phase, there is a lot of misunderstanding and committing errors
- The basic objective of validation is to ensure that the SRS specifies the actual requirements accurately and clearly.
- A related objective is to check that the SRS document is itself of good quality.
- Usually the type of errors that occur in an SRS are classified in four types: omission, inconsistency, incorrect fact, and ambiguity.
- **Omission** is a common error in requirements. If some user requirement might simply not included in the SRS; the omitted requirement may be related to the system in some form.
- **Inconsistency** can be due to contradictions within the requirements themselves or to incompatibility of the stated requirements with the actual requirements.
- **Incorrect fact** occurs when some fact recorded in the SRS is not correct.
- **Ambiguity** occurs when there are some requirements that have multiple meanings.
- Some projects have collected data about requirement errors.
- Consider real-time flight control software project where a total of about 80 errors were detected in requirements document, out of which about 23% had errors in documentation. Of the remaining, the distribution with error type was:

Omission	Incorrect Fact	Inconsistency	Ambiguity
32%	49%	13%	5%

- Besides improving the quality of the SRS (e.g., no clerical errors), the validation should focus on uncovering these types of errors.
- Inspections and reviews are suitable for requirements validation.
- **Requirements review** is a review by a group of people to find errors and other concerns in the requirements specifications.
- The review group includes author of the requirements document, designer, and people responsible for maintaining the requirements document and a software quality engineer.
- Review process should also consider factors affecting quality, such as testability & readability.
- Review team must go through each requirement, find errors, discuss and agree on the nature of the error.
- Sometimes special purpose tools for modeling and analysis are used.
- These tools focus on internal consistency and completeness.
