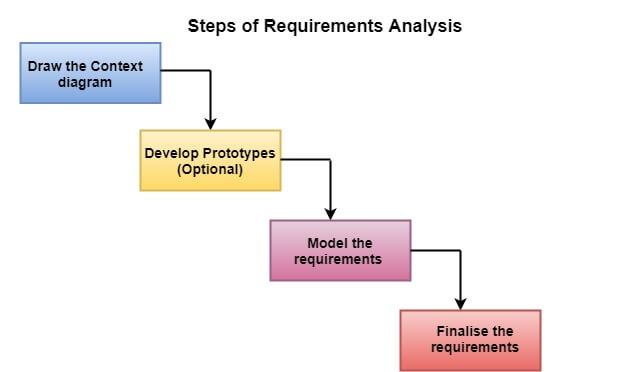
**UNIT II**

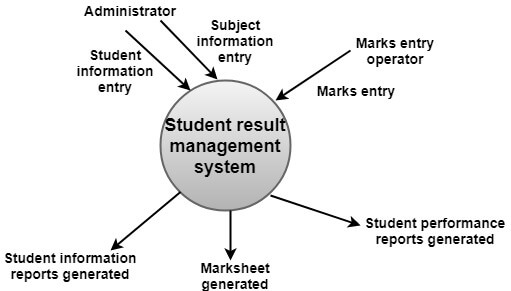
**Requirement analysis :**

Requirement analysis is significant and essential activity after elicitation. We analyze, refine, and scrutinize the gathered requirements to make consistent and unambiguous requirements. This activity reviews all requirements and may provide a graphical view of the entire system. After the completion of the analysis, it is expected that the understandability of the project may improve significantly. Here, we may also use the interaction with the customer to clarify points of confusion and to understand which requirements are more important than others.

**The various steps of requirement analysis are shown in fig:**



**(i) Draw the context diagram:** The context diagram is a simple model that defines the boundaries and interfaces of the proposed systems with the external world. It identifies the entities outside the proposed system that interact with the system. The context diagram of student result management system is given below:



**(ii) Development of a Prototype (optional):** One effective way to find out what the customer wants is to construct a prototype, something that looks and preferably acts as part of the system they say they want.

We can use their feedback to modify the prototype until the customer is satisfied continuously. Hence, the prototype helps the client to visualize the proposed system and increase the understanding of the requirements. When developers and users are not sure about some of the elements, a prototype may help both the parties to take a final decision.

Some projects are developed for the general market. In such cases, the prototype should be shown to some representative sample of the population of potential purchasers. Even though a person who tries out a prototype may not buy the final system, but their feedback may allow us to make the product more attractive to others.

The prototype should be built quickly and at a relatively low cost. Hence it will always have limitations and would not be acceptable in the final system. This is an optional activity.

**(iii) Model the requirements:** This process usually consists of various graphical representations of the functions, data entities, external entities, and the relationships between them. The graphical view may help to find incorrect, inconsistent, missing, and superfluous requirements. Such models include the Data Flow diagram, Entity-Relationship diagram, Data Dictionaries, State-transition diagrams, etc.

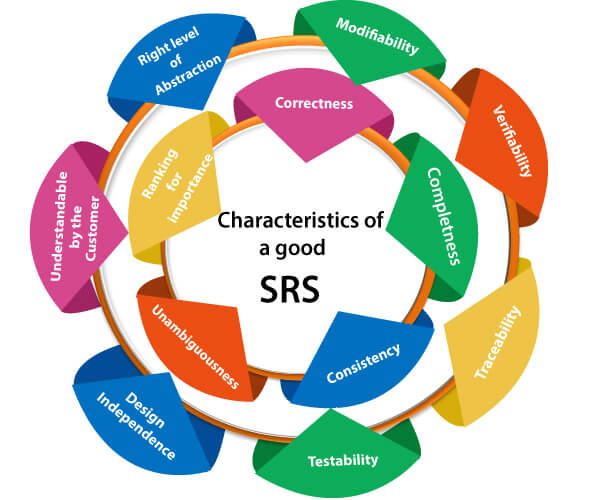
**(iv) Finalise the requirements:** After modeling the requirements, we will have a better understanding of the system behavior. The inconsistencies and ambiguities have been identified and corrected. The flow of data amongst various modules has been analyzed. Elicitation and analyze activities have provided better insight into the system. Now we finalize the analyzed requirements, and the next step is to document these requirements in a prescribed format.

Software Requirements

The production of the requirements stage of the software development process is **Software Requirements Specifications (SRS)** (also called a **requirements document**). This report lays a foundation for software engineering activities and is constructing when entire requirements are elicited and analyzed. **SRS** is a formal report, which acts as a representation of software that enables the customers to review whether it (SRS) is according to their requirements. Also, it comprises user requirements for a system as well as detailed specifications of the system requirements.

The SRS is a specification for a specific software product, program, or set of applications that perform particular functions in a specific environment. It serves several goals depending on who is writing it. First, the SRS could be written by the client of a system. Second, the SRS could be written by a developer of the system. The two methods create entirely various situations and establish different purposes for the document altogether. The first case, SRS, is used to define the needs and expectation of the users. The second case, SRS, is written for various purposes and serves as a contract document between customer and developer.

Characteristics of good SRS



**Following are the features of a good SRS document:**

**1. Correctness:** User review is used to provide the accuracy of requirements stated in the SRS. SRS is said to be perfect if it covers all the needs that are truly expected from the system.

**2. Completeness:** The SRS is complete if, and only if, it includes the following elements:

**(1).** All essential requirements, whether relating to functionality, performance, design, constraints, attributes, or external interfaces.

**(2).** Definition of their responses of the software to all realizable classes of input data in all available categories of situations.

**(3).** Full labels and references to all figures, tables, and diagrams in the SRS and definitions of all terms and units of measure.

**3. Consistency:** The SRS is consistent if, and only if, no subset of individual requirements described in its conflict. There are three types of possible conflict in the SRS:

**(1).** The specified characteristics of real-world objects may conflicts. For example,

(a) The format of an output report may be described in one requirement as tabular but in another as textual.

(b) One condition may state that all lights shall be green while another states that all lights shall be blue.

**(2).** There may be a reasonable or temporal conflict between the two specified actions. For example,

(a) One requirement may determine that the program will add two inputs, and another may determine that the program will multiply them.

(b) One condition may state that "A" must always follow "B," while other requires that "A and B" co-occurs.

**(3).** Two or more requirements may define the same real-world object but use different terms for that object. For example, a program's request for user input may be called a "prompt" in one requirement's and a "cue" in another. The use of standard terminology and descriptions promotes consistency.

**4. Unambiguousness:** SRS is unambiguous when every fixed requirement has only one interpretation. This suggests that each element is uniquely interpreted. In case there is a method used with multiple definitions, the requirements report should determine the implications in the SRS so that it is clear and simple to understand.

**5. Ranking for importance and stability:** The SRS is ranked for importance and stability if each requirement in it has an identifier to indicate either the significance or stability of that particular requirement.

Typically, all requirements are not equally important. Some prerequisites may be essential, especially for life-critical applications, while others may be desirable. Each element should be identified to make these differences clear and explicit. Another way to rank requirements is to distinguish classes of items as essential, conditional, and optional.

**6. Modifiability:** SRS should be made as modifiable as likely and should be capable of quickly obtain changes to the system to some extent. Modifications should be perfectly indexed and cross-referenced.

**7. Verifiability:** SRS is correct when the specified requirements can be verified with a cost-effective system to check whether the final software meets those requirements. The requirements are verified with the help of reviews.

**8. Traceability:** The SRS is traceable if the origin of each of the requirements is clear and if it facilitates the referencing of each condition in future development or enhancement documentation.

**There are two types of Traceability:**

**1. Backward Traceability:** This depends upon each requirement explicitly referencing its source in earlier documents.

**2. Forward Traceability:** This depends upon each element in the SRS having a unique name or reference number.

The forward traceability of the SRS is especially crucial when the software product enters the operation and maintenance phase. As code and design document is modified, it is necessary to be able to ascertain the complete set of requirements that may be concerned by those modifications.

**9. Design Independence:** There should be an option to select from multiple design alternatives for the final system. More specifically, the SRS should not contain any implementation details.

**10. Testability:** An SRS should be written in such a method that it is simple to generate test cases and test plans from the report.

**11. Understandable by the customer:** An end user may be an expert in his/her explicit domain but might not be trained in computer science. Hence, the purpose of formal notations and symbols should be avoided too as much extent as possible. The language should be kept simple and clear.

## Properties of a good SRS document

**The essential properties of a good SRS document are the following:**

**Concise:** The SRS report should be concise and at the same time, unambiguous, consistent, and complete. Verbose and irrelevant descriptions decrease readability and also increase error possibilities.

**Structured:** It should be well-structured. A well-structured document is simple to understand and modify. In practice, the SRS document undergoes several revisions to cope up with the user requirements. Often, user requirements evolve over a period of time. Therefore, to make the modifications to the SRS document easy, it is vital to make the report well-structured.

**Black-box view:** It should only define what the system should do and refrain from stating how to do these. This means that the SRS document should define the external behavior of the system and not discuss the implementation issues. The SRS report should view the system to be developed as a black box and should define the externally visible behavior of the system. For this reason, the SRS report is also known as the black-box specification of a system.

**Conceptual integrity:** It should show conceptual integrity so that the reader can merely understand it. Response to undesired events: It should characterize acceptable responses to unwanted events. These are called system response to exceptional conditions.

**Verifiable:** All requirements of the system, as documented in the SRS document, should be correct. This means that it should be possible to decide whether or not requirements have been met in an implementation.

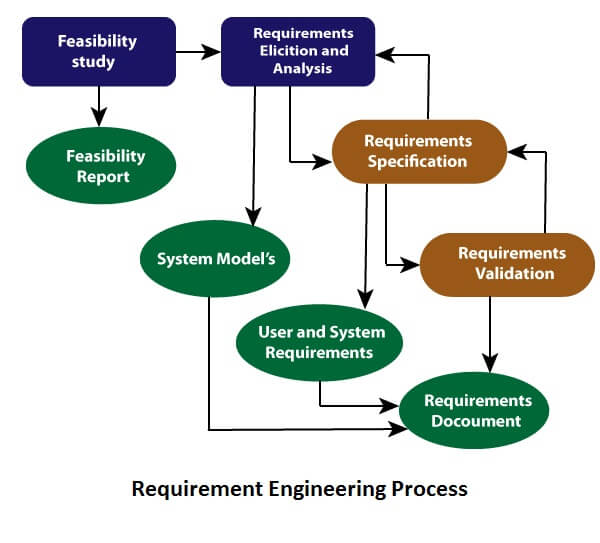
**Requirements engineering:**

**Requirements engineering (RE)** refers to the process of defining, documenting, and maintaining requirements in the engineering design process. Requirement engineering provides the appropriate mechanism to understand what the customer desires, analyzing the need, and assessing feasibility, negotiating a reasonable solution, specifying the solution clearly, validating the specifications and managing the requirements as they are transformed into a working system. Thus, requirement engineering is the disciplined application of proven principles, methods, tools, and notation to describe a proposed system's intended behavior and its associated constraints.

## Requirement Engineering Process

It is a four-step process, which includes -

1. Feasibility Study
2. Requirement Elicitation and Analysis
3. Software Requirement Specification
4. Software Requirement Validation
5. Software Requirement Management



### **1. Feasibility Study:**

The objective behind the feasibility study is to create the reasons for developing the software that is acceptable to users, flexible to change and conformable to established standards.

**Types of Feasibility:**

1. **Technical Feasibility** - Technical feasibility evaluates the current technologies, which are needed to accomplish customer requirements within the time and budget.
2. **Operational Feasibility** - Operational feasibility assesses the range in which the required software performs a series of levels to solve business problems and customer requirements.
3. **Economic Feasibility** - Economic feasibility decides whether the necessary software can generate financial profits for an organization.

### **2. Requirement Elicitation and Analysis:**

This is also known as the **gathering of requirements**. Here, requirements are identified with the help of customers and existing systems processes, if available.

Analysis of requirements starts with requirement elicitation. The requirements are analyzed to identify inconsistencies, defects, omission, etc. We describe requirements in terms of relationships and also resolve conflicts if any.

**Problems of Elicitation and Analysis**

* Getting all, and only, the right people involved.
* Stakeholders often don't know what they want
* Stakeholders express requirements in their terms.
* Stakeholders may have conflicting requirements.
* Requirement change during the analysis process.
* Organizational and political factors may influence system requirements.

### **3. Software Requirement Specification:**

Software requirement specification is a kind of document which is created by a software analyst after the requirements collected from the various sources - the requirement received by the customer written in ordinary language. It is the job of the analyst to write the requirement in technical language so that they can be understood and beneficial by the development team.

The models used at this stage include ER diagrams, data flow diagrams (DFDs), function decomposition diagrams (FDDs), data dictionaries, etc.

* **Data Flow Diagrams:** Data Flow Diagrams (DFDs) are used widely for modeling the requirements. DFD shows the flow of data through a system. The system may be a company, an organization, a set of procedures, a computer hardware system, a software system, or any combination of the preceding. The DFD is also known as a data flow graph or bubble chart.
* **Data Dictionaries:** Data Dictionaries are simply repositories to store information about all data items defined in DFDs. At the requirements stage, the data dictionary should at least define customer data items, to ensure that the customer and developers use the same definition and terminologies.
* **Entity-Relationship Diagrams:** Another tool for requirement specification is the entity-relationship diagram, often called an "**E-R diagram**." It is a detailed logical representation of the data for the organization and uses three main constructs i.e. data entities, relationships, and their associated attributes.

### **4. Software Requirement Validation:**

After requirement specifications developed, the requirements discussed in this document are validated. The user might demand illegal, impossible solution or experts may misinterpret the needs. Requirements can be the check against the following conditions -

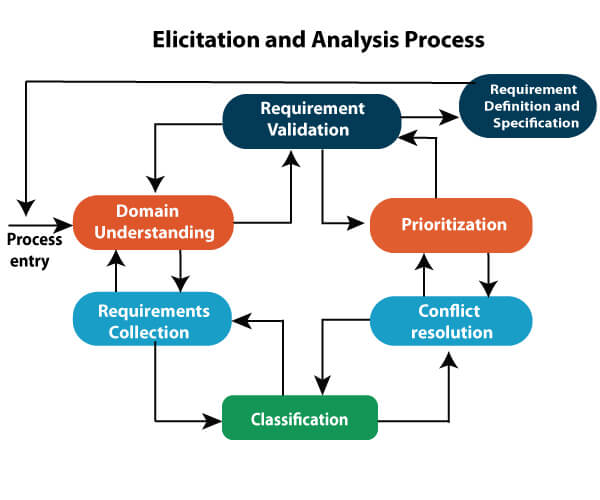
* If they can practically implement
* If they are correct and as per the functionality and specially of software
* If there are any ambiguities
* If they are full
* If they can describe

**Requirements Validation Techniques**

* **Requirements reviews/inspections:** systematic manual analysis of the requirements.
* **Prototyping:** Using an executable model of the system to check requirements.
* **Test-case generation:** Developing tests for requirements to check testability.
* **Automated consistency analysis:** checking for the consistency of structured requirements descriptions.

### **Software Requirement Management:**

Requirement management is the process of managing changing requirements during the requirements engineering process and system development.



Developing a Use cases

Developing a use case is the practice of creating a written description that outlines how a system or process responds to user behaviors. Written from a hypothetical user's perspective, it summarizes the ideal functionality of a product and helps to envision potential user scenarios, both good and bad. Generally speaking, use cases consist of the following common elements:

* A description of the user
* Desirable behaviors of the user
* The user's overall goal
* The specific steps the user takes to complete their goal
* The response of the process based on a user's actions

For example, a use case for a website might detail a user's characteristics, what they wish to accomplish on the site, how the site responds to user inputs and whether the user is successful in their intent. Studying this use case can help determine the suitability of the website for its purpose and identify potential areas of improvement.

Developing a use case is important because it helps overcome ambiguous goals, especially when a project involves newer processes or technologies. It allows you not only to define requirements but also to maintain consistency across them. In addition, developing a use case can provide the following advantages:

* **Ideal reactions to certain behaviors:** A use case explains a user's behaviors to the system, guiding it on how to respond.
* **Error prediction:** A use case can also help the system with predicting mistakes before they occur.
* **A List of goals:** A use case can also help provide the system with a list of goals, along with steps on how to achieve those specific goals.
* **Sets of rules:** These rules can guide you on how best to use technology to meet user goals.
* **Project clarity:** A use case can also help project managers better understand the client's requirements.

**How to plan use case?**

Following example will illustrate on how to plan use cases:

**Use Case:**What is the main objective of this use case. For eg. Adding a software component, adding certain functionality etc.

**Primary Actor:**Who will have the access to this use case. In the above examples, administrators will have the access.

**Scope:**Scope of the use case

**Level:**At what level the implementation of the use case be.

**Flow:**What will be the flow of the functionality that needs to be there. More precisely, the work flow of the use case.

Some other things that can be included in the use cases are:

* **Preconditions**
* **Postconditions**
* **Brief course of action**
* **Time Period**
* **Steps for develop Use cases**

# Developing a Use Case

|  |
| --- |
| When developing use cases you should start with a functional partition—a listing of the major functional categories of the application. This will help identify what areas need to be focused on. [(10)](http://www.umsl.edu/~sauterv/analysis/488_f01_papers/Prossman/Bibliography.htm) |

|  |
| --- |
| **Step 1:  Identify who is going to be using the system directly. These are the Actors.**    The main component of use case development is actors. An actor is a specific role played by a system user and represents a category of users that demonstrates similar behaviors when using the system. The actors may be people or computer systems. [(10)](http://www.umsl.edu/~sauterv/analysis/488_f01_papers/Prossman/Bibliography.htm) A primary actor is one having a goal requiring the assistance of the system. A secondary actor is one from which the system needs assistance to satisfy its goal.  One of the actors is designated as the system under discussion. [(6)](http://www.umsl.edu/~sauterv/analysis/488_f01_papers/Prossman/Bibliography.htm) A person can play several roles and thereby represent several actors, such as computer-system operator or end user. |

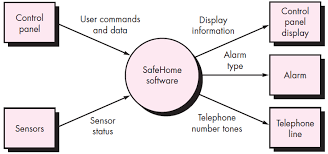
|  |
| --- |
| **Step 2: Pick one of those Actors.**    To identify a target system’s use case, we identify the system actors. A good starting point is to check the system design and identify who it is supposed to help. [(12)](http://www.umsl.edu/~sauterv/analysis/488_f01_papers/Prossman/Bibliography.htm) |

|  |
| --- |
| **Step 3: Define what that Actor wants to do with the system. Each of these things that the actor wants to do with the system become a Use Case.**    The things that the actors want to do with the system become goals. The goal is the end outcome of the actions of the user. There are two types of goals. The first type is a rigid goal. This goal must be completely satisfied and describes a target system’s minimum requirement. The second type of goal is a soft goal. This usually describes a desired property for a target system and does not need to be completely satisfied. [(12)](http://www.umsl.edu/~sauterv/analysis/488_f01_papers/Prossman/Bibliography.htm) To identify use cases, we can read the requirement specification from an actor’s perspective and carry on discussions with those users who will function as actors. By defining everything that every actor will be able to do in interaction with the system, the complete functionality of the system is defined. [(4)](http://www.umsl.edu/~sauterv/analysis/488_f01_papers/Prossman/Bibliography.htm) |

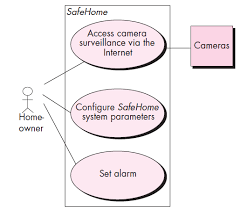
|  |
| --- |
| **Step 4: For each of those Use Cases decide on the most usual course when that Actor is using the system. What normally happens.**    A use case has one basic course and several alternative courses. The basic course is the simplest course, the one in which a request is delivered without any difficulty. [(12)](http://www.umsl.edu/~sauterv/analysis/488_f01_papers/Prossman/Bibliography.htm) There may be alternative courses that describe variants of the basic course and the errors that can occur. These are documented as extensions to the use case. |

|  |
| --- |
| **Step 5: Describe that basic course in the description for the use case.**    The use scenario is written from the user’s perspective in view in easy to understand language. This step is very similar to documenting a process flow. The steps necessary to achieve the identified goal are written out. |

|  |
| --- |
| **Step 6: Once you’re happy with the basic course now consider the alternatives and add those as extending use cases.**    The extensions are written in the same manner as the original use case but they provide alternatives to the simplest path. |

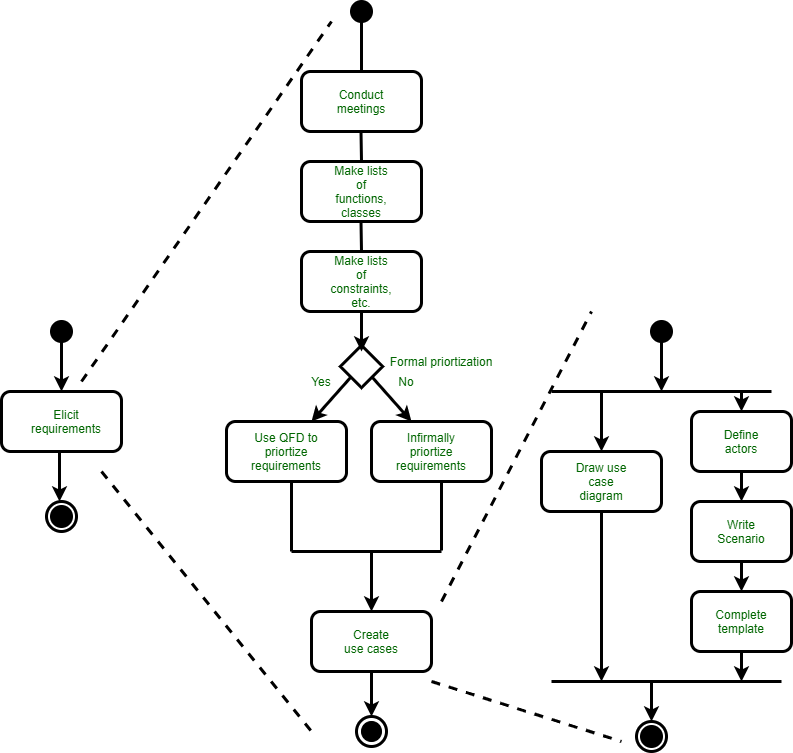
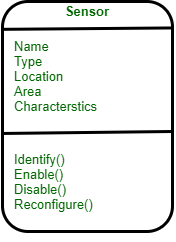


Safehome system



Requirements for a computer-based system can be seen in many different ways. Some software people argue that it’s worth using a number of different modes of representation while others believe that it’s best to select one mode of representation.

The specific **elements of the requirements model** are dedicated to the analysis modeling method that is to be used.

* **Scenario-based elements :**  
  Using a scenario-based approach, system is described from user’s point of view. **For example**, basic use cases and their corresponding use-case diagrams evolve into more elaborate template-based use cases. Figure 1(a) depicts a UML activity diagram for eliciting requirements and representing them using use cases. There are three levels of elaboration.
* **Class-based elements :**  
  A collection of things that have similar attributes and common behaviors i.e., objects are categorized into classes. **For example**, a UML case diagram can be used to depict a Sensor class for the SafeHome security function. Note that diagram lists attributes of sensors and operations that can be applied to modify these attributes. In addition to class diagrams, other analysis modeling elements depict manner in which classes collaborate with one another and relationships and interactions between classes.
* **Behavioral elements :**  
  Effect of behavior of computer-based system can be seen on design that is chosen and implementation approach that is applied. Modeling elements that depict behavior must be provided by requirements model.
* 
* *Figure 1(a):UML activity diagrams for eliciting requirements*
* 
* *Class diagram for sensor*
* Method for representing behavior of a system by depicting its states and events that cause system to change state is state diagram. A state is an externally observable mode of behavior. In addition, state diagram indicates actions taken as a consequence of a particular event.
* **Flow-oriented elements :**  
  As it flows through a computer-based system information is transformed. System accepts input, applies functions to transform it, and produces output in a various forms. Input may be a control signal transmitted by a transducer, a series of numbers typed by human operator, a packet of information transmitted on a network link, or a voluminous data file retrieved from secondary storage. Transform may compromise a single logical comparison, a complex numerical algorithm, or a rule-inference approach of an expert system. Output produce a 200-page report or may light a single LED. In effect, we can create a flow model for any computer-based system, regardless of size and complexity.

## Analysis Patterns

Anyone who has done requirements engineering on a number of software projects will note that some issues repeat across all projects within a certain application area. These patterns of analysis provide solutions (e.g., a class, a function, or a behaviour) inside the application domain that can be reused when modelling several applications.

By referencing the pattern name, analysis patterns are integrated into the analysis model. They are also stored in a repository so that requirements engineers can find and apply them using search facilities. Information about an analysis pattern (and other sorts of patterns) is contained in a standard template.

## Negotiating requiremnts

The inception, elicitation, and elaboration tasks in an ideal requirements engineering setting determine customer requirements in sufficient depth to proceed to later software engineering activities. You might have to negotiate with one or more stakeholders. Most of the time, stakeholders are expected to balance functionality, performance, and other product or system attributes against cost and time-to-market.

The goal of this discussion is to create a project plan that meets the objectives of stakeholders while also reflecting the real-world restrictions (e.g., time, personnel, and budget) imposed on the software team. The successful negotiations aim for a “win-win” outcome. That is, stakeholders, benefit from a system or product that meets the majority of their needs, while you benefit from working within realistic and reasonable budgets and schedules.

At the start of each software process iteration, Boehm defines a series of negotiating actions. Rather than defining a single customer communication activity, the following are defined:

1. Identifying the major stakeholders in the system or subsystem.
2. Establishing the stakeholders’ “win conditions.”
3. Negotiation of the win conditions of the stakeholders in order to reconcile them into a set of win-win conditions for all people involved.

**Validating Requirements**

Each aspect of the requirements model is checked for consistency, omissions, and ambiguity as it is developed. The model’s requirements are prioritised by stakeholders and bundled into requirements packages that will be implemented as software increments.

The following questions are addressed by an examination of the requirements model:

* Is each requirement aligned with the overall system/product objectives?
* Were all requirements expressed at the appropriate level of abstraction? Do some criteria, in other words, give a level of technical information that is inappropriate at this stage?
* Is the requirement truly necessary, or is it an optional feature that may or may not be critical to the system’s goal?
* Is each requirement well defined and unambiguous?
* Is each requirement attributed? Is there a source noted for each requirement?
* Are there any requirements that conflict with others?
* Is each requirement attainable in the technical environment in which the system or product will be housed?
* Is each requirement, once implemented, testable?
* Does the requirements model accurately represent the information, functionality, and behaviour of the system to be built?
* Has the requirements model been “partitioned” in such a way that progressively more detailed information about the system is exposed?
* Have requirements patterns been used to reduce the complexity of the requirements model?
* Have all patterns been validated properly? Are all patterns in accordance with the requirements of the customers?