**Unit-II**

**Chapter-1**

**REQUIREMENT ENGINEERING**

**Introduction:** Requirements engineering helps software engineers better understand the problems they are trying to solve. Building an elegant computer solution that ignores the customer's needs helps no one. It includes the set of tasks that helps to understand the business impact of the software, need of the customer and the interaction of the end-user with the software. Thus, it is very important to understand the customer's wants and needs before designing or building a computer-based solution. The requirements engineering process begins with inception, moves on to elicitation, elaboration, negotiation, problem specification, and ends with review or validation of the specification. The intent of requirements engineering is to produce a written understanding of the customer's problem.

Requirements Engineering, must be adapted to the needs of a specific process, project, product, or people doing the work. It is a software engineering action, that begins during the software engineering communication activity and continues into the modeling activity. Hence, it is essential that the software engineering team understand the requirements of a problem before the team tries to solve the problem.

**1.1 Requirements Engineering Tasks:**

The requirements engineering process consists of seven distinct functions: Inception, Elicitation, Elaboration, Negotiation, Specification, Validation and Management. Requirements engineering functions can occur in parallel according to the needs of the project.

**Inception:** Most projects begin with the identification of a business need or when a new market or service is needed. Stakeholders will study and identify the need and perform a rough feasibility analysis and identify a working description of the project’s scope.

During Inception, the software engineers ask some context-free questions to establish a basic understanding of the problem, the people who want a solution, the nature of the solution desired and the effectiveness of the collaboration between customers and developers.

**Elicitation:** Elicitation process finds out from customers or users, what the product objectives are, what is to be done, how the product fits into business needs, and how the product is used on a day to day basis. It is difficult to gain a clear understanding of the customer needs because

* Problems of scope: The system boundary is ill-defined ie., not clear with overall system objectives.
* Problems of Understanding: The customer/user are not clear about their needs. They may have poor understanding of the computing environment and problem domain. Poor communication while specifying the requirements, specifying unambiguous requirements may also lead to problems.
* Problems of volatility: The requirements may change over time.

**Elaboration:** Elaboration focuses on developing a refined technical model of software functions, features, and constraints using the information obtained during inception and elicitation. Elaboration is an analysis modeling action. It consists of a number of modeling and refinement tasks. User scenarios are created which describes the interaction of end-user with the system. The user scenarios are used to identify the analysis classes. The attributes and methods of each class and the relationship between classes are identified. It can be followed by a variety of UML diagrams for more clarification. Thus elaboration results into an analysis model that defines the informational, functional and behavioural domain of the problem.

**Negotiation:** Negotiation categorizes the requirements and organizes it into subsets, relations among the requirements are identified. The requirements are reviewed for correctness and they are prioritized based on customer needs.

The conflicting requirements proposed by the customers are reconciled by asking the customers/users/stakeholders to rank the requirements and discussed the conflicts in priority. Risks associated with the requirements are identified and analyzed. Using an iterative approach the requirements are eliminated, combined or modified so that the stakeholders are satisfied.

**Specification:** Specification is the written work products produced describing the function, performance, and development constraints for a computer-based system. It can be a written document, a set of graphical models, a formal mathematical model, a collection of usage scenarios, a prototype or a combination of these. Thus, it serves as the foundation for the software engineering activities.

**Validation:** Requirement Validation is a process of assessing the quality of the work products produced during requirements engineering . The requirements are checked for ambiguity, inconsistency, omissions, errors detected and corrected. Formal technical reviews mechanism is used to examine the specification work products to ensure requirement quality and that all work products conform to agreed upon standards for the process, project, and products.

**Requirements Management:** Requirements Management is a set of activities that help project team to identify; control, and track requirements and changes as project proceeds.Many of these activities are identical to those that make up the software configuration management (SCM) process. Requirements are first identified, tagged with a unique identifier and classified by type (functional, data, behavioral, interface, or output).Traceability tables (e.g., features, source, dependency, subsystem, interface) are developed and updated any time a requirement is modified).Database systems are invaluable in helping software teams track requirement changes.

Some of the Traceability Tables are:

* Features traceability table shows how requirements relate to customer observable features
* Source traceability table identifies source of each requirement
* Dependency traceability table indicates the relations among requirements
* Subsystem traceability table categorizes the requirements by subsystem
* Interface traceability table shows requirement relations to internal and external interfaces

**1.2 Initiating Requirements Engineering Process:**

The steps required to initiate requirements engineering are as follows:

* Identifying the stakeholders

A stakeholder is anyone who is benefited in a direct or indirect way from the system which is being developed. Business operations managers, product managers, marketing people, end- users, product engineers, software engineers, support and maintenance etc are all stake holders. Different stake holders have a different view of the system and achieve different benefits from the system when it is developed. Thus a list of stakeholders who contribute input as requirements are made which may also grow in the meantime.

* **Recognizing the existence of multiple stakeholder viewpoints**

The requirements of the system are explored from different points of view.The information from multiple view points are collected. The emerging requirements may be conflicting. Thus, the requirements engineer categorizes the information from the stakeholders in such a way that it allow the decision makers to choose an internally consistent set of requirements of a system.

* **Working towards collaboration among stakeholders**

The requirements engineer has to identify the areas of commonality and conflicts and inconsistency of requirements among stakeholders. The stake holders collaborate and provide their view of requirements. Some senior person like business manager or senior techonologist makes the final decision about requirements.

* **Asking the first questions**

These context-free questions focus on customer, stakeholders, overall goals, and benefits of the system .It helps to identify all stakeholders who have interest in the software.

* + Who is behind the request for work?
  + Who will use the solution?
  + What will be the economic benefit of a successful solution?
  + Is there another source for the solution needed?

The next set of questions enables developer to better understand the problem and the customer's perceptions of the solution.

* + How would you characterize good output form a successful solution?
  + What problem(s) will this solution address?
  + Can you describe the business environment in which the solution will be used?
  + Will special performance constraints affect the way the solution is approached?

The final set of questions focuses on communication effectiveness

* + Are you the best person to give "official" answers to these questions?
  + Are my questions relevant to your problem?
  + Am I asking too many questions?
  + Can anyone else provide additional information?
  + Should I be asking you anything else?

These questions initiate the communication that is essential for successful elicitation.

**Chapter-2**

**BUILDING ANALYSIS MODEL**

**Introduction:** The analysis model is the first technical representation of a system. Analysis modeling uses a combination of text and diagrams to represent software requirements (data, function, and behavior) in an understandable way. Building analysis models helps make it easier to uncover requirement inconsistencies and omissions.

Two types of analysis modeling are commonly used: structured analysis and object-oriented analysis. Scenario-based modeling represents the system from the user's point of view. Flow-oriented modeling shows how data are transformed inside the system by processing functions. Class-based modeling defines objects, attributes, and relationships. Behavioral modeling uses state transition diagrams to show the impact of events on the system states. Analysis work products must be reviewed for completeness, correctness, and consistency.

**2.1 Guidelines that were needed to be followed while building Analysis Model:**

* The Products of Analysis must be highly maintainable, especially the software requirements specification Document.
* Problems of size must be dealt with using an effective method of partitioning.
* Graphics should be used whenever possible.
* The logical (essential) and physical (implementation) considerations must be differentiated.
* Must find something to help with requirements partitioning and document the partitioning before specification.
* Some way to track and evaluate user interfaces must be devised.
* Tools that describe logic and policy better than narrative text must be devised.

The analysis model bridges the gap between the system level description which describes the overall system functionality and software design. The system functionality is achieved by applying the software, hardware, data, human and other system elements. The software design describes the software application architecture, user interface and component level structure. The Analysis model must achieve three Primary Objectives. They are

* To describe what the customer requires.
* To establish a basis for the creation of a software design.
* To devise a set of requirements that can be validated once the software is built.

**2.2 Data Modeling Concepts:**

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



**Data objects, attributes and relationships**

1. **Data Objects:**

A data object is a repsentation of any composite information that must be understood by the software. A data object can be an external entity (e.g., anything that produces or consumes information), a thing (e.g., a report or a display), an occurrence (e.g., a telephone call) or event (e.g., an alarm), a role (e.g., salesperson), an organizational unit (e.g., accounting department), a place (e.g., a warehouse), or a structure (e.g., a file). For example, a person or a car (Figure 12.2) can be viewed as a data object in the sense that either can be defined in terms of a set of attributes. The data object description incorporates the data object and all of its attributes. A data object encapsulates data only—there is no reference within a data object to operations that act on the data. Therefore, the data object can be represented as a table.

1. **Data Attributes:**

Attributes define the properties of a data object. They are used to name a data object instance, describe its characteristics, or make reference to another data object. The set of attributes that is appropriate for a given data object is determined through an understanding of the problem context. The attributes for **car** might serve well for an application that would be used by a Department of Motor Vehicles, but these attributes would be useless for an automobile company that needs manufacturing control software. In the latter case, the attributes for **car** might also include ID number, body type and color, but many additional attributes (e.g., interior code, drive train type, trim package designator, transmission type) would have to be added to make car a meaningful object in the manufacturing control context.



Tabular representation of data objects

1. **Relationships:**

Data objects are connected to one another in different ways. Relationships indicate the manner in which data objects are connected to one another. Consider two data objects, **person** and **car**. A connection is established between **person** and **car** because the two objects are related. We can define a set of object/relationship pairs that define the relevant relationships.

**For example**,

• A person *owns* a car

• A person *is insured to* *drive* a car.

The relationship *owns* and *insured to drive* define the relevant connections between **person** and **car**. The directionality of the relationship reduces the ambiguity and misinterpretation.

1. **Cardinality and Modality:**

The elements of data modeling—data objects, attributes, and relationships— provide the basis for understanding the information domain of a problem. We must understand how many occurrences of **object X** are related to how many occurrences of **object Y.** This leads to a data modeling concept called *cardinality.*

**Cardinality.** The data model must be capable of representing the number of occurrences objects in a given relationship. Cardinality is the specification of the number of occurrences of one [object] that can be related to the number of occurrences of another [object].

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

• One-to-many (l:N)—One occurrence of [object] 'A' can relate to one or many occurrences of [object] 'B,' but an occurrence of 'B' can relate to only one occurrence of 'A.'

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

• Many-to-many (M:N)—An occurrence of [object] 'A' can relate to one or more occurrences

of 'B,' while an occurrence of 'B' can relate to one or more occurrences of 'A.'

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

Cardinality specifies how the number of occurrences of one object are related to the number of occurrences of another object (1:1, 1:N, M:N)

**Modality:**

The **modality** provides an indication of whether or not a particular data object must participate in the relationship.The *modality* of a relationship is 0 if there is no explicit need for the relationship to occur or the relationship is optional. The modality is 1 if an occurrence of the relationship is mandatory. To illustrate, consider software that is used by a local telephone company to process requests for field service. A customer indicates that there is a problem. If the problem is diagnosed as relatively simple, a single repair action occurs. However, if the problem is complex, multiple repair actions may be required. Thus, Modality is zero (0) for an optional object relationship and one (1) for a mandatory relationship.



The relationship, cardinality, and modality between the data objects **customer** and **repair action.**

**2.3 Class based Modeling:**

The class-based elements of an analysis model are classes and objects, attributes, operations, packages, CRC models collaboration diagrams. A formal guidelines for identifying these elements for an actual problem is as follows:

1. **Identifying Analysis Classes**

* The problem statement is examined and nouns are found that fit the following categories (i.e., grammatical parse)
  + *External entities* (systems, devices, people) that produce or consume information to be used by the computer-based system
  + *Things* (e.g., reports, displays, letters, signals) that are part of the information domain for the problem.
  + *Occurrences* or *events*5 (e.g., a property transfer or the completion of a series of robot movements) that occur within the context of system operation.
  + *Roles* (e.g., manager, engineer, salesperson) played by people who interact with the system.
  + *Organizational units* (e.g., division, group, team) that are relevant to an application
  + *Places* (e.g., laboratory, control room, aircraft cockpit) that establish the context of the problem and the overall function of the system.
  + *Structures* (e.g., sensors, vehicles, computers) that define a class of objects or in the extreme, related classes of objects.



Class manifestation

*SafeHome* security system example is used to illustrate how analysis classes might be defined during the early stages of analysis by performing a "grammatical parse" on a processing narrative for the *SafeHome* system. The processing narrative is reproduced:

*SafeHome* software enables the homeowner to configure the security system when it is installed, monitors all sensors connected to the security system, and interacts with the homeowner through a keypad and function keys contained in the *SafeHome* control panel. During installation, the *SafeHome* control panel is used to "program" and configure the system. Each sensor is assigned a number and type, a master password is programmed for arming and disarming the system, and telephone number(s) are input for dialing when a sensor event occurs. When a sensor event is sensed by the software, it rings an audible alarm attached to the system. After a delay time that is specified by the homeowner during system configuration activities, the software dials a telephone number of a monitoring service, provides information about the location, reporting and the nature of the event that has been detected. The number will be redialed every 20 seconds until telephone connection is obtained. All interaction with *SafeHome* is managed by a user-interaction subsystem that reads input provided through the keypad and function keys, displays prompting messages on the LCD display, displays system status information on the LCD display. Keyboard interaction takes the following form . . .

Extracting the nouns, we can propose a number of potential objects:

|  |  |
| --- | --- |
| **Potential Object/Class** | **General Classification** |
| Homeowner | role or external entity |
| sensor | external entity |
| control panel | external entity |
| installation | occurrence |
| system (alias security system) | thing |
| number, type | not objects, attributes of sensor |
| master password | thing |
| telephone number | thing |
| sensor event | occurrence |
| audible alarm | external entity |
| monitoring service | organizational unit or external entity |

The list would be continued until all nouns in the processing narrative have been considered. Each entry in the list a *potential* object. Coad and Yourdon suggest six selection characteristics that should be used as an analyst considers each potential object for inclusion in the analysis model

* + Contains information that should be retained
  + Provides needed services
  + Contains multiple attributes
  + Has common set of attributes that apply to all class instances
  + Has common set of operations that apply to all object instances
  + Represents external entity that produces or consumes information

The potential class should satisfy all or almost all of these characteristics for inclusion into an analysis model. A later evaluation may sometimes discard a class. The first step of class-based modeling is definition of objects, and decisions (even subjective ones) must be made. With this in mind, we apply the selection characteristics to the list of potential *SafeHome* objects:

|  |  |
| --- | --- |
| **Potential Object/Class** | **Characteristic Number That Applies** |
| homeowner | rejected: 1, 2 fail even though 6 applies |
| sensor | accepted: all apply |
| control panel | accepted: all apply |
| installation | rejected |
| system (alias security system) | accepted: all apply |
| number, type | rejected: 3 fails, attributes of sensor |
| master password | rejected: 3 fails |
| telephone number | rejected: 3 fails |
| sensor event | accepted: all apply |
| audible alarm | accepted: 2, 3, 4, 5, 6 apply |
| monitoring service | rejected: 1, 2 fail even though 6 applies |

It should be noted that (1) the preceding list is not all-inclusive, additional objects would have to be added to complete the model; (2) some of the rejected potential objects will become attributes for those objects that were accepted (e.g., number and type are attributes of **sensor**, and master password and telephone number may become attributes of **system**); (3) different statements of the problem might cause different "accept or reject" decisions to be made (e.g., if each homeowner had an individual password or was identified by voice print, the **homeowner** object would satisfy characteristics 1 and 2 and would have been accepted).

1. **Specifying Class Attributes**

Attributes are the set of dataobjects that fully define the class within the context of the problem. To develop a meaningful set of attributes for an object, the processing narrative (or statement of scope) for the problem must be studied and things that reasonably "belong" to the object are selected. In addition, the following question should be answered for each object: "What data items (composite and/or elementary) fully define this object in the context of the problem at hand?"

Considering the **system** object defined for *SafeHome.*  the homeowner can configure the security system to reflect sensor information, alarm response information, activation/deactivation information, identification information, and so forth. the composite data items can be represented in the following manner:

**sensor information = sensor type + sensor number + alarm threshold**

**alarm response information = delay time + telephone number + alarm type**

**activation/deactivation information = master password + number of allowable tries + temporary password identification information = system ID + verification phone number + system status**

Each of the data items to the right of the equal sign could be further defined to an elementary

level, but for our purposes, they constitute a reasonable list of attributes for the system object 

Class diagram for the system class

1. **Defining Operations**

Operations define the behavior of an object and an operation changes one or more attribute values that are contained within the object. they can generally be divided into three broad categories: (1) operations that manipulate data in some way (e.g., adding, deleting, reformatting, selecting), (2) operations that perform a computation, and (3) operations that monitor an object for the occurrence of a controlling event. the analyst can again study the processing narrative (or statement of scope)

To identify operations, the grammatical parse is again studied and verbs are isolated. Someof these verbs will be legitimate operations and can be easily connected to a specific object. For example, from the *SafeHome* processing narrative , "sensor is assigned a number and type" or that "a master password is programmed for arming and disarming the system." Indicate a number of things:

• That an *assign* operation is relevant for the **sensor** object.

• That a *program* operation will be applied to the **system** object.

• That *arm* and *disarm* are operations that apply to **system**

Further, it is likely that the operation *program* will be divided into a number of more specific suboperations required to configure the system. For example, *program* implies specifying phone numbers, configuring system characteristics (e.g., creating the sensor table, entering alarm characteristics), and entering password(s). But *program* is specified as a single operation. Thus the verbs in the processing narrative helps identifying operations reasonably belonging to each class .Operations can also be divided into sub operations as needed.

1. **Class-Responsibility-Collaborator (CRC) Modeling :**

*Classresponsibility-collaborator* (CRC) modeling [WIR90] provides a simple means for identifying and organizing the classes that are relevant to system or product requirements. According to Ambler , a CRC model is a collection of standard index cards that represent classes. The cards are divided into three sections. Along the top of the card the name of the class is written. In the body of the card the class responsibilities are listed on the left and the collaborators on the right. In reality, the CRC model may make use of actual or virtual *index cards.* The intent is to develop an organized representation of classes. *Responsibilities* are the attributes and operations that are relevant for the class. A responsibility is “anything the class knows or does” . *Collaborators* are those classes that are required to provide a class with the information needed to complete a responsibility. A collaboration implies either a request for information or a request for some action.

Thus, a set of index cards that represent the system classes are developed, one class per card. Cards are divide into three sections (class name, class responsibilities, class collaborators).Once a complete CRC card set is developed it is reviewed examining the usage scenarios.

**d.i Classes**

Classes must be identified and can be categorized as follows:

Entity Classes or model or business classes are extracted from the statement of the problem. These classes represent things that can be stored in a database and persists throughout the duration of the application.

Boundary Classes are used to create interface that the user sees and interacts with as the software is used.Boundary classes display the entity objects to the users.

Controller classes manage a unit of work from start to finish. Thus, they are designed to manage

* The creation and updation of entity object
* The instantation of the boundary objects as they obtain information from entity objects
* Complex communication between sets and objects, and
* Validation of data communicated between objects/user and application.

**d.i Responsibilities**

The guidelines for Allocating Responsibilities to Classes are as follows:

* System intelligence should be evenly distributed across classes to best address the needs of the problem
* Each responsibility should be stated as generally as possible
* Information and the behavior related to it should reside within the same class.
* Information about one thing should be localized with a single class, not distributed across multiple classes.
* Responsibilities should be shared among related classes, when appropriate

**d.k Collaborations**

Classes fulfill their responsibilities in one of the two ways.

* A class can use its own operations to manipulate its own attributes, thereby fulfilling a particular responsibility
* A class can collaborate with other class.

Collaborations are identified by determining whether a class can fulfill each responsibility itself. If at any time, a class cannot fulfill a responsibility on its own, it needs to interact with another class. Hence , a collaboration. For example, A server object interacts with a client object to fulfill some responsibility.

As part of the activation procedure in the *SafeHome* application, the **control panel** object must determine whether any sensors are open. A responsibility named *determine-sensor- status* is defined. If sensors are open **control panel** must set a status attribute to “not ready.” Sensor information can be acquired from the **sensor** object. Therefore, the responsibility *determine-sensor-status* can be fulfilled only if **control panel** works in collaboration with **sensor**.

To help in the identification of collaborators, the analyst can examine three different generic relationships between classes :(1) the *is-part-of* relationship, (2) the *has-knowledge-of* relationship, and (3) the *depends-upon* relationship. By creating a class-relationship diagram the connections necessary to identify these relationships can be developed. Each of the three generic relations is considered briefly in the paragraphs that follow.

All classes that are part of an aggregate class are connected to the aggregate class via an is-part-of relationship. Consider the classes defined for the video game, the class **player-body** is-part-of **player,** as are **player-arms, player-legs,** and **player-head.**

When one class must acquire information from another class, the has-knowledge of relationship is established. The *determine-sensor-status* responsibility noted earlier is an example of a has-knowledge-of relationship. The depends-upon relationship implies that two classes have a dependency that is not achieved by has-knowledge-of or is-part-of. For example, **player-head** must always be connected to **player- body** (unless the video game is particularly violent), yet each object could exist without direct knowledge of the other. An attribute of the **player-head** object called center-position is determined from the center position of **player-body**. This information is obtained via a third object, **player**, that acquires it from **player-body**. Hence, **player-head** depends-upon **player-body**. In all cases, the collaborator class name is recorded on the CRC model index card next to the responsibility that has spawned the collaboration. Therefore, the index card contains a list of responsibilities and the corresponding collaborations that enable the responsibilities to be fulfilled

After developing a CRC model, it must be reviewed and continues until all the use-cases have been completed. The review is as follows:

* Each review participant is given a subset of the CRC cards (collaborating cards must be separated)
* All use-case scenarios and use-case diagrams should be organized into categories
* Review leader chooses a use-case scenario and begins reading it out loud
* Each time a named object is read a token is passed to the reviewer holding the object's card
* When the reviewer receives the token, he or she is asked to describe the responsibilities listed on the card
* The group determines whether one of the responsibilities on the card satisfy the use-case requirement or not
* If the responsibilities and collaborations on the index card cannot accommodate the use-case requirements then modifications need to be made to the card set

1. **Associations and Dependencies**

An association defines a relationship between classes. The association may be further defined by indicating multiplicity/cardinality. Multiplicity defines how many of one class are related to how many of another class.Association **multiplicity** or cardinality can be indicated in a UML class diagram (e.g., 0..1, 1..1, 0.., 1..)

Dependency - client/server relationship between two classes ie. When a class depends on another class.Dependency relationships are indicated in class diagrams using stereotype names surrounded by angle brackets (e.g., «stereotype»). Stereotype is an “extensibility mechanism that allows to define a special modeling element whose semantics are custom-defined.

1. **Analysis Packages**

Categorization is an important part of analysis modeling. Various elements of the analysis model are categorisedin a manner that packages them as a grouping-called an analysis package- that is given a representative name. Thus, Analysis packages are made up of classes having the same categorization. In class diagrams , visibility of class elements can be indicated using a + before a package name in each package indicates the public visibility, - private visibility.