

UNIT – II

Requirements engineering:

Designing and building computer software is challenging, creative, and just plain fun. In fact, building software is so compelling that many software developers want to jump right in before they have a clear understanding of what is needed. They argue that things will become clear as they build, that project stakeholders will be able to understand need only after examining early iterations of the software, that things change so rapidly that any attempt to understand requirements in detail is a waste of time, that the bottom line is producing a working program, and that all else is secondary.

The broad spectrum of tasks and techniques that lead to an understanding of requirements is called requirements engineering. From a software process perspective, requirements engineering is a major software engineering action that begins during the communication activity and continues into the modeling activity. It must be adapted to the needs of the process, the project, the product, and the people doing the work.

Requirements engineering builds a bridge to design and construction. But where does the bridge originate? One could argue that it begins at the feet of the project stakeholders (e.g., managers, customers, and end users), where business need is defined, user scenarios are described, functions and features are delineated and project constraints are identified.

Requirements engineering encompasses seven distinct tasks: inception, elicitation, elaboration, negotiation, specification, validation, and management. It is important to note that some of these tasks occur in parallel and all are adapted to the needs of the project.

Requirements engineering tasks:

1. Inception:

How does a software project get started? Is there a single event that becomes the catalyst for a new computer-based system or product, or does the need evolve over time? There are no definitive answers to these questions. In some cases, a casual conversation is all that is needed to precipitate a major software engineering effort. But in general, most projects begin when a business need is identified or a potential new market or service is discovered. Stakeholders from the business community (e.g., business managers, marketing people, and product managers) define a business case for the idea, try to identify the breadth and depth of the market, do a rough feasibility analysis, and identify a working description of the project's

scope. All of this information is subject to change, but it is sufficient to precipitate discussions with the software engineering organization. At project inception, you establish a basic understanding of the problem, the people who want a solution, the nature of the solution that is desired, and the effectiveness of preliminary communication and collaboration between the other stakeholders and the software team

2. Elicitation:

It certainly seems simple enough—ask the customer, the users, and others what the objectives for the system or product are, what is to be accomplished, how the system or product fits into the needs of the business, and finally, how the system or product is to be used on a day-to-day basis. But it isn't simple—it's very hard.

An important part of elicitation is to establish business goals [Cle10]. Your job is to engage stakeholders and to encourage them to share their goals honestly. Once the goals have been captured, a prioritization mechanism should be established, and a design rationale for a potential architecture (that meets stakeholder goals) can be created.

Christel and Kang [Cri92] identify a number of problems that are encountered as elicitation occurs. Problems of scope occur when the boundary of the system is ill-defined or the customers and users specify unnecessary technical detail that may confuse, rather than clarify, overall system objectives. Problems of understanding are encountered when customers and users are not completely sure of what is needed, have a poor understanding of the capabilities and limitations of their computing environment, don't have a full understanding of the problem domain, have trouble communicating needs, omit information that is believed to be "obvious," specify requirements that conflict with the needs of other customers and users, or specify requirements that are ambiguous or untestable. Problems of volatility occur when the requirements change over time.

3. Elaboration:

The information obtained from the customer during inception and elicitation is expanded and refined during elaboration. This task focuses on developing a refined requirements model that identifies various aspects of software function, behavior, and information.

Elaboration is driven by the creation and refinement of user scenarios that describe how the end user (and other actors) will interact with the system. Each user scenario is parsed to extract analysis classes—business domain entities that are visible to the end user. The

attributes of each analysis class are defined, and the services⁴ that are required by each class are identified. The relationships and collaboration between classes are identified, and a variety of supplementary diagrams are produced.

4. Negotiation:

It isn't unusual for customers and users to ask for more than can be achieved, given limited business resources. It's also relatively common for different customers or users to propose conflicting requirements, arguing that their version is "essential for our special needs."

You have to reconcile these conflicts through a process of negotiation. Customers, users, and other stakeholders are asked to rank requirements and then discuss conflicts in priority. Using an iterative approach that prioritizes requirements, assesses their cost and risk, and addresses internal conflicts, requirements are eliminated, combined, and/or modified so that each party achieves some measure of satisfaction.

5. Specification:

In the context of computer-based systems (and software), the term specification means different things to different people. A specification can be a written document, a set of graphical models, a formal mathematical model, and a collection of usage scenarios, a prototype, or any combination of these.

Some suggest that a "standard template" [Som97] should be developed and used for a specification, arguing that this leads to requirements that are presented in a consistent and therefore more understandable manner. However, it is sometimes necessary to remain flexible when a specification is to be developed. For large systems, a written document, combining natural language descriptions and graphical models may be the best approach. However, usage scenarios may be all that are required for smaller products or systems that reside within well-understood technical environments.

6. Validation: The work products produced as a consequence of requirements engineering are assessed for quality during a validation step. Requirements validation examines the specification⁵ to ensure that all software requirements have been stated unambiguously; that inconsistencies, omissions, and errors have been detected and corrected; and that the work products conform to the standards established for the process, the project, and the product.

The primary requirements validation mechanism is the technical review. The review team that validates requirements includes software engineers, customers, users, and other stakeholders who examine the specification looking for errors in content or interpretation,

areas where clarification may be required, missing information, inconsistencies, conflicting requirements, or unrealistic (unachievable) requirements.

To illustrate some of the problems that occur during requirements validation, consider two seemingly innocuous requirements:

1. The software should be user friendly.
2. The probability of a successful unauthorized database intrusion should be less than 0.0001.

7. Requirement management: Requirements for computer-based systems change, and the desire to change requirements persists throughout the life of the system. Requirements management is a set of activities that help the project team identify, control, and track requirements and changes to requirements at any time as the project proceeds. Many of these activities are identical to the software configuration management (SCM) techniques.

Establishing the Groundwork:

In an ideal setting, stakeholders and software engineers work together on the same team. In such cases, requirements engineering is simply a matter of conducting meaningful conversations with colleagues who are well-known members of the team. But reality is often quite different.

Customer(s) or end users may be located in a different city or country, may have only a vague idea of what is required. We discuss the steps required to establish the groundwork for an understanding of Software requirements—to get the project started in a way that will keep it moving forward toward a successful solution:

1. Identifying Stakeholders: Sommerville and Sawyer [Som97] define a stakeholder as “anyone who benefits in a direct or indirect way from the system which is being developed.” We have already identified the usual suspects: business operations managers, product managers, marketing people, internal and external customers, end users, consultants, product engineers, software engineers, support and maintenance engineers, and others. Each stakeholder has a different view of the system, achieves different benefits when the system is successfully developed, and is open to different risks if the development effort should fail.

At inception, you should create a list of people who will contribute input as requirements are elicited (Section 8.3). The initial list will grow as stakeholders are contacted because every stakeholder will be asked: “Whom else do you think I should talk to?”

2. Recognizing multiple viewpoints: Because many different stakeholders exist, the requirements of the system will be explored from many different points of view. For example, the marketing group is interested in functions and features that will excite the potential market, making the new system easy to sell. Business managers are interested in a feature set that can be built within budget and that will be ready to meet defined market windows. End users may want features that are familiar to them and that are easy to learn and use. Software engineers may be concerned with functions that are invisible to nontechnical stakeholders but that enable an infrastructure that supports more marketable functions and features. Support engineers may focus on the maintainability of the software.

Each of these constituencies (and others) will contribute information to the requirements engineering process. As information from multiple viewpoints is collected, emerging requirements may be inconsistent or may conflict with one another. You should categorize all stakeholder information (including inconsistent and conflicting requirements) in a way that will allow decision makers to choose an internally consistent set of requirements for the system

3. Working towards collaboration: If five stakeholders are involved in a software project, you may have five (or more) different opinions about the proper set of requirements. Throughout earlier chapters, we have noted that customers (and other stakeholders) should collaborate among themselves (avoiding petty turf battles) and with software engineering practitioners if a successful system is to result. But how is this collaboration accomplished?

The job of a requirements engineer is to identify areas of commonality (i.e., requirements on which all stakeholders agree) and areas of conflict or inconsistency (i.e., requirements that are desired by one stakeholder but conflict with the needs of another stakeholder). It is, of course, the latter category that presents a challenge.

Collaboration does not necessarily mean that requirements are defined by committee. In many cases, stakeholders collaborate by providing their view of requirements, but a strong “project champion” (e.g., a business manager or a senior technologist) may make the final decision about which requirements make the cut.

4. Asking the First Questions: Questions asked at the inception of the project should be “context free” [Gau89]. The first set of context-free questions focuses on the customer and other stakeholders, the overall project goals and benefits. For example, you might ask:

- Who is behind the request for this work?
- Who will use the solution?

- What will be the economic benefit of a successful solution?
- Is there another source for the solution that you need?

These questions help to identify all stakeholders who will have interest in the software to be built. In addition, the questions identify the measurable benefit of a successful implementation and possible alternatives to custom software development

Eliciting requirements:

Requirements elicitation (also called requirements gathering) combines elements of problem solving, elaboration, negotiation, and specification. In order to encourage a collaborative, team-oriented approach to requirements gathering, stakeholders work together to identify the problem, propose elements of the solution, negotiate different approaches, and specify a preliminary set of solution requirements.

1. Collaborative Requirements Gathering:

Many different approaches to collaborative requirements gathering have been proposed. Each makes use of a slightly different scenario, but all apply some variation on the following basic guidelines:

- Meetings (either real or virtual) are conducted and attended by both software engineers and other stakeholders.
- Rules for preparation and participation are established.
- An agenda is suggested that is formal enough to cover all important points but informal enough to encourage the free flow of ideas.
- A “facilitator” (can be a customer, a developer, or an outsider) controls the meeting.
- A “definition mechanism” (can be work sheets, flip charts, or wall stickers or an electronic bulletin board, chat room, or virtual forum) is used.

The goal is to identify the problem, propose elements of the solution, negotiate different approaches, and specify a preliminary set of solution requirements.


A one- or two-page “product request” is generated during inception. A meeting place, time, and date are selected; a facilitator is chosen; and attendees from the software team and other stakeholder organizations are invited to participate. The product request is distributed to all attendees before the meeting date.

As an example,¹⁰ consider an excerpt from a product request written by a marketing person involved in the **SafeHome** project. This person writes the following narrative about the home security function that is to be part of **SafeHome**:

Our research indicates that the market for home management systems is growing at a rate of 40 percent per year. The first **SafeHome** function we bring to market should be the home security function. Most people are familiar with “alarm systems” so this would be an easy sell.

The home security function would protect against and/or recognize a variety of undesirable “situations” such as illegal entry, fire, flooding, carbon monoxide levels, and others. It’ll use our wireless sensors to detect each situation, can be programmed by the homeowner, and will automatically telephone a monitoring agency when a situation is detected.

SAFEHOME



Conducting a Requirements-Gathering Meeting

The scene: A meeting room. The first requirements-gathering meeting is in progress.

The players: Jamie Lazar, software team member; Vinod Raman, software team member; Ed Robbins, software team member; Doug Miller, software engineering manager; three members of marketing; a product engineering representative; and a facilitator.

The conversation:

Facilitator (pointing at whiteboard): So that’s the current list of objects and services for the home security function.

Marketing person: That about covers it from our point of view.

Vinod: Didn’t someone mention that they wanted all *SafeHome* functionality to be accessible via the Internet? That would include the home security function, no?

Marketing person: Yes, that’s right . . . we’ll have to add that functionality and the appropriate objects.

Facilitator: Does that also add some constraints?

Jamie: It does, both technical and legal.

Production rep: Meaning?

Jamie: We better make sure an outsider can’t hack into the system, disarm it, and rob the place or worse. Heavy liability on our part.

Doug: Very true.

Marketing: But we still need that . . . just be sure to stop an outsider from getting in.

Ed: That’s easier said than done and . . .

Facilitator (interrupting): I don’t want to debate this issue now. Let’s note it as an action item and proceed. (Doug, serving as the recorder for the meeting, makes an appropriate note.)

Facilitator: I have a feeling there’s still more to consider here. (The group spends the next 20 minutes refining and expanding the details of the home security function.)

2. Quality Function Deployment:

Quality function deployment (QFD) is a quality management technique that translates the needs of the customer into technical requirements for software. QFD “concentrates on maximizing customer satisfaction from the software engineering process” [Zul92]. To accomplish this, QFD emphasizes an understanding of what is valuable to the customer and then deploys these values throughout the engineering process.

Within the context of QFD, normal requirements identify the objectives and goals that are stated for a product or system during meetings with the customer. If these requirements are present, the customer is satisfied. Expected requirements are implicit to the product or system and may be so fundamental that the customer does not explicitly state them. Their absence will be a cause for significant dissatisfaction. Exciting requirements go beyond the customer's expectations and prove to be very satisfying when present.

Although QFD concepts can be applied across the entire software process [Par96a]; specific QFD techniques are applicable to the requirements elicitation activity. QFD uses customer interviews and observation, surveys, and examination of historical data (e.g., problem reports) as raw data for the requirements gathering activity. These data are then translated into a table of requirements—called the customer voice table—that is reviewed with the customer and other stakeholders. A variety of diagrams, matrices, and evaluation methods are then used to extract expected requirements and to attempt to derive exciting requirements.

As requirements are gathered, an overall vision of system functions and features begin to materialize. However, it is difficult to move into more technical software engineering activities until you understand how these functions and features will be used by different classes of end users. To accomplish this, developers and users can create a set of scenarios that identify a thread of usage for the system to be constructed. The scenarios, often called use cases [Jac92], provide a description of how the system will be used.

3. Usage Scenarios:

As requirements are gathered, an overall vision of system functions and features begin to materialize. However, it is difficult to move into more technical software engineering activities until you understand how these functions and features will be used by different classes of end users. To accomplish this, developers and users can create a set of scenarios that identify a thread of usage for the system to be constructed. The scenarios, often called use cases [Jac92], provide a description of how the system will be used.



Developing a Preliminary User Scenario

The scene: A meeting room, continuing the first requirements gathering meeting.

The players: Jamie Lazar, software team member; Vinod Raman, software team member; Ed Robbins, software team member; Doug Miller, software engineering manager; three members of marketing; a product engineering representative; and a facilitator.

The conversation:

Facilitator: We've been talking about security for access to *SafeHome* functionality that will be accessible via the Internet. I'd like to try something. Let's develop a usage scenario for access to the home security function.

Jamie: How?

Facilitator: We can do it a couple of different ways, but for now, I'd like to keep things really informal. Tell us (he points at a marketing person) how you envision accessing the system.

Marketing person: Um . . . well, this is the kind of thing I'd do if I was away from home and I had to let someone into the house, say a housekeeper or repair guy, who didn't have the security code.

Facilitator (smiling): That's the reason you'd do it . . . tell me how you'd actually do this.

Marketing person: Um . . . the first thing I'd need is a PC. I'd log on to a website we'd maintain for all users of *SafeHome*. I'd provide my user ID and . . .

Vinod (interrupting): The Web page would have to be secure, encrypted, to guarantee that we're safe and . . .

Facilitator (interrupting): That's good information, Vinod, but it's technical. Let's just focus on how the end user will use this capability. OK?

Vinod: No problem.

Marketing person: So as I was saying, I'd log on to a website and provide my user ID and two levels of passwords.

Jamie: What if I forget my password?

Facilitator (interrupting): Good point, Jamie, but let's not address that now. We'll make a note of that and call it an *exception*. I'm sure there'll be others.

Marketing person: After I enter the passwords, a screen representing all *SafeHome* functions will appear. I'd select the home security function. The system might request that I verify who I am, say, by asking for my address or phone number or something. It would then display a picture of the security system control panel along with a list of functions that I can perform—arm the system, disarm the system, disarm one or more sensors. I suppose it might also allow me to reconfigure security zones and other things like that, but I'm not sure.

(As the marketing person continues talking, Doug takes copious notes; these form the basis for the first informal usage scenario. Alternatively, the marketing person could have been asked to write the scenario, but this would be done outside the meeting.)

4. Elicitation Work products:

The work products produced as a consequence of requirements elicitation will vary depending on the size of the system or product to be built. For most systems, the work products include:

- (1) A statement of need and feasibility,
- (2) A bounded statement of scope for the system or product,
- (3) A list of customers, users, and other stakeholders who participated in requirements elicitation,
- (4) A description of the system's technical environment,
- (5) A list of requirements (preferably organized by function) and the domain constraints that applies to each,
- (6) A set of usage scenarios that provide insight into the use of the system or product under different operating conditions, and
- (7) Any prototypes developed to better define requirements. Each of these work products is reviewed by all people who have participated in requirements elicitation.

Developing use cases:

Alistair Cockburn [Coc01b] notes that “a use case captures a contract . . . [that] describes the system’s behaviour under various conditions as the system responds to a request from one of its stakeholders . . .” In essence, a use case tells a stylized story about how an end user (playing one of a number of possible roles) interacts with the system under a specific set of circumstances. The story may be narrative text, an outline of tasks or interactions, a template-based description, or a diagrammatic representation. Regardless of its form, a use case depicts the software or system from the end user’s point of view.

The first step in writing a use case is to define the set of “actors” that will be involved in the story. Actors are the different people (or devices) that use the system or product within the context of the function and behaviour that is to be described. Actors represent the roles that people (or devices) play as the system operates. Defined somewhat more formally, an actor is anything that communicates with the system or product and that is external to the system itself. Every actor has one or more goals when using the system.

It is important to note that an actor and an end user are not necessarily the same thing. A typical user may play a number of different roles when using a system, whereas an actor represents a class of external entities (often, but not always, people) that play just one role in the context of the use case. As an example, consider a machine operator (a user) who interacts with the control computer for a manufacturing cell that contains a number of robots and numerically controlled machines. After careful review of requirements, the software for the control computer requires four different modes (roles) for interaction: programming mode, test mode, monitoring mode, and troubleshooting mode. Therefore, four actors can be defined: programmer, tester, monitor, and trouble shooter. In some cases, the machine operator can play all of these roles. In others, different people may play the role of each actor.

Because requirements elicitation is an evolutionary activity, not all actors are identified during the first iteration. It is possible to identify primary actors [Jac92] during the first iteration and secondary actors as more is learned about the system. Primary actors interact to achieve required system function and derive the intended benefit from the system. They work directly and frequently with the software. Secondary actors support the system so that primary actors can do their work.

Once actors have been identified, use cases can be developed. Jacobson [Jac92] suggests a number of questions that should be answered by a use case:

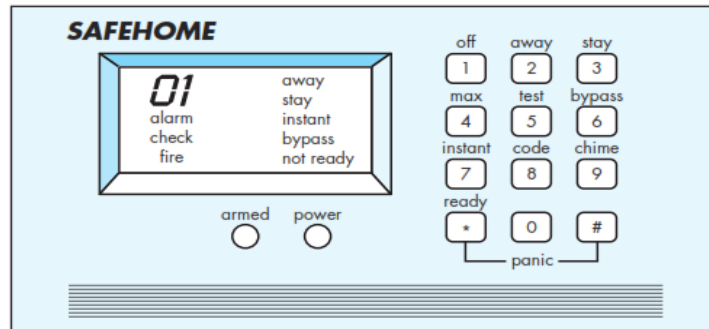
- Who is the primary actor, the secondary actor(s)?
- What are the actor's goals?
- What preconditions should exist before the story begins?
- What main tasks or functions are performed by the actor?
- What exceptions might be considered as the story is described?
- What variations in the actor's interaction are possible?
- What system information will the actor acquire, produce, or change?
- Will the actor have to inform the system about changes in the external environment?
- What information does the actor desire from the system?
- Does the actor wish to be informed about unexpected changes?

Recalling basic **SafeHome** requirements, we define four actors: homeowner (a user), setup manager (likely the same person as homeowner, but playing a different role), sensors (devices attached to the system), and the monitoring and response subsystem (the central station that monitors the **SafeHome** home security function). For the purposes of this example, we consider only the home-owner actor. The homeowner actor interacts with the home security function in a number of different ways using either the alarm control panel or a PC. The homeowner (1) enters a password to allow all other interactions, (2) inquires about the status of a security zone, (3) inquires about the status of a sensor, (4) presses the panic button in an emergency, and (5) activates/deactivates the security system.

Considering the situation in which the homeowner uses the control panel, the basic use case for system activation follows:

1. The homeowner If the system is not ready, a not ready message is displayed on the LCD display, and the homeowner must physically close windows or doors so that the not ready message disappear, observes the **SafeHome** control panel (Figure) to determine if the system is ready for input.

SafeHome
control panel



2. The homeowner uses the keypad to key in a four-digit password. The password is compared with the valid password stored in the system. If the password is incorrect, the control panel will beep once and reset itself for additional input. If the password is correct, the control panel awaits further action.
3. The homeowner selects and keys in stay or away (see Figure 8.1) to activate the system. Stay activates only perimeter sensors (inside motion detecting sensors are deactivated). Away activates all sensors.
4. When activation occurs, a red alarm light can be observed by the homeowner.

The basic use case presents a high-level story that describes the interaction between the actor and the system.

In many instances, uses cases are further elaborated to provide considerably more detail about the interaction. For example, Cockburn [Coc01b] suggests the following template for detailed descriptions of use cases:

Use case: Initiate Monitoring

Primary actor: Homeowner.

Goal in context: To set the system to monitor sensors when the homeowner leaves the house or remains inside

Preconditions: System has been programmed for a password and to recognize various sensors

Trigger: The homeowner decides to “set” the system, that is, to turn on the alarm functions

Scenario:

1. Homeowner: observes control panel
2. Homeowner: enters password
3. Homeowner: selects “stay” or “away”
4. Homeowner: observes read alarm light to indicate that **SafeHome** has been armed

Exceptions:

1. **Control panel is not ready:** homeowner checks all sensors to determine which are open; closes them.
2. **Password is incorrect** (control panel beeps once): homeowner re-centers correct password.
3. **Password not recognized:** monitoring and response subsystem must be contacted to reprogram password.
4. **Stay is selected:** control panel beeps twice and a stay light is lit; perimeter sensors are activated.
5. **Away is selected:** control panel beeps three times and an away light is lit; all sensors are activated.

Priority:	Essential, must be implemented
When available:	First increment
use:	Frequency of
actor:	Many times per day
	Channel to
	Via control panel interface
Secondary actors:	Support technician, sensors

Channels to secondary actors:

Support technician: phone line

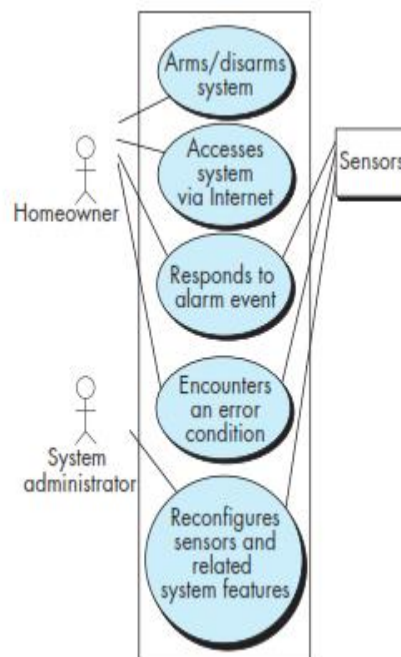
Sensors: hardwired and radio frequency interfaces

Open issues:

1. Should there be a way to activate the system without the use of a password or with an abbreviated password?
2. Should the control panel display additional text messages?
3. How much time does the homeowner have to enter the password from the time the first key is pressed?
4. Is there a way to deactivate the system before it actually activates?

Use cases for other homeowner interactions would be developed in a similar manner. It is important to review each use case with care. If some element of the interaction is ambiguous, it is likely that a review of the use case will indicate a problem.

UML use case diagram for
SafeHome
home security function



Building the Analysis model:

The intent of the analysis model is to provide a description of the required informational, functional, and behavioral domains for a computer-based system. The model changes dynamically as you learn more about the system to be built, and other stakeholders understand more about what they really require. For that reason, the analysis model is a snapshot of requirements at any given time. You should expect it to change.

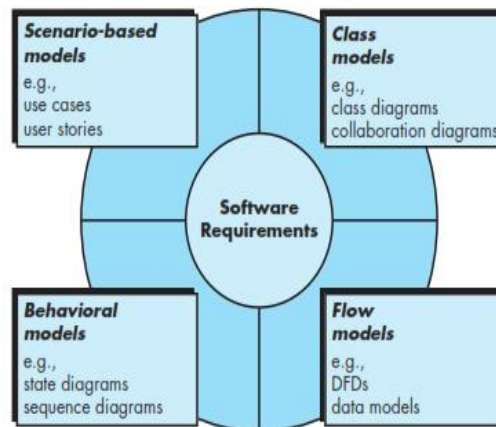
As the analysis model evolves, certain elements will become relatively stable, providing a solid foundation for the design tasks that follow. However, other elements of the model may be more volatile, indicating that stakeholders do not yet fully understand requirements for the system.

Elements of the Analysis Model:

There are many different ways to look at the requirements for a computer-based system. Some software people argue that it's best to select one mode of representation (e.g., the use case) and apply it to the exclusion of all other modes. Other practitioners believe that it's worthwhile to use a number of different modes of representation to depict the analysis model. Different modes of representation force you to consider requirements from different viewpoints—an approach that has a higher probability of uncovering omissions,

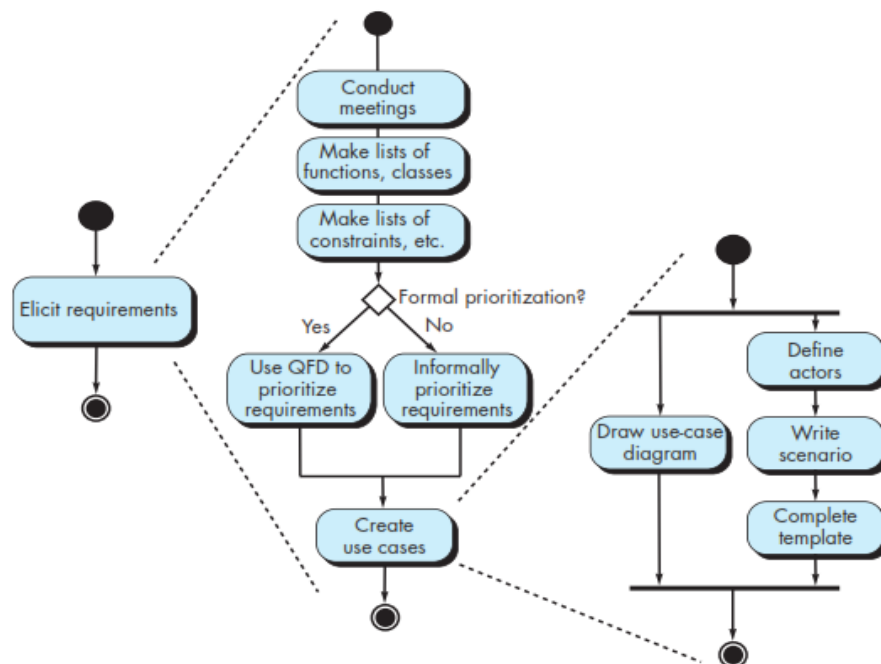
inconsistencies, and ambiguity. A set of generic elements is common to most analysis models.

Elements of the analysis model



Scenario-based elements. The system is described from the user's point of view using a scenario-based approach. For example, basic use cases and their corresponding use case diagrams evolve into more elaborate template-based use cases. Scenario-based elements of the requirements model are often the first part of the model that is developed. As such, they serve as input for the creation of other modeling elements. Figure 8.3 depicts a UML activity diagram for eliciting requirements and representing them using use cases. Three levels of elaboration are shown, culminating in a scenario-based representation.

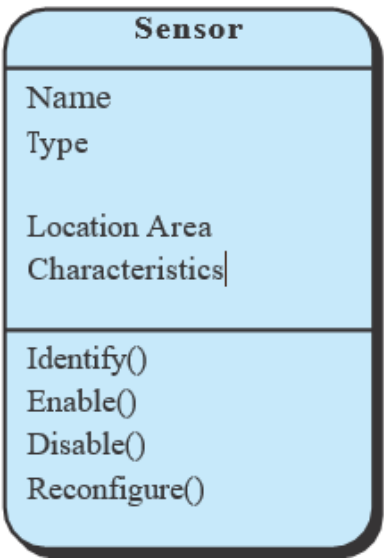
UML activity diagrams for eliciting requirements



Class-based elements. Each usage scenario implies a set of objects that are manipulated as an actor interacts with the system. These objects are categorized into

classes—a collection of things that have similar attributes and common behaviors. For example, a UML class diagram can be used to depict a Sensor class for the **SafeHome** security function (Figure 8.4). Note that the diagram lists the attributes of sensors (e.g., name, type) and the operations (e.g., identify, enable) that can be applied to modify these attributes. In addition to class diagrams, other analysis modeling elements depict the manner in which classes collaborate with one another and the relationships and interactions between classes.

Class diagram
for sensor

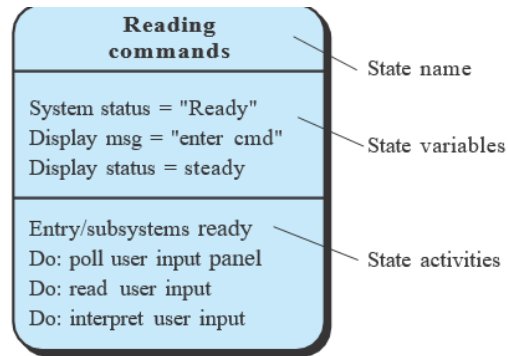


Behavioral elements. The behavior of a computer-based system can have a profound effect on the design that is chosen and the implementation approach that is applied. Therefore, the requirements model must provide modeling elements that depict behavior.

The state diagram is one method for representing the behaviour of a system by depicting its states and the events that cause the system to change state. A state is any observable mode of behavior. In addition, the state diagram indicates what actions (e.g., process activation) are taken as a consequence of a particular event.

To illustrate the use of a state diagram, consider software embedded within the **SafeHome** control panel that is responsible for reading user input. A simplified UML state diagram is shown below.

UML state
diagram
notation



Negotiating requirements:

In an ideal requirements engineering context, the *inception, elicitation, and elaboration tasks determine customer requirements in sufficient detail* to proceed to subsequent software engineering activities. Unfortunately, this rarely happens. In reality, you may have to enter into a negotiation with one or more stakeholders. In most cases, *stakeholders are asked to balance functionality, performance, and other product or system characteristics against cost and time-to-market*. The intent of this negotiation is to develop a project plan that meets stakeholder needs while *at the same time reflecting the real-world constraints (e.g., time, people, and budget)* that have been placed on the software team.

The best negotiations strive for a “win-win” result. That is, stakeholders win by getting the system or product that satisfies the majority of their needs and you (as a member of the software team) win by working to realistic and achievable budgets and deadlines.

Boehm [Boe98] defines a set of negotiation activities at the beginning of each software process iteration. Rather than a single customer communication activity, the following activities are defined:

1. Identification of the system or subsystem’s key stakeholders.
2. Determination of the stakeholders’ “win conditions.”
2. Negotiation of the stakeholders’ win conditions to reconcile them into a set of win-win conditions for all concerned (including the software team)

Successful completion of these initial steps achieves a win-win result, which becomes the key criterion for proceeding to subsequent software engineering activities.

Validating Requirements:

As each element of the requirements model is created, it is examined for inconsistency, omissions, and ambiguity. The requirements represented by the model are prioritized by stakeholders and grouped within requirements packages that will be implemented as software increments. A review of the requirements model addresses the following questions:

- Is each requirement consistent with the overall objectives for the system or product?
- Have all requirements been specified at the proper level of abstraction?

That is, do some requirements provide a level of technical detail that is inappropriate at this stage?

- Is the requirement really necessary or does it represent an add-on feature that may not be essential to the objective of the system?
- Is each requirement bounded and unambiguous?
- Does each requirement have attribution? That is, is a source (generally, a specific individual) noted for each requirement?
- Do any requirements conflict with other requirements?
- Is each requirement achievable in the technical environment that will house the system or product?
- Is each requirement testable, once implemented?
- Does the requirements model properly reflect the information, function, and behavior of the system to be built?
- Has the requirements model been “partitioned” in a way that exposes progressively more detailed information about the system?
- Have requirements patterns been used to simplify the requirements model? Have all patterns been properly validated? Are all patterns consistent with customer requirements?

These and other questions should be asked and answered to ensure that the requirements model is an accurate reflection of stakeholder needs and that it provides a solid foundation for design

Agile Requirements:

Scenario-Based Modeling:

Although the success of a computer-based system or product is measured in many ways, user satisfaction resides at the top of the list. If you understand how end users (and other actors) want to interact with a system, your software team will be better able to properly characterize requirements and build meaningful analysis and design models. Hence, requirements modeling with UML begins with the creation of scenarios in the form of use cases, activity diagrams, and swimlane diagrams.

1. Creating a preliminary Use Case

Alistair Cockburn characterizes a use case as a “contract for behavior” [Coc01b]. As we discussed in Chapter 8, the “contract” defines the way in which an actor uses a computer-based system to accomplish some goal. In essence, a use case captures the interactions that occur between producers and consumers of information and the system itself. In this section, we examine how use cases are developed as part of the analysis modeling activity.

Earlier we noted that a use case describes a specific usage scenario in straightforward language from the point of view of a defined actor. But how do you know

- (1) What to write about,
- (2) How much to write about it,
- (3) How detailed to make your description, and
- (4) How to organize the description? These are the questions that must be answered if use cases are to provide value as a requirements modeling tool.

What to Write About? The first two requirements engineering tasks—inception and elicitation—provide you with the information you’ll need to begin writing use cases. Requirements-gathering meetings, quality function deployment (QFD), and other requirements engineering mechanisms are used to identify stakeholders, define the scope of the problem, specify overall operational goals, establish priorities, outline all known functional requirements, and describe the things (objects) that will be manipulated by the system.

To begin developing a set of use cases, list the functions or activities performed by a specific actor. You can obtain these from a list of required system functions, through conversations with stakeholders, or by an evaluation of activity diagrams.

The **SafeHome** home surveillance function (subsystem) discussed in the sidebar identifies the following functions (an abbreviated list) that are performed by the homeowner actor:

- Select camera to view
- Request thumbnails from all cameras
- Display camera views in a PC window.
- Control pan and zoom for a specific camera
- Selectively record camera output
- Replay camera output
- Access camera surveillance via the Internet

Use case: Access camera surveillance via the Internet display camera views (ACS-DCV) .

Actor: homeowner

1. The homeowner logs onto the **SafeHome** Products website.
2. The homeowner enters his or her user ID.
3. The homeowner enters two passwords (each at least eight characters in length).
4. The system displays all major function buttons.
5. The homeowner selects the “surveillance” from the major function buttons.
6. The homeowner selects “pick a camera.”
7. The system displays the floor plan of the house.
8. The homeowner selects a camera icon from the floor plan.
9. The homeowner selects the “view” button.
10. The system displays a viewing window that is identified by the camera ID.
11. The system displays video output within the viewing window at one frame per second.

2. Refining a preliminary Use Case :

A description of alternative interactions is essential for a complete understanding of the function that is being described by a use case. Therefore, each step in the primary scenario is evaluated by asking the following questions:

- Can the actor take some other action at this point?
- Is it possible that the actor will encounter some error condition at this point? If so, what might it be?
- Is it possible that the actor will encounter some other behavior at this point (e.g., behavior that is invoked by some event outside the actor’s control)? If so, what might it be?

Answers to these questions result in the creation of a set of secondary scenarios that are part of the original use case but represent alternative behavior. For example, consider *steps 6 and 7* in the primary scenario presented earlier:

6. The homeowner selects “pick a camera.”
7. The system displays the floor plan of the house.

Can the actor take some other action at this point? The answer is yes. Referring to the free-flowing narrative, the actor may choose to view thumbnail snapshots of all cameras simultaneously. Hence, one secondary scenario might be *“View thumbnail snapshots for all cameras.”*

Is it possible that the actor will encounter some error condition at this point? Any number of error conditions can occur as a computer-based system operates. In this context, we consider only error conditions that are likely as a direct result of the action described in step 6 or step 7. Again the answer to the question is yes. A floor plan with camera icons may have never been configured. Hence, selecting “pick a camera” results in an error condition: “No floor plan configured for this house.” This error condition becomes a secondary scenario.

Is it possible that the actor will encounter some other behavior at this point? Again the answer to the question is yes. As *steps 6 and 7* occur, the system may encounter an alarm condition. This would result in the system displaying a special alarm notification (type, location, system action) and providing the actor with a number of options relevant to the nature of the alarm. Because this secondary scenario can occur at any time for virtually all interactions, it will not become part of the *ACS- DCV* use case. Rather, a separate use case—Alarm condition encountered—would be developed and referenced from other use cases as required.

Each of the situations described in the preceding paragraphs is characterized as a use case exception. An exception describes a situation (either a failure condition or an alternative chosen by the actor) that causes the system to exhibit somewhat different behavior.

Cockburn [Coc01b] recommends a “brainstorming” session to derive a reasonably complete set of exceptions for each use case. In addition to the three generic questions suggested earlier in this section, the following issues should also be explored:

- Are there cases in which some “validation function” occurs during this use case? This implies that validation function is invoked and a potential error condition might occur.
- Are there cases in which a supporting function (or actor) will fail to respond appropriately? For example, a user action awaits a response but the function that is to respond times out.
- Can poor system performance result in unexpected or improper user actions? For example, a Web-based interface responds too slowly, resulting in a user making multiple selects on a processing button. These selects queue inappropriately and ultimately generate an error condition.

The list of extensions developed as a consequence of asking and answering these questions should be “rationalized” [Co01b] using the following criteria: an exception should be noted

within the use case if the software can detect the condition described and then handle the condition once it has been detected. In some cases, an exception will precipitate the development of another use case.

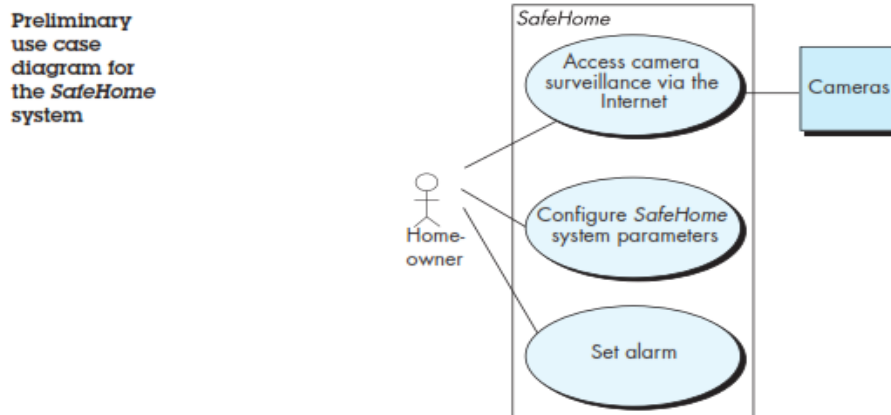
Writing a Formal Use Case:

The informal use cases are sometimes sufficient for requirements modeling. However, when a use case involves a critical activity or describes a complex set of steps with a significant number of exceptions, a more formal approach may be desirable.

The ACS-DCV use case shown in the sidebar follows a typical outline for formal use cases. The goal in context identifies the overall scope of the use case.

The precondition describes what is known to be true before the use case is initiated. The trigger identifies the event or condition that “gets the use case started” [Coc01b]. The scenario lists the specific actions that are required by the actor and the appropriate system responses. Exceptions identify the situations uncovered as the preliminary use case is refined. Additional headings may or may not be included and are reasonably self-explanatory.

Preliminary use case for the **SafeHome** system.



In many cases, there is no need to create a graphical representation of a usage scenario. However, diagrammatic representation can facilitate understanding, particularly when the scenario is complex. As we noted earlier in this book, UML does provide use case diagramming capability. Above Figure depicts a preliminary use case diagram for the **SafeHome** product. Each use case is represented by an oval. Only the ACS-DCV use case has been discussed in this section.

Every modeling notation has limitations, and the use case is no exception. Like any other form of written description, a use case is only as good as its author(s). If the description is unclear, the use case can be misleading or ambiguous. A use case focuses on function and behavioral requirements and is generally inappropriate for non-functional requirements. For situations in which the requirements model must have significant detail and precision (e.g., safety critical systems), a use case may not be sufficient.