**UNDERSTANDING REQUIREMENTS:**

Understanding the requirements of a problem is among the most difficult tasks that face a software engineer. When you first think about it, developing a clear understanding of requirements doesn’t seem that hard. After all, doesn’t the customer know what is required? Shouldn’t the end users have a good understanding of the features and functions that will provide benefit? Surprisingly, in many instances the answer to these questions is “no.” And even if customers and end users are explicit in their needs, those needs will change throughout the project.

**8.1 REQUIREMENTS E NGINEERING:**

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.

**Inception:**

**Inception:**

A software project can start in various ways, sometimes from a simple conversation or a gradually evolving need. Generally, most projects begin when a business need is identified or a new market or service opportunity is discovered. Business stakeholders, such as managers and product marketers, develop a business case, estimate market potential, conduct a rough feasibility study, and outline the project scope. Although this initial information may change, it initiates discussions with the software engineering team.

During the inception phase, the goal is to establish a basic understanding of the problem, identify the stakeholders, outline the desired solution, and assess the effectiveness of early communication and collaboration among all involved parties.

**Elicitation**:

Elicitation, the process of gathering requirements from stakeholders, seems straightforward—ask about system objectives, how it fits business needs, and daily use. However, it's incredibly challenging. Stakeholders may struggle to articulate their goals clearly, leading to several difficulties:

1. **Scope Problems:** Unclear system boundaries or unnecessary technical details can obscure project objectives.
2. **Understanding Issues:** Stakeholders may not fully grasp their own needs, the capabilities of their computing environment, or the intricacies of the problem domain. Communication difficulties, omitted information, conflicting requirements, and ambiguities also hinder clarity.
3. **Volatility:** Requirements may change over time due to evolving circumstances.

To address these challenges and gain a clear understanding of what the customer wants, it's crucial to engage stakeholders openly, establish and prioritize business goals early, and maintain a structured approach to requirements gathering. This ensures that all parties involved have a shared understanding and that the software solution aligns with stakeholder expectations and business objectives.

### Elaboration

During the elaboration phase of software development, the initial information gathered from customers during inception and elicitation is expanded and refined. This phase focuses on developing a detailed requirements model that specifies various aspects of software function, behaviour, and information.

* **Refined Requirements Model:** Chapters 9 through 11 of the development process are dedicated to refining the requirements model. This model identifies the software's functionalities, how it behaves, and the types of information it manages.
* **User Scenarios:** Elaboration involves creating and refining user scenarios that describe how end users and other actors interact with the system. These scenarios are analyzed to identify and define analysis classes—business domain entities visible to the end user.
* **Attributes and Services:** Each analysis class's attributes and required services are defined. Relationships and collaborations between classes are identified, and supplementary diagrams are produced to visualize these relationships.

### Negotiation

Negotiation is a critical aspect of requirements management, especially when stakeholders have conflicting or unrealistic demands:

* **Conflict Resolution:** Customers and users often request more than what can be feasibly delivered given business constraints. Conflicting requirements may arise, with stakeholders arguing for their needs as essential.
* **Prioritization and Iterative Approach:** Stakeholders rank requirements based on importance, and conflicts are resolved through negotiation. An iterative approach is used to prioritize requirements, assess their costs and risks, and reconcile internal conflicts.
* **Modification and Combination:** Requirements may be modified, combined, or eliminated to achieve a balance that satisfies all parties involved to some extent.

### Specification

The specification phase involves defining what the software system will do and how it will perform:

* **Various Forms of Specification:** Specifications can take different forms, such as written documents, graphical models, formal mathematical models, usage scenarios, prototypes, or a combination of these.
* **Flexibility in Specification:** While some advocate for a standard template for specifications to ensure consistency and clarity, flexibility is crucial. The choice of specification format depends on the size and complexity of the system, as well as the technical environment in which it will operate.
* **Examples:** For large systems, a comprehensive written document combining natural language descriptions and graphical models may be necessary. In contrast, smaller systems may only require usage scenarios within well-understood technical environments.

### Validation of Requirements

Requirements validation ensures the quality and accuracy of the software requirements specification:

* **Purpose:** It confirms that all requirements are clearly stated and free from ambiguities, inconsistencies, and omissions. The goal is to ensure the specification aligns with established standards for the project and product.
* **Mechanism:** Validation primarily involves technical reviews conducted by a team of stakeholders and software engineers. They identify errors, ambiguities, and missing details in the specification, using analysis models to maintain consistency across requirements.
* **Challenges:** Vague requirements like "user-friendly" need clarification, while quantitative requirements, such as security thresholds, require assessment for feasibility and effectiveness.
* **Verification Techniques:** Techniques include both qualitative (e.g., user surveys) and quantitative methods to verify compliance with requirements. Critical quality requirements are quantified to assess risk and ensure they are effectively addressed.

In essence, requirements validation is crucial for ensuring that the software development process starts with a clear, comprehensive, and feasible set of requirements that meet stakeholder expectations and project goals.

**Requirements 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. 6 Many of these activities are identical to the software configuration management (SCM) techniques discussed in Chapter 29Top of FormBottom of Form

**8.2 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, may have conflicting opinions about the system to be built, may have limited technical knowledge, and may have limited time to interact with the requirements engineer.

None of these things are desirable, but all are fairly common, and you are often forced to work within the constraints imposed by this situation. In the sections that follow, 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.

**8.2.1 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?”.

**8.2.2 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. There are several things that can make it hard to elicit requirements for software that satisfies its users: project goals are unclear, stakeholders’ priorities differ, people have unspoken assumptions, stakeholders interpret meanings differently, and requirements are stated in a way that makes them difficult to verify [Ale11]. The goal of effective requirements engineering is to eliminate or at least reduce these problems.

**8.2.3 Working toward 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.

**8.2.4 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 t 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. The next set of questions enables you to gain a better understanding of the problem and allows the customer to voice his or her perceptions about a solution:

How would you characterize “good” output that would be generated by a successful solution?

• What problem(s) will this solution address?

• Can you show me (or describe) the business environment in which the solution will be used?

• Will special performance issues or constraints affect the way the solution is approached?

The final set of questions focuses on the effectiveness of the communication activity itself. Gause and Weinberg [Gau89] call these “meta-questions” and propose the following (abbreviated) list:

• Are you the right person to answer these questions? Are your answers “official”?

• Are my questions relevant to the problem that you have?

• Am I asking too many questions?

• Can anyone else provide additional information?

• Should I be asking you anything else?

These questions (and others) will help to “break the ice” and initiate the communication that is essential to successful elicitation. But a question-and-answer meeting format is not an approach that has been overwhelmingly successful. In fact, the Q&A session should be used for the first encounter only and then replaced by a requirements elicitation format that combines elements of problem solving, negotiation, and specification. An approach of this type is presented in Section 8.3.

**8.2.5 Nonfunctional Requirements:**

A nonfunctional requirement (NFR) can be described as a quality attribute, a performance attribute, a security attribute, or a general constraint on a system. These are often not easy for stakeholders to articulate. Chung [Chu09] suggests that there is a lopsided emphasis on functionality of the software, yet the software may not be useful or usable without the necessary non-functional characteristics. In Section 8.3.2, we discuss a technique called quality function deployment (QFD). Quality function deployment attempts to translate unspoken customer needs or goals into system requirements. Nonfunctional requirements are often listed separately in a software requirements specification.

As an adjunct to QFD, it is possible to defi ne a two-phase approach [Hne11] that can assist a software team and other stakeholders in identifying nonfunctional requirements. During the first phase, a set of software engineering guidelines is established for the system to be built. These include guidelines for best practice, but also address architectural style (Chapter 13) and the use of design patterns (Chapter 16). A list of NFRs (e.g., requirements that address usability, testability, security or maintainability) is then developed. A simple table lists NFRs as column labels and software engineering guidelines as row labels. A relationship matrix compares each guideline to all others, helping the team to assess whether each pair of guidelines is complementary, overlapping, conflicting, or independent.

In the second phase, the team prioritizes each nonfunctional requirement by creating a homogeneous set of nonfunctional requirements using a set of decision rules [Hne11] that establish which guidelines to implement and which to reject.

**8.2.6 Traceability:**

Traceability is a software engineering term that refers to documented links between software engineering work products (e.g., requirements and test cases). A traceability matrix allows a requirements engineer to represent the relationship between requirements and other software engineering work products. Rows of the traceability matrix are labelled using requirement names and columns can be labelled with the name of a software engineering work product (e.g., a design element or a test case). A matrix cell is marked to indicate the presence of a link between the two.

The traceability matrices can support a variety of engineering development activities. They can provide continuity for developers as a project moves from one project phase to another, regardless of the process model being used. Traceability matrices often can be used to ensure the engineering work products have taken all requirements into account.

As the number of requirements and the number of work products grows, it becomes increasingly difficult to keep the traceability matrix up to date. Nonetheless, it is important to create some means for tracking the impact and evolution of the product requirements [Got11].

**8.3 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 [Zah90].

**8.3.1 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 (Section 8.2). 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 Safe Home project. This person writes the following narrative about the home security function that is to be part of Safe Home.

Our research indicates that the market for home management systems is growing at a rate of 40 percent per year. The first Safe Home 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, fi re, 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.

In reality, others would contribute to this narrative during the requirements-gathering meeting and considerably more information would be available. But even with additional information, ambiguity is present, omissions are likely to exist, and errors might occur. For now, the preceding “functional description” will suffice.

While reviewing the product request in the days before the meeting, each attendee is asked to make a list of objects that are part of the environment that surrounds the system, other objects that are to be produced by the system, and objects that are used by the system to perform its functions. In addition, each attendee is asked to make another list of services (processes or functions) that manipulate or interact with the objects. Finally, lists of constraints (e.g., cost, size, business rules) and performance criteria (e.g., speed, accuracy) are also developed. The attendees are informed that the lists are not expected to be exhaustive but are expected to reflect each person’s perception of the system.

Objects described for Safe Home might include the control panel, smoke detectors, window and door sensors, motion detectors, an alarm, an event (a sensor has been activated), a display, a PC, telephone numbers, a telephone call, and so on. The list of services might include configuring the system, setting the alarm, monitoring the sensors, dialling the phone, programming the control panel, and reading the display (note that services act on objects). In a similar fashion, each attendee will develop lists of constraints (e.g., the system must recognize when sensors are not operating, must be user friendly, must interface directly to a standard phone line) and performance criteria (e.g., a sensor event should be recognized within one second, and an event priority scheme should be implemented).

The lists of objects can be pinned to the walls of the room using large sheets of paper, stuck to the walls using adhesive-backed sheets, or written on a wall board. Alternatively, the lists may have been posted on a group forum, at an internal website, or posed in a social networking environment for review prior to the meeting. Ideally, each listed entry should be capable of being manipulated separately so that lists can be combined, entries can be deleted, and additions can be made. At this stage, critique and debate are strictly prohibited.

After individual lists are presented in one topic area, the group creates a combined list by eliminating redundant entries, adding any new ideas that come up during the discussion, but not deleting anything. After you create combined lists for all topic areas, discussion—coordinated by the facilitator—ensues. The combined list is shortened, lengthened, or reworded to properly reflect the product or system to be developed. The objective is to develop a consensus list of objects, services, constraints, and performance for the system to be built.

In many cases, an object or service described on a list will require further explanation. To accomplish this, stakeholders develop mini-specific cations for entries on the lists or by creating a use case (Section 8.4) that involves the object or service. For example, the mini-spec for the Safe Home object Control Panel might be:

The control panel is a wall-mounted unit that is approximately 230 x 130 mm in size. The control panel has wireless connectivity to sensors and a PC. User interaction occurs through a keypad containing 12 keys. A 75 x 75 mm OLED colour display provides user feedback. Software provides interactive prompts, echo, and similar functions.

The mini-specs are presented to all stakeholders for discussion. Additions, deletions, and further elaboration are made. In some cases, the development of mini-specs will uncover new objects, services, constraints, or performance requirements that will be added to the original lists. During all discussions, the team may raise an issue that cannot be resolved during the meeting. An issues list is maintained so that these ideas will be acted on later.

Many stakeholder concerns (e.g., accuracy, data accessibility, security) are the basis for nonfunctional system requirements (Section 8.2). As stakeholders enunciate these concerns, software engineers must consider them within the context of the system to be built. Among the questions that must be answered [Lag10] are as follows:

• Can we build the system? • Will this development process allow us to beat our competitors to market? • Do adequate resources exist to build and maintain the proposed system? • Will the system performance meet the needs of our customers? The answers to these and other questions will evolve over time.

**8.3.2 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 [Aka04].

**8.3.3 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. Use cases are discussed in greater detail in Section 8.4.

**8.3.4 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.

**8.3.5 Agile Requirements Elicitation:**

Within the context of an agile process, requirements are elicited by asking all stakeholders to create user stories. Each user story describes a simple system requirement written from the user’s perspective. User stories can be written on small note cards, making it easy for developers to select and manage a subset of requirements to implement for the next product increment. Proponents claim that using note cards written in the user’s own language allows developers to shift their focus to communication with stakeholders on the selected requirements rather than their own agenda [Mai10a].

Although the agile approach to requirements elicitation is attractive for many software teams, critics argue that a consideration of overall business goals and nonfunctional requirements is often lacking. In some cases, rework is required to accommodate performance and security issues. In addition, user stories may not provide a sufficient basis for system evolution over time

**8.3.6 Service-Oriented Methods:**

Service-oriented development views a system as an aggregation of services. A service can be “as simple as providing a single function, for example, a request/ response-based mechanism that provides a series of random numbers, or can be an aggregation of complex elements, such as the Web service API” [Mic12]. Requirements elicitation in service-oriented development focuses on the definition of services to be rendered by an application. As a metaphor, consider the service provided when you visit a fi ne hotel. A doorperson greets guests. A valet parks their cars. The desk clerk checks the guests in. A bellhop manages the bags. The concierge assists guest with local arrangements. Each contact or touchpoint between a guest and a hotel employee is designed to enhance the hotel visit and represents a service offered.

Most service design methods emphasize understanding the customer, thinking creatively, and building solutions quickly [Mai10b]. To achieve these goals, requirements elicitation can include ethnographic studies,11 innovation workshops, and early low-fidelity prototypes. Techniques for eliciting requirements must also acquire information about the brand and the stakeholders’ perceptions of it. In addition to studying how the brand is used by customers, analysts need strategies to discover and document requirements about the desired qualities of new user experiences. User stories are helpful in this regard The requirements for touchpoints should be characterized in a manner that indicates achievement of the overall service requirements. This suggests that each requirement should be traceable to a specific service.

**8.4 DEVELOPING USE CASES:**

In a book that discusses how to write effective 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 defied 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.

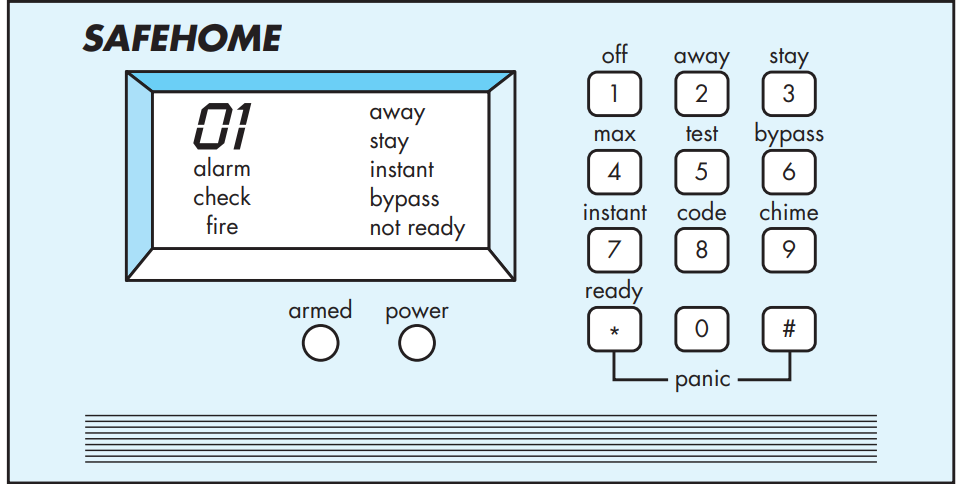
It is important to note that an actor and an end user are not necessarily the same thing. A typical user may play a number of different roles when using a system, whereas an actor represents a class of external entities (often, but not always, people) that play just one role in the context of the use 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 defi ned: programmer, tester, monitor, and troubleshooter. 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 12 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?

**Safe Home control panel:**



Recalling basic Safe Home 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 Safe Home, home security function). For the purposes of this example, we consider only the homeowner 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) inquiries about the status of a security zone,

(3) inquiries 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: 13

1. The homeowner observes the Safe Home control panel (Figure 8.1) to determine if the system is ready for input. 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 disappears. [A not ready message implies that a sensor is open; i.e., that a door or window is open.]
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 Safe Home 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-enters 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

**Frequency of use:** Many times, per day

**Channel to actor:** 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:

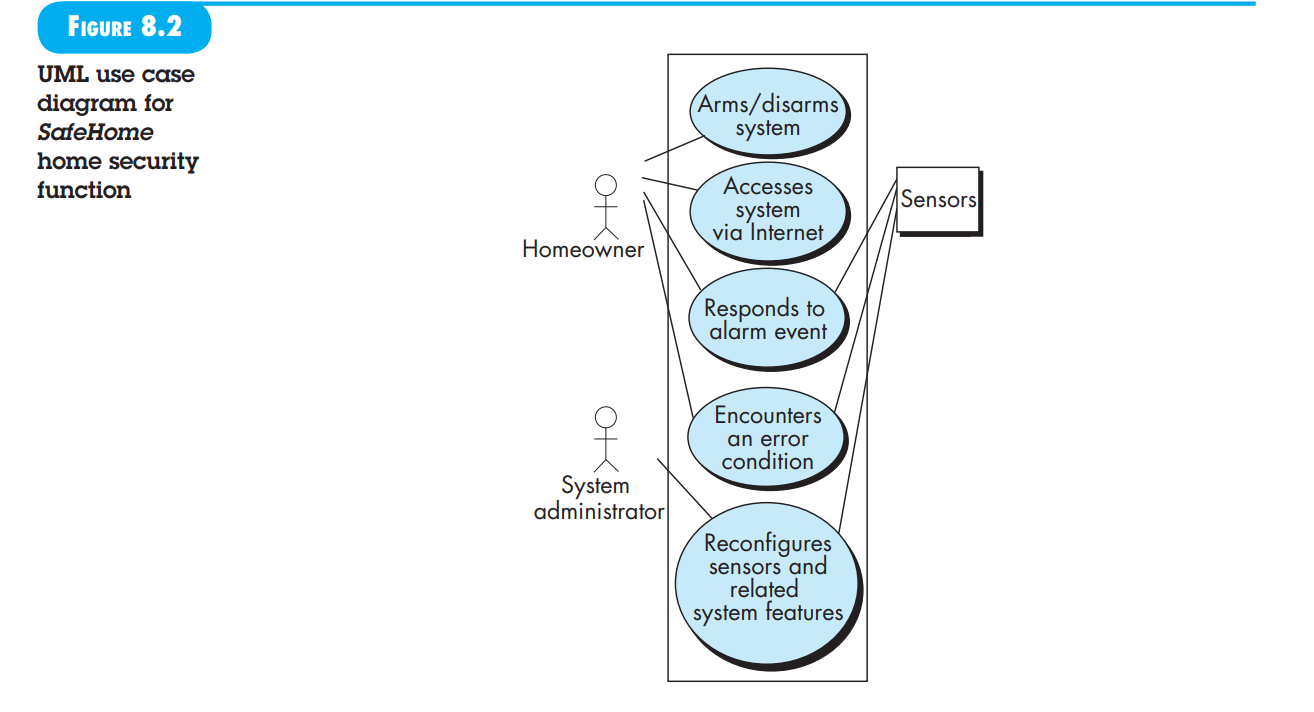
1. Should there be a way to activate the system without the use of a password or with an abbreviated password?

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.



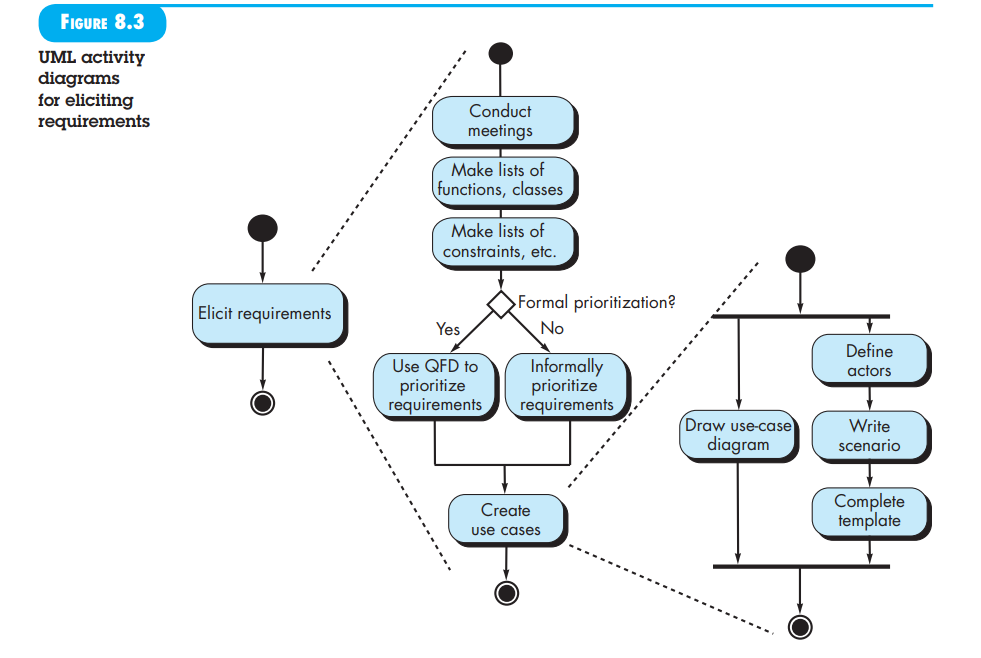
**8.5 B UILDING THE A NALYSIS MODEL:**

he intent of the analysis model is to provide a description of the required informational, functional, and behavioural 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. The analysis model and the methods that are used to build it are presented in detail in Chapters 9 to 11. We present a brief overview in the sections that follow

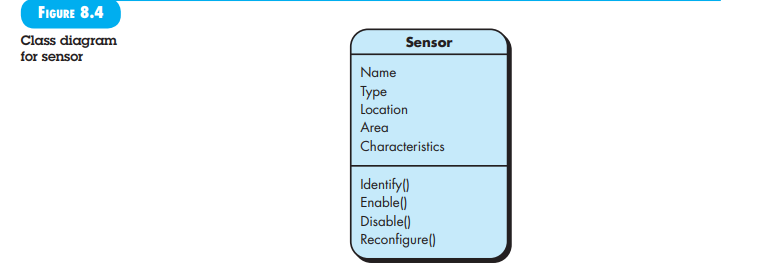
**8.5.1 Elements of the Analysis Model:**

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.

****

**Scenario-based elements.** The system is described from the user’s point of view using a scenario-based approach. For example, basic use cases (Section 8.4) and their corresponding use case diagrams ( Figure 8.2 ) evolve into more elaborate template-based use cases. Scenario-based elements of the requirements model are often the fi rst part of the model that is developed. As such, they serve as input for the creation of other modeling elements. Figure 8.3depicts a UML activity diagram 17 for eliciting requirements and representing them using use cases. Three levels of elaboration are shown, culminating in a scenario-based representation.

**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 behaviours. For example, a UML class diagram can be used to depict a Sensor class for the Safe Home 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 modelling elements depict the manner in which classes collaborate with one another and the relationships and interactions between classes. These are discussed in more detail in Chapter 10.



**Behavioural elements:** The behaviour 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 modelling elements that depict behaviour. 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 behaviour. 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 Safe Home control panel that is responsible for reading user input. A simplify ed UML state diagram is shown in Figure 8.5. In addition to behavioural representations of the system as a whole, the behaviour of individual classes can also be modelled. Further discussion of behavioural modelling is presented in Chapter 11.