Unit: 2

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.

1. Inception

Inception is the initial phase where a software project gets started. The need for a new system or product might arise from a casual conversation, an identified business need, or the discovery of a new market or service. During this phase, stakeholders from the business community (e.g., business managers, marketing people) define the business case, perform a rough feasibility analysis, and outline the project's scope. The goal is to establish a basic understanding of the problem, the desired solution, and the preliminary communication and collaboration between stakeholders and the software team.

2. Elicitation

Elicitation involves gathering requirements from stakeholders such as customers and users. It aims to understand the objectives for the system or product, its role in the business, and its day-to-day usage. This task is challenging due to potential issues like poorly defined system boundaries, unclear requirements, and changing needs over time. Establishing business goals and engaging stakeholders to share their goals honestly is crucial. Prioritization mechanisms and design rationales for potential architectures are developed based on these goals.

3. Elaboration

Elaboration takes the information obtained during inception and elicitation and refines it. This task focuses on developing a detailed requirements model that identifies software functions, behaviors, and information. It involves creating and refining user scenarios that describe interactions with the system. Analysis classes (business domain entities visible to the end user) are defined, along with their attributes and required services. Relationships and collaborations between classes are identified, and supplementary diagrams are produced to visualize the system.

4. Negotiation

Negotiation addresses conflicts that arise when stakeholders ask for more than can be achieved within the given resources or propose conflicting requirements. Stakeholders rank requirements and discuss conflicts in priority. An iterative approach is used to prioritize requirements, assess their cost and risk, and resolve internal conflicts. The goal is to eliminate, combine, or modify requirements so that each party achieves some measure of satisfaction.

5. Specification

Specification involves documenting the requirements for the system. This can take various forms, such as written documents, graphical models, formal mathematical models, usage scenarios, or prototypes. The approach depends on the size and complexity of the system. For large systems, a combination of natural language descriptions and graphical models might be used, while smaller systems might only require usage scenarios. The specification should be presented consistently but remain flexible to accommodate different project needs.

6. Validation

Validation ensures that the requirements have been stated unambiguously, and that inconsistencies, omissions, and errors have been detected and corrected. The work products produced during requirements engineering are assessed for quality. The primary validation mechanism is the technical review, where a team of software engineers, customers, users, and other stakeholders examines the specification for errors in content or interpretation, missing information, inconsistencies, and deviations from business rules and standards.

7. Management

Requirements management involves activities that help the project team identify, control, and track requirements and changes to them throughout the project's life. As requirements for computer-based systems change, managing these changes is crucial to maintaining the project's progress and alignment with business goals. Requirements management ensures that changes are systematically handled and that the impact on the project is well understood and managed.

8.2 ESTABLISHING THE GROUNDWORK

- **Challenges in Requirements Engineering**: In an ideal scenario, stakeholders and engineers work closely on the same team, making requirements engineering straightforward. In reality, stakeholders may be in different locations, have unclear or conflicting requirements, limited technical knowledge, or restricted time to interact.
- **Identifying Stakeholders**: Stakeholders are individuals who benefit from the system being developed. These include business managers, product managers, marketers, customers, end users, engineers, and support staff. Identifying stakeholders involves listing them and expanding this list by asking each for further recommendations.
- **Recognizing Multiple Viewpoints**: Different stakeholders have different perspectives and needs. For instance, marketing may focus on features that attract users, while business managers are concerned with budget and deadlines. Collecting and categorizing these viewpoints helps manage inconsistencies and conflicts in requirements.
- **Working toward Collaboration**: Effective collaboration is key, even when stakeholders have differing opinions. Techniques like “priority points” allow stakeholders to vote on the importance of requirements, helping to resolve conflicts and prioritize requirements. A project champion may make final decisions on requirements.
- **Asking the First Questions**: Initial questions should focus on understanding the project’s context, goals, and benefits. Questions include:
    - Who requested the work?
    - Who will use the solution?
    - What are the economic benefits?
    - Are there alternative solutions?
    - Follow-up questions should explore the problem, desired outcomes, and any constraints.
- **Nonfunctional Requirements (NFRs)**: NFRs include quality, performance, security, and constraints. Identifying NFRs can be challenging, but techniques like Quality Function Deployment (QFD) and a two-phase approach help define and prioritize these requirements.
- **Traceability**: Traceability involves linking requirements to other work products like design elements and test cases using a traceability matrix. This ensures that all requirements are addressed throughout the development process and helps maintain continuity as the project progresses.

8.3 ELICITING REQUIREMENTS

8.3.1 Collaborative Requirements Gathering

- **Meeting Structure**: Held with both software engineers and stakeholders, using predefined rules, agendas, and facilitators.
- **Preparation**: Attendees create lists of objects, services, constraints, and performance criteria, which are reviewed before the meeting.
- **Definition Mechanism**: Utilizes various tools (e.g., worksheets, digital boards) to document and manage information.
- **Combining Lists**: After presenting individual lists, the group merges them to form a consensus on requirements, avoiding premature critique.
- **Mini-Specifications**: Detailed descriptions (mini-specs) are developed for specific objects or services to clarify requirements.
- **Issue Tracking**: Issues that arise but cannot be resolved during the meeting are documented for future resolution.
- **Nonfunctional Requirements**: Considerations like accuracy, data accessibility, and security are addressed in the context of system requirements.

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 fine 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

Safe Home control panel

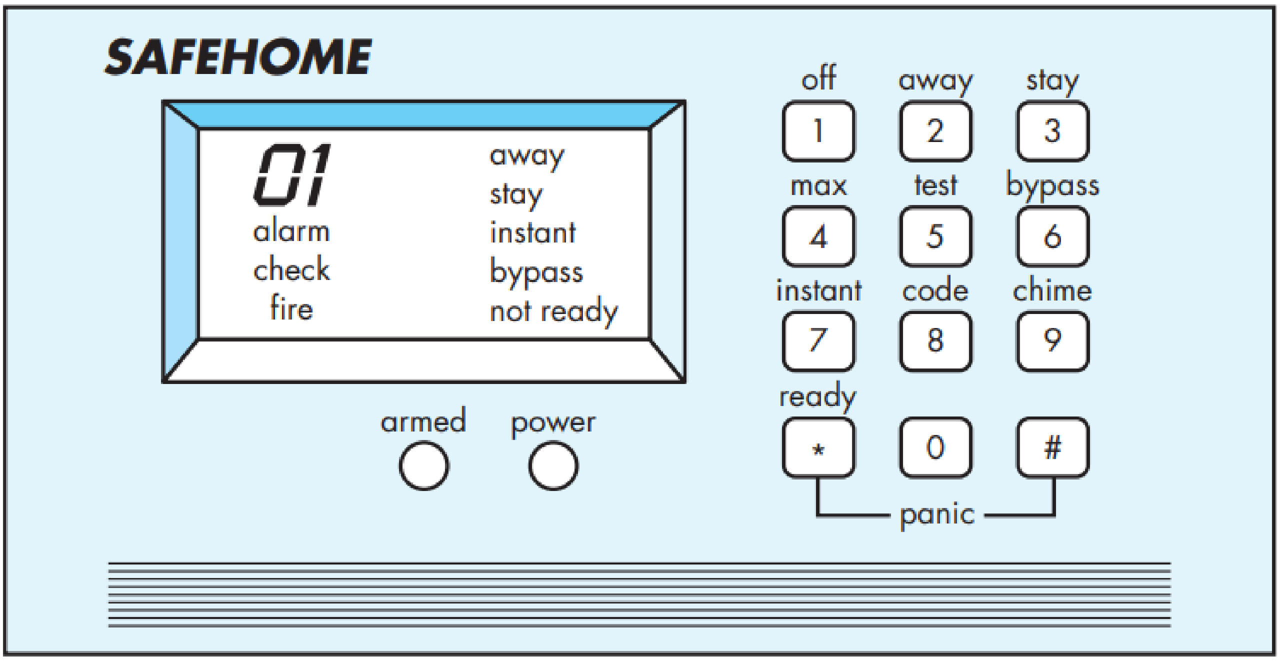
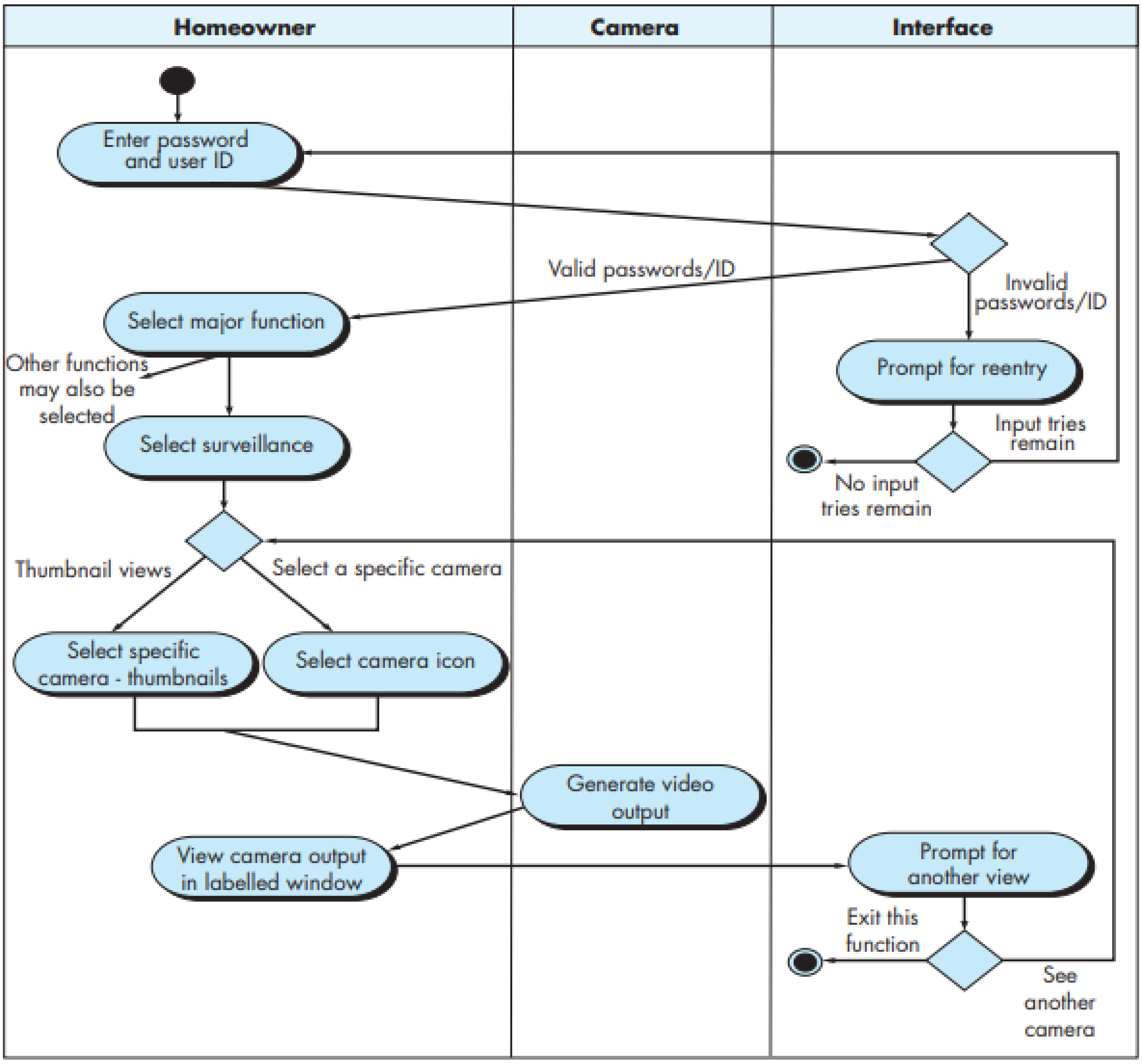


FIGURE 9.6 Swimlane diagram for Access camera surveillance via the Internet—display camera views function.



SafeHome Basic Requirements and Use Cases

Actors:

1. **Homeowner**: A user who interacts with the system.
2. **Setup Manager**: Likely the same person as the homeowner but playing a different role.
3. **Sensors**: Devices attached to the system.
4. **Monitoring and Response Subsystem**: The central station that monitors the SafeHome security function.

Homeowner Interactions:

1. **Enters a password**: Allows 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.
5. **Activates/Deactivates the security system**.

Use Case: System Activation

1. **Observing the Control Panel**:
    - The homeowner checks the SafeHome control panel to determine if the system is ready for input.
    - If the system is not ready, a "not ready" message is displayed on the LCD screen. The homeowner must physically close windows or doors to make the message disappear. [A "not ready" message implies that a sensor is open (e.g., a door or window is open).]
2. **Entering Password**:
    - The homeowner uses the keypad to input a four-digit password.
    - The password is compared with the valid password stored in the system. If the password is incorrect, the control panel beeps once and resets itself for additional input. If the password is correct, the control panel awaits further action.
3. **Selecting Activation Mode**:
    - The homeowner selects and inputs "stay" or "away" to activate the system.
        - **Stay**: Activates only perimeter sensors (inside motion-detecting sensors are deactivated).
        - **Away**: Activates all sensors.
4. **Observing Alarm Light**:
    - When activation occurs, a red alarm light is visible to the homeowner, indicating that SafeHome has been armed.

Use Case Details:

- **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**: The system has been programmed with a password and is configured to recognize various sensors.
- **Trigger**: The homeowner decides to activate the system.

Scenario:

1. Homeowner observes the control panel.

2. Homeowner enters the password.
3. Homeowner selects "stay" or "away."
4. Homeowner observes the red alarm light, indicating that SafeHome has been armed.

Exceptions:

1. **Control Panel Not Ready**: The homeowner checks all sensors to determine which are open and closes them.
2. **Incorrect Password**: The control panel beeps once, and the homeowner re-enters the correct password.
3. **Password Not Recognized**: Contact the monitoring and response subsystem to reprogram the password.
4. **Stay Selected**: The control panel beeps twice, and a stay light is lit; only perimeter sensors are activated.
5. **Away Selected**: The 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 using 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 fully activates?

FIGURE 8.2

UML use case diagram for *SafeHome* home security function

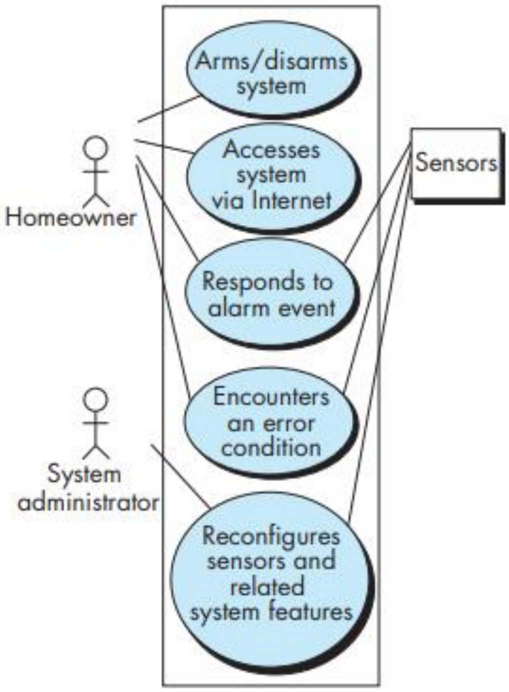
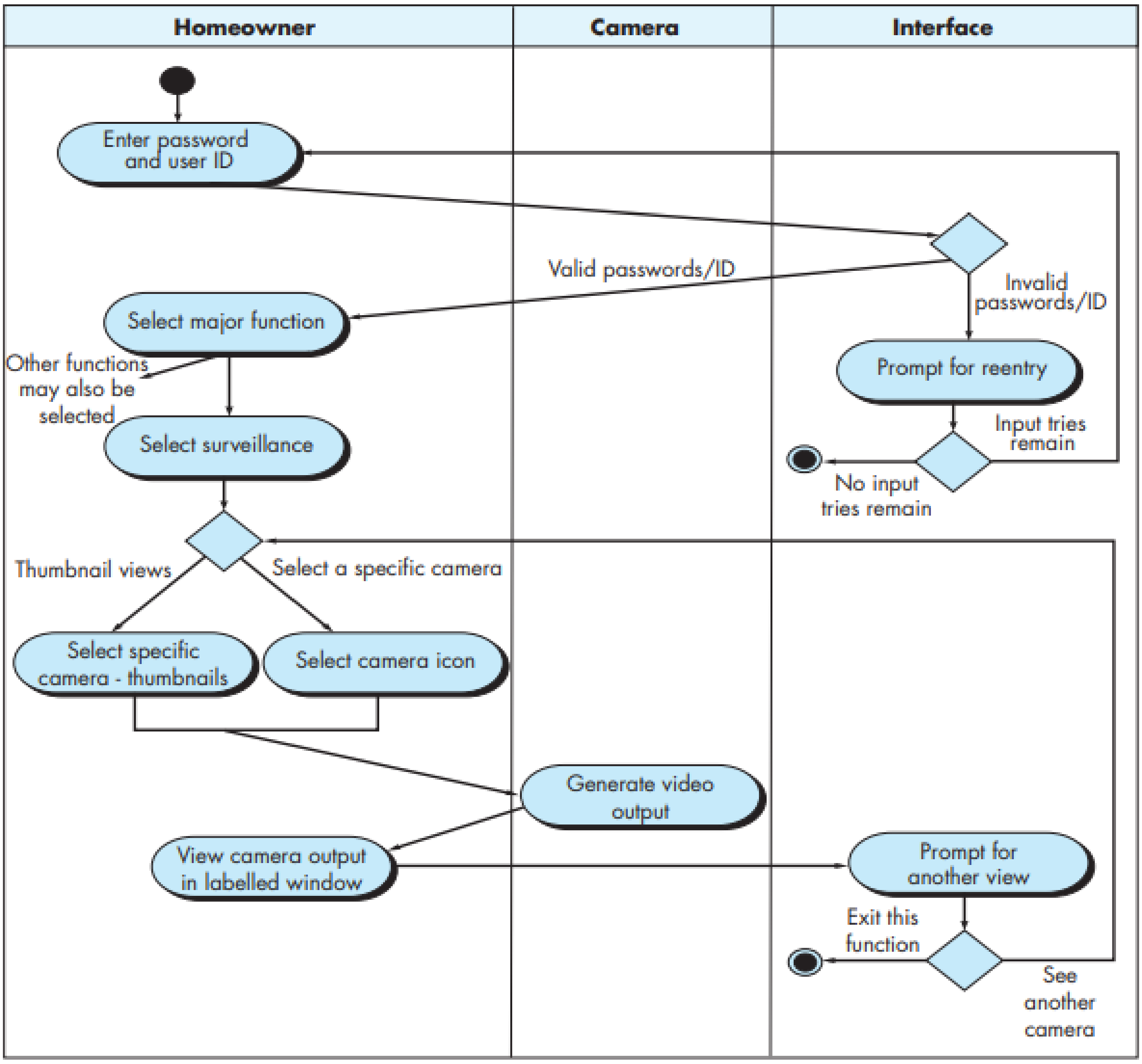

Arms/disarms system
Accesses system via Internet
Responds to alarm event
Encounters an error condition
Reconfigures sensors and related system features
Homeowner
System administrator
Sensors

FIGURE 9.6 Swimlane diagram for Access camera surveillance via the Internet—display camera views function.



Use Case Diagram for Library Management System

(i) University Library System

Use Case: Borrow a Book

Primary Actor: Library Patron (Student/Faculty)

Goal in Context: To borrow a book from the university library.

Preconditions:

- The patron must have a valid library account.
- The book must be available in the library.

Trigger: The patron decides to borrow a book.

Scenario:

1. **Patron**: Logs into their library account on the library’s website or app.
2. **Patron**: Searches for the book by title, author, or ISBN.
3. **System**: Displays search results with availability status.
4. **Patron**: Selects the book they wish to borrow.
5. **System**: Displays the book’s details, including availability.
6. **Patron**: Clicks the "Borrow" button.
7. **System**: Checks if the book is available and if the patron’s account is in good standing.
8. **System**: Updates the book’s status to "Checked Out" and assigns a due date.
9. **System**: Sends a confirmation email to the patron with borrowing details and due date.
10. **Patron**: Receives the confirmation and retrieves the book from the library (if applicable) or accesses it digitally.

Exceptions:

1. **Book Not Available**:
   - The system informs the patron that the book is currently checked out and provides options to place a hold or search for similar books.
2. **Account Issues**:
   - If the patron’s account has overdue books or fines, the system prevents the borrowing and provides instructions for resolving the issue.
3. **System Error**:
   - If there is a technical issue, the system displays an error message and advises the patron to try again later or contact library support.

Priority: Essential

When Available: First increment

Frequency of Use: Regularly

Channel to Actor: Library website or app, in-person at the library

Secondary Actors:

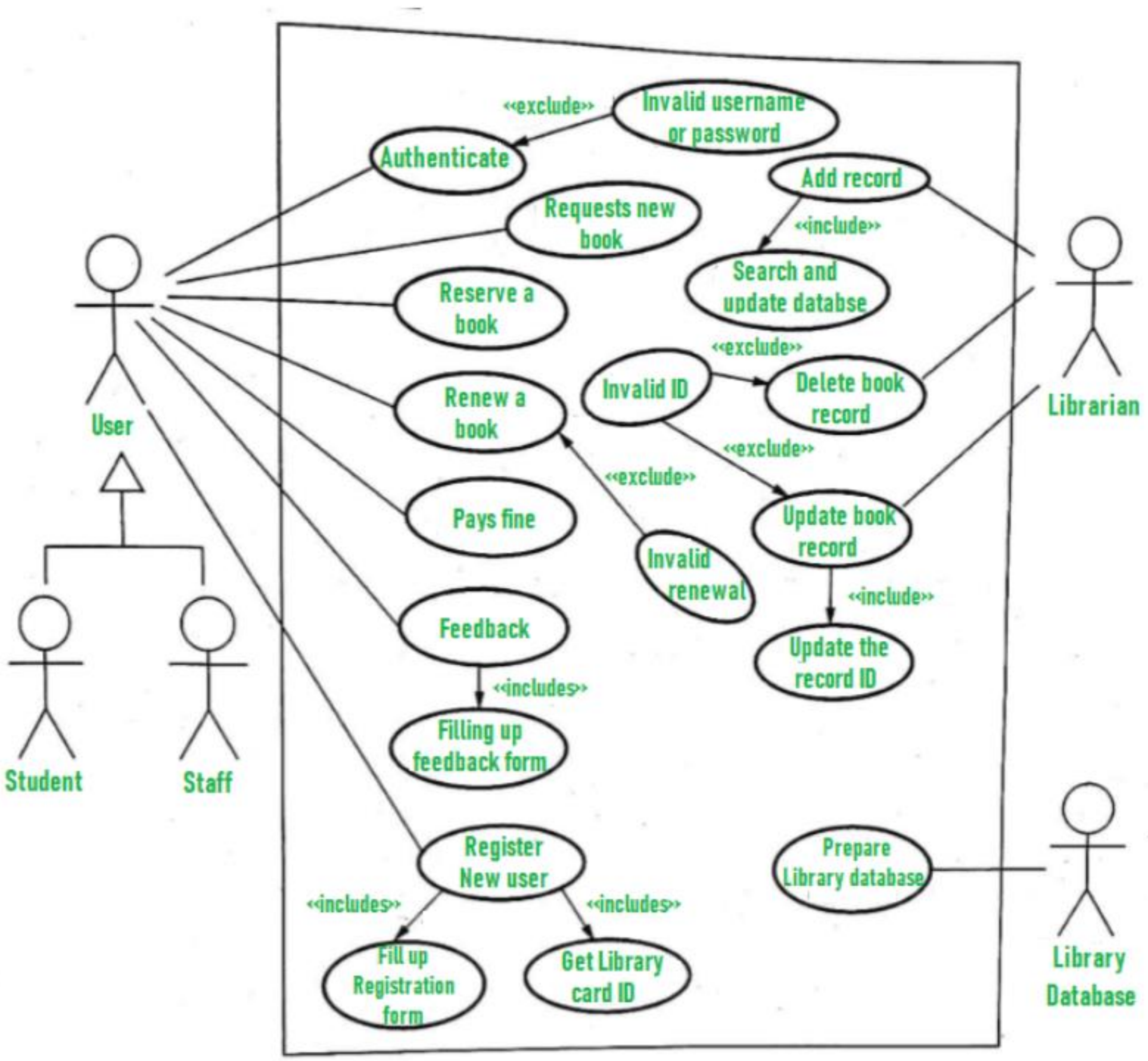
- **Library Staff**: For assistance.
- **Database**: For book availability and account management.

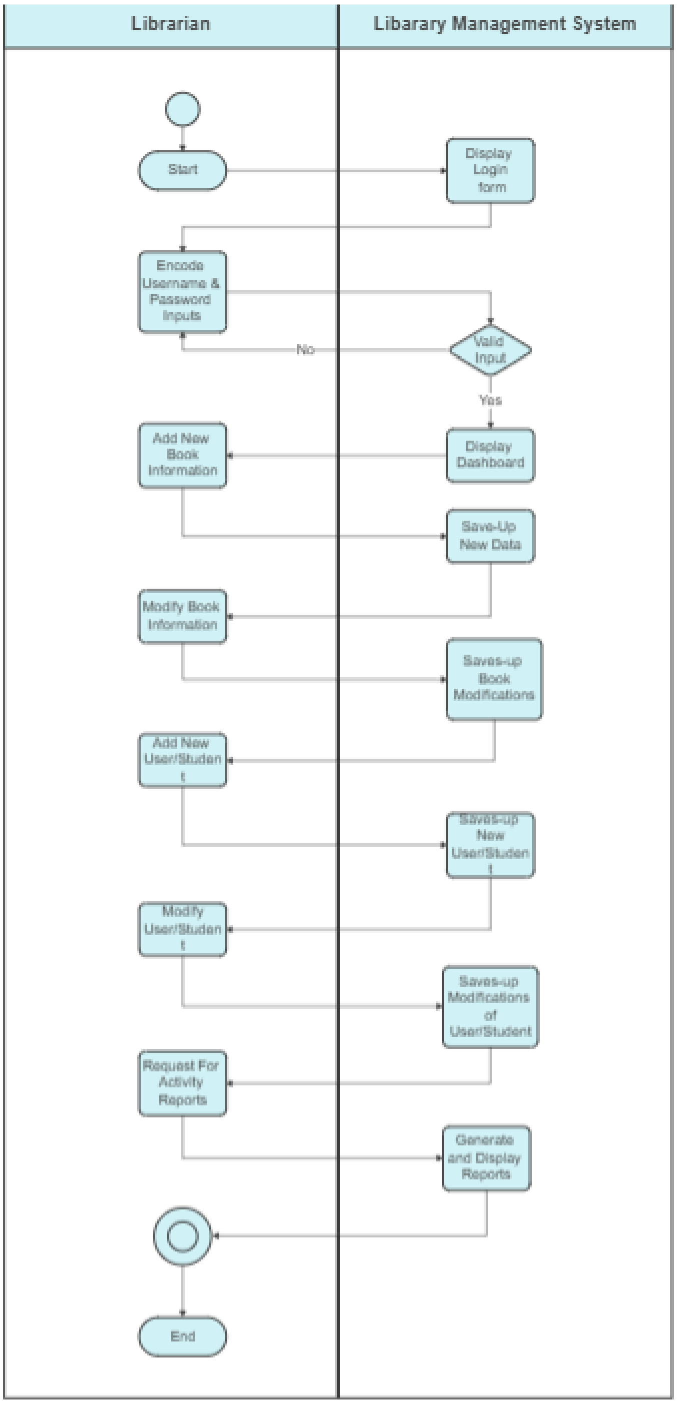
Channels to Secondary Actors:

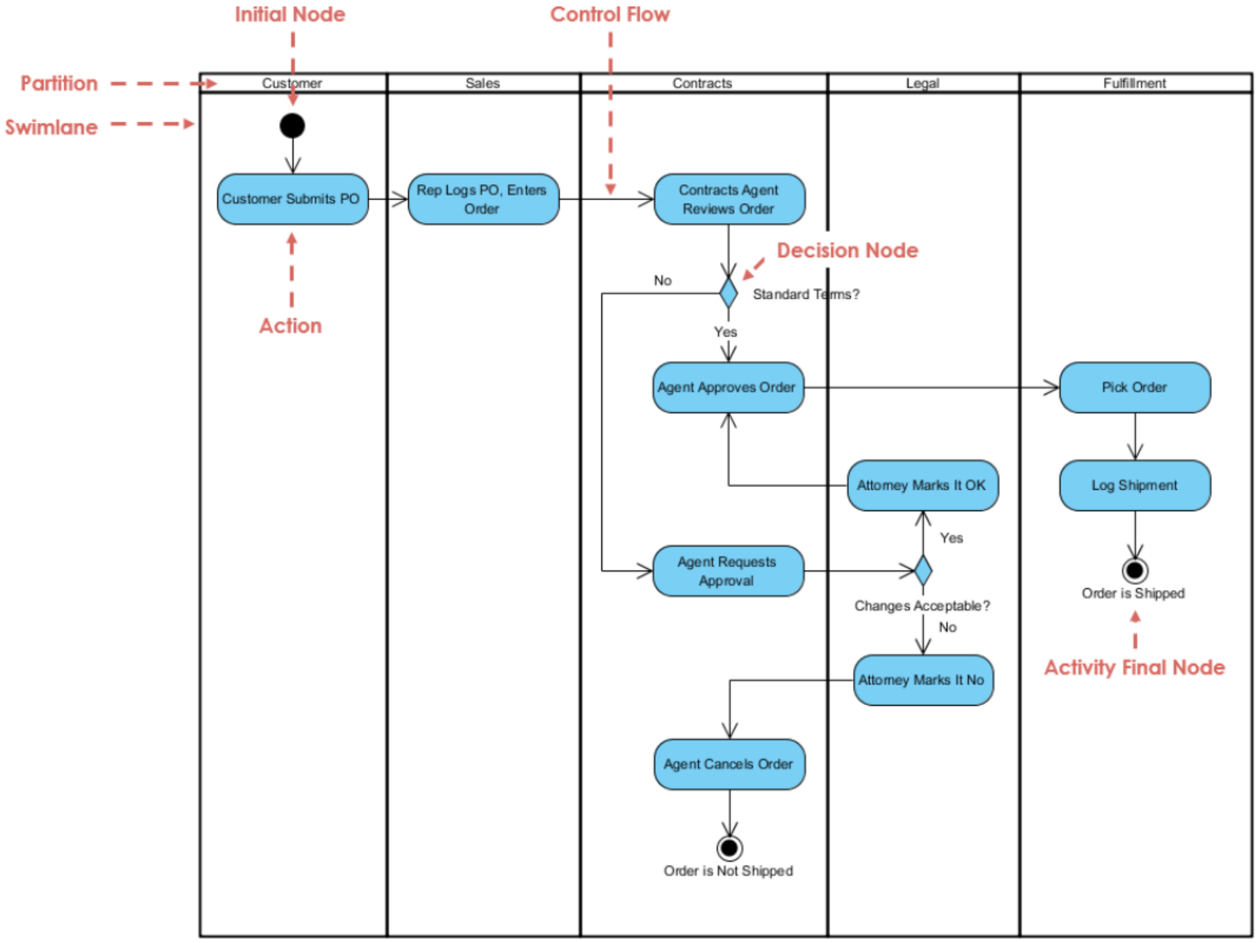
- **Library Staff**: In-person or via phone/email.
- **Database**: Automated updates and queries.

Open Issues:

1. Should there be a way to activate the system without using 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 fully activates?


«exclude»
Invalid username or password
Authenticate
Add record
Requests new book
«include»
Search and update databse
Reserve a book
«exclude»
Invalid ID
Delete book record
Librarian
User
Renew a book
«exclude»
«exclude»
Update book record
Pays fine
Invalid renewal
«include»
Feedback
Update the record ID
«includes»
Filling up feedback form
Student
Staff
Register New user
Prepare Library database
«includes»
«includes»
Fill up Registration form
Get Library card ID
Library Database


Librarian
Libarary Management System
Start
Display Login form
Encode Username & Password Inputs
Valid Input
No
Yes
Display Dashboard
Add New Book Information
Save-Up New Data
Modify Book Information
Saves-up Book Modifications
Add New User/Studen t
Saves-up New User/Studen t
Modify User/Studen t
Saves-up Modifications of User/Student
Request For Activity Reports
Generate and Display Reports
End



(ii) Buying a Stock Using an Online Brokerage Account

Use Case: Purchase Stock

Primary Actor: Investor (User with a brokerage account)

Goal in Context: To buy stocks through an online brokerage account.

Preconditions:

- The investor must have a valid brokerage account with sufficient funds.
- The stock must be available for trading.

Trigger: The investor decides to buy a stock.

Scenario:

- **Investor**: Logs into their brokerage account via the website or app.
- **Investor**: Navigates to the "Trade" or "Buy Stocks" section.

- **Investor**: Searches for the stock by ticker symbol or company name.
- **System**: Displays the stock's current price and other relevant details.
- **Investor**: Selects the stock and specifies the number of shares to buy.
- **Investor**: Chooses the type of order (e.g., market order, limit order).
- **System**: Displays the total cost, including any fees or commissions.
- **Investor**: Confirms the purchase details and submits the order.
- **System**: Processes the order and checks for sufficient funds.
- **System**: Executes the order and updates the investor's portfolio.
- **System**: Sends a confirmation email or notification with the transaction details.

Exceptions:

1. **Insufficient Funds**:
   - The system alerts the investor that there are not enough funds to complete the purchase.
2. **Stock Not Available**:
   - If the stock is not available for trading, the system informs the investor and may suggest alternative stocks.
3. **Order Processing Error**:
   - If there is a technical issue, the system notifies the investor and provides instructions for retrying or contacting support.

Priority: High

When Available: First increment

Frequency of Use: As needed

Channel to Actor: Online brokerage website or app

Secondary Actors:

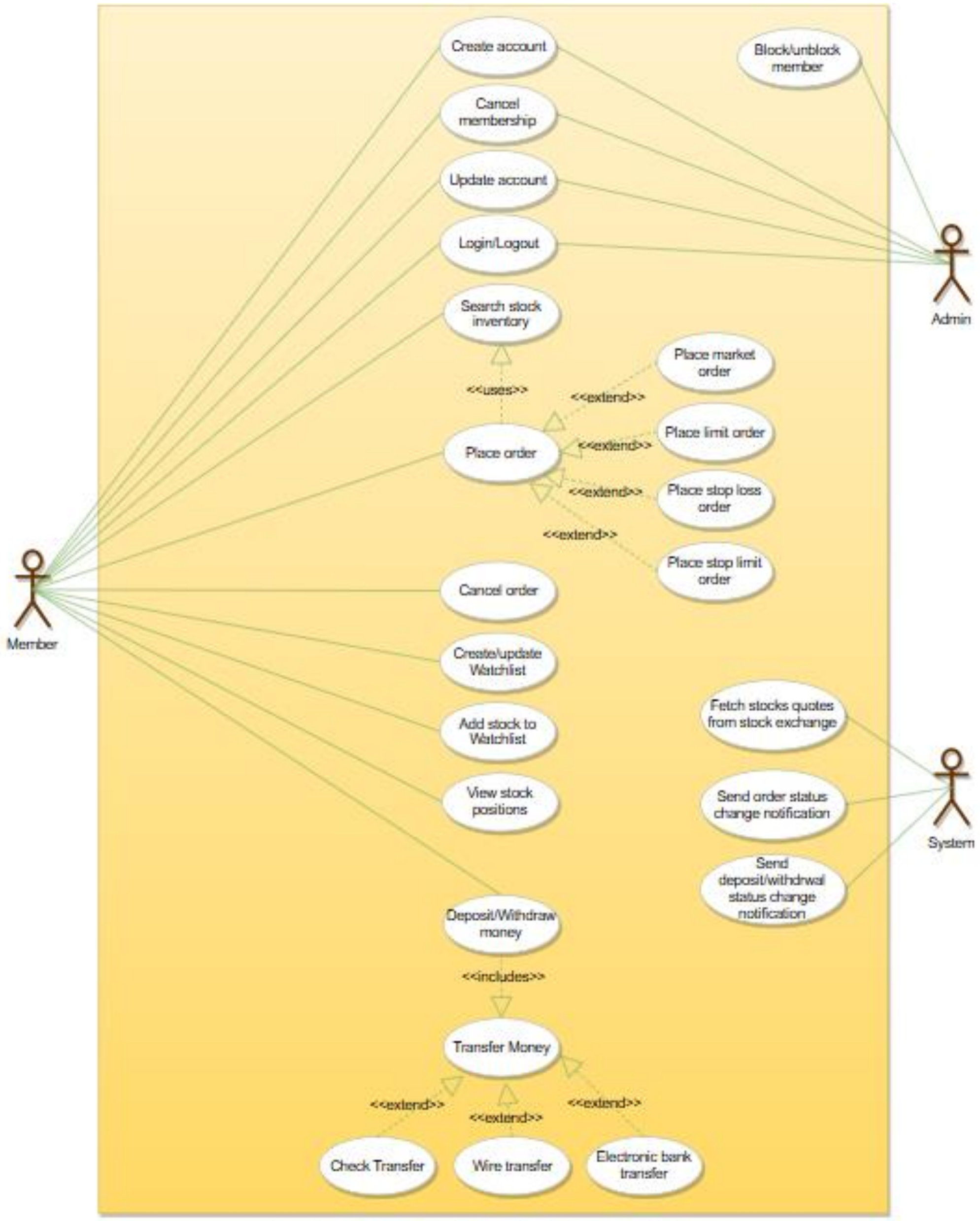
- **Stock Exchange**: For executing trades.
- **Payment Gateway**: For processing funds.

Channels to Secondary Actors:

- **Stock Exchange**: Automated trading systems.
- **Payment Gateway**: Financial transaction processing.

Open Issues:

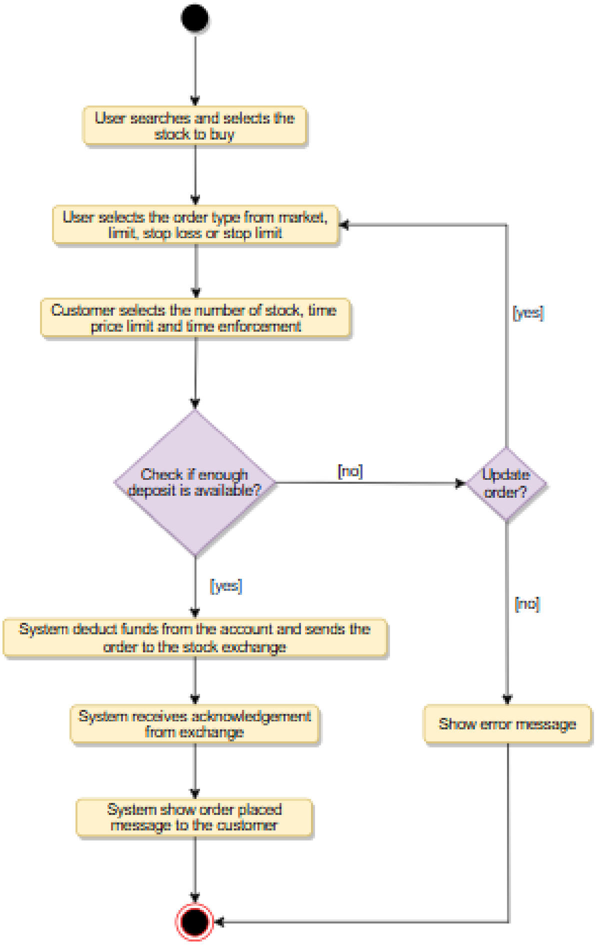
1. Should there be a way to activate the system without using a password or with an abbreviated password?
2. Should the control panel display additional text messages?
3. How much time does the homeowner have to enter the password from the time the first key is pressed?
4. Is there a way to deactivate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



Use Case Diagram for Online Stock Brokerage System

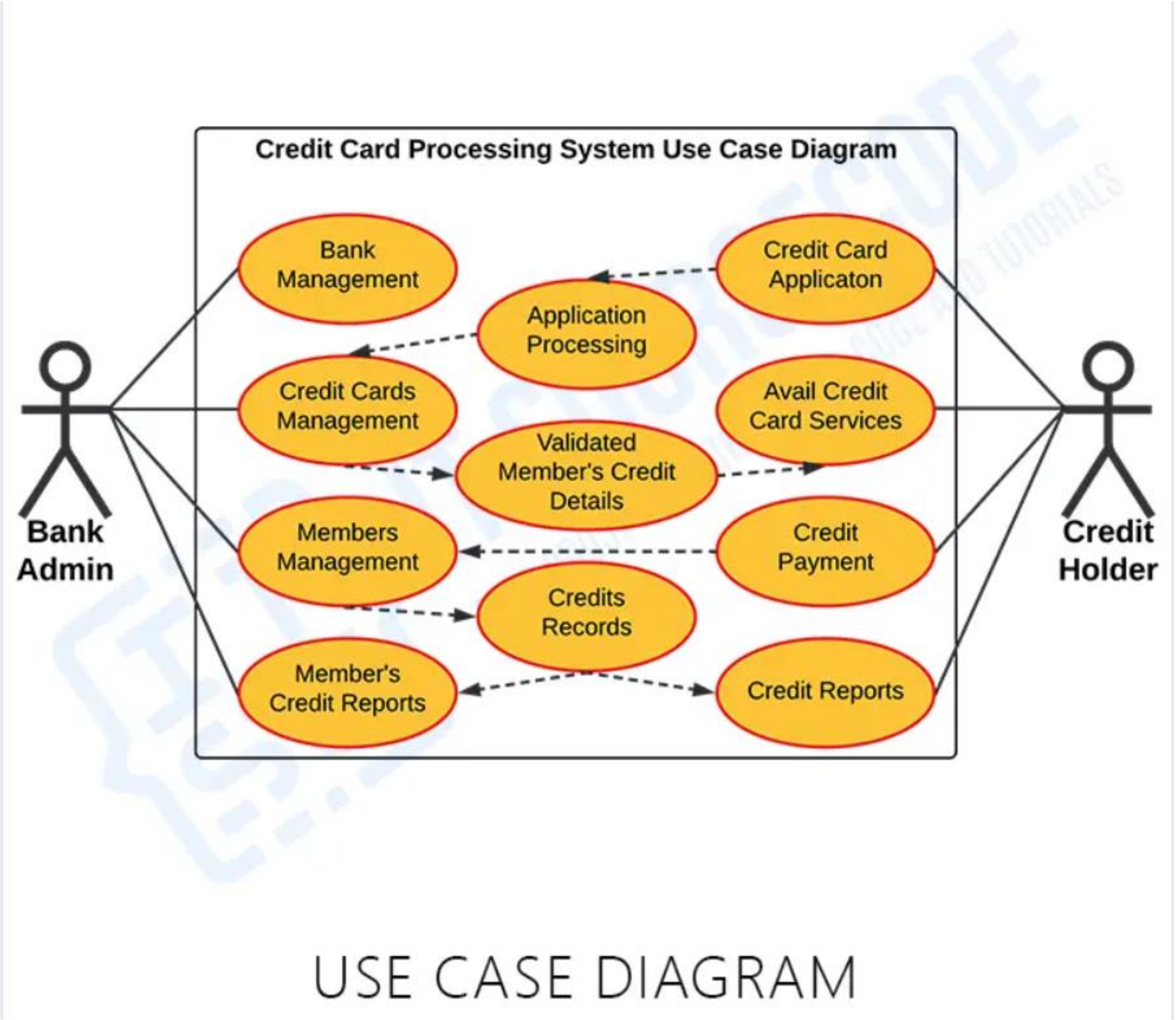
Activity Diagrams

Place a buy order: Any system user can perform this activity. Here are the steps to place a buy order:


User searches and selects the stock to buy
User selects the order type from market, limit, stop loss or stop limit
Customer selects the number of stock, time price limit and time enforcement
Check if enough deposit is available?
[no]
Update order?
[yes]
[yes]
[no]
System deduct funds from the account and sends the order to the stock exchange
System receives acknowledgement from exchange
System show order placed message to the customer
Show error message

Activity Diagram for Online Stock Brokerage System Buy Order

(iii) Using Credit Card at a Restaurant



Use Case: Pay Bill with Credit Card

Primary Actor: Customer (Cardholder)

Goal in Context: To pay for a meal at a restaurant using a credit card.

Preconditions:

- The customer must have a valid credit card with available credit.
- The restaurant must accept credit card payments.

Trigger: The customer decides to pay the bill.

Scenario:

1. **Customer**: Receives the bill from the restaurant server.
2. **Customer**: Provides the credit card to the server or inserts/swipes it at the payment terminal.
3. **Server/Payment Terminal**: Enters the bill amount into the payment system.
4. **Customer**: Reviews the amount and confirms.
5. **Payment System**: Processes the credit card transaction.
6. **Payment System**: Contacts the credit card issuer for authorization.
7. **Credit Card Issuer**: Validates the card details and available credit.

8. **Credit Card Issuer**: Approves or declines the transaction.
9. **Payment System**: Completes the transaction and prints a receipt or sends an electronic receipt.
10. **Customer**: Signs the receipt if required or confirms payment electronically.

Exceptions:

1. **Card Declined**:
   - If the credit card is declined, the system notifies the customer, who may use an alternative payment method.
2. **Payment Terminal Issue**:
   - If there is a malfunction with the payment terminal, the server may use a manual method to process the payment or retry the transaction.
3. **Receipt Error**:
   - If there is an issue with printing or sending the receipt, the system alerts the server to provide a manual receipt or follow up later.

Priority: Essential

When Available: First increment

Frequency of Use: Daily, depending on dining frequency

Channel to Actor: Payment terminal or credit card reader

Secondary Actors:

- **Credit Card Issuer**: For authorization.
- **Payment Processor**: For handling transactions.

Channels to Secondary Actors:

- **Credit Card Issuer**: Authorization and validation services.
- **Payment Processor**: Transaction handling and settlement.

Open Issues:

1. Should there be a way to activate the system without using 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 fully activates?

8.5 BUILDING THE A NALYSIS MODEL

8.5 Building the Analysis Model

Overview: The analysis model is a snapshot of the requirements for a system at a given time. It evolves as more is learned about the system and stakeholders' needs become clearer. The model includes various elements that collectively provide a comprehensive understanding of the system.

8.5.1 Elements of the Analysis Model

1. **Scenario-Based Elements**

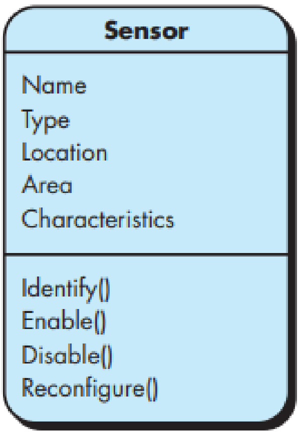
The system is described from the user’s point of view using a scenario-based approach. For example, basic use cases (Section 8.4) and their corresponding use case diagrams (Figure 8.2) evolve into more elaborate template-based use cases. Scenario-based elements of the requirements model are often the first 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.

2. **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. These are discussed in more detail in Chapter 10.

FIGURE 8.4

Class diagram for sensor

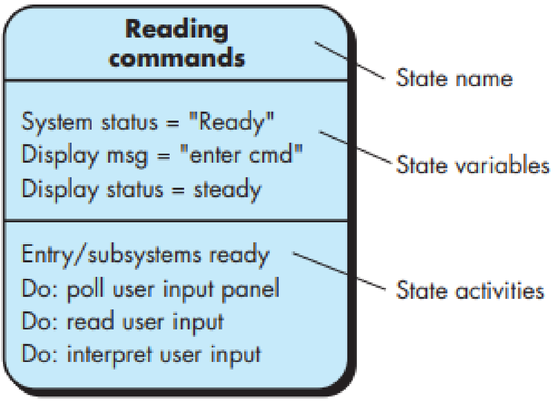


3. **Behavioral Elements**

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 behavior 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 in Figure 8.5 . In addition to behavioral representations of the system as a whole, the behavior of individual classes can also be modeled. Further discussion of behavioural modeling is presented in Chapter 11.

FIGURE 8.5

UML state diagram notation



Example Analysis Model Elements

Scenario-Based Example: University Library System

- **Use Case**: “Borrow a Book”
  - **Actors**: Library Patron
  - **Scenario**:
    1. Patron logs into their account.
    2. Searches for a book.
    3. Selects and borrows the book.
  - **Exceptions**:
    - Book not available
    - Account issues

Class-Based Example: University Library System

- **Class Diagram**:
  - **Classes**: Book, Patron, LibraryAccount
  - **Attributes**: Book has title, author, availability; Patron has name, libraryID.
  - **Operations**: BorrowBook(), ReturnBook()

Behavioral Example: SafeHome Security System

- **State Diagram**:
  - **States**: “Ready,” “Not Ready,” “Armed Stay,” “Armed Away”
  - **Events**: Entering password, selecting activation mode
  - **Actions**: System updates status, triggers alarm

8.5.2 Analysis Patterns

9.1 REQUIREMENTS A NALYSIS

Requirements analysis specifies the operational characteristics of software, its interface with other system elements, and the constraints it must meet. This phase elaborates on the basic requirements established during the inception, elicitation, and negotiation tasks of requirements engineering (refer to Chapter 8).

The requirements modeling process results in one or more of the following types of models:

- **Scenario-Based Models**: These depict requirements from the perspective of various system “actors” or users.
- **Class-Oriented Models**: These represent object-oriented classes (attributes and operations) and their collaboration to fulfill system requirements.
- **Behavioral and Patterns-Based Models**: These illustrate how the software behaves in response to external “events.”
- **Data Models**: These show the information domain relevant to the problem.
- **Flow-Oriented Models**: These represent functional elements of the system and how data is transformed as it moves through the system.

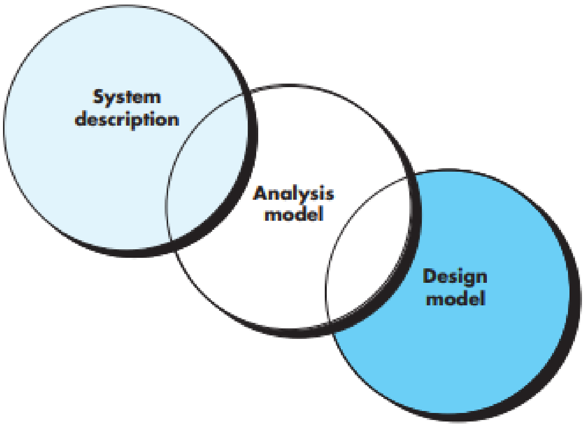
These models provide a software designer with information that can be translated into architectural, interface, and component-level designs. They also offer a basis for assessing quality once the software is built.

Quote: “Any one ‘view’ of requirements is insufficient to understand or describe the desired behavior of a complex system.”
— Alan M. Davis

9.1.1 Overall Objectives and Philosophy

FIGURE 9.1

The requirements model as a bridge between the system description and the design model



Throughout analysis modeling, your primary focus is on what, not how. You need to determine:

- What user interaction occurs in specific circumstances
- What objects the system manipulates
- What functions the system must perform
- What behaviors the system exhibits
- What interfaces are defined
- What constraints apply

Complete specification of requirements may not be possible at this stage. The customer might be unsure of precisely what is required for certain aspects of the system, and the developer might be unsure that a specific approach will adequately accomplish function and performance. These uncertainties advocate for an iterative approach to requirements analysis and modeling. The analyst should model what is known and use that model as the foundation for designing the software increment.

The requirements model must achieve three primary objectives:

1. Describe what the customer requires.
2. Establish a basis for the creation of a software design.
3. Define a set of requirements that can be validated once the software is built.

The analysis model bridges the gap between a system-level description, which outlines overall system or business functionality achieved by applying software, hardware, data, human, and other system elements, and a software design (covered in Chapters 12 through 18) that details the software's application architecture, user interface, and component-level structure. This relationship is illustrated in Figure 9.1.

It is important to note that all elements of the requirements model will be directly traceable to parts of the design model. A clear division of tasks between analysis and design is not always possible. Some design invariably occurs during analysis, and some analysis will be conducted during design.

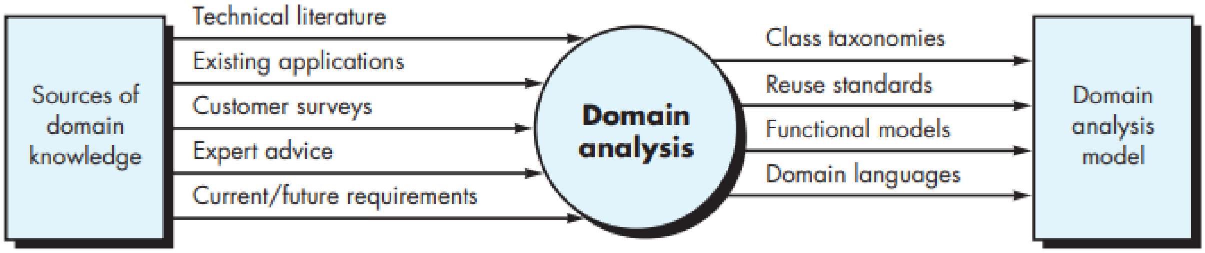
9.1.2 Analysis Rules of Thumb

Arlow and Neustadt [Arl02] suggest several worthwhile rules of thumb for creating the analysis model:

- **Focus on Visible Requirements**: The model should concentrate on requirements that are visible within the problem or business domain. Maintain a high level of abstraction and avoid getting bogged down in details that explain how the system will work.
- **Enhance Understanding**: Each element of the requirements model should contribute to an overall understanding of software requirements, providing insight into the information domain, function, and behavior of the system.
- **Delay Nonfunctional Models**: Postpone consideration of infrastructure and other nonfunctional models until the design phase. While a database may be required, details about the classes needed to implement it, the functions to access it, and its behavior should be addressed only after completing problem domain analysis.
- **Minimize Coupling**: Strive to minimize coupling throughout the system. While it's important to represent relationships between classes and functions, efforts should be made to reduce the level of interconnectedness if it becomes too high.
- **Provide Stakeholder Value**: Ensure that the requirements model offers value to all stakeholders. Different constituencies have various uses for the model. For instance, business stakeholders can use the model to validate requirements, designers can use it as a basis for design, and QA personnel can use it to plan acceptance tests.
- **Simplicity**: Keep the model as simple as possible. Avoid adding diagrams that do not provide new information and refrain from using complex notational forms when a simple list will suffice.

9.1.3 Domain Analysis

FIGURE 9.2 Input and output for domain analysis



In the discussion of requirements engineering (Chapter 8), we noted that analysis patterns often recur across many applications within a specific business domain. If these patterns are defined and categorized in a manner that allows you to recognize and apply them to solve common problems, the creation of the analysis model is expedited. More importantly, the likelihood of applying design patterns and executable software components grows dramatically. This improves time-to-market and reduces development costs.

But how are analysis patterns and classes recognized in the first place? Who defines them, categorizes them, and readies them for use on subsequent projects? The answers to these questions lie in domain analysis. Firesmith [Fir93] describes domain analysis in the following way:

"Software domain analysis is the identification, analysis, and specification of common requirements from a specific application domain, typically for reuse on multiple projects within that application domain... [Object-oriented domain analysis is] the identification, analysis, and specification of common, reusable capabilities within a specific application domain, in terms of common objects, classes, subassemblies, and frameworks."

The "specific application domain" can range from avionics to banking, from multimedia video games to software embedded within medical devices. The goal of domain analysis is straightforward: to find or create those analysis classes and/or analysis patterns that are broadly applicable so that they may be reused.

Using terminology introduced earlier in this book, domain analysis can be viewed as an umbrella activity for the software process. This means domain analysis is an ongoing software engineering activity not connected to any one software project. In a way, the role of a domain analyst is similar to the role of a master toolsmith in a heavy manufacturing environment. The job of the toolsmith is to design and build tools that may be used by many people doing similar but not necessarily the same jobs. Similarly, the role of the domain analyst is to discover and define analysis patterns, analysis classes, and related information that may be used by many people working on similar but not necessarily the same applications.

Figure 9.2 [Arn89] illustrates key inputs and outputs for the domain analysis process. Sources of domain knowledge are surveyed in an attempt to identify objects that can be reused across the domain.

9.2 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.

9.2.1 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.

In Chapter 8, 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 (Section 9.3.1) developed as part of requirements modelling.

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.

As further conversations with the stakeholder (who plays the role of a homeowner) progress, the requirements-gathering team develops use cases for each of the functions noted. In general, use cases are written first in an informal narrative fashion. If more formality is required, the same use case is rewritten using a structured format similar to the one proposed in Chapter 8 and reproduced later in this section as a sidebar.

To illustrate, consider the function access camera surveillance via the Internet—display camera views (ACS-DCV). The stakeholder who takes on the role of the homeowner actor might write the following narrative:

Use case: Access camera surveillance via the Internet—display camera views (ACS-DCV)
Actor: Homeowner
If I'm at a remote location, I can use any PC with appropriate browser software to log on to the SafeHome Products website. I enter my user ID and two levels of passwords and once I'm validated, I have access to all functionality for my installed SafeHome system. To access a specific camera view, I select "surveillance" from the major function buttons displayed. I then select "pick a camera" and the floor plan of the house is displayed. I then select the camera that I'm interested in. Alternatively, I can look at thumbnail snapshots from all cameras simultaneously by selecting "all cameras" as my viewing choice. Once I choose a camera, I select "view" and a one-frame-per-second view appears in a viewing window that is identified by the camera ID. If I want to switch cameras, I select "pick a camera" and the original viewing window disappears and the floor plan of the house is displayed again. I then select the camera that I'm interested in. A new viewing window appears.

A variation of a narrative use case presents the interaction as an ordered sequence of user actions. Each action is represented as a declarative sentence. Revisiting the ACS-DCV function, you would write:

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.

It is important to note that this sequential presentation does not consider any alternative interactions (the narrative is more free-flowing and did represent a few alternatives). Use cases of this type are sometimes referred to as primary scenarios [Sch98a].

9.2.2 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 [Sch98a]:

- 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 behaviour at this point (e.g., behaviour 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 behaviour. 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 selections 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" [Coc01b] 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 to handle the condition noted.

9.2.3 Writing a Formal Use Case

The informal use cases presented in Section 9.2.1 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 (Section 9.2.2). Additional headings may or may not be included and are reasonably self-explanatory.

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. Figure 9.4 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 nonfunctional 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.

However, scenario-based modeling is appropriate for a significant majority of all situations that you will encounter as a software engineer. If developed properly, the use case can provide substantial benefit as a modeling tool.

Suggest who might be stakeholders in a hospital management system. Explain why it is almost inevitable that the requirements of different stakeholders will conflict in some way.

In a Hospital Management System (HMS), stakeholders include anyone who has an interest or investment in the system's development, deployment, and usage. Here are some key stakeholders and why their requirements might conflict:

Key Stakeholders:

1. **Hospital Administration**
   - **Responsibilities:** Overseeing hospital operations, budgeting, and strategic planning.
   - **Requirements:** Cost-effective solutions, efficient management of hospital resources, and comprehensive reporting tools.
2. **Doctors and Medical Staff**
   - **Responsibilities:** Providing medical care, diagnosis, treatment, and patient management.
   - **Requirements:** User-friendly interfaces for patient records, quick access to patient history, and tools for collaboration with other medical staff.
3. **Nurses and Support Staff**
   - **Responsibilities:** Assisting doctors, patient care, and administrative support.
   - **Requirements:** Easy-to-use patient management tools, scheduling systems, and efficient workflow management.
4. **Patients**
   - **Responsibilities:** Receiving care, managing personal health records, and communicating with healthcare providers.
   - **Requirements:** Secure access to personal health records, appointment scheduling, and communication channels with healthcare providers.
5. **IT Staff**
   - **Responsibilities:** Maintaining the HMS, ensuring data security, and system integration.
   - **Requirements:** Scalable architecture, robust security measures, and ease of maintenance and updates.
6. **Regulatory Bodies**
   - **Responsibilities:** Ensuring compliance with healthcare regulations and standards.
   - **Requirements:** Adherence to legal standards, data privacy, and accurate reporting mechanisms.
7. **Insurance Companies**
   - **Responsibilities:** Processing claims, managing patient insurance information, and providing coverage.

    - **Requirements:** Accurate billing, seamless integration with hospital records, and efficient claim processing.
8. **Pharmacy Staff**
    - **Responsibilities:** Managing medications, dispensing drugs, and maintaining inventory.
    - **Requirements:** Integration with patient records, inventory management, and prescription tracking.
9. **Vendors and Suppliers**
    - **Responsibilities:** Providing medical equipment, supplies, and services.
    - **Requirements:** Efficient order processing, inventory management, and timely payment systems.

Conflict of Requirements:

1. **Different Priorities:**
    - **Example:** Hospital administration might prioritize cost-effectiveness, while doctors and medical staff might require advanced (and potentially expensive) diagnostic tools.
2. **Usability vs. Security:**
    - **Example:** IT staff may implement stringent security measures to protect patient data, but these measures might make the system cumbersome for doctors and nurses who need quick access during emergencies.
3. **Compliance vs. User Convenience:**
    - **Example:** Regulatory bodies require detailed reporting and compliance with healthcare standards, which might add extra steps for doctors and nurses, thereby slowing down their workflow.
4. **Customization vs. Standardization:**
    - **Example:** Different departments might need customized features to suit their specific workflows, while the IT department might push for a standardized system to ease maintenance and integration.
5. **Immediate vs. Long-term Needs:**
    - **Example:** Doctors might need immediate improvements to address current patient care issues, while the administration might focus on long-term strategic goals that do not align with short-term needs.
6. **Resource Allocation:**
    - **Example:** Financial resources might be limited, leading to conflicts over whether to invest in new medical equipment, enhance IT infrastructure, or expand patient care facilities.

8.6 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, 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."**
3. **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. Fricker [Fri10] and his colleagues suggest replacing the traditional handoff of requirements specifications to software teams with a bidirectional communication process called handshaking. In handshaking, the software team proposes solutions to requirements, describes their impact, and communicates their intentions to customer representatives. The customer representatives review the proposed solutions, focusing on missing features and seeking clarification of novel requirements. Requirements are determined to be good enough if the customers accept the proposed solution.

Handshaking allows detailed requirements to be delegated to software teams. The teams need to elicit requirements from customers (e.g., product users and domain experts), thereby improving product acceptance. Handshaking tends to improve identification, analysis, and selection of variants and promotes win-win outcomes.