Understanding Requirements: Requirements Engineering, Establishing the ground work, Eliciting Requirements, Developing use cases, Building the requirements model, Negotiating Requirements, Validating Requirements Textbook 1: Chapter 5: 5.1 to 5.7

Requirements Engineering

Requirements engineering provides the appropriate mechanism for understanding what the customer wants, analyzing need, assessing feasibility, negotiating a reasonable solution, specifying the solution unambiguously, validating the specification, and managing the requirements as they are transformed into an operational system . It 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: In general, most projects begin when a business need is identified or a potential new market or service is discovered. Stakeholders from the business community (e.g., business managers, marketing people, product managers) define a business case for the idea, try to identify the breadth and depth of the market, do a rough feasibility analysis, and identify a working description of the project's scope. All of this information is subject to change, but it is sufficient to precipitate discussions with the software engineering organization.

At project inception, you establish a basic understanding of the problem, the people who want a solution, the nature of the solution that is desired, and the effectiveness of preliminary communication and collaboration between the other stakeholders and the software team.

Elicitation: Means Figuring Out What exactly we need as per user or customer. Its hard thing because: According to Christel and kang

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

Elaboration. The information obtained from the customer during inception and elicitation is expanded and refined during elaboration. This task focuses on developing a refined requirements model (Chapters 6 and 7) that identifies various aspects of software function, behavior, and information.

Elaboration is driven by the creation and refinement of user scenarios that describe how the end user (and other actors) will interact with the system. Each user scenario is parsed to extract analysis classes—business domain entities that are visible to the end user. The attributes of each analysis class are defined, and the services that are required by each class are identified. The relationships and collaboration between classes are identified, and a variety of supplementary diagrams are produced.

Negotiation. It isn't unusual for customers and users to ask for more than can be achieved, given limited business resources. It's also relatively common for different customers or users to propose conflicting requirements, arguing that their version is "essential for our special needs." You have to reconcile these conflicts through a process of negotiation. Customers, users, and other stakeholders are asked to rank requirements and then discuss conflicts in priority. Using an iterative approach that prioritizes requirements, assesses their cost and risk, and addresses internal conflicts, requirements are eliminated, combined, and/or modified so that each party achieves some measure of satisfaction.

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

Validation. The products produced in effect of requirements engineering are assessed for quality during a validation step. Requirements validation examines the specification to ensure that all software requirements have been stated unambiguously; that inconsistencies, omissions(not included), and errors have been detected and corrected; and that the products confirm to the standards established for the process, the project, and the product.

The primary requirements validation mechanism is the technical review . The review team that validates requirements includes software engineers, customers, users, and other stakeholders who examine the specification looking for errors in content, areas where clarification may be required, missing information, inconsistencies, conflicting requirements, or unrealistic (unachievable) requirements.

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.

Establishing the ground work

Identifying Stakeholders

- **Definition of Stakeholder**: A stakeholder is anyone who benefits from the system being developed, either directly or indirectly.

- **Usual Stakeholders**:
    - **Business Operations Managers**: Oversee business processes and operations.
    - **Product Managers**: Define product features and roadmap.
    - **Marketing Personnel**: Promote and sell the product.
    - **Internal and External Customers**: Users and purchasers of the product.
    - **End Users**: Individuals who will use the system.
    - **Consultants**: Provide expert advice and recommendations.
    - **Product Engineers**: Design and develop the product.
    - **Software Engineers**: Implement the technical aspects of the system.
    - **Support and Maintenance Engineers**: Ensure the system runs smoothly and fix issues.
- **Stakeholder Identification**:
    - Start with a preliminary list of stakeholders.
    - Expand this list by asking each stakeholder for additional contacts, ensuring comprehensive coverage.

Recognizing Multiple Viewpoints

- **Importance of Multiple Viewpoints**: Different stakeholders have different priorities and needs.
    - **Marketing**: Wants features that attract and retain customers.
    - **Business Managers**: Focus on cost, budget, and timing.
    - **End Users**: Prefer user-friendly and intuitive features.
    - **Software Engineers**: Emphasize robust technical infrastructure.
    - **Support Engineers**: Focus on the ease of maintenance and support.

- **Handling Multiple Viewpoints**:
    - Collect all stakeholder inputs.
    - Identify and manage inconsistencies and conflicts in requirements.
    - Help decision-makers prioritize and choose the most critical and feasible requirements.

Working toward Collaboration

- **Challenges of Multiple Opinions**: Different stakeholders may have different, sometimes conflicting, views on requirements.
- **Role of Requirements Engineer**:
    - Find common ground among stakeholders.
    - Identify and resolve conflicts in requirements.

- Facilitate collaboration through communication and negotiation.
- Sometimes a project champion (e.g., a senior manager) makes the final decision on which requirements to prioritize.

Asking the First Questions

- **Context-Free Questions**: These initial questions help understand the project's broader context without specific technical constraints.
  - **Stakeholder Identification**:
    - Who requested the work?
    - Who will use the solution?
    - What economic benefits will the solution provide?
    - Are there alternative solutions available?
  - **Understanding the Problem**:
    - Define what good output from the solution looks like.
    - Identify the problems the solution aims to solve.
    - Describe the business environment where the solution will be used.
    - Identify any special performance issues or constraints.
  - **Communication Effectiveness (Meta-questions)**:
    - Are you the right person to answer these questions? Are your answers official?
    - Are my questions relevant to your problem?
    - Am I overwhelming you with too many questions?
    - Can anyone else provide additional information?
    - Is there anything else I should be asking?

Eliciting Requirements

Requirements Elicitation Overview

Requirements elicitation, also known as requirements gathering, is a crucial phase in software engineering that combines elements of problem-solving, elaboration, negotiation, and specification. The aim is to gather comprehensive and accurate requirements by encouraging a collaborative, team-oriented approach where stakeholders work together to identify problems, propose solutions, negotiate approaches, and specify preliminary requirements.

Collaborative Requirements Gathering

Various approaches to collaborative requirements gathering exist, but all follow basic guidelines to ensure effective communication and thorough requirement collection:

- **Meeting Participation**: Both software engineers and stakeholders attend the meetings.
- **Preparation and Participation Rules**: Established rules ensure organized and productive meetings.
- **Structured but Flexible Agenda**: An agenda that covers key points while allowing free flow of ideas.
- **Facilitator Role**: A facilitator (customer, developer, or outsider) guides the meeting to maintain focus.
- **Definition Mechanism**: Tools like worksheets, flip charts, wall stickers, or electronic forums are used to document ideas.

The goal is to create an environment conducive to identifying the problem, proposing solutions, negotiating approaches, and specifying a preliminary set of requirements. A typical scenario involves initial meetings to establish project scope, followed by more detailed meetings where stakeholders review and refine requirements.

Example Scenario: In the SafeHome project, a marketing person might draft a product request outlining the home security function. Stakeholders review this request, list relevant objects, services, constraints, and performance criteria before the meeting. These lists are then used to facilitate discussion, combine ideas, and develop a consensus on the system requirements.

Quality Function Deployment (QFD)

Quality Function Deployment (QFD) is a technique that translates customer needs into technical requirements, emphasizing customer satisfaction throughout the software engineering process. QFD identifies three types of requirements:

- **Normal Requirements:** 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. Examples of normal requirements might be requested types of graphical displays, specific system functions, and defined levels of performance.
- **Expected Requirements**: Implicit, fundamental requirements if missing, it causes dissatisfaction (e.g., ease of use, reliability). These 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. Examples of expected requirements are: ease of human/machine interaction, overall operational correctness and reliability, and ease of software installation.
- **Exciting Requirements**: Unexpected features that delight customers (e.g., advanced functionalities).These features go beyond the customer's expectations and prove to be very satisfying when present. For example, software for a new mobile phone comes with standard features, but is

coupled with a set of unexpected capabilities (e.g., multitouch screen, visual voice mail) that delight every user of the product.

QFD uses customer interviews, surveys, and historical data to gather raw requirements, which are then reviewed and refined with the customer to ensure alignment with their needs.

Usage Scenarios

Usage scenarios, or use cases, describe how the system will be used by different user classes. These scenarios help developers understand system functions and features from the user's perspective, providing a basis for moving into technical software engineering activities.

Elicitation Work Products

The work products from requirements elicitation vary based on the project's size but generally include:

- **Statement of Need and Feasibility**: Describes the project's purpose and feasibility.
- **Bounded Statement of Scope**: Defines the project's scope and boundaries.
- **List of Stakeholders**: Identifies all participants in the requirements gathering process.
- **Description of Technical Environment**: Outlines the system's operating environment.
- **List of Requirements**: Organized by function, including domain constraints.
- **Usage Scenarios**: Provide insights into system use under various conditions.
- **Prototypes**: Developed to clarify and refine requirements.

Developing use cases

Defining Use Cases

According to Alistair Cockburn, a use case captures a contract describing the system’s behavior in response to requests from stakeholders. Essentially, a use case tells a stylized story about how an end user interacts with the system under specific circumstances. The story can be presented as narrative text, task outlines, template-based descriptions, or diagrams, depicting the software from the end user’s perspective.

Identifying Actors

The first step in writing a use case is defining the set of "actors" involved. Actors are roles played by people or devices interacting with the system. An actor is anything external to the system that communicates with it, aiming to achieve specific goals.

Key Points About Actors:

- **Roles vs. End Users**: An actor represents a role, not necessarily a specific person. One user can play multiple roles.
- **Primary Actors**: Directly interact with the system to achieve main functions.
- **Secondary Actors**: Support the primary actors in achieving their goals.

Example: In a manufacturing cell with robots and numerically controlled machines, the machine operator may play roles such as programmer, tester, monitor, and troubleshooter, each representing a distinct actor.

Developing Use Cases

Once actors are identified, use cases can be developed. They answer key questions about the actors' interactions with the system:

1. Who are the primary and secondary actors?
2. What are the actors' goals?
3. What preconditions exist before the story begins?
4. What main tasks or functions are performed by the actors?
5. What exceptions might occur during the interaction?
6. What variations in interaction are possible?
7. What system information will the actors acquire, produce, or change?
8. Will the actors inform the system about changes in the external environment?
9. What information do the actors desire from the system?
10. Do the actors need to be informed about unexpected changes?

Example Use Case: SafeHome System

Actors: Homeowner, setup manager, sensors, monitoring and response subsystem.

Homeowner Interactions:

- Entering a password for access.
- Inquiring about the status of security zones and sensors.

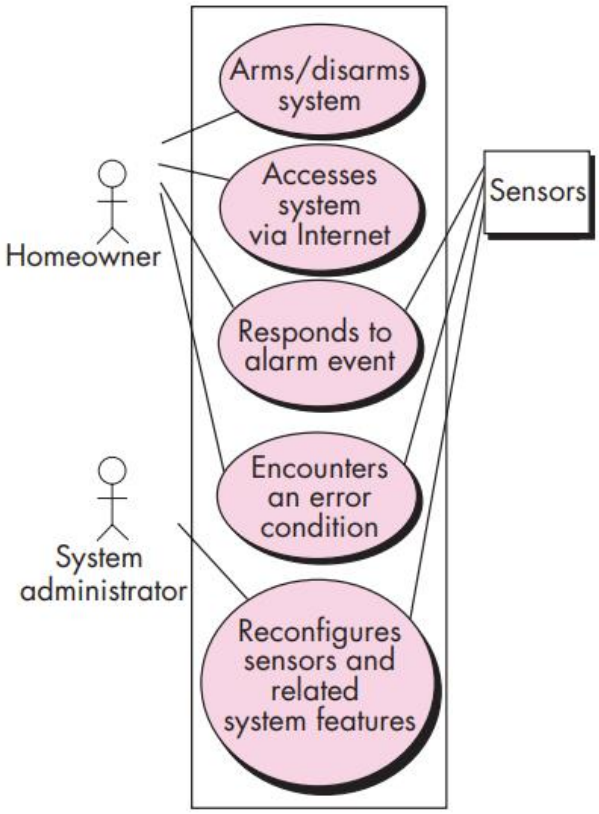
- Activating/deactivating the system.
- Pressing the panic button in emergencies.

Basic Use Case for System Activation:

1. The homeowner checks if the system is ready.
2. The homeowner enters a password.
3. The homeowner selects "stay" or "away" to activate the system.
4. A red alarm light indicates activation.

Detailed Use Case Template Use Case: **InitiateMonitoring**

- **Primary Actor**: Homeowner
- **Goal in Context**: Set the system to monitor sensors.
- **Preconditions**: System is programmed with a password and sensor recognition.
- **Trigger**: Homeowner decides to activate the system.
- **Scenario**:
    1. Homeowner observes control panel.
    2. Homeowner enters password.
    3. Homeowner selects "stay" or "away."
    4. Homeowner observes the alarm light.
- **Exceptions**:
    1. System not ready: homeowner checks and closes sensors.
    2. Incorrect password: re-enter correct password.
    3. Unrecognized password: contact support.
    4. "Stay" selected: perimeter sensors activated.
    5. "Away" selected: all sensors activated.
- **Priority**: Essential
- **Frequency of Use**: Multiple times per day
- **Channels to Actors**: Control panel interface
- **Secondary Actors**: Support technician, sensors
- **Open Issues**:
    1. Alternative password methods.
    2. Additional text messages on the control panel.
    3. Password entry time limit.
    4. Early deactivation options.



Building the requirements model

Analysis Model Overview

The intent of the analysis model is to describe the required informational, functional, and behavioral domains for a computer-based system. It evolves dynamically as more information about the system is gathered and as stakeholders refine their understanding of their requirements. Therefore, the analysis model is a snapshot of requirements at any given time and is expected to change.

Stability and Volatility in the Model

- **Stable Elements**: Form a solid foundation for design tasks.
- **Volatile Elements**: Indicate areas where stakeholders are still uncertain about requirements.

Elements of the Requirements Model

There are various ways to represent the requirements for a computer-based system. Some practitioners prefer a single mode (e.g., use cases), while others advocate for multiple modes to provide different perspectives and uncover potential issues.

Common Elements in Most Requirements Models

1. **Scenario-Based Elements**:
    - Describe the system from the user's point of view.
    - Often the first part of the model developed, serving as input for other elements.
    - Examples: Basic use cases, use-case diagrams, and template-based use cases.
    - **Figure 5.3**: UML activity diagram for eliciting requirements and representing them using use cases.
2. **Class-Based Elements**:
    - Each usage scenario implies a set of objects manipulated by actors.
    - Objects are categorized into classes with similar attributes and behaviors.
    - Examples: UML class diagrams depicting attributes and operations.
    - **Figure 5.4**: UML class diagram for the SafeHome Sensor class.
3. **Behavioral Elements**:
    - Depict the system's behavior, influencing design and implementation.
    - Examples: State diagrams representing states, events, and actions.
    - **Figure 5.5**: UML state diagram for the SafeHome control panel software.

Analysis Patterns

Recurrent problems across projects within a specific application domain lead to the development of analysis patterns. These patterns provide reusable solutions and accelerate the development of abstract analysis models.

Benefits of Analysis Patterns

1. **Speed up development**:
    - Provide reusable models with examples.
    - Describe advantages and limitations.
2. **Facilitate transformation into a design model**:
    - Suggest design patterns and reliable solutions for common problems.

Integration of Analysis Patterns

- Integrated into the analysis model by referencing the pattern name.
- Stored in a repository for easy access by requirements engineers.
- Presented in a standard template format.

UML activity diagrams for eliciting requirements

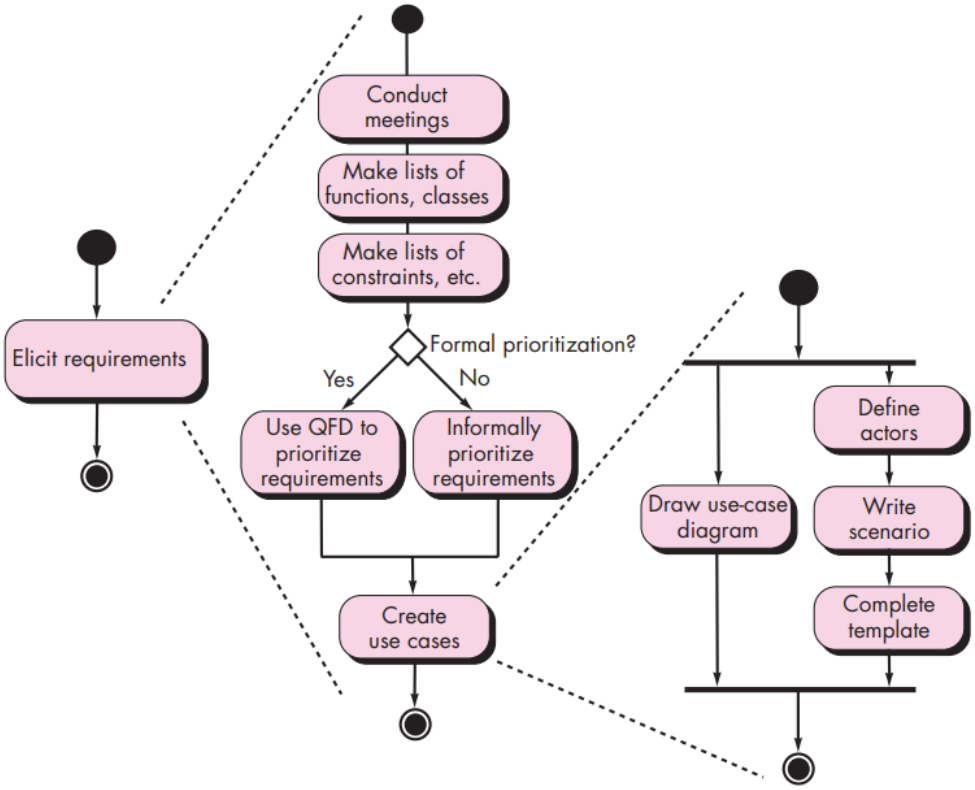
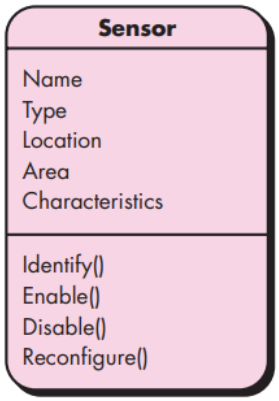


FIGURE 5.4

Class diagram for sensor



Negotiating Requirements

In an ideal requirements engineering context, inception, elicitation, and elaboration tasks determine customer requirements in sufficient detail to proceed to subsequent software engineering activities. However, this rarely happens in reality. Often, negotiation with one or more stakeholders is necessary.

Balancing Needs and Constraints

Stakeholders must balance functionality, performance, and other product or system characteristics against cost and time-to-market. The negotiation aims to develop a project plan that meets stakeholder needs while reflecting real-world constraints (e.g., time, people, budget) on the software team.

Win-Win Negotiation Strategy

The best negotiations strive for a "win-win" result, where:

- **Stakeholders** win by getting a system or product that satisfies most of their needs.
- **Software team** wins by working within realistic and achievable budgets and deadlines.

Boehm's Negotiation Activities

Barry Boehm defines a set of negotiation activities at the beginning of each software process iteration. These activities include:

1. **Identification of Key Stakeholders**: Determine who the key stakeholders are for the system or subsystem.
2. **Determination of Stakeholders' "Win Conditions"**: Understand what each stakeholder needs to achieve a "win."
3. **Negotiation of Win Conditions**: Reconcile stakeholders' win conditions into a set of win-win conditions for all concerned, including the software team.

Validating Requirements

As each element of the requirements model is created, it is critical to examine it for inconsistencies, omissions, and ambiguity. Requirements are then prioritized by stakeholders and grouped within requirements packages for implementation as software increments. The following questions should guide the review of the requirements model:

1. **Consistency with Overall Objectives**:
    - Is each requirement aligned with the overall goals and objectives of the system or product?
2. **Proper Level of Abstraction**:
    - Are all requirements specified at the appropriate level of abstraction?
    - Are there requirements that include too much technical detail for this stage of development?
3. **Necessity and Priority**:
    - Is the requirement essential to the system's objectives, or is it an additional feature that may not be critical?
4. **Clarity and Boundaries**:
    - Is each requirement clearly defined and unambiguous?
    - Are the boundaries and scope of each requirement well-defined?
5. **Attribution**:
    - Does each requirement have clear attribution? Is the source (typically a specific individual or stakeholder) noted for each requirement?
6. **Conflict Check**:
    - Do any requirements conflict with each other?
7. **Feasibility**:
    - Is each requirement achievable within the technical environment that will house the system or product?
8. **Testability**:
    - Can each requirement be tested once it is implemented?
9. **Reflection of System Information, Function, and Behavior**:
    - Does the requirements model accurately reflect the necessary information, function, and behavior of the system to be built?

Requirements Modeling Scenarios, Information and Analysis classes: Requirement Analysis, Scenario based modeling, UML models that supplement the Use Case, Data modeling Concepts class Based Modeling.

Requirement Analysis

Requirements Analysis: An Overview

Definition:

- Requirements analysis is the process of determining the software's operational characteristics, its interface with other system elements, and the constraints it must meet.

Key Objectives:

1. **Describe Customer Needs:** Clearly document what the customer requires from the system.
2. **Basis for Design:** Establish a foundation for creating the software design.
3. **Validation:** Define requirements that can be validated once the software is built.

Philosophy:

- Focus on **what** the system must do, not **how** it will do it.
- Accept that complete specification may not be possible at this stage due to uncertainties from both the customer and the developer.

Iterative Approach:

- Model what is known and use it as the basis for the software design incrementally.

Types of Requirements Models

1. **Scenario-Based Models:**
    - Depict requirements from the perspective of various system actors.
    - Example: Use cases showing how users interact with the system, such as logging in or processing an order.
2. **Data Models:**
    - Illustrate the information domain of the problem.
    - Example: Entity-Relationship diagrams showing data attributes and relationships, such as a customer table linked to an order table.

3. **Class-Oriented Models:**
    - Represent object-oriented classes, their attributes, operations, and collaborations.
    - Example: UML class diagrams detailing the properties and methods of objects like Customer, Order, and Product.
4. **Flow-Oriented Models:**
    - Represent functional elements of the system and how data is transformed as it moves through the system.
    - Example: Data flow diagrams showing processes like Process Order and Generate Invoice.
5. **Behavioral Models:**
    - Show how the software behaves in response to external events.
    - Example: State diagrams illustrating the states and transitions of an Order object (e.g., New, Processed, Shipped, Delivered).

Rules of Thumb for Analysis

1. **Focus on the Problem Domain:**
    - Maintain a high level of abstraction and avoid excessive details on implementation specifics.
    - Example: Focus on what data needs to be stored rather than how it will be stored in a database.
2. **Enhance Understanding:**
    - Ensure each element of the model adds to the overall understanding of the system's requirements.
    - Example: Use clear and concise diagrams that convey essential information without clutter.
3. **Delay Nonfunctional Details:**
    - Consider infrastructure and other nonfunctional elements only after completing problem domain analysis.
    - Example: Identify the need for a database but defer the design of database schemas to the design phase.
4. **Minimize Coupling:**
    - Reduce the level of interconnectedness between system components.
    - Example: Design classes and functions to be as independent as possible to simplify maintenance and enhance modularity.
5. **Stakeholder Value:**
    - Ensure the model provides value to all stakeholders, including business stakeholders, designers, and QA teams.

- Example: Business stakeholders use the model to validate requirements; designers use it as a basis for design; QA teams use it to plan acceptance tests.

6. **Simplicity:**
   - Keep the model as simple as possible, avoiding unnecessary diagrams and complex notations.
   - Example: Use simple lists or tables where appropriate, rather than overly complex diagrams.

Domain Analysis

Definition:

- Domain analysis is the identification, analysis, and specification of common requirements within a specific application domain for reuse in multiple projects.

Goals:

- Identify reusable analysis patterns and classes to expedite the creation of analysis models.
- Improve time-to-market and reduce development costs.

Process:

- Survey sources of domain knowledge to identify reusable objects and patterns.
- Example: In the banking domain, identify common elements such as accounts, transactions, and customers.

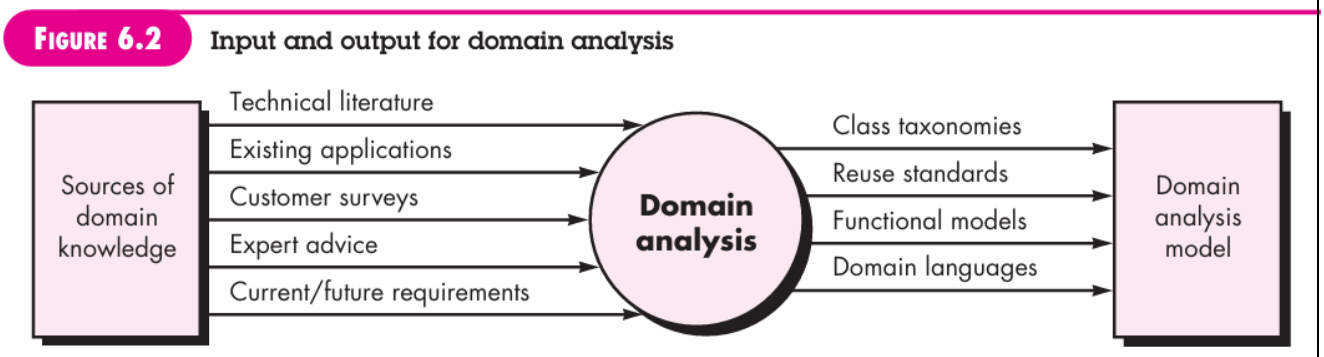


FIGURE 6.2 Input and output for domain analysis

Role of Domain Analyst:

- Discover and define analysis patterns, analysis classes, and related information for reuse.
- Example: A domain analyst in healthcare might develop reusable models for patient records, appointments, and billing.

Approaches to Analysis Modeling

1. **Structured Analysis:**
    - Treats data and processes as separate entities.
    - Models data attributes and relationships, and how data is transformed by processes.
    - Example: Data flow diagrams and entity-relationship diagrams.
2. **Object-Oriented Analysis:**
    - Defines classes and their collaborations to meet customer requirements.
    - Uses UML and the Unified Process.
    - Example: UML class diagrams, sequence diagrams, and use case diagrams.

Combining Approaches:

- A hybrid approach can be used, selecting the best elements from both structured and object-oriented analysis to suit the project needs.
- Example: Use scenario-based and class-oriented models together to capture both user interactions and object relationships.