Chapter 4 Requirement Phase

Contents:

Rapid Prototyping method, Specification phase, Specification Document, Formal methods of developing specification document, Examples of other semi - formal methods of using Finite-State-Machines, Petri nets and E- Language. Analysis phase: Use case Modeling, Class Modeling, Dynamic Modeling, Testing during OO Analysis.

Rapid Prototyping:

- Rapid prototyping is a development approach where software is swiftly created to demonstrate the core functionalities of the intended product. Unlike traditional development methods that involve extensive planning and implementation of all features, rapid prototyping focuses on quickly building a functional model that captures the essence of the final product. Here's a detailed breakdown of the rapid prototyping process:
- **Identifying Key Functionalities:** The first step involves understanding the essential features and functionalities required in the final product. This typically involves discussions with the client or stakeholders to determine their needs and priorities. For example, in the case of an apartment management system, key functionalities might include tenant management, occupancy reporting, and payment tracking.
- **Building the Prototype:** Based on the identified functionalities, developers create a prototype that includes only the most critical features. For instance, in the apartment management system example, the prototype might include screens for adding new tenants and generating occupancy reports. However, non-essential features like error-checking mechanisms or complex tax calculations are omitted at this stage to expedite development.
- **User Feedback and Iteration:** The prototype is then presented to the client and intended users for evaluation. Users interact with the prototype, providing feedback on its usability, functionality, and overall suitability. Developers observe these interactions and take notes on areas that require improvement. This feedback loop is crucial for refining the prototype and ensuring it aligns with the client's needs.
- **Speed and Agility:** Rapid prototyping emphasizes speed and agility in development. The focus is on delivering a functional prototype quickly, even if it has imperfections or lacks certain features. The goal is to facilitate rapid understanding and alignment between the development team and the client. Therefore, developers prioritize building the prototype as efficiently as possible, using languages and tools conducive to rapid development.
- **Adaptability and Change Management:** Another key aspect of rapid prototyping is the prototype's adaptability to change. If the initial version of the prototype does not fully meet the client's requirements, developers quickly iterate on it to incorporate feedback and make necessary adjustments. This iterative approach ensures that the prototype evolves rapidly to better satisfy the client's needs.
- **Language and Tools Selection:** Rapid prototyping often employs fourth-generation languages (4GL) and interpreted languages that enable fast development and iteration. Languages like HTML, Perl, Smalltalk, Prolog, and Lisp are commonly used for rapid prototyping due to their flexibility and ease of modification. The choice of language and tools prioritizes the ability to produce and modify prototypes quickly.

- **Effectiveness for User Interface Development:** Rapid prototyping is particularly effective for developing user interfaces. By quickly building and iterating on interface designs, developers can gather early feedback from users, refine interface elements, and ensure usability and user satisfaction.

Specification Phase:

The requirements analysis and specification phase starts after the feasibility study stage is complete and the project has been found to be financially viable and technically feasible. The requirements analysis and specification phase ends when the requirements specification document has been developed and reviewed. The requirements specification document is usually called as the software requirements specification (SRS) document. The goal of the requirements analysis and specification phase can be stated in a nutshell as follows.

The goal of the requirements analysis and specification phase is to clearly understand the customer requirements and to systematically organise the requirements into a document called the Software Requirements Specification (SRS) document

Who carries out requirements analysis and specification?

Requirements analysis and specification activity is usually carried out by a few experienced members of the development team and it normally requires them to spend some time at the customer site. The engineers who gather and analyse customer requirements and then write the requirements specification document are known as system analysts in the software industry parlance. System analysts collect data pertaining to the product to be developed and analyse the collected data to conceptualise what exactly needs to be done. After understanding the precise user requirements, the analysts analyse the requirements to weed out inconsistencies, anomalies and incompleteness. They then proceed to write the software requirements specification (SRS) document.

The SRS document is the final outcome of the requirements analysis and specification phase.

REQUIREMENTS GATHERING AND ANALYSIS

- The complete set of requirements are almost never available in the form of a single document from the customer.
- The primary objective of the requirement's gathering task is to collect the requirements from the stakeholders.
- A stakeholder is a source of the requirements and is usually a person, or a group of persons who either directly or indirectly are concerned with the software.
- However, in practice it is very difficult to gather all the necessary information from a large number of stakeholders and from information scattered across several pieces of documents.
- Gathering requirements turns out to be especially challenging if there is no working model of the software being developed.
- Suppose a customer wants to automate some activity in his organisation that is currently being carried out manually.

Availability of a working model is usually of great help in requirements gathering.

For example, if the project involves automating the existing accounting activities of an organisation, then the task of the system analyst becomes a lot easier as he can immediately obtain the input and output forms and the details of the operational procedures.

In this context, consider that it is required to develop a software to automate the book-keeping activities involved in the operation of a certain office.

However, if a project involves developing something new for which no working model exists, then the requirements gathering activity becomes all the more difficult.

Typically, even before visiting the customer site, requirements gathering activity is started by studying the existing documents to collect all possible information about the system to be developed.

During visit to the customer site, the analysts normally interview the end-users and customer representatives,1 carry out requirements gathering activities such as questionnaire surveys, task analysis, scenario analysis, and form analysis.

In the following, we briefly discuss the important ways in which an experienced analyst gathers requirements:

- **Studying existing documentation:** The analyst usually studies all the available documents regarding the system to be developed before visiting the customer site.
    - Customers usually provide statement of purpose (SoP) document to the developers.
    - Typically, these documents might discuss issues such as the context in which the software is required, the basic purpose, the stakeholders, features of any similar software developed elsewhere, etc.
- **Interview:** Typically, there are many different categories of users of a software.
    - Each category of users typically requires a different set of features from the software.
    - Therefore, it is important for the analyst to first identify the different categories of users and then determine the requirements of each.
    - For example, the different categories of users of a library automation software could be the library members, the librarians, and the accountants.
    - In this technique, the analyst consolidates the requirements as understood by him in a document and then circulates it for the comments of the various categories of users.
    - This procedure is repeated till the different users agree on the set of requirements.
- **Task analysis:** The users usually have a black-box view of a software and consider the software as something that provides a set of services (functionalities).

We can therefore say that the software performs various tasks of the users.

In this context, the analyst tries to identify and understand the different tasks to be performed by the software.

For each identified task, the analyst tries to formulate the different steps necessary to realise the required functionality in consultation with the users.

Overall description of organisation of SRS document Product perspective:

This section needs to briefly state as to whether the software is intended to be a replacement for a certain existing system, or it is a new software. If the software being developed would be used as a component of a larger system, a simple schematic diagram can be given to show the major components of the overall system, subsystem interconnections, and external interfaces can be helpful.

- **Product features**: This section should summarize the major ways in which the software would be used. Details should be provided in Section 3 of the document. So, only a brief summary should be presented here.
- **User classes:** Various user classes that are expected to use this software are identified and described here. The different classes of users are identified by the types of functionalities that they are expected to invoke, or their levels of expertise in using computers.
- **Operating environment:** This section should discuss in some detail the hardware platform on which the software would run, the operating system, and other application software with which the developed software would interact.
- **Design and implementation constraints:** In this section, the different constraints on the design and implementation are discussed. These might include—corporate or regulatory policies; hardware limitations (timing requirements, memory requirements); interfaces to other applications; specific technologies, tools, and databases to be used; specific programming language to be used; specific communication protocols to be used; security considerations; design conventions or programming standards.
- **User documentation:** This section should list out the types of user documentation, such as user manuals, on-line help, and trouble-shooting manuals that will be delivered to the customer along with the software. Functional requirements for organisation of SRS document This section can classify the functionalities either based on the specific functionalities invoked by different users, or the functionalities that are available in different modes, etc., depending what may be appropriate.
- User class 1
    - Functional requirement 1.1
    - Functional requirement 1.2
- User class 2
    - Functional requirement 2.1
    - Functional requirement 2.2
- **External interface requirements User interfaces:** This section should describe a high-level description of various interfaces and various principles to be followed. The user interface description may include sample screen images, any GUI standards or style guides that are to be followed, screen layout constraints, standard push buttons (e.g., help) that will appear on every screen, keyboard shortcuts, error message display standards, etc. The details of the user interface design should be documented in a separate user interface specification document. Hardware interfaces: This section should describe the interface between the software and the hardware components of the system. This section may include the description of the supported device types, the nature of the data and control interactions between the software and the hardware, and the communication protocols to be used.
- **Software interfaces:** This section should describe the connections between this software and other specific software components, including databases, operating systems, tools, libraries,

and integrated commercial components, etc. Identify the data items that would be input to the software and the data that would be output should be identified and the purpose of each should be described.

- **Communications interfaces:** This section should describe the requirements associated with any type of communications required by the software, such as e-mail, web access, network server communications protocols, etc. This section should define any pertinent message formatting to be used. It should also identify any communication standards that will be used, such as TCP sockets, FTP, HTTP, or SHTTP. Specify any com munication security or encryption issues that may be relevant, and also the data transfer rates, and synchronisation mechanisms.

Other non-functional requirements for organisation of SRS document

It describes the non-functional requirements other than the design and implementation constraints and the external interface requirements that have been described in Sections 2 and 4 respectively. Performance requirements: Aspects such as number of transactions to be completed per second should be specified here. Some performance requirements may be specific to individual functional requirements or features. These should also be specified here.

- **Safety requirements:** Those requirements that are concerned with possible loss or damage that could result from the use of the software are specified here. For example, recovery after power failure, handling software and hardware failures, etc. may be documented here.
- **Security requirements:** This section should specify any requirements regarding security or privacy requirements on data used or created by the software. Any user identity authentication requirements should be described here. It should also refer to any external policies or regulations concerning the security issues. Define any security or privacy certifications that must be satisfied. For software that have distinct modes of operation, in the functional requirements section, the different modes of operation can be listed and, in each mode, the specific functionalities that are available for invocation can be organised as follows.
- Functional requirements
- Operation mode 1
    - Functional requirement 1.1
    - Functional requirement 1.2 2.
    - Operation mode 2
- Functional requirement 2.1
- Functional requirement 2.2

Specification of the behaviour may not be necessary for all systems. It is usually necessary for those systems in which the system behaviour depends on the state in which the system is, and the system transits among a set of states depending on some prespecified conditions and events. The behaviour of a system can be specified using either the finite state machine (FSM) formalism and any other alternate formalisms. The FSMs can used to specify the possible states (modes) of the system and the transition among these states due to occurrence of events

Why Spend Time and Resource to Develop an SRS Document?

A well-formulated SRS document finds a variety of usage other than the primary intended usage as a basis for starting the software development work. In the following subsection, we identify the important uses of a well-formulated SRS document:

Forms an agreement between the customers and the developers: A good SRS document sets the stage for the customers to form their expectation about the software and the developers about what is expected from the software. Reduces future reworks: The process of preparation of the SRS document forces the stakeholders to rigorously think about all of the requirements before design and development get underway. This reduces later redesign, recoding, and retesting. Careful review of the SRS document can reveal omissions, misunderstandings, and inconsistencies early in the development cycle.

Provides a basis for estimating costs and schedules: Project managers usually estimate the size of the software from an analysis of the SRS document. Based on this estimate they make other estimations such as the effort required to develop the software and the total cost of development. The SRS document also serves as a basis for price negotiations with the customer. The project manager also uses the SRS document for work scheduling.

Provides a baseline for validation and verification: The SRS document provides a baseline against which compliance of the developed software can be checked. It is also used by the test engineers to create the test plan.

Facilitates future extensions: The SRS document usually serves as a basis for planning future enhancements. Before we discuss about how to write an SRS document, we first discuss the characteristics of a good SRS document and the pitfalls that one must consciously avoid while writing an SRS document

Important Categories of Customer Requirements

A good SRS document, should properly categorize and organise the requirements into different sections [IEEE830]. As per the IEEE 830 guidelines, the important categories of user requirements are the following.

An SRS document should clearly document the following aspects of a software:

• Functional requirements

• Non-functional requirements — Design and implementation constraints — External interfaces required — Other non-functional requirements

• Goals of implementation.

In the following subsections, we briefly describe the different categories of requirements.

Table 4.1: Decision Table for the LMS Problem

Conditions

Valid selection	NO	YES	YES	YES
New member	-	YES	NO	NO
Renewal	-	NO	YES	NO
Cancellation	-	NO	NO	YES
Actions				

a. **Functional requirements:** The functional requirements capture the functionalities required by the users from the system that it is useful to consider a software as offering a set of functions {fi} to the user. These functions can be considered similar to a mathematical function f : I → O, meaning that a function transforms an element (ii) in the input domain (I) to a value (oi) in the output (O). This functional view of a system is shown schematically in Figure 4.1. Each function fi of the system can be considered as reading certain data ii, and then transforming a set of input data (ii) to the corresponding set of output data (oi). The functional requirements of the system, should clearly describe each functionality that the system would support along with the corresponding input and output data set.

2. **Non-functional requirements :**The non-functional requirements are non-negotiable obligations that must be supported by the software. The non-functional requirements capture those requirements of the customer that cannot be expressed as functions (i.e., accepting input data and producing output data). Non-functional requirements usually address aspects concerning external interfaces, user interfaces, maintainability, portability, usability, maximum number of concurrent users, timing, and throughput (transactions per second, etc.). The non-functional requirements can be critical in the sense that any failure by the developed software to achieve some minimum defined level in these requirements can be considered as a failure and make the software unacceptable by the customer. The IEEE 830 standard recommends that out of the various non-functional requirements, the external interfaces, and the design and implementation constraints should be documented in two different sections. The remaining non-functional requirements should be documented later in a section and these should include the performance and security requirements. In the following subsections, we discuss the different categories of non-functional requirements that are described under three different sections:
3. **Design and implementation constraints:** Design and implementation constraints are an important category of non-functional requirements describe any items or issues that will limit the options available to the developers. Some of the example constraints can be—corporate or regulatory policies that needs to be honoured; hardware limitations; interfaces with other applications; specific technologies, tools, and databases to be used; specific communications protocols to be used; security considerations; design conventions or programming standards to be followed, etc. Consider an example of a constraint that can be included in this section—Oracle DBMS needs to be used as this would facilitate easy interfacing with other applications that are already operational in the organisation.
4. **External interfaces required:** Examples of external interfaces are— hardware, software and communication interfaces, user interfaces, report formats, etc. To specify the user interfaces, each interface between the software and the users must be described. The description may include sample screen images, any GUI standards or style guides that are to be followed,

screen layout constraints, standard buttons and functions (e.g., help) that will appear on every screen, keyboard shortcuts, error message display standards, and so on. One example of a user interface requirement of a software can be that it should be usable by factory shop floor workers who may not even have a high school degree. The details of the user interface design such as screen designs, menu structure, navigation diagram, etc. should be documented in a separate user interface specification document.

Formal Methods for developing the specification document

Formal methods offer a rigorous approach to developing software specification documents by using mathematical techniques to specify, model, and verify system requirements. These methods provide a way to express requirements precisely, avoiding ambiguity and inconsistency. Here's how formal methods can be applied to develop specification documents:

1. **Anna:** Anna is a formal language designed specifically for specifying Ada programs. It provides a structured way to describe system requirements and behaviors, making it a formal method for documenting specifications.
2. **Gist:** Gist aims to formalize processes in a manner that resembles natural language constructs, making it easier for users to describe system behavior. Despite its goal of readability, Gist specifications can still be challenging to interpret, highlighting the need for tools to aid in translation.
3. **Vienna Definition Method (VDM):** VDM is a formal method based on denotational semantics, offering a comprehensive framework for software development. It extends beyond specifications to encompass design and implementation aspects, making it suitable for documenting system requirements at various stages of the development process.
4. **Communicating Sequential Processes (CSP):** CSP describes system behavior in terms of sequences of events and message passing between processes. It provides a way to model concurrent systems and specify their behavior, offering executable specifications and a structured path from specification to implementation.
5. **Mathematical Notations:** Formal methods often involve using mathematical notations to describe system requirements. This can include using set theory, predicate logic, formal languages (such as Z or B), or formal specification languages (such as TLA+ or Alloy) to define system properties and behaviors precisely.
6. **Formal Specification Languages:** Formal specification languages provide a structured way to express system requirements using mathematical constructs. These languages typically have well-defined syntax and semantics, allowing for rigorous specification of system behavior, data structures, and constraints. Examples include:
7. **Z:** A formal specification language based on set theory and first-order logic, commonly used for specifying software requirements.
8. **B:** A formal method that combines mathematics, logic, and set theory to model system behavior and properties.
9. **TLA+ (Temporal Logic of Actions):** A specification language used for describing concurrent and distributed systems, enabling precise specification of system properties and behaviours.

10. **Alloy:** A lightweight formal modelling language that allows for the specification and analysis of system designs using first-order relational logic.
11. **Modelling Tools:** Formal methods often involve the use of modelling tools that support the creation and analysis of formal specifications. These tools provide capabilities for writing, editing, and validating formal specifications, as well as for performing automated analysis and verification of system properties.
12. **Verification Techniques:** Formal methods enable the verification of system properties against formal specifications. This can involve techniques such as model checking, theorem proving, and simulation-based verification to ensure that the system satisfies its requirements and behaves as intended.
13. **Refinement:** Formal methods support the refinement of specifications from abstract to concrete levels. This involves progressively adding detail to the specification while preserving its correctness and consistency, ultimately leading to the implementation of the system.
14. **Tool Support:** Various tools and frameworks support the application of formal methods in software specification. These tools provide features for writing formal specifications, performing analysis and verification, and generating documentation from formal models.

Finate state Machines:

This Problem is originally formulated by the M202 team at the Open University, United Kingdom [Brady, 1977]

Imagine a safe with a combination lock that can be in one of three positions: 1, 2, or 3. The dial on the lock can be turned left (L) or right (R). This means there are six possible movements: 1L, 1R, 2L, 2R, 3L, and 3R.

The combination to unlock the safe is 1L, 3R, 2L. Any other dial movement will trigger an alarm.

- Here's how the Finite State Machine works:
- Set of States (J): The lock can be in one of five states: Safe Locked, A, B, Safe Unlocked, or Sound Alarm.
- Set of Inputs (K): The inputs represent the movements of the dial: 1L, 1R, 2L, 2R, 3L, and 3R.
- Transition Function (T): The transition function specifies the next state based on the current state and input.

For example, if the lock is in the Safe Locked state and the input is 1L, the next state is A. But if any other input is received (e.g., 1R or 3L), the next state is Sound Alarm.

If the correct combination is entered (1L, 3R, 2L), the sequence of transitions leads from Safe Locked to A to B to Safe Unlocked.

- Initial State (S): The initial state is Safe Locked.
- Final States (F): There are two final states: Safe Unlocked and Sound Alarm.

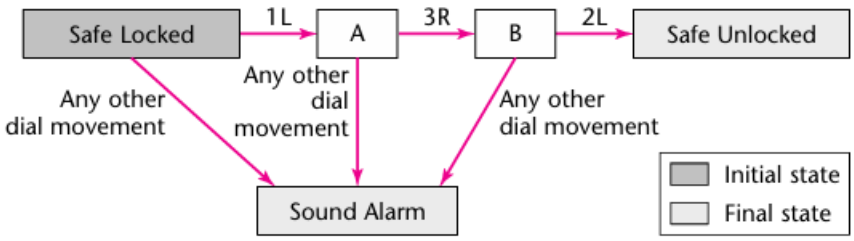


Table of Next States

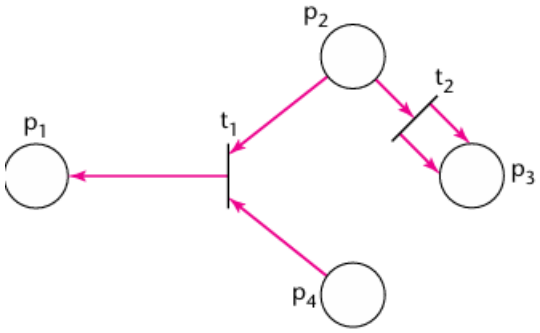
Dial Movement \ Current State	Safe Locked	A	B
1L	A	Sound alarm	Sound alarm
1R	Sound alarm	Sound alarm	Sound alarm
2L	Sound alarm	Sound alarm	Safe unlocked
2R	Sound alarm	Sound alarm	Sound alarm
3L	Sound alarm	Sound alarm	Sound alarm
3R	Sound alarm	B	Sound alarm

- Transition Function (T):The transition function tells us what state the system moves to based on the current state and input. It's represented in a table or tabular form, detailing all possible transitions.
- Initial State (S): The initial state is where the system starts, in this case, it's Safe Locked.
- Final States (F):These are the states where the system ends up after processing inputs.
- For the combination lock, the final states are Safe Unlocked and Sound Alarm.
- Formal Definition: A Finite State Machine is formally defined as a 5-tuple (J, K, T, S, F).J represents the set of states, K is the set of inputs, T is the transition function, S is the initial state, and F is the set of final states.
- Usage: Finite State Machines are widely used in computing, especially in user interfaces like menus. Each menu option corresponds to a state, and selecting an option triggers a transition to another state.
- Transition Rules: Transition rules describe how the system moves from one state to another based on an event and possibly additional conditions (predicates).Each transition rule has the form: "current state [event] and [condition] ⇒ next state."
- Extension with Predicates: To enhance FSMs, we can add a set of predicates that evaluate the global state of the system. These predicates determine whether certain transitions should occur based on the system's overall state.

Petri Nets

Petri nets were developed by Carl Adam Petri, a German mathematician and computer scientist. He introduced Petri nets in his doctoral dissertation titled "Kommunikation mit Automaten" (Communication with Automata) in 1962. Petri nets have since become a widely used formalism for modeling and analyzing concurrent systems in various fields, including computer science, engineering, and systems biology.

A Petri net comprises four fundamental components: places (P), transitions (T), an input function (I), and an output function (O). Let's illustrate this with a Petri net depicted in Figure 12.18:



The set of places, P, includes {p1, p2, p3, p4}.

Transitions, T, are represented by {t1, t2}.

Input functions for transitions are defined as follows:

I(t1) = {p2, p4}

I(t2) = {p2}

Output functions for transitions are defined as:

O(t1) = {p1}

O(t2) = {p3, p3}

In a more formal sense (Peterson, 1981), a Petri net structure is represented as a 4-tuple, C = (P, T, I, O):

P: A finite set of places.

T: A finite set of transitions, disjoint from P.

I: The input function, mapping transitions to bags of places.

O: The output function, mapping transitions to bags of places.

A marked Petri net involves assigning tokens to the Petri net's places. In Figure 12.19, the marking includes four tokens distributed among the places. For example, there's one token in p1 and two in p2. Transition t1 is enabled because there are tokens in p2 and p4. If t1 fires, one token each is removed from p2 and p4, and one is placed in p1.

Markings are represented as functions from the set of places to nonnegative integers, denoted as M: $P \rightarrow \{0, 1, 2, ...\}$. A marked Petri net is then represented as a 5-tuple (P, T, I, O, M).

Petri nets operate no deterministically; if multiple transitions are enabled, any one of them can fire. After a transition fires, the marking of the Petri net changes accordingly.

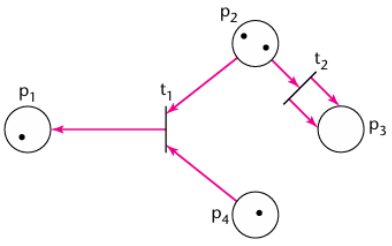


FIGURE 12.19
A marked Petri net.

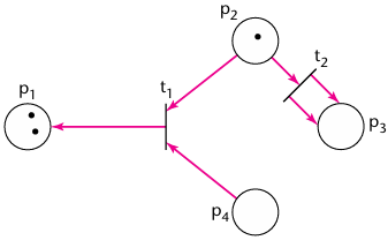


FIGURE 12.20
The Petri net of Figure 12.19 after transition t_1 fires.

A marked Petri net involves the assignment of tokens to the places within the Petri net, as shown in Figure 12.19. Each place may contain a certain number of tokens, indicated by a marking. For instance, in Figure 12.19, there are four tokens distributed among the places p1, p2, p3, and p4. This marking can be represented by the vector (1, 2, 0, 1), indicating the token count in each place.

Transitions in a Petri net are considered enabled (ready to fire) if each of their input places contains the required number of tokens. In Figure 12.19, both transitions t1 and t2 are enabled. If a transition fires, tokens are consumed from its input places and produced in its output places according to the net's structure.

Petri nets operate no deterministically, meaning that if multiple transitions are enabled simultaneously, any one of them can fire. After a transition fires, the marking of the Petri net changes

accordingly. For example, if transition t1 fires in Figure 12.19, tokens are removed from p2 and p4, and one token is placed in p1, resulting in a new marking (2, 1, 0, 0) as shown in Figure 12.20.

Formally, a marking M of a Petri net C = (P, T, I, O) is a function from the set of places P to the set of nonnegative integers, denoted as M: P → {0, 1, 2, ...}. A marked Petri net is then represented as a 5-tuple (P, T, I, O, M), capturing both the structure of the net and its current marking.

Z language:

Z notation, often referred to simply as "Z," is a formal specification language that has been increasingly popular in recent years [Spivey, 2001]. However, mastering Z requires a solid understanding of set theory, functions, and discrete mathematics, including concepts like first-order logic. Even for individuals with a background in these areas (which often includes most computer science majors), learning Z can be challenging initially. This difficulty arises not only from the need to understand typical set theoretic and logic symbols like ∃, ∩, and ⇒ but also from the use of numerous specialized symbols such as ⨁, , | →, and | →.

Use Case Modelling:

This is a technique used in software development and systems engineering to describe the functional requirements of a system. It focuses on understanding and documenting how a system is supposed to work from the perspective of the end users. In essence, it helps answer the question: “What should the system do to meet the needs and goals of its users?

Key Concepts of Use Case Modeling

Functional Requirements: Functional requirements are the features, actions, and behaviors a system must have to fulfill its intended purpose. Use case modeling is primarily concerned with defining and capturing these requirements in a structured manner.

End User’s Perspective: Use case modeling starts by looking at the system from the viewpoint of the people or entities (referred to as “actors”) who will interact with the system. It’s essential to understand how these actors will use the system to achieve their objectives or perform their tasks.

Interactions: Use case modeling emphasizes capturing the interactions between these end users (actors) and the system. It’s not just about what the system does in isolation; it’s about how it responds to user actions or requests.

The Basics of Use Cases:

A use case is a description of how a system interacts with one or more external entities, called actors, to achieve a specific goal.

A use case can be written in textual or diagrammatic form, depending on the level of detail and complexity required.

A use case should capture the essential and relevant aspects of the interaction, such as the preconditions, postconditions, main flow, alternative flows, and exceptions.

Entity Class Modelling:

FIGURE 13.4 An exception scenario (the missing responsibilities and the use of the passive voice will be corrected in the next iteration).

1. User A presses the Up floor button at floor 3 to request an elevator. User A wishes to go to floor 1.
2. The Up floor button is turned on.
3. An elevator arrives at floor 3. It contains User B, who has entered the elevator at floor 1 and pressed the elevator button for floor 9.
4. The elevator doors open.
5. The timer starts.
   User A enters the elevator.
6. User A presses the elevator button for floor 1.
7. The elevator button for floor 1 is turned on.
8. The elevator doors close after a timeout.
9. The Up floor button is turned off.
10. The elevator travels to floor 9.
11. The elevator button for floor 9 is turned off.
12. The elevator doors open to allow User B to exit from the elevator.
13. The timer starts.
    User B exits from the elevator.
14. The elevator doors close after a timeout.
15. The elevator proceeds to floor 1 with User A.

Entity class modeling involves extracting entity classes and their attributes, typically represented in a UML class diagram. This step focuses solely on identifying the attributes of entity classes, deferring the assignment of methods to the classes until the object-oriented design phase. However, this process can be challenging, as accurately identifying entity classes and their attributes is crucial for the success of the overall analysis.

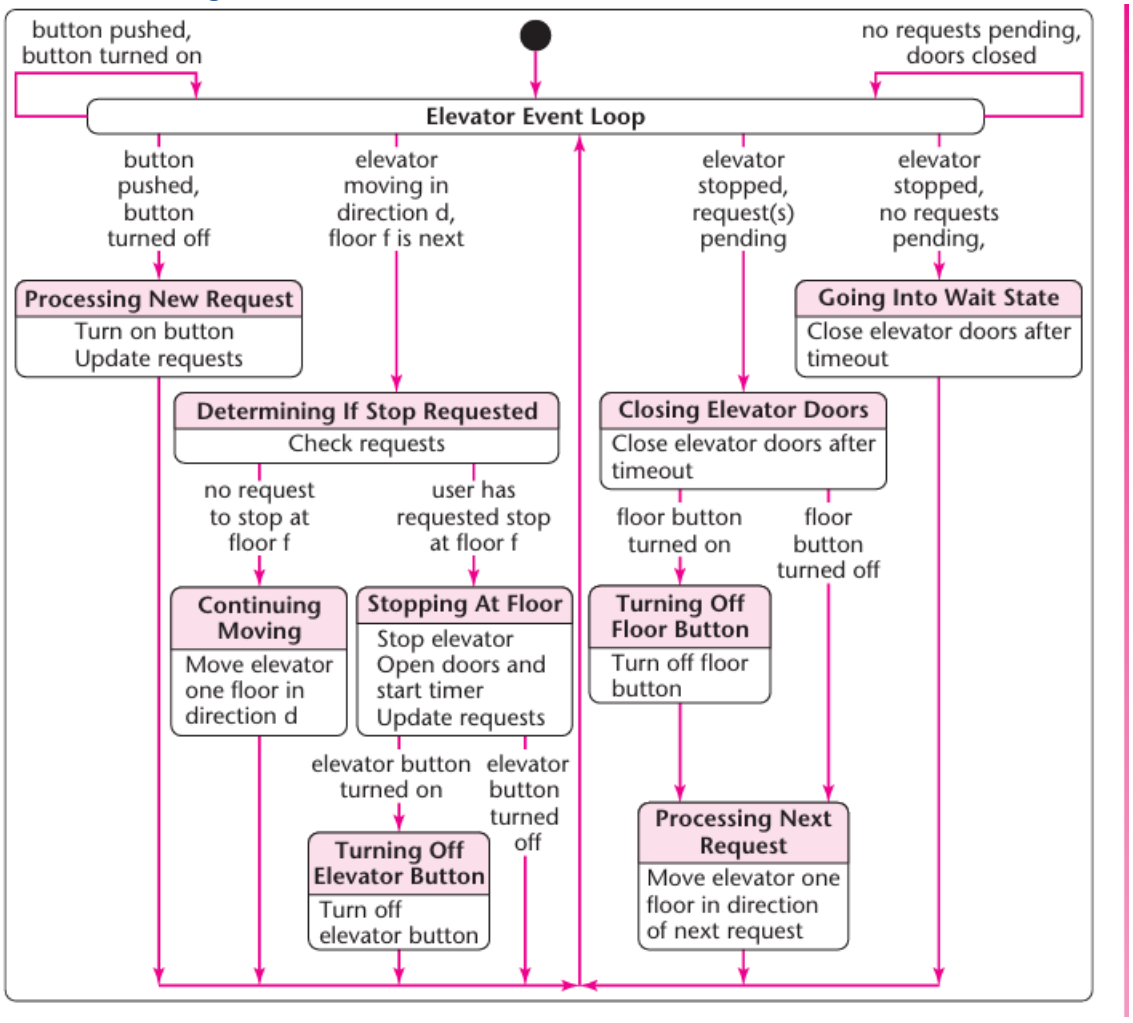
One method of determining entity classes is to derive them from the use cases by studying both normal and exceptional scenarios. For instance, scenarios like those depicted in Figures 13.3 and 13.4 suggest candidate entity classes such as elevator buttons, floor buttons, elevators, doors, and timers. However, it's essential to avoid inferring too many candidate entity classes, as it's easier to add new classes later than to remove unnecessary ones.

Another effective approach, particularly when developers have domain expertise, is using CRC cards. However, for those lacking domain knowledge, noun extraction is recommended. The noun extraction method involves describing the software product in a paragraph and then identifying the nouns within it as candidate entity classes.

The two-stage noun extraction method begins with describing the software product and then identifying the nouns within that description. Abstract nouns, which represent concepts without physical existence, are usually not suitable as entity classes but may serve as attributes. From the identified nouns, candidate entity classes are determined.

However, this approach has limitations. For example, in the elevator problem case study, the initial extraction process may overlook critical components like the elevator controller. Consequently, while noun extraction provides a starting point for identifying entity classes, it's not comprehensive and should be supplemented with other techniques and domain knowledge to ensure accuracy.

Dynamic Modelling:



Dynamic modeling: aims to create statecharts, which describe the behavior of a system similar to finite state machines, for each class. Let's take the Elevator Controller Class as an example. The statechart provides a visual representation of the states and transitions of the class.

In dynamic modeling, two approaches are compared: Finite State Machines (FSMs) and UML statecharts. FSMs are formal techniques represented by mathematical rules, while UML statecharts are less formal and distribute the state, event, and predicate aspects across the diagram. Although UML statecharts are semiformal, as current versions of OOA are graphical, future versions may become more formal as the object-oriented paradigm evolves.

The equivalence between a statechart and State Transition Diagrams (STDs) can be demonstrated by considering various scenarios. For instance, if a user presses the "Up" floor button at floor 3, the

system transitions accordingly in both the statechart and STD representations. The statechart in Figure 13.7 is constructed from generalized scenarios, ensuring it captures various possible events and transitions accurately.

Each event in the scenario is generalized to model potential actions and states in the system. For example, pressing any button leads to either turning on or off the button, depending on its current state. Similarly, events like the elevator arriving at a floor or no requests pending lead to corresponding state transitions and actions in the statechart. These generalizations ensure that the statechart accurately represents the system's behavior across different scenarios.

Test Flow in oo design

The test workflow in object-oriented analysis involves verifying and validating the system's behavior and functionality. It ensures that the system meets its requirements and performs as expected. The test workflow typically includes the following steps:

- **Identifying Test Cases:** Test cases are created based on the system requirements and specifications. These test cases cover various scenarios, including normal operations, boundary conditions, and exceptional cases.
- **Test Plan Development:** A comprehensive test plan is developed, outlining the testing approach, objectives, resources, schedule, and responsibilities. The test plan defines the testing scope and strategies to be employed.
- **Test Environment Setup:** The test environment is set up to replicate the system's runtime environment as closely as possible. This includes hardware, software, databases, networks, and other dependencies necessary for testing.
- **Test Case Execution:** Test cases are executed according to the test plan. Testers run the system with different inputs and conditions, observing its behavior and comparing it against expected outcomes.
- **Defect Reporting:** Any defects or discrepancies identified during testing are documented and reported to the development team. Defect reports typically include detailed descriptions of the issue, steps to reproduce, severity, and priority.
- **Defect Resolution:** The development team investigates reported defects and works on resolving them. Once fixed, the defects are retested to ensure that the issues have been adequately addressed.
- **Regression Testing:** After resolving defects, regression testing is performed to verify that the fixes have not introduced new issues or affected existing functionality. Regression tests re-run previously executed test cases to confirm system stability.
- **Test Completion:** Once all test cases have been executed, defects resolved, and regression testing completed, the testing phase is concluded. A test summary report may be prepared, summarizing the testing activities, results, and any remaining open issues.
- **Acceptance Testing:** In some cases, user acceptance testing (UAT) may be conducted by stakeholders to validate that the system meets their requirements and expectations. UAT helps ensure that the system is ready for deployment.

- **Documentation:** Test documentation, including test plans, test cases, test results, and defect reports, is updated and maintained throughout the testing process. Clear and comprehensive documentation aids in knowledge transfer and future maintenance efforts.