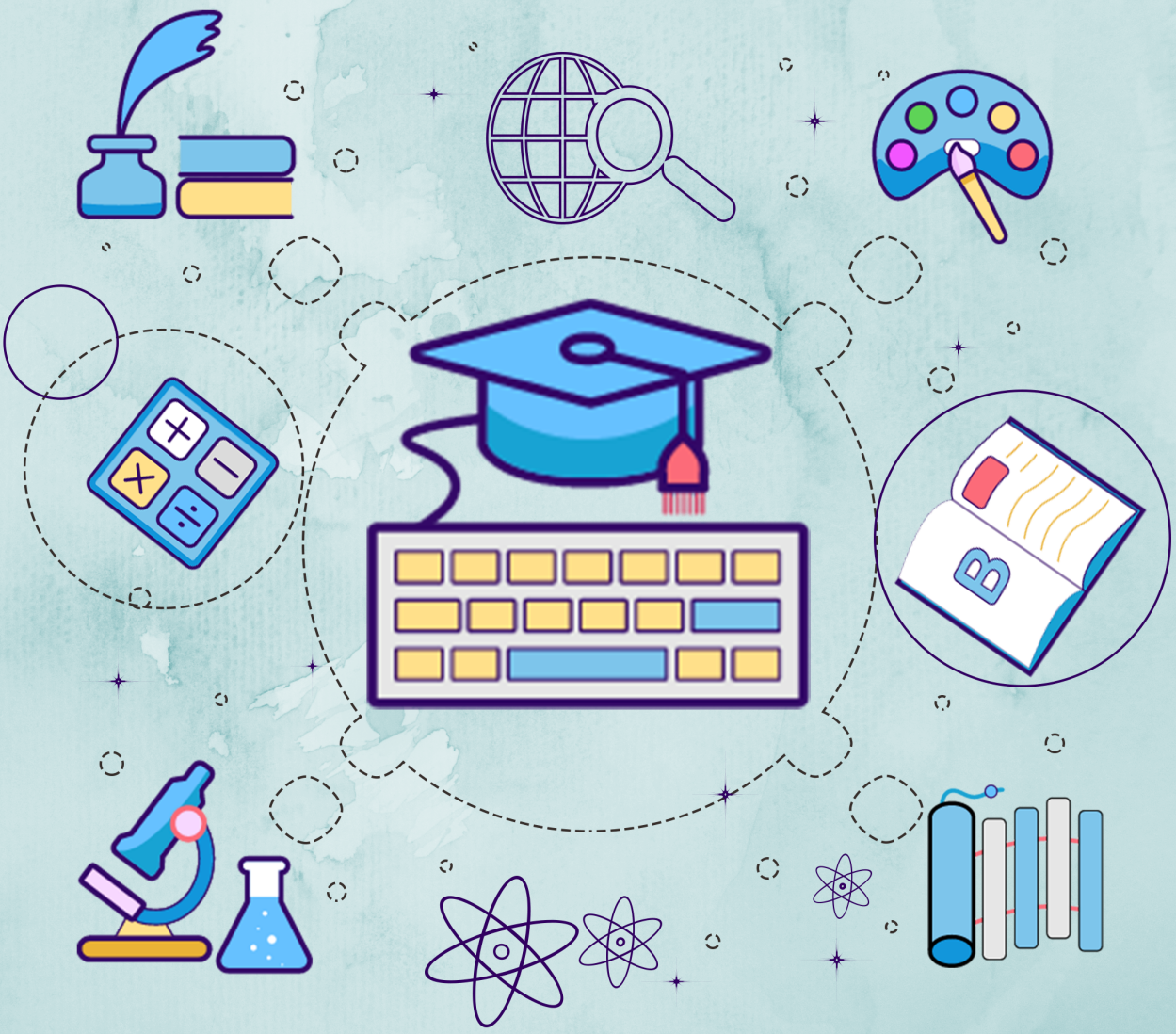
APJ ABDUL KALAM TECHNOLOGICAL UNIVERSITY

KeralaNotes



SYLLABUS | STUDY MATERIALS | TEXTBOOK

PDF | SOLVED QUESTION PAPERS

www.keralanotes.com

KeralaNotes

KTU STUDY MATERIALS

MANAGEMENT OF SOFTWARE SYSTEMS

CST 309

Module 2

Related Link :

- KTU S5 STUDY MATERIALS
- KTU S5 NOTES
- KTU S5 SYLLABUS
- KTU S5 TEXTBOOK PDF
- KTU S5 PREVIOUS YEAR SOLVED QUESTION PAPER

MODULE – 2
REQUIREMENTS ANALYSIS AND DESIGN

- **Requirements** → the descriptions of the services that a system should provide and the constraints on its operation.

- **Requirements Engineering (RE)**
  - The process of finding out, analyzing, documenting and checking the services and constraints of a system.
  - The first stage of the software engineering process.

User Requirements

- High-level abstract requirements
- Statements, in a natural language plus diagrams, of what services the system is expected to provide to system users and the constraints under which it must operate.

System Requirements

- Detailed description of what the system should do.
- Detailed descriptions of the software system's functions, services, and operational constraints.
- System requirements document (sometimes called a functional specification) should define exactly what is to be implemented.
- Often classified as functional or non-functional requirements.

User requirements definition

1. The Mentcare system shall generate monthly management reports showing the cost of drugs prescribed by each clinic during that month.

System requirements specification

1.1 On the last working day of each month, a summary of the drugs prescribed, their cost and the prescribing clinics shall be generated.
1.2 The system shall generate the report for printing after 17.30 on the last working day of the month.
1.3 A report shall be created for each clinic and shall list the individual drug names, the total number of prescriptions, the number of doses prescribed and the total cost of the prescribed drugs.
1.4 If drugs are available in different dose units (e.g. 10mg, 20mg, etc.) separate reports shall be created for each dose unit.
1.5 Access to drug cost reports shall be restricted to authorized users as listed on a management access control list.

Fig: Mental health care patient information system (Mentcare) shows how a user requirement may be expanded into several system requirements

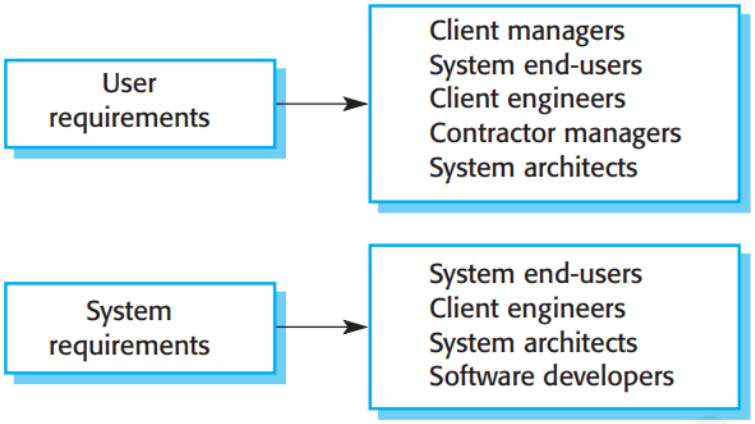


Fig: Readers of different types of requirements specification

Functional Requirements

- Statements of services the system should provide, how the system should react to particular inputs, and how the system should behave in particular situations.
- Explicitly state what the system should not do.

Non-functional Requirements

- Constraints on the services or functions offered by the system.
- Include timing constraints, constraints on the development process, and constraints imposed by standards.

Functional Requirements

- Describe what the system should do.
- Depends on the type of software being developed, the expected users of the software, and the general approach taken by the organization when writing requirements.
- When expressed as user requirements, it should be written in natural language so that system users and managers can understand them.
- Functional system requirements are written for system developers.

Lakshmi M B

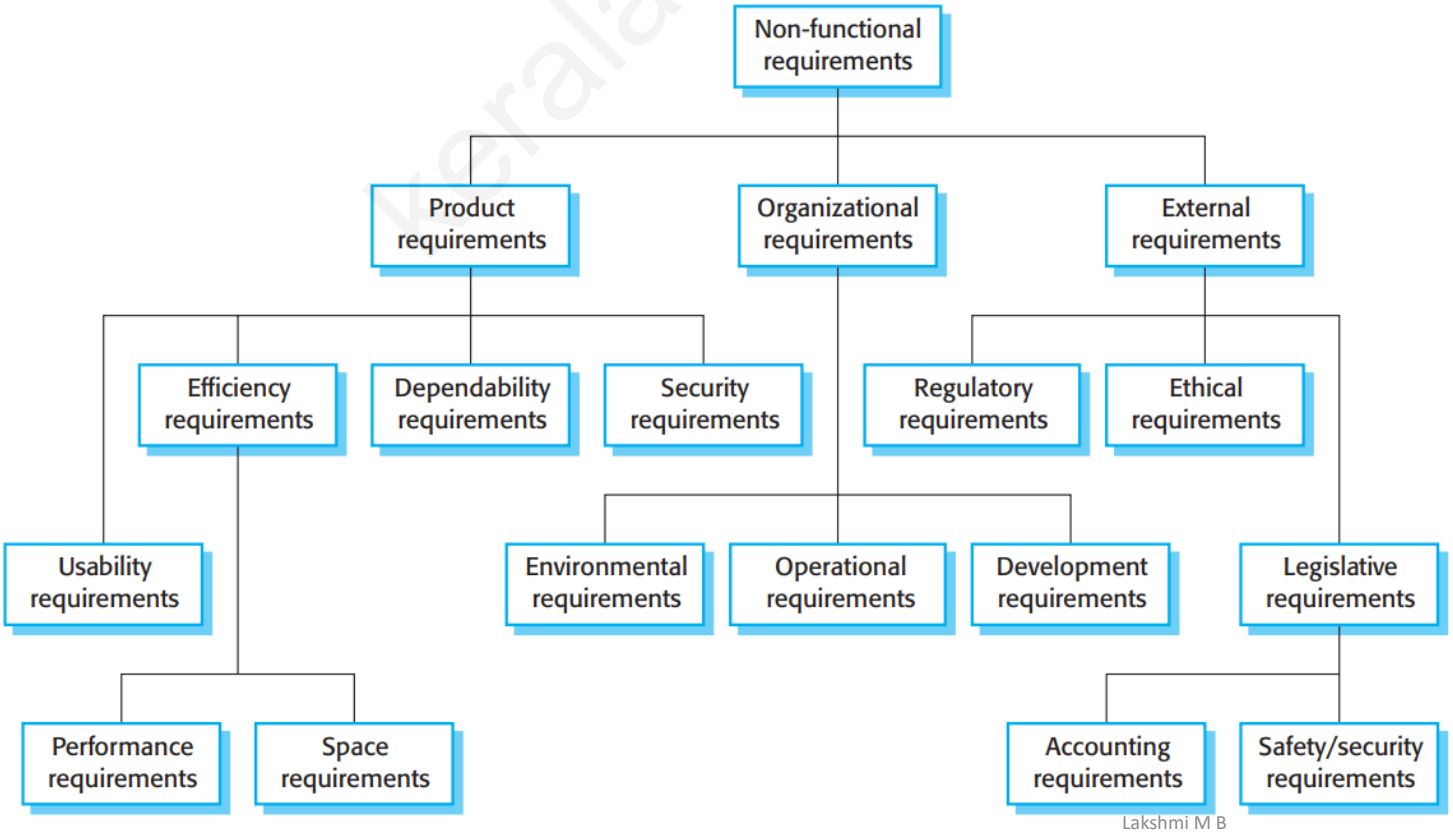
Examples for functional requirements for the Mentcare system:

1. A user shall be able to search the appointments lists for all clinics.
2. The system shall generate each day, for each clinic, a list of patients who are expected to attend appointments that day.
3. Each staff member using the system shall be uniquely identified by his or her eight-digit employee number.

Lakshmi M B

Non- functional Requirements

- Usually specify or constrain characteristics of the system as a whole.
- Often more critical than individual functional requirements.
- Failing to meet a non-functional requirement can mean that the whole system is unusable.
- Arise through user needs because of budget constraints, organizational policies, the need for interoperability with other software or hardware systems, or external factors such as safety regulations or privacy legislation.
- The implementation of these requirements may be spread throughout the system, for two reasons:
  1. May affect the overall architecture of a system rather than the individual components.
  2. May generate several, related functional requirements that define new system services that are required if the non-functional requirement is to be implemented.



1. **Product requirements** → specify or constrain the runtime behavior of the software.
   - Examples include how fast the system must execute, how much memory it requires, etc.
2. **Organizational requirements** → broad system requirements derived from policies and procedures in the customer's and developer's organizations.
   - Examples include operational process requirements that define how the system will be used; development process requirements that specify the programming language; the development environment or process standards to be used; and environmental requirements that specify the operating environment of the system.
3. **External requirements** → derived from factors external to the system and its development process.
   - Example include regulatory requirements that set out what must be done for the system to be approved for use by a regulator, such as a nuclear safety authority; legislative requirements that must be followed to ensure that the system operates within the law; and ethical requirements that ensure that the system will be acceptable to its users and the general public.

PRODUCT REQUIREMENT
The Mentcare system shall be available to all clinics during normal working hours (Mon–Fri, 08:30–17:30). Downtime within normal working hours shall not exceed 5 seconds in any one day.

ORGANIZATIONAL REQUIREMENT
Users of the Mentcare system shall identify themselves using their health authority identity card.

EXTERNAL REQUIREMENT
The system shall implement patient privacy provisions as set out in HStan-03-2006-priv.

Property	Measure
Speed	Processed transactions/second User/event response time Screen refresh time
Size	Megabytes/Number of ROM chips
Ease of use	Training time Number of help frames
Reliability	Mean time to failure Probability of unavailability Rate of failure occurrence Availability
Robustness	Time to restart after failure Percentage of events causing failure Probability of data corruption on failure
Portability	Percentage of target dependent statements Number of target systems

Fig: Metrics for specifying non-functional requirements

Requirements Engineering Processes

- RE involves three key activities:
  1. Discovering requirements by interacting with stakeholders (elicitation and analysis)
  2. Converting these requirements into a standard form (specification)
  3. Checking that the requirements actually define the system that the customer wants (validation)
- The output of the RE process is a system requirements document.

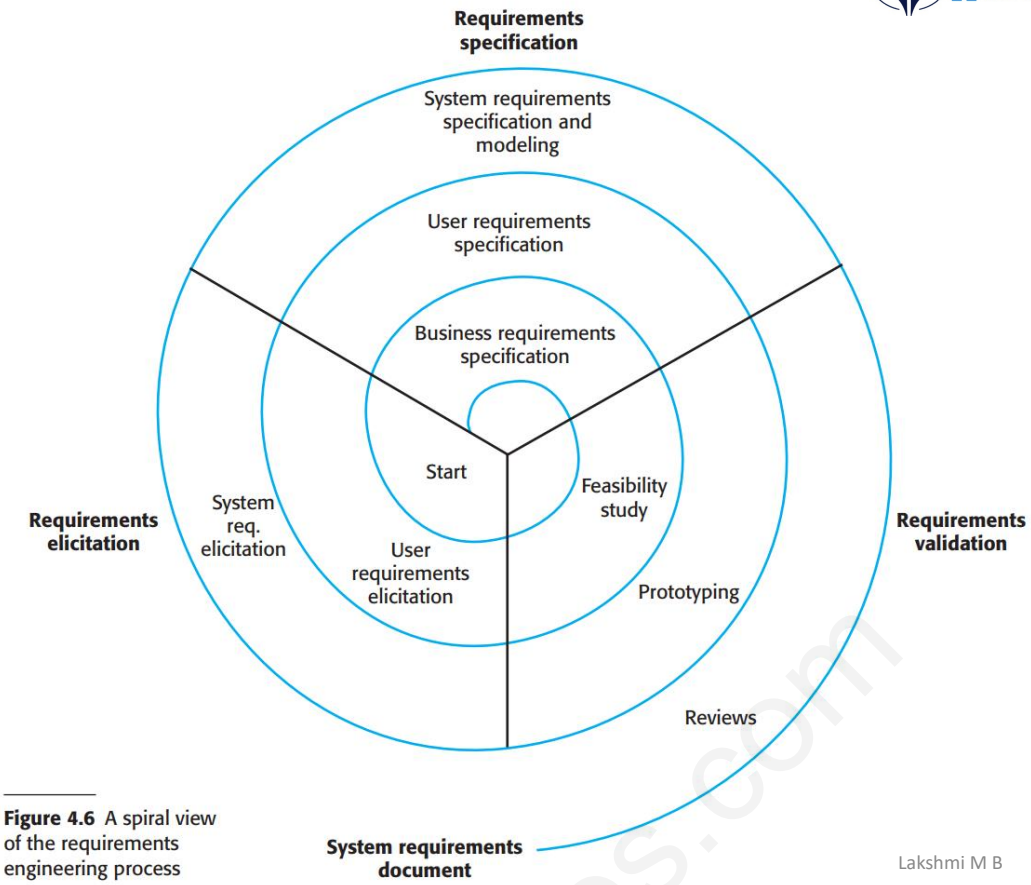


Figure 4.6 A spiral view of the requirements engineering process

Lakshmi M B

REQUIREMENTS ELICITATION

- An iterative process that can be represented as a spiral of activities—requirements discovery, requirements classification and organization, requirements negotiation, and requirements documentation.
- Aim → to understand the work that stakeholders do and how they might use a new system to help support that work.
- Software engineers work with stakeholders to find out about the application domain, work activities, the services and system features that stakeholders want, the required performance of the system, hardware constraints, etc.

- Difficult process due to several reasons:
  1. Stakeholders often don't know what they want from a computer system except in the most general terms.
  2. Stakeholders in a system naturally express requirements in their own terms and with implicit knowledge of their own work.
  3. Different stakeholders, with diverse requirements, may express their requirements in different ways.
  4. Political factors may influence the requirements of a system.
  5. The economic and business environment in which the analysis takes place is dynamic. New requirements may emerge from new stakeholders who were not originally consulted.

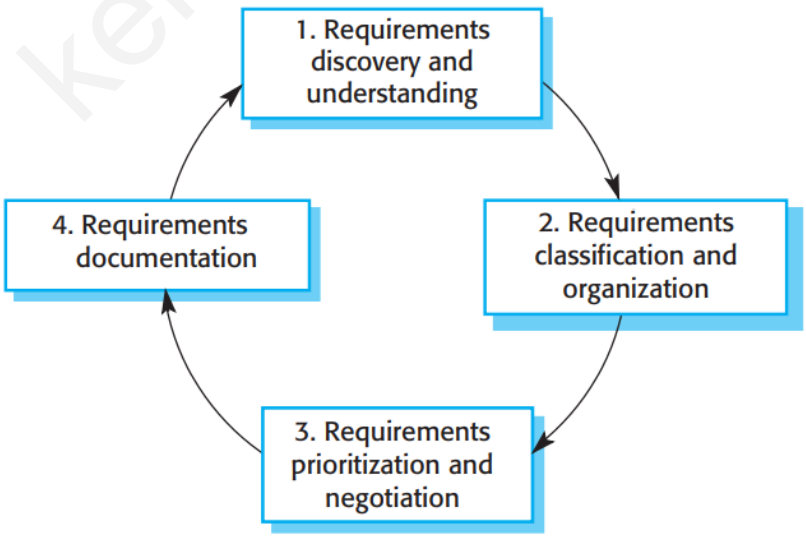


Fig: The requirements elicitation and analysis process

1. **Requirements discovery and understanding** → interacting with stakeholders of the system to discover their requirements.
2. **Requirements classification and organization** → takes the unstructured collection of requirements, groups related requirements and organizes them into coherent clusters.
3. **Requirements prioritization and negotiation** → prioritizing requirements and finding and resolving requirements conflicts through negotiation.
4. **Requirements documentation** → The requirements are documented and input into the next round of the spiral.

Requirements Elicitation Techniques

1. **Interviewing** → where you talk to people about what they do.

2. **Observation or ethnography** → where you watch people doing their job to see what artifacts they use, how they use them, and so on.

1. Interviewing

- Interviews may be of two types:
  1. **Closed interviews** → where the stakeholder answers a predefined set of questions.
  2. **Open interviews** → in which there is no predefined agenda.

- To be an effective interviewer, you should bear two things in mind:
  1. You should be open-minded, avoid preconceived ideas about the requirements, and willing to listen to stakeholders.
  2. You should prompt the interviewee to get discussions going by using a springboard question or a requirements proposal, or by working together on a prototype system.

Lakshmi. M. B

2. Ethnography

- An observational technique that can be used to understand operational processes and help derive requirements for software to support these processes.
- Value → it helps discover implicit system requirements that reflect the actual ways that people work, rather than the formal processes defined by the organization.
- Effective for discovering two types of requirements:
  1. Requirements derived from the way in which people actually work, rather than the way in which business process definitions say they ought to work.
  2. Requirements derived from cooperation and awareness of other people's activities.
- Ethnography can be combined with the development of a system prototype.

Lakshmi. M. B

Stories and Scenarios

- **Stories** → written as narrative text and present a high-level description of system use. The advantage of stories is that everyone can easily relate to them.
- **Scenarios** → usually structured with specific information collected such as inputs and outputs. Theses are descriptions of example user interaction sessions. A scenario starts with an outline of the interaction. During the elicitation process, details are added to create a complete description of that interaction. A scenario may include:
    1. A description of what the system and users expect when the scenario starts.
    2. A description of the normal flow of events in the scenario.
    3. A description of what can go wrong and how resulting problems can be handled.
    4. Information about other activities that might be going on at the same time.
    5. A description of the system state when the scenario ends.

Lakshmi. M. B

REQUIREMENTS VALIDATION

- The process of checking that requirements define the system that the customer really wants.
- Overlaps with elicitation and analysis as it is concerned with finding problems with the requirements.

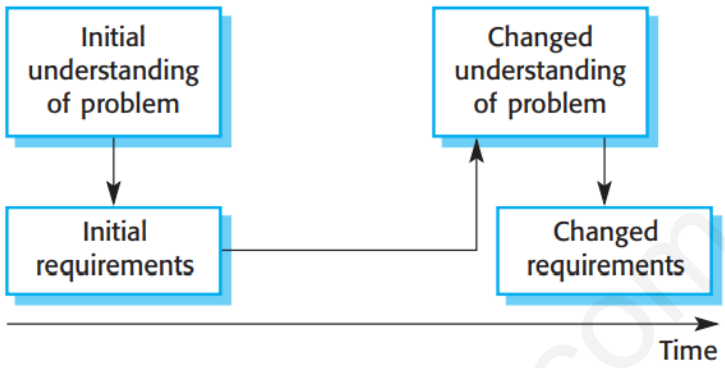
Lakshmi. M. B

- Different types of checks should be carried out on the requirements in the requirements document:
  1. **Validity checks** → check that the requirements reflect the real needs of system users.
  2. **Consistency checks** → requirements in the document should not conflict.
  3. **Completeness checks** → requirements document should include requirements that define all functions and the constraints intended by the system user.
  4. **Realism checks** → checked to ensure that they can be implemented within the proposed budget for the system.
  5. **Verifiability** → system requirements should always be written so that they are verifiable.

- Requirements validation techniques that are used either individually or in conjunction with one another:
  1. **Requirements reviews** → the requirements are analyzed systematically by a team of reviewers who check for errors and inconsistencies.
  2. **Prototyping** → involves developing an executable model of a system and using this with end-users and customers to see if it meets their needs and expectations.
  3. **Test-case generation** → requirements should be testable. Developing tests from the user requirements before any code is written is an integral part of test-driven development.

REQUIREMENTS CHANGE

- One reason → systems are often developed to address problems that cannot be completely defined.



Lakshmi. M. B

- Most changes to system requirements arise because of changes to the business environment of the system:
  1. The business and technical environment of the system always changes after installation.
  2. Requirements imposed by system customers because of organizational and budgetary constraints may conflict with end-user requirements.
  3. Priorities given to different requirements of diverse stakeholders may be conflicting or contradictory.

Lakshmi. M. B

- As requirements are evolving, you need to keep track of individual requirements and maintain links between dependent requirements so that you can assess the impact of requirements changes. This formal process for making change proposals and linking these to system requirements is known as **requirements management.**
- It is often better for an independent authority, who can balance the needs of all stakeholders, to decide on the changes that should be accepted.

Requirements management planning

- Concerned with establishing how a set of evolving requirements will be managed.
- Issues that have to be decided:
  1. **Requirements identification** → each requirement must be uniquely identified.
  2. **A change management process** → the set of activities that assess the impact and cost of changes.
  3. **Traceability policies** → defines the relationships between each requirement and between the requirements and the system design that should be recorded. Also define how these records should be maintained.
  4. **Tool support** → range from specialist requirements management systems to shared spreadsheets and simple database systems.

- Requirements management needs automated support, and the software tools for this should be chosen during the planning phase. You need tool support for:

1. **Requirements storage** → requirements should be maintained in a secure, managed data store that is accessible to everyone involved in the requirements engineering process.
2. **Change management** → simplified if active tool support is available.
3. **Traceability management** → tool support for traceability allows related requirements to be discovered.

Requirements change management

- Should be applied to all proposed changes to a system's requirements after the requirements document has been approved.
- It is essential because you need to decide if the benefits of implementing new requirements are justified by the costs of implementation.
- The advantage of using a formal process for change management is that all change proposals are treated consistently and changes to the requirements document are made in a controlled way.

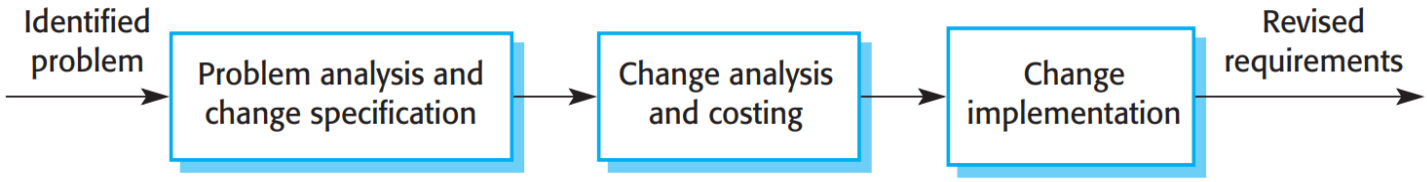


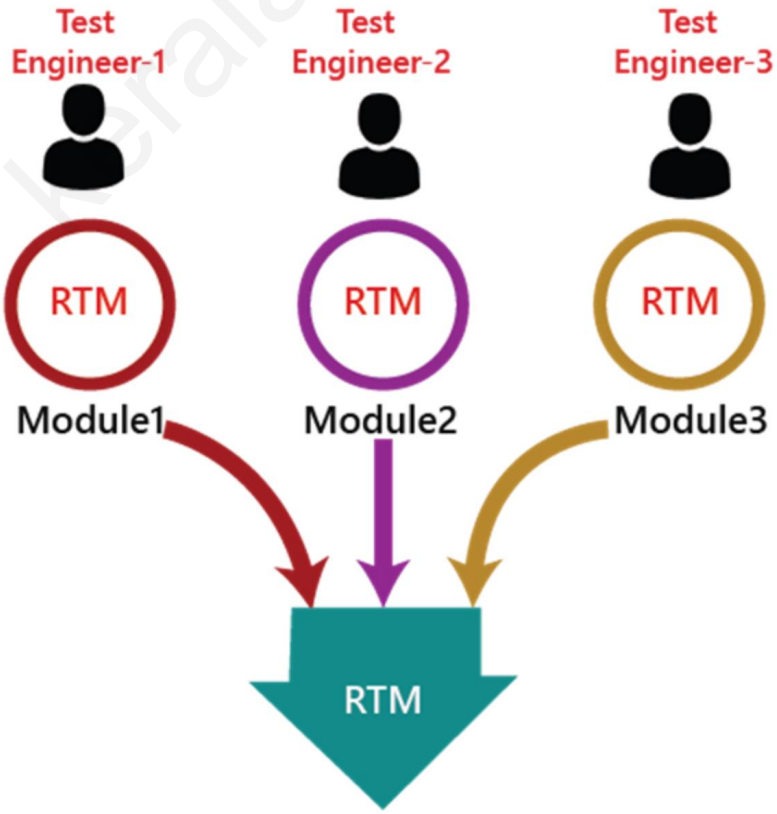
Figure 4.19
Requirements change management

- 3 principal stages to a change management process:
  1. **Problem analysis and change specification** → starts with an identified requirements problem or, sometimes, with a specific change proposal. It is then analyzed to check whether it is valid. This analysis is fed back to the change requestor who may respond with a more specific requirements change proposal, or decide to withdraw the request.
  2. **Change analysis and costing** → assessed using traceability information and general knowledge of the system requirements. The cost of making the change is estimated in terms of modifications to the requirements document. Once this analysis is completed, a decision is made as to whether or not to proceed with the requirements change.
  3. **Change implementation** → The requirements document and, the system design and implementation, are modified. Organize the requirements document so that changes can be done without extensive rewriting or reorganization. Changeability in documents is achieved by minimizing external references and making the document sections as modular as possible. Thus, individual sections can be changed and replaced without affecting other parts of the document.

TRACEABILITY MATRIX

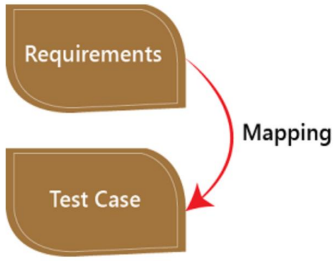
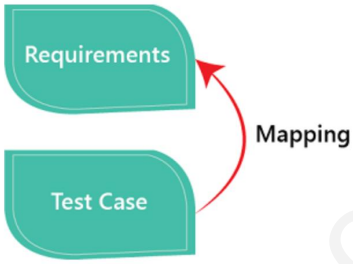
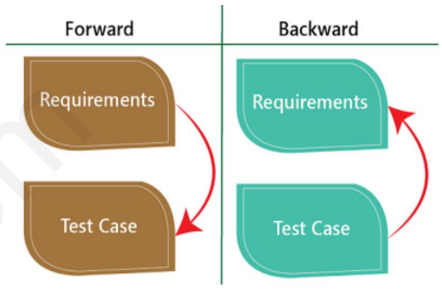
- A table type document that is used in the development of software application to trace requirements.
- It can be used for both forward (from Requirements to Design or Coding) and backward (from Coding to Requirements) tracing.
- It is also known as **Requirement Traceability Matrix (RTM) or Cross Reference Matrix (CRM).**
- It is prepared before the test execution process to ensure that every requirement is covered in the form of a Test case so that we don't miss out any testing.
- We map all the requirements and corresponding test cases to ensure that we have written all the test cases for each condition.

Lakshmi. M. B



	A	B	C	D	E
1	RTM Template				
2	**Requirement number**	**Module name**	**High level requirement**	**Low level requirement**	**Test case name**
3	2	Loan	**2.1 Personal loan**	2.1.1--> personal loan for private employee	beta-2.0-personal loan
4				2.1.2--> personal loan for government employee	
5				2.1.3--> personal loan for jobless people	
6			**2.2 Car loan**	2.2.1--> car loan for private employee	
7				——	
8			**2.3 Home loan**	——	
9				——	
10				——	
11					

- The traceability matrix can be classified into three different types which are as follows:
  1. Forward traceability
  2. Backward or reverse traceability
  3. Bi-directional traceability

Forward Traceability	Backward Traceability	Bi-directional Traceability
• Used to ensure that every business's needs or requirements are executed correctly in the application and also tested rigorously. • Requirements are mapped into the forward direction to the test cases.	• Used to check that we are not increasing the space of the product by enhancing the design elements, code, test other things which are not mentioned in the business needs. • Requirements are mapped into the backward direction to the test cases.	• A combination of forwarding and backward traceability matrix. • Used to make sure that all the business needs are executed in the test cases. • Also evaluates the modification in the requirement which is occurring due to the bugs in the application.
		

- Goals of Traceability Matrix:
  - It helps in tracing the documents that are developed during various phases of SDLC.
  - It ensures that the software completely meets the customer's requirements.
  - It helps in detecting the root cause of any bug.
- Advantages of RTM:
  - With the help of the RTM document, we can display the complete test execution and bugs status based on requirements.
  - It is used to show the missing requirements or conflicts in documents.
  - We can ensure the complete test coverage, which means all the modules are tested.
  - It will also consider the efforts of the testing teamwork towards reworking or reconsidering on the test cases.

DEVELOPING USE CASES

- **Use case** → a list of actions or event steps typically defining the interactions between a role (known in the Unified Modeling Language (UML) as an actor) and a system to achieve a goal. The actor can be a human or other external system.
- A use case tells a stylized story about how an end user interacts with the system under a specific set of circumstances.
- Regardless of its form, a use case depicts the software or system from the end user's point of view.

Lakshmi M B

- The first step in writing a use case is to define the set of "actors" that will be involved in the story.
- Actors represent the roles that people (or devices) play as the system operates. An actor is anything that communicates with the system or product and that is external to the system itself.
- An actor and an end user are not necessarily the same thing.
- A typical user may play a number of different roles when using a system, whereas an actor represents a class of external entities (often, but not always, people) that play just one role in the context of the use case.

Lakshmi M B

- Ex: consider a machine operator (a user) who interacts with the control computer for a manufacturing cell that contains a number of robots and numerically controlled machines.
- After careful review of requirements, the software for the control computer requires 4 different modes (roles) for interaction: programming mode, test mode, monitoring mode, and troubleshooting mode. Therefore, 4 actors can be defined → programmer, tester, monitor, and troubleshooter.

- It is possible to identify primary actors during the first iteration and secondary actors as more is learned about the system.
- **Primary actors** → interact to achieve required system function and derive the intended benefit from the system. They work directly and frequently with the software.
- **Secondary actors** → support the system so that primary actors can do their work.
- Once actors have been identified, use cases can be developed.

- A number of questions should be answered by a use case: -
  1. Who is the primary actor, the secondary actor(s)?
  2. What are the actor's goals?
  3. What preconditions should exist before the story begins?
  4. What main tasks or functions are performed by the actor?
  5. What exceptions might be considered as the story is described?
  6. What variations in the actor's interaction are possible?
  7. What system information will the actor acquire, produce, or change?
  8. Will the actor have to inform the system about changes in the external environment?
  9. What information does the actor desire from the system?
  10. Does the actor wish to be informed about unexpected changes?

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

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

1. The homeowner observes the *SafeHome* control panel (Figure 8.1) to determine if the system is ready for input. If the system is not ready, a *not ready* message is displayed on the LCD display, and the homeowner must physically close windows or doors so that the *not ready* message disappears. [A *not ready* message implies that a sensor is open; i.e., that a door or window is open.]
2. The homeowner uses the keypad to key in a four-digit password. The password is compared with the valid password stored in the system. If the password is incorrect, the control panel will beep once and reset itself for additional input. If the password is correct, the control panel awaits further action.
3. The homeowner selects and keys in *stay* or *away* (see Figure 8.1) to activate the system. *Stay* activates only perimeter sensors (inside motion detecting sensors are deactivated). *Away* activates all sensors.
4. When activation occurs, a red alarm light can be observed by the homeowner.

- Template for detailed descriptions of use cases:

Use case:	*InitiateMonitoring*
Primary actor:	Homeowner.
Goal in context:	To set the system to monitor sensors when the homeowner leaves the house or remains inside.
Preconditions:	System has been programmed for a password and to recognize various sensors.
Trigger:	The homeowner decides to "set" the system, that is, to turn on the alarm functions.

Scenario:

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

Exceptions:

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

Priority: Essential, must be implemented
When available: First increment
Frequency of use: Many times per day
Channel to actor: Via control panel interface
Secondary actors: Support technician, sensors
Channels to secondary actors:
Support technician: phone line
Sensors: hardwired and radio frequency interfaces

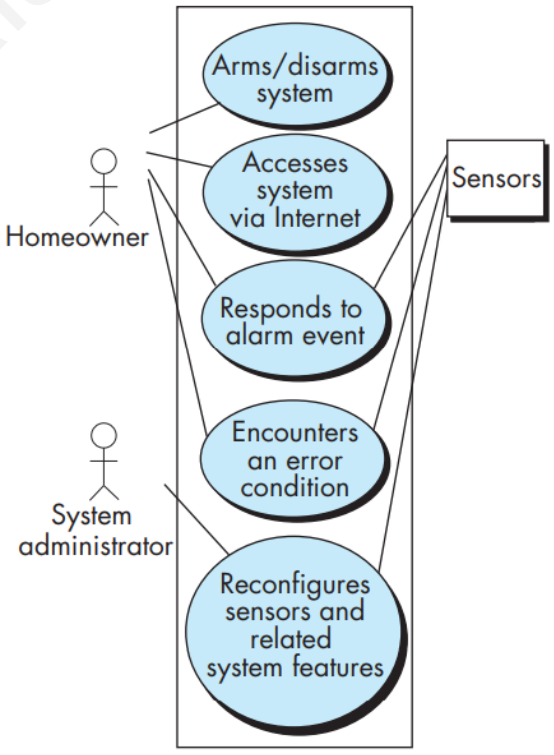
Open issues:

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

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

FIGURE 8.2

UML use case diagram for *SafeHome* home security function



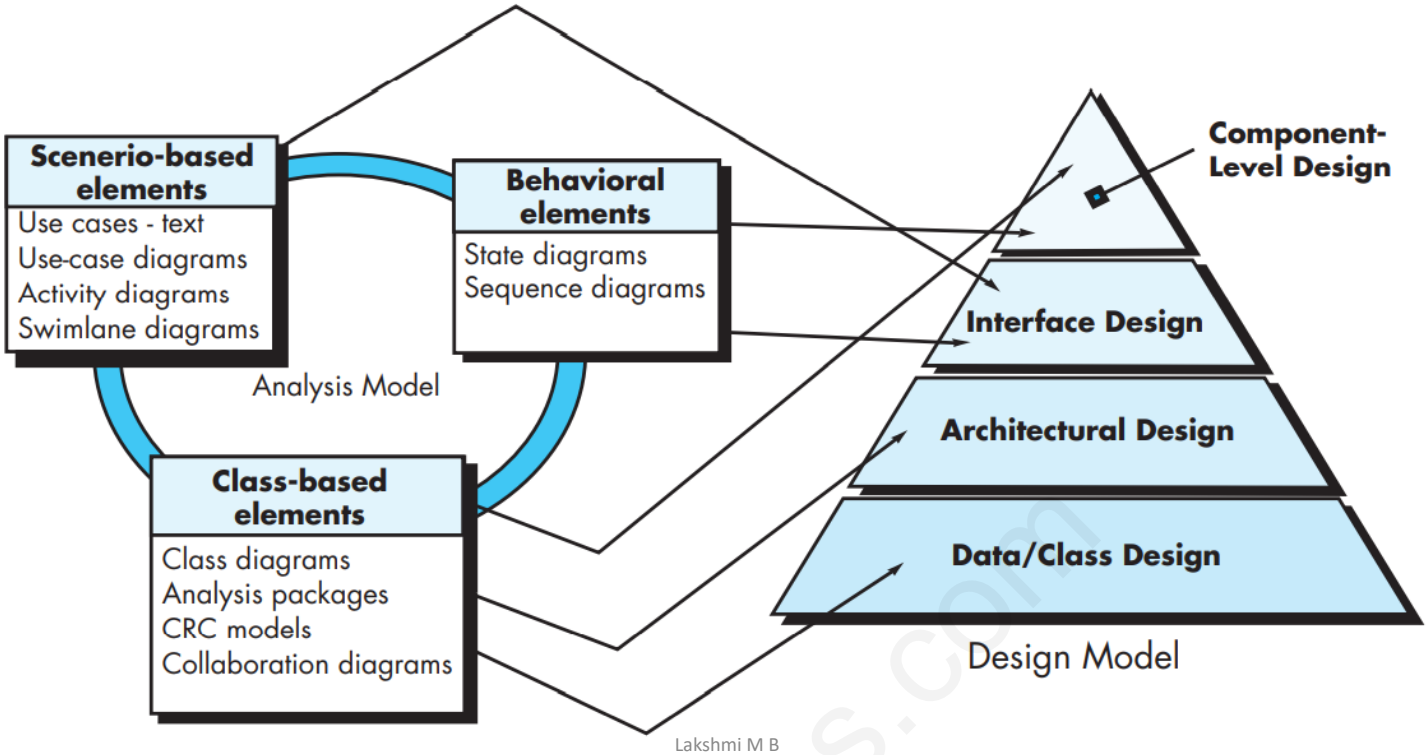
DESIGN CONCEPTS

- Software design encompasses the set of principles, concepts, and practices that lead to the development of a high-quality system or product.
- The goal of design is to produce a model or representation that exhibits firmness, commodity, and delight.
- It changes continually as new methods, better analysis, and broader understanding evolve.

DESIGN WITHIN THE CONTEXT OF SOFTWARE ENGINEERING

- Beginning once software requirements have been analyzed and modeled, software design is the last software engineering action within the modeling activity and sets the stage for construction (code generation and testing).
- The flow of information during software design is illustrated in the figure.

FIGURE 12.1 Translating the requirements model into the design model



- **Data/Class Design** → transforms class models into design class realizations and the requisite data structures required to implement the software. The objects and relationships provide the basis for the data design activity.
- **Architectural Design** → defines the relationship between major structural elements of the software, the architectural styles and patterns that can be used to achieve the requirements defined for the system, and the constraints that affect the way in which architecture can be implemented. The architectural design representation—the framework of a computer-based system—is derived from the requirements model.

- **Interface Design** → describes how the software communicates with systems that interoperate with it, and with humans who use it. An interface implies a flow of information (e.g., data and/or control) and a specific type of behavior. Therefore, usage scenarios and behavioral models provide much of the information required for interface design.
- **Component-Level Design** → transforms structural elements of the software architecture into a procedural description of software components. Information obtained from the class-based models and behavioral models serve as the basis for component design.

DESIGN PROCESS

- Software design is an iterative process through which requirements are translated into a "blueprint" for constructing the software.

Software Quality Guidelines and Attributes

- 3 characteristics used for evaluation of quality:
  1. The design should implement all of the explicit requirements contained in the requirements model, and it must accommodate all of the implicit requirements desired by stakeholders.
  2. The design should be readable and understandable for those who generate code and test the software.
  3. The design should provide a complete picture of the software, addressing the data, functional, and behavioral domains from an implementation perspective.

Quality Guidelines:

1. A design should exhibit an architecture that (1) has been created using recognizable architectural styles or patterns, (2) is composed of components that exhibit good design, and (3) can be implemented in an evolutionary fashion, thereby facilitating implementation and testing.
2. A design should be modular; i.e., the software should be logically partitioned into elements or subsystems.
3. A design should contain distinct representations of data, architecture, interfaces, and components.
4. A design should lead to data structures that are appropriate for the classes to be implemented and are drawn from recognizable data patterns.
5. A design should lead to components that exhibit independent functional characteristics.
6. A design should lead to interfaces that reduce the complexity of connections between components and with the external environment.
7. A design should be derived using a repeatable method that is driven by information obtained during software requirements analysis.
8. A design should be represented using a notation that effectively communicates its meaning.

Lakshmi M B

Quality Attributes:

1. **Functionality** → assessed by evaluating the feature set and capabilities of the program, the generality of the functions that are delivered, and the security of the overall system.
2. **Usability** → assessed by considering human factors , overall aesthetics, consistency, and documentation.
3. **Reliability** → evaluated by measuring the frequency and severity of failure, the accuracy of output results, the mean-time-to-failure (MTTF), the ability to recover from failure, and the predictability of the program.
4. **Performance** → measured using processing speed, response time, resource consumption, throughput, and efficiency.
5. **Supportability** → combines extensibility, adaptability, and serviceability.

Lakshmi M B

DESIGN CONCEPTS

- An overview of fundamental software design concepts:
  1. Abstraction
  2. Architecture
  3. Patterns
  4. Separation of Concerns
  5. Modularity
  6. Information Hiding
  7. Functional Independence
  8. Refinement
  9. Aspects
  10. Refactoring
  11. Object-Oriented Design Concepts
  12. Design Classes
  13. Dependency Inversion
  14. Design for Test

Lakshmi M B

1. Abstraction:

- At the highest level of abstraction, a solution is stated in broad terms using the language of the problem environment.
- At lower levels of abstraction, a more detailed description of the solution is provided.
- Create both procedural and data abstractions.
- **Procedural abstraction** → a sequence of instructions that have a specific and limited function.
- **Data abstraction** → a named collection of data that describes a data object.

Lakshmi M B

2. Architecture:

- Architecture is the structure or organization of program components (modules), the manner in which these components interact, and the structure of data that are used by the components.
- One goal of software design is to derive an architectural rendering of a system.
- A set of architectural patterns enables a software engineer to reuse design-level concepts.
- **Structural properties** → "the components of a system (e.g., modules, objects, filters) and the manner in which those components are packaged and interact with one another."
- **Extra-functional properties** → address "how the design architecture achieves requirements for performance, capacity, reliability, security, adaptability, and other system characteristics.
- **Families of related systems** → "draw upon repeatable patterns that are commonly encountered in the design of families of similar systems."

Lakshmi M B

- **Structural models** → represent architecture as an organized collection of program components.
- **Framework models** → increase the level of design abstraction by attempting to identify repeatable architectural design frameworks (patterns) that are encountered in similar types of applications.
- **Dynamic models** → address the behavioral aspects of the program architecture, indicating how the structure or system configuration may change as a function of external events.
- **Process models** → focus on the design of the business or technical process that the system must accommodate.
- **Functional models** → used to represent the functional hierarchy of a system.

Lakshmi M B

3. Patterns:

- A named nugget of insight which conveys the essence of a proven solution to a recurring problem within a certain context amidst competing concerns.
- Describes a design structure that solves a particular design problem within a specific context and amid “forces” that may have an impact on the manner in which the pattern is applied and used.
- These provide a description that enables a designer to determine:

  (1) whether the pattern is applicable to the current work,

  (2) whether the pattern can be reused (hence, saving design time), and

  (3) whether the pattern can serve as a guide for developing a similar, but functionally or structurally different pattern.

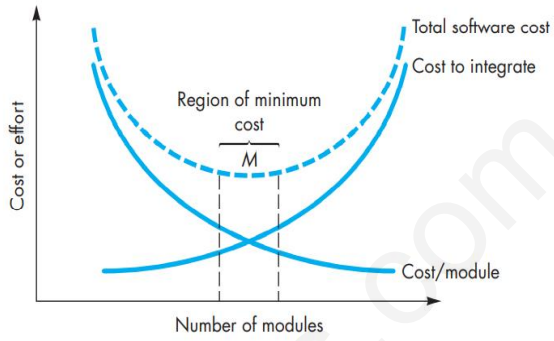
4. Separation of Concerns:

- Suggests that any complex problem can be more easily handled if it is subdivided into pieces that can each be solved and/or optimized independently.
- **Concern** → a feature or behavior that is specified as part of the requirements model for the software.
- By separating concerns into smaller, and therefore more manageable pieces, a problem takes less effort and time to solve.

5. Modularity:

- Software is divided into separately named and addressable components, sometimes called modules, that are integrated to satisfy problem requirements.
- **Modularity** → the single attribute of software that allows a program to be intellectually manageable".

FIGURE 12.2
Modularity and software cost



Lakshmi M B

6. Information Hiding :

- The principle of information hiding suggests that modules should be specified and designed so that information (algorithms and data) contained within a module is inaccessible to other modules that have no need for such information.
- The intent of information hiding is to hide the details of data structures and procedural processing behind a module interface. Knowledge of the details need not be known by users of the module.

7. Functional Independence :

- Functional independence is achieved by developing modules with " single-minded" function and an "aversion" to excessive interaction with other modules.
- Functional independence is a key to good design, and design is the key to software quality.
- Independence is assessed using 2 qualitative criteria:
  1. **Cohesion** → an indication of the relative functional strength of a module.
  2. **Coupling** → an indication of the relative interdependence among modules.

8. Refinement :

- Stepwise refinement is a top-down design strategy.
- It is actually a process of elaboration.
- You begin with a statement of function (or description of information) that is defi ned at a high level of abstraction.
- You then elaborate on the original statement, providing more and more detail as each successive refinement (elaboration) occurs.
- Abstraction and refinement are complementary concepts. Abstraction enables you to specify procedure and data internally but suppress the need for "outsiders" to have knowledge of low-level details. Refinement helps you to reveal low-level details as design progresses

9. Aspects :

- As design begins, requirements are refined into a modular design representation.
- Consider 2 requirements, A and B. Requirement A **crosscuts** requirement B "if a software decomposition [refinement] has been chosen in which B cannot be satisfied without taking A into account".
- An aspect is a representation of a crosscutting concern.
- An aspect is implemented as a separate module (component) rather than as software fragments.

10. Refactoring :

- A reorganization technique that simplifies the design (or code) of a component without changing its function or behavior.
- Refactoring is the process of changing a software system in such a way that it does not alter the external behavior of the code [design] yet improves its internal structure.
- When software is refactored, the existing design is examined for redundancy, unused design elements, inefficient or unnecessary algorithms, poorly constructed or inappropriate data structures, or any other design failure that can be corrected to yield a better design.

11. Object-Oriented Design Concepts :

- OO design concepts –
  - classes and objects,
  - inheritance,
  - messages, and
  - polymorphism.

12. Design Classes:

- These refine the analysis classes by providing design detail that will enable the classes to be implemented, and implement a software infrastructure that supports the business solution.
- 5 different types of design classes:
  1. **User interface classes** → define all abstractions that are necessary for human-computer interaction (HCI) and often implement the HCI in the context of a metaphor.
  2. **Business domain classes** → identify the attributes and services (methods) that are required to implement some element of the business domain that was defined by one or more analysis classes.
  3. **Process classes** → implement lower-level business abstractions required to fully manage the business domain classes.
  4. **Persistent classes** → represent data stores (e.g., a database) that will persist beyond the execution of the software.
  5. **System classes** → implement software management and control functions that enable the system to operate and communicate within its computing environment and with the outside world.

- 4 characteristics of a well-formed design class:
  1. Complete and sufficient
  2. Primitiveness
  3. High cohesion
  4. Low coupling

13. Dependency Inversion:

- The dependency inversion principle states: High-level modules (classes) should not depend [directly] upon low-level modules. Both should depend on abstractions. Abstractions should not depend on details. Details should depend on abstractions.

14. Design for Test:

- Whether to design and then test, or test before implementing a code.

THE DESIGN MODEL

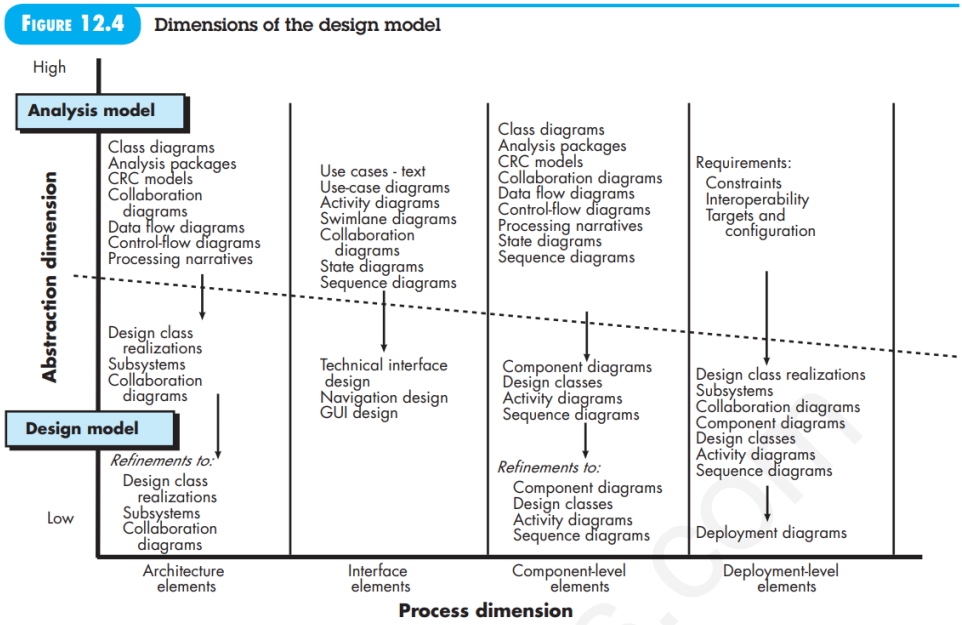


FIGURE 12.4 Dimensions of the design model

Lakshmi M B

- The design model can be viewed in two different dimensions:
  1. **Process dimension** → indicates the evolution of the design model as design tasks are executed as part of the software process.
  2. **Abstraction dimension** → represents the level of detail as each element of the analysis model is transformed into a design equivalent and then refi ned iteratively.
- The design model has 4 major elements:
  1. Data Design Elements,
  2. Architectural Design Elements,
  3. Interface Design Elements, and
  4. Component-Level Design Elements.

Lakshmi M B

1. Data Design Elements:

- Data design (or data architecting) creates a model of data and/or information that is represented at a high level of abstraction (the customer/user's view of data).
- Then refined into progressively more implementation-specific representations that can be processed by the computer-based system.
- At the program-component level, the design of data structures and the associated algorithms is essential to the creation of high- quality applications.
- At the application level, the translation of a data model (derived as part of requirements engineering) into a database is pivotal to achieving the business objectives of a system.
- At the business level, the collection of information stored in disparate databases and reorganized into a "data warehouse" enables data mining or knowledge discovery that can have an impact on the success of the business itself.

Lakshmi M B

2. Architectural Design Elements:

- The architectural design for software is the equivalent to the fl oor plan of a house.
- The architectural model is derived from 3 sources:
  1. information about the application domain for the software to be built;
  2. specific requirements model elements such as use cases or analysis classes, their relationships and collaborations for the problem at hand; and
  3. the availability of architectural styles and patterns.
- The architectural design element is usually depicted as a set of interconnected subsystems, often derived from analysis packages within the requirements model.
- Each subsystem may have its own architecture.

Lakshmi M B

3. Interface Design Elements:

- The interface design for software is analogous to a set of detailed drawings (and specifications) for the doors, windows, and external utilities of a house.
- The interface design elements for software depict information flows into and out of a system and how it is communicated among the components defined as part of the architecture.
- There are 3 important elements of interface design:
  1. the user interface (UI),
  2. external interfaces to other systems, devices, networks, or other producers or consumers of information, and
  3. internal interfaces between various design components.
- These interface design elements allow the software to communicate externally and enable internal communication and collaboration among the components that populate the software architecture.

4. Component-Level Design Elements:

- The component-level design for software is the equivalent to a set of detailed drawings (and specifications) for each room in a house.
- The component-level design for software fully describes the internal detail of each software component.
- To accomplish this, the component-level design defines data structures for all local data objects and algorithmic detail for all processing that occurs within a component and an interface that allows access to all component operations (behaviors).
- The design details of a component can be modeled at many different levels of abstraction.

4. Development-Level Design Elements:

- Deployment-level design elements indicate how software functionality and subsystems will be allocated within the physical computing environment that will support the software.

Lakshmi M B