

SOFTWARE ENGINEERING

Chapter 2

Software Engineering Requirements & Development of Analysis & Design Models

Detailed Study Notes with MCQs and Broad Questions

For Engineering Students

Contents

2.1 Software Engineering Core Principles; Communication, Planning, Modeling, Construction & Deployment Principles

2.2 Requirements Engineering Tasks, Initiating the Requirements Process

2.3 Analysis Approaches & Preparation of the Analysis Model (Data Modeling, Concepts, Object-Oriented Analysis, Flow-Oriented, Class-Based and Behavioral Models)

2.4 Design Approaches & Preparation of the Design Model (Design Concepts, Design Model, Pattern-Based Design)

Multiple Choice Questions (15)

Answer Key

Broad Questions (10)

2.1 Software Engineering Core Principles & Framework-Activity Principles

Introduction

A **principle** is an important underlying law or assumption required in a system of thought. Principles of software engineering guide practitioners in the technical and management work of building software. They do not guarantee success, but ignoring them invites failure. Practice, on the other hand, is a collection of concepts, principles, methods and tools that a software engineer calls upon on a daily basis. Principles are the foundation of practice, and they are applied to each framework activity: **communication, planning, modeling, construction** and **deployment**.

Core Principles of Software Engineering (David Hooker)

No.	Principle	Meaning
1	The Reason It All Exists	A software system exists to provide value to its users. Every decision must be made keeping this in mind.
2	KISS (Keep It Simple, Stupid!)	Software design is not a haphazard process. Keep the design as simple as possible, but not simpler; simple designs are easier to understand, test and maintain.
3	Maintain the Vision	A clear vision is essential to the success of a software project. Compromising the architectural vision weakens the system; an architect should hold the vision.
4	What You Produce, Others Will Consume	Always specify, design and implement knowing someone else will have to understand and maintain what you have done. Write readable code and clear documentation.
5	Be Open to the Future	A system with a long lifetime has more value. Never design yourself into a corner; always ask 'what if?' and be ready to adapt.
6	Plan Ahead for Reuse	Reuse of code and designs saves time and effort. Planning for reuse requires foresight but pays off over the life of the system.
7	Think!	Placing clear, complete thought before action almost always produces better results. When you think, you gain knowledge and can give better answers.

A. Communication Principles

Before customer requirements can be analysed, modeled or specified, they must be gathered through the communication activity. Communication with the customer and other stakeholders is difficult because of different goals, different vocabularies and conflicting priorities. The following principles make communication effective.

- 1 **Listen:** Focus on the speaker's words rather than formulating a response; ask for clarification, but never interrupt.

- 2 **Prepare before you communicate:** Spend time to understand the problem before meeting; research the business domain and prepare the agenda.
- 3 **Someone should facilitate the activity:** A facilitator (a leader) keeps the conversation moving in a productive direction, mediates conflicts and ensures the principles are followed.
- 4 **Face-to-face communication is best:** It works better when supported by some other representation such as a drawing, a document or a screen mock-up.
- 5 **Take notes and document decisions:** Things have a way of falling through the cracks; someone should act as a recorder.
- 6 **Strive for collaboration:** Collective knowledge of team members is used to describe product functions; collaboration builds trust.
- 7 **Stay focused, modularize your discussion:** The more people involved, the more likely the discussion is to jump from topic to topic; the facilitator should keep it modular and leave a topic only when it is resolved.
- 8 **If something is unclear, draw a picture:** Sketches and diagrams often clear up confusion.
- 9 **Move on:** (a) Once you agree on something, move on; (b) if you cannot agree, move on; (c) if a feature is unclear and no one can clarify it right now, move on.
- 10 **Negotiation is not a contest or a game:** It works best when both parties win; each side must respect the other.

B. Planning Principles

The planning activity creates a 'road map' that helps the team get to its destination. It may be as informal as a sketch or as formal as a detailed schedule. These ten principles apply to project planning.

No.	Planning Principle	Explanation
1	Understand the scope of the project	Scope tells the team where it is going; without it you cannot select a route.
2	Involve the customer in planning	The customer sets priorities and defines delivery dates and constraints.
3	Recognize that planning is iterative	As work begins, changes are inevitable; the plan must be adjusted.
4	Estimate based on what you know	Estimates indicate the effort, cost and duration based on the team's current understanding.
5	Consider risk as you define the plan	Identify risks and create contingency plans; high-impact risks affect the schedule.
6	Be realistic	People do not work 100 percent every day; include noise, change and omission in plans.
7	Adjust granularity as you plan	Fine granularity (hours/days) for near-term tasks; coarse granularity for later tasks.

No.	Planning Principle	Explanation
8	Define how you intend to ensure quality	Include reviews, inspections and testing in the plan.
9	Describe how you intend to accommodate change	Define how change requests will be assessed and approved.
10	Track the plan frequently and make adjustments as required	Check daily or weekly; slippage of one day is easier to fix than slippage of a month.

C. Modeling Principles

Models are built to better understand the actual entity to be built. We create **analysis models** to understand requirements and **design models** to describe the solution. Agile modeling principles (Scott Ambler) say: model with a purpose, use multiple models, travel light, content is more important than representation, know the models and the tools, and adapt locally.

Requirements (Analysis) Modeling Principles

- 1 The **information domain** of a problem must be represented and understood (data flowing in, out and stored).
- 2 The **functions** that the software performs must be defined.
- 3 The **behavior** of the software (as a consequence of external events) must be represented.
- 4 Models depicting information, function and behavior must be **partitioned** in a manner that uncovers detail in a layered (hierarchical) fashion.
- 5 The analysis task should move from **essential information toward implementation detail**.

Design Modeling Principles

- Design should be **traceable** to the analysis model.
- Always consider the **architecture** of the system to be built.
- Design of **data** is as important as design of processing functions.
- **Interfaces** (both internal and external) must be designed with care.
- **User interface** design should be tuned to the needs of the end user and stress ease of use.
- **Component-level design** should be functionally independent (high cohesion).
- **Components** should be loosely coupled to one another and to the external environment.
- Design representations should be **easily understandable**.
- Design should be developed **iteratively**; with each iteration the designer should strive for greater simplicity.

D. Construction Principles

The construction activity encompasses a set of **coding** and **testing** tasks that lead to operational software that is ready for delivery.

Coding Principles	Testing Principles
Preparation: understand the problem, design principles and language; pick a programming environment; create unit tests.	All tests should be traceable to customer requirements .
Coding: constrain the algorithm by structured programming; select data structures to meet the design; understand the software architecture; keep conditional logic simple; write self-documenting code; follow coding standards.	Tests should be planned long before testing begins.
Validation: conduct a code walkthrough; perform unit tests and correct errors; refactor the code.	The Pareto principle applies: 80 percent of errors are found in 20 percent of the components.
Use meaningful names and comments; avoid deeply nested loops.	Testing should begin 'in the small' (unit) and progress toward testing 'in the large' (system).
Keep modules short and cohesive.	Exhaustive testing is not possible ; select adequate coverage.
	Testing should be conducted by an independent third party for greater effectiveness.

E. Deployment Principles

The deployment activity includes **delivery, support and feedback**. Because modern software process models are evolutionary or incremental, deployment happens not once but several times.

- 1 **Customer expectations for the software must be managed:** Do not promise more than you can deliver; avoid conflict through honest communication.
- 2 **A complete delivery package should be assembled and tested:** Executable software, support data files, manuals, installation instructions, and so on.
- 3 **A support regime must be established before the software is delivered:** Provide help desk and responsive channels so users get timely answers.
- 4 **Appropriate instructional materials must be provided to end users:** Training aids, troubleshooting guides and a 'what is different' description of the new release.
- 5 **Buggy software should be fixed first, delivered later:** Delivering low-quality software damages reputation; fix defects before the release.

Remember: Communication - listen and collaborate. Planning - scope, risk, iteration. Modeling - understand and partition. Construction - code clean and test early. Deployment - manage expectations and support users.

2.2 Requirements Engineering Tasks & Initiating the Requirements Process

What is a Requirement?

A **requirement** is a condition or capability that a system must satisfy or possess to solve a problem or achieve an objective of a user or stakeholder. **Requirements engineering (RE)** is the process of discovering, analysing, documenting, validating and managing these requirements. It builds a bridge to design and construction, and helps the software engineer understand the problem to be solved, what impact the software will have on the business, what the customer wants, and how end users will interact with the software.

Types of Requirements

Type	Description	Example
Functional requirements	Describe what the system should do - services, functions and behaviour for specific inputs.	The system shall allow a student to register for an examination online.
Non-functional requirements	Quality attributes and constraints - performance, security, usability, reliability, portability, maintainability.	The system shall respond within 2 seconds for 95 percent of requests.
User requirements	High-level statements, usually in natural language, written for customers and users.	The user wants to view the account balance.
System requirements	Detailed descriptions of the system's functions, services and constraints, written for developers.	SRS-level detailed specification.
Domain requirements	Derived from the application domain and reflect domain characteristics.	Medical software must follow patient-record privacy rules.

Requirements Engineering Tasks

RE is accomplished through seven distinct tasks. Some of these tasks occur in parallel and all are adapted to the needs of the project.

No.	Task	Description
1	Inception	Establishes a basic understanding of the problem, the people who want a solution, the nature of the solution desired, and the effectiveness of preliminary communication and collaboration.
2	Elicitation	Draws out requirements from stakeholders; addresses problems of scope, understanding and volatility.
3	Elaboration	Expands and refines information obtained during inception and elicitation into a refined requirements model (user scenarios, functions, features, constraints).
4	Negotiation	Reconciles conflicting requirements; prioritises requirements, assesses cost and risk, and reaches a 'win-win' agreement.

No.	Task	Description
5	Specification	Produces a written work product - a written document, set of graphical models, formal mathematical model, collection of usage scenarios, a prototype, or any combination (the SRS).
6	Validation	Examines the specification for errors, omissions, ambiguity, inconsistency and conflicts using formal technical reviews.
7	Requirements management	Activities that help the team identify, control and track requirements and changes to requirements as the project proceeds.

Initiating the Requirements Process (Inception)

In general, most projects begin when a business need is identified or a new market or service opportunity is discovered. At inception, the software team asks a set of context-free questions to establish a basic understanding of the problem. The goal is to establish a basic understanding and an effective, collaborative working relationship with the customer.

Steps in Initiating the Process

- 1 **Identifying stakeholders:** A stakeholder is anyone who benefits in a direct or indirect way from the system being developed - business operations managers, product managers, marketing people, internal and external customers, end users, consultants, product engineers, software engineers and support engineers. Each stakeholder has a different view of the system.
- 2 **Recognizing multiple viewpoints:** Different stakeholders see the system from different angles (marketing: features that excite the market; business managers: features within budget and on time; end users: easy-to-learn features; support engineers: maintainability). The requirements engineer must categorise all information and conflicts so decisions can be made.
- 3 **Working toward collaboration:** The job of the requirements engineer is to identify areas of commonality and areas of conflict. A collaborative approach leads to a consistent set of requirements, often by letting stakeholders vote or by a 'champion' deciding.
- 4 **Asking the first questions:** Ask context-free questions about the customer, goals and benefits; then about the problem and the solution; then about the effectiveness of the communication itself.

The First Set of Questions

Group	Sample Questions
Stakeholders and goals	Who is behind the request for this work? Who will use the solution? What will be the economic benefit of a successful solution? Is there another source for the solution?
Problem and solution	How would you characterise 'good' output? What problems will this solution address? Show me the business environment. Will special performance issues or constraints affect the way the solution is approached?
Meta-questions	Are you the right person to answer these questions? Are my questions relevant to the problem? Am I asking too many questions? Can anyone else provide additional information? Should I be asking you anything else?

Elicitation of Requirements

Requirements elicitation is a combination of problem solving, elaboration, negotiation and specification. It is difficult because of **problems of scope** (unclear boundaries), **problems of understanding** (customers are not sure what they need) and **problems of volatility** (requirements change over time).

Elicitation Techniques

Technique	Description
Interviews	Structured or unstructured conversations with stakeholders.
Questionnaires / surveys	Used to collect data from a large number of users.
Collaborative Requirements Gathering (FAST/JAD)	Joint meetings of customers and developers with a facilitator; participants prepare lists of objects, services, constraints and performance criteria and agree on a consolidated list. Ground rules for the meeting are set in advance.
Quality Function Deployment (QFD)	Translates the needs of the customer into technical requirements; identifies normal , expected and exciting requirements.
Usage scenarios / use cases	Describe how end users interact with the system.
Observation (ethnography)	Observing users at work to learn how they actually perform tasks.
Prototyping	Showing a working model to elicit feedback.
Document analysis	Studying existing documents, forms and systems.

QFD Requirement Type	Meaning	Example
Normal requirements	Objectives and goals stated for a product during meetings with the customer; satisfaction increases when present.	Graphical displays, specific system functions
Expected requirements	Implicit and so basic that the customer does not state them; their absence causes dissatisfaction.	Ease of human-machine interaction, reliability, easy installation
Exciting requirements	Features that go beyond the customer's expectations and prove very satisfying when present.	Multi-touch gestures, voice commands in a word processor

Use Cases

A **use case** describes how an actor interacts with the system to achieve a goal. An **actor** is a person, device or another system that interacts with the software. A use case diagram shows actors, use cases (ovals) and the system boundary (rectangle), along with relationships such as *include* and *extend*.

Use Case Element	Content
Name	Short verb-phrase, e.g., 'Withdraw Cash'
Primary actor	The actor that initiates the use case
Preconditions	State of the system before the use case starts

Use Case Element	Content
Main flow	Numbered steps of normal interaction
Alternate flows	Exceptions and variations
Postconditions	State of the system after successful completion

Negotiating Requirements

- Identify the key stakeholders - the people who will be involved in the negotiation.
- Determine each stakeholder's 'win conditions'.
- Negotiate: work toward a set of requirements that lead to 'win-win' for all parties.
- Prioritise requirements and resolve conflicts about cost, schedule and risk.

Specification: Software Requirements Specification (SRS)

The SRS is the official statement of what is required of the system developers. It is a **contract** between the customer and the developer. A good SRS is correct, unambiguous, complete, consistent, ranked for importance and/or stability, verifiable, modifiable and traceable (IEEE 830).

SRS Section	Contents
1. Introduction	Purpose, scope, definitions, acronyms, references, overview.
2. Overall description	Product perspective, product functions, user characteristics, constraints, assumptions and dependencies.
3. Specific requirements	External interface requirements, functional requirements, performance requirements, design constraints, quality attributes.
4. Appendices and index	Supporting information.

Validation of Requirements

Requirements validation examines the specification to ensure all requirements have been stated unambiguously, inconsistencies, omissions and errors have been detected and corrected, and the work products conform to standards. The primary mechanism is the **formal technical review**. A review checklist asks questions such as:

- Is each requirement consistent with the overall objective of the system?
- Have all requirements been specified at the proper level of abstraction?
- Is the requirement really necessary or an add-on feature?
- Is each requirement bounded and unambiguous?
- Does each requirement have attribution (source)?
- Do any requirements conflict with other requirements?
- Is each requirement achievable in the technical environment?

- Is each requirement testable once implemented?
- Does the requirements model properly reflect the information, function and behaviour of the system?

Requirements Management

Requirements keep changing throughout the project. Requirements management is a set of activities that help the project team identify, control and track requirements and changes. Key tools are **unique identifiers**, **traceability tables** (features, source, dependency, subsystem, interface) and a **change control** process with a change control board.

Poly Notes Hub

2.3 Analysis Approaches & Preparation of the Analysis Model

Introduction to Requirements Analysis

Requirements analysis results in the specification of software's operational characteristics, indicates the software's interface with other system elements, and establishes constraints that software must meet. It allows the software engineer to refine the software allocation and build models of the data, functional and behavioural domains that will be treated by software. The analysis model provides the software designer with a representation of information, function and behaviour that can be translated into data, architectural, interface and component-level designs.

Objectives of the Analysis Model

- To describe what the customer requires.
- To establish a basis for the creation of a software design.
- To define a set of requirements that can be validated once the software is built.

Rules of Thumb for Analysis Modeling

- The model should focus on requirements visible within the problem or business domain; the level of abstraction should be relatively high.
- Each element of the analysis model should add to an overall understanding of the requirements.
- Delay consideration of infrastructure and non-functional models until the design phase.
- Minimise coupling throughout the system.
- Be certain that the analysis model provides value to all stakeholders.
- Keep the model as simple as it can be.

Analysis Approaches

Aspect	Structured Analysis	Object-Oriented Analysis
Viewpoint	Treats data and processes as separate entities.	Combines data and operations into objects (classes).
Main focus	Functions (what the system does) and data flow.	Objects, classes, attributes, operations and relationships.
Key models	ERD, DFD, data dictionary, process specifications, state-transition diagram.	Use cases, class diagrams, CRC cards, sequence and state diagrams.
Decomposition	Top-down functional decomposition.	Object decomposition and encapsulation.
Reusability	Low to moderate.	High (inheritance, polymorphism).
Change impact	Changes in data structures affect many functions.	Changes are localised within classes.
Pioneers	DeMarco, Yourdon, Constantine	Booch, Rumbaugh, Jacobson, Coad

Elements of the Analysis Model

Element	Purpose	Representations
Scenario-based elements	Describe the system from the user's point of view.	Use cases, activity diagrams, swimlane diagrams, user stories
Flow-oriented elements	Show how data objects are transformed as they flow through the system.	Data flow diagrams, control flow diagrams, processing narratives
Class-based elements	Define classes, attributes and operations that the system manipulates.	Class diagrams, analysis packages, CRC models, collaboration diagrams
Behavioral elements	Show how the software behaves in response to external events.	State diagrams, sequence diagrams
Data model elements (Data modeling)	Show data objects and relationships.	Entity-relationship diagrams (ERD), data dictionary

A. Data Modeling Concepts

Data modeling examines data objects independently of processing. It focuses on the data domain and creates a model at the customer's level of abstraction. The most common notation is the **Entity-Relationship Diagram (ERD)**, introduced by Peter Chen.

Basic Concepts

Concept	Description	Example
Data object	Representation of composite information that must be understood by the software; has a number of different properties or attributes.	Customer, Book, Student, Invoice
Attributes	Define the properties of a data object; used to name an instance, describe it or make reference to another instance; one or more attributes act as an identifier (key) .	Student: roll_no (key), name, branch, address
Relationships	Indicate the manner in which data objects are connected to one another.	Student <i>enrols in</i> Course; Customer <i>places</i> Order
Cardinality	Specifies the number of occurrences of one object that can be related to the number of occurrences of another: one-to-one (1:1), one-to-many (1:N), many-to-many (M:N).	One teacher teaches many students (1:N)
Modality	Indicates whether a relationship is mandatory (modality 1) or optional (modality 0).	An order must belong to a customer (1); a customer may have no orders (0)

ER Diagram Symbols

Symbol	Represents
Rectangle	Entity (data object)
Ellipse (oval)	Attribute (key attribute is underlined)

Symbol	Represents
Diamond	Relationship between entities
Lines	Connect attributes to entities and entities to relationships, with cardinality labels
Double rectangle / double ellipse	Weak entity / multivalued attribute

B. Flow-Oriented Modeling

Flow-oriented modeling shows how data objects are transformed by processing functions. It is the basis of **structured analysis**. A **Data Flow Diagram (DFD)** depicts the flow of data through the system, and the transformations applied to data as it moves from input to output.

DFD Notation

Symbol	Name	Meaning
Rectangle (square)	External entity	Source or sink of data outside the system (customer, bank).
Circle or rounded rectangle (bubble)	Process	Transforms input data into output data; named with a verb phrase.
Arrow	Data flow	Movement of data; labeled with the data name.
Open-ended rectangle / two parallel lines	Data store	Repository where data is held (file or database).

Levels of DFD

- **Level 0 (context diagram):** The entire system is represented as a single bubble with inputs and outputs and external entities.
- **Level 1:** The context process is refined into major subprocesses (about 5 to 9) showing data stores.
- **Level 2 and lower:** Each bubble is decomposed further until the processes are simple enough to be described by a short process specification (mini-spec).

Rules for Drawing DFDs

- Every process must have at least one input and one output.
- Data cannot flow directly between two external entities, or two data stores.
- Data flows must be named; processes must be numbered.
- Balance must be maintained between parent and child levels (inputs and outputs of the parent equal those of the child diagram).
- Avoid showing control or decision logic in a DFD.

Other Flow-Oriented Components

Component	Purpose
Control flow diagram (CFD)	Models event-driven (real-time) systems; shows control items (events and signals) and control processes.
Process specification (PSPEC)	Describes the processing of each bubble using structured English, decision tables or pseudo-code.
Control specification (CSPEC)	Describes the behaviour of the system through state diagrams and decision tables.
Data dictionary	Organised listing of all data elements with precise definitions of name, aliases, use, content and format.

Data Dictionary Notation

Notation	Meaning
=	is composed of
+	and (sequence)
[]	either-or (selection)
{ }	repetition (iteration)
()	optional data
* *	comment

Example: *Name = first_name + (middle_name) + last_name*

C. Scenario-Based Modeling (Use Cases and Activity Diagrams)

Scenario-based modeling is the first element of the analysis model because requirements are best understood from the user's point of view. It consists of **use cases** (text and diagram) supplemented by **activity diagrams** (flowchart-like representation of a procedural flow) and **swimlane diagrams** (activity diagram divided into lanes showing which actor is responsible for each activity).

D. Object-Oriented Analysis (OOA)

The intent of object-oriented analysis is to define all classes that are relevant to the problem to be solved - the operations and attributes associated with them, the relationships and behaviours associated with them.

Steps in OOA

- 1 Elicit customer requirements and build **use cases**.
- 2 Identify **classes** (nouns in use case descriptions) with attributes and operations.
- 3 Define **class hierarchy** and relationships (association, aggregation, composition, generalization).
- 4 Build an **object-relationship model** (class diagram).
- 5 Model **object behaviour** using state and sequence diagrams.
- 6 Review the OO analysis model against use cases and refine.

Basic OO Concepts

Concept	Meaning
Object	An instance of a class; has state (attributes), behaviour (operations) and identity.
Class	A template or blueprint describing a set of objects with similar attributes and operations.
Encapsulation	Packaging of data and operations into a class; hides internal details.
Inheritance	A subclass acquires attributes and operations from a superclass; supports reuse.
Polymorphism	The same operation behaves differently on different classes.
Abstraction	Focusing on essential features while ignoring unnecessary details.
Message	A request sent to an object to invoke one of its operations.

E. Class-Based Modeling

Class-based modeling represents the objects that the system will manipulate, the operations (methods or services) that will be applied to the objects, the relationships between objects, and the collaborations that occur between the classes that are defined. Its elements are classes, objects, attributes, operations, **CRC models**, collaboration diagrams and packages.

Identifying Analysis Classes

Classes are found by performing a grammatical parse of the use case narrative: nouns become candidate classes, verbs become operations. Candidate classes manifest in several ways:

Class Category	Examples
External entities	Other systems, devices or people that produce or consume information.
Things	Reports, displays, letters, signals.
Occurrences or events	Property transfer, completion of a series of robot movements.
Roles	Manager, engineer, salesperson (people who interact with the system).
Organisational units	Division, group, team.
Places	Manufacturing floor, loading dock.
Structures	Sensors, four-wheeled vehicles, computers.

Criteria for Selecting Classes (Coad and Yourdon)

- **Retained information:** the potential class is useful only if information about it must be remembered.
- **Needed services:** it must have a set of identifiable operations.
- **Multiple attributes:** a class with a single attribute is better represented as an attribute of another class.
- **Common attributes and operations** apply to all instances.

- **Essential requirements:** external entities that appear in the problem space and produce or consume information are almost always defined as classes.

CRC Modeling (Class-Responsibility-Collaborator)

A CRC model is a collection of standard index cards that represent classes. Each card is divided into three sections: the class name on top, the **responsibilities** (attributes and operations of the class) on the left, and the **collaborators** (other classes required to fulfil responsibilities) on the right.

Class: FloorPlan	
Responsibility	Collaborator
Defines floor plan name / type	(none)
Manages floor plan positioning	Wall
Scales floor plan for display	Camera
Incorporates walls, doors and windows	Wall, Door, Window

Class Types in CRC Modeling

- **Entity classes (model classes):** extracted directly from the problem statement (e.g., Student, Sensor).
- **Boundary classes:** create the interface (e.g., interactive screen, printed reports) that the user sees and interacts with.
- **Controller classes:** manage a 'unit of work' - creation or update of entity objects, communication between sets of objects, validation of data.

Relationships Between Classes

Relationship	Meaning	Example
Association	A simple link between classes.	Student - Course
Aggregation	'Has-a' relationship; the part can exist independently.	Department has Teachers
Composition	Strong aggregation; the part cannot exist without the whole.	House has Rooms
Generalization (inheritance)	'Is-a' relationship.	Car is a Vehicle
Dependency	One class uses another temporarily.	Report depends on Printer

F. Behavioral Modeling

The behavioural model indicates how software will respond to external events or stimuli. It is created in steps: (1) evaluate all use cases to fully understand the sequence of interaction; (2) identify events that drive the interaction and understand how these events relate to specific objects; (3) create a sequence for each use case; (4) build a state diagram for the system; (5) review the behavioural model

to verify accuracy and consistency.

State Diagram

A **state** is an externally observable mode of behaviour. A **state diagram** (state-transition diagram) represents active states for each class and the events (triggers) that cause changes between active states. A **transition** is the movement from one state to another, caused by an event and optionally guarded by a condition.

Notation	Meaning
Rounded rectangle	State
Solid black circle	Initial pseudostate
Circle with inner solid circle (bull's eye)	Final state
Arrow labelled event [guard] / action	Transition

Example (ATM): Idle -> (card inserted) -> Verifying PIN -> (PIN valid) -> Selecting Transaction -> Processing -> (card ejected) -> Idle.

Sequence Diagram

A sequence diagram shows how events cause flow from one object to another as a function of time. Objects are placed horizontally across the top; **lifelines** descend vertically; **messages** are horizontal arrows between lifelines, drawn in time order from top to bottom; **activation bars** show when an object is active.

Aspect	State Diagram	Sequence Diagram
Shows	Life history of a single object or system	Interaction between multiple objects
Focus	States and transitions	Messages in time order
Basis	Events and conditions	Use case scenario

Analysis Pattern and Analysis Package

- **Analysis pattern:** a reusable solution to a recurring problem in the application domain (e.g., Actor-Role).
- **Analysis package:** a grouping of related analysis classes (e.g., a package of classes for the user interface).

Putting the Analysis Model Together

Step	Action	Output
1	Gather requirements and prepare use cases	Use case diagrams / narratives
2	Develop data model	ERD, data dictionary
3	Develop functional (flow) model	DFD levels 0, 1, 2; PSPECs

Step	Action	Output
4	Develop class-based model	Class diagrams, CRC cards
5	Develop behavioural model	State and sequence diagrams
6	Review and validate	Approved analysis model, input to SRS and design

Poly Notes Hub

2.4 Design Approaches & Preparation of the Design Model

Introduction to Software Design

Software design is the last software engineering action within the modeling activity and sets the stage for construction. It is the process by which requirements are translated into a 'blueprint' for constructing the software. The initial blueprint depicts a holistic view of software, and is then refined into representations that are ever closer to code. Design is where **quality is established**: it provides representations of software that can be assessed for quality, serves as the only way to translate requirements accurately into a finished product, and without design we risk building an unstable system.

Design Approaches

Approach	Description	Notations / Techniques
Function-oriented (structured) design	Decomposes the system into functions or procedures; data flows between them. Derived from DFDs by transform or transaction mapping.	Structure charts, DFD, pseudo-code
Object-oriented design	The system is a collection of interacting objects; designs classes, interfaces and collaborations.	UML class, sequence, component and deployment diagrams
Data-structure-oriented design	Structure of the program is derived from the structure of input and output data.	Jackson Structured Programming
Component-based design	Assembles the system from reusable, pre-built components.	Component diagrams, interfaces
Pattern-based design	Applies proven solutions (patterns) to recurring design problems.	Design patterns, architectural styles
Top-down and bottom-up design	Top-down: begin with high-level modules and refine. Bottom-up: begin with basic components and combine.	Stepwise refinement; component libraries

Goals of a Good Design (McGlaughlin's Criteria)

- The design must implement all explicit requirements in the analysis model and accommodate the implicit requirements.
- It must be a readable, understandable guide for those who generate code, test and support the software.
- It should provide a complete picture of the software, addressing the data, functional and behavioural domains from an implementation perspective.

Quality Guidelines

- A design should exhibit an architecture created using recognisable architectural styles or patterns, composed of components with good design characteristics, and implementable in an evolutionary fashion.

- A design should be modular; the software should be logically partitioned into elements.
- A design should contain distinct representations of data, architecture, interfaces and components.
- A design should lead to data structures appropriate for the classes to be implemented.
- A design should lead to components that exhibit independent functional characteristics.
- A design should lead to interfaces that reduce complexity of connections between components and with the external environment.
- A design should be derived using a repeatable method driven by information obtained during requirements analysis.
- A design should be represented using a notation that effectively communicates its meaning.

Quality Attributes (FURPS)

Attribute	Meaning
Functionality	Feature set and capabilities, generality of functions, security of the overall system.
Usability	Human factors, aesthetics, consistency, documentation.
Reliability	Frequency and severity of failure, accuracy of outputs, mean-time-to-failure, ability to recover.
Performance	Processing speed, response time, resource consumption, throughput, efficiency.
Supportability	Extensibility, adaptability, serviceability, maintainability, testability, compatibility, configurability.

A. Design Concepts

Concept	Explanation
Abstraction	Allows us to concentrate on a problem at some level of generalisation without regard to irrelevant low-level details. Procedural abstraction is a named sequence of instructions with a specific function (e.g., 'open a door'); data abstraction is a named collection of data describing a data object (e.g., door with type, swing direction, weight).
Architecture	The overall structure of the software and the ways in which that structure provides conceptual integrity for a system; the structure of program components, their interactions and the data structures they use.
Patterns	A design pattern describes a design structure that solves a particular design problem within a specific context and amid the forces that may have an impact.
Separation of concerns	A complex problem is divided into pieces (concerns) that can be solved and optimised independently.
Modularity	Software is divided into separately named and addressable components called modules that are integrated to satisfy problem requirements. 'Modularity is the single attribute of software that allows a program to be intellectually manageable.'

Concept	Explanation
Information hiding	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 it.
Functional independence	Achieved by developing modules with 'single-minded' function and an 'aversion' to excessive interaction with other modules; measured using cohesion and coupling.
Stepwise refinement	Top-down strategy by Niklaus Wirth; a hierarchy is developed by decomposing a macroscopic statement of function in a stepwise fashion until programming language statements are reached.
Aspects	A representation of a crosscutting concern (e.g., security, logging) that must be accommodated by multiple components.
Refactoring	A reorganisation technique that simplifies the design (or code) of a component without changing its function or behaviour.
OO design concepts	Classes and objects, inheritance, messages, polymorphism.
Design classes	Refine analysis classes by providing design detail that will enable the classes to be implemented; five types: user interface, business domain, process, persistent and system classes.

Cohesion and Coupling

Cohesion is an indication of the relative functional strength of a module: how closely the elements within a module are related. **Coupling** is an indication of the relative interdependence among modules. Good design has **high cohesion and low coupling**.

Cohesion (best to worst)	Description
Functional	All elements contribute to a single, well-defined task (best).
Sequential	Output of one element is the input of the next.
Communicational	Elements operate on the same data.
Procedural	Elements are executed in a specific order but are not necessarily related.
Temporal	Elements are executed at the same time (e.g., initialisation).
Logical	Elements perform similar logical functions chosen by a flag.
Coincidental	Elements have no meaningful relationship (worst).

Coupling (best to worst)	Description
Data coupling	Modules communicate by passing simple data parameters (best).
Stamp coupling	Modules share a composite data structure.
Control coupling	One module passes a control flag that affects the logic of another.
External coupling	Modules share an externally imposed format, protocol or device.
Common coupling	Modules share global data.
Content coupling	One module modifies or relies on the internals of another (worst).

Design Classes (Five Types)

Class Type	Description	Example
User interface classes	Abstractions necessary for human-computer interaction.	Menu, Window, Button
Business domain classes	Refinements of analysis classes that identify business entities.	Customer, Order
Process classes	Implement lower-level business abstractions required to manage business domain classes.	Billing process
Persistent classes	Data stores (database) that persist beyond execution of the software.	Database access class
System classes	Implement software management and control functions that enable the system to operate.	Communication, file management

Characteristics of a Well-Formed Design Class

- **Complete and sufficient:** has all attributes and methods required.
- **Primitiveness:** each method accomplishes one service.
- **High cohesion:** focused on one thing.
- **Low coupling:** minimal collaboration with other classes.

B. The Design Model

The design model can be viewed in two dimensions: the **process dimension** (evolution of the design model as tasks are executed) and the **abstraction dimension** (the level of detail as each element of the analysis model is transformed into a design equivalent and then refined). It is developed from the analysis model, and each element of the analysis model feeds a design element.

Design Model Element	Derived From	Description
Data design (class design)	Class-based model, data dictionary, ERD	Transforms the information domain model into data structures and classes at a high level of abstraction; refined for implementation; also leads to database design.
Architectural design	Analysis classes, their relationships, collaborations and behaviour; architectural styles and patterns	Defines the relationship between the major structural elements of the software, architectural styles and patterns, and constraints that affect the way the architecture can be implemented.
Interface design	Scenario-based and behavioural elements, data flow	Describes how the software communicates with systems that interoperate with it, with humans who use it (UI), and between components. Includes user interface , external interfaces and internal interfaces .

Design Model Element	Derived From	Description
Component-level design	Class-based, flow-oriented and behavioural elements	Transforms structural elements of the architecture into a procedural description of software components; uses class diagrams, flowcharts, pseudo-code, decision tables.
Deployment-level design	Architecture and component design	Indicates how software functionality and subsystems will be allocated within the physical computing environment (UML deployment diagrams).

Mapping Analysis to Design

Analysis Model	Design Model
Scenario-based elements (use cases)	Interface design, component-level design
Class-based elements	Data / class design, component-level design
Flow-oriented elements (DFD)	Architectural design, component-level design
Behavioural elements (state diagrams)	Interface design, component-level design

Architectural Design Basics

- **Software architecture:** The structure of a system composed of software components, the externally visible properties of those components, and the relationships among them.
- **Architectural styles:** Data-centered, data-flow (pipe-and-filter), call-and-return (main program/subprogram, remote procedure call), object-oriented, and layered architectures.
- **Transform and transaction flow mapping:** Techniques for converting a DFD into a program structure (structure chart).

Architectural Style	Description	Example
Data-centered	A central data store is accessed by client software.	Database systems, blackboard
Data-flow	Input data is transformed by a series of computational components (filters) connected by pipes.	Compilers, batch processing
Call-and-return	Program structure is a hierarchy of control; main program invokes subprograms.	Modular programs, RPC
Object-oriented	Components encapsulate data and operations; communicate by message passing.	Java/C++ applications
Layered	Layers each perform operations progressively closer to the machine instruction set.	OS, network protocol stack, MVC

Component-Level Design

A **component** is a modular, deployable and replaceable part of a system that encapsulates implementation and exposes a set of interfaces. In OO views, a component contains a set of

collaborating classes; in the conventional view, it is a functional element with processing logic, internal data structures and an interface. Design uses structured programming constructs (sequence, condition, repetition) and tools such as flowcharts, pseudo-code (program design language) and decision tables.

User Interface Design Golden Rules (Theo Mandel)

- **Place the user in control:** flexible interaction, interruptible and undoable actions, hide technical internals.
- **Reduce the user's memory load:** reduce demand on short-term memory, establish meaningful defaults, define intuitive shortcuts.
- **Make the interface consistent:** allow the user to put the current task in a meaningful context, maintain consistency across the family of applications.

C. Pattern-Based Software Design

Throughout the design process, a software engineer should look for every opportunity to reuse existing design patterns (when they meet the needs of the design) rather than creating new ones. A **design pattern** is a named nugget of insight that conveys the essence of a proven solution to a recurring problem within a certain context amid competing concerns.

Kinds of Patterns

Kind	Level of Abstraction	Examples
Architectural patterns	Describe broad-based design problems solved using a structural approach.	Layered, MVC, pipe-and-filter, client-server
Design patterns	Address a specific element of the design such as an aggregation of components, to solve a design problem.	Singleton, Factory, Observer, Adapter
Idioms (coding patterns)	Language-specific implementation of an algorithm or data structure.	Iterator idiom in Java; RAII in C++

Pattern Template (Description of a Design Pattern)

Element	Description
Name	Short, meaningful identifier.
Intent	What the pattern does and the problem it solves.
Also-known-as	Other names for the pattern.
Motivation	An example scenario illustrating the problem.
Applicability	Situations in which the pattern can be used.
Structure	Class diagram of the solution.
Participants	Classes and objects with their responsibilities.
Collaborations	How participants work together.

Element	Description
Consequences	Results, trade-offs and benefits.
Implementation, sample code, known uses, related patterns	Guidance, code fragments, real systems, and links to related patterns.

Gang of Four (GoF) Pattern Categories

Category	Purpose	Patterns
Creational	How objects are created.	Singleton, Factory Method, Abstract Factory, Builder, Prototype
Structural	How classes and objects are composed into larger structures.	Adapter, Bridge, Composite, Decorator, Facade, Flyweight, Proxy
Behavioral	How objects interact and distribute responsibility.	Observer, Strategy, Command, Iterator, State, Template Method, Mediator, Chain of Responsibility

Using Patterns in Design

- **Architectural patterns** are used to define the overall structure; **component-level patterns** to define the components; **user interface patterns** for interaction design.
- **Frameworks:** A framework is not an architectural pattern but a skeleton of infrastructure that can be adapted by the implementer; it consists of a collection of plug-in points ('hooks' or 'slots') that enable it to be adapted to a specific problem domain.

Pattern-Based Design Process

- 1 Examine the requirements model and develop a problem statement that describes the problem at each level of abstraction.
- 2 Search for patterns in pattern repositories (pattern languages) that match the problem and context.
- 3 Choose the most suitable pattern by comparing forces, consequences and context.
- 4 Adapt the pattern to the problem; combine with other patterns when needed.
- 5 Validate the design, then refine into components and code.

Pattern-Organising Tables and Common Mistakes

Item	Description
Pattern-organising table	A table that groups patterns by database, application, implementation or infrastructure problem so that designers can find the right one quickly.
Not enough understanding	Applying a pattern without understanding the problem.
Wrong context	Using a pattern where its forces do not apply.
Pattern overuse	Forcing patterns everywhere, creating needless complexity.
Ignoring forces	Missing the trade-offs among the competing concerns of the pattern.

Design Review and Deliverables

- **Design specification:** Description of architecture, data, interface and component designs (the Software Design Document, SDD).
- **Design reviews:** Formal technical reviews assess whether the design meets requirements, is traceable, and follows quality guidelines.
- **Traceability:** Every design element should map back to a requirement.

Poly Notes Hub

Multiple Choice Questions

Choose the correct option for each question. The answer key is given on the next page.

Q1. Which of the following is the first of Hooker's core principles of software engineering?

- (a) KISS
- (b) Think!
- (c) The Reason It All Exists
- (d) Plan Ahead for Reuse

Q2. Which RE task establishes a basic understanding of the problem and the people who want a solution?

- (a) Elicitation
- (b) Inception
- (c) Validation
- (d) Negotiation

Q3. The official statement of what is required of the system developers is called:

- (a) SDD
- (b) SRS
- (c) DFD
- (d) ERD

Q4. In QFD, features whose absence causes dissatisfaction because they are so basic are called:

- (a) Normal requirements
- (b) Exciting requirements
- (c) Expected requirements
- (d) Volatile requirements

Q5. Which of the following is a non-functional requirement?

- (a) Student can register online
- (b) System shall respond in under 2 seconds
- (c) Admin can add a course
- (d) User can print receipts

Q6. In data modeling, 'cardinality' specifies:

- (a) Whether a relationship is optional
- (b) The number of occurrences of one object related to another
- (c) The data type of an attribute
- (d) The name of a data object

Q7. In an ER diagram, a relationship is represented by a:

- (a) Rectangle
- (b) Ellipse
- (c) Diamond
- (d) Circle

Q8. A Level-0 DFD is also known as the:

- (a) Context diagram
- (b) Process specification
- (c) State diagram
- (d) Class diagram

Q9. In a CRC card, the 'C' that stands for classes required to fulfil responsibilities is:

- (a) Class name
- (b) Collaborator
- (c) Cardinality
- (d) Constraint

Q10. Which diagram shows interaction of objects arranged in time order?

- (a) State diagram
- (b) ER diagram
- (c) Sequence diagram
- (d) DFD

Q11. A good design has:

- (a) Low cohesion and high coupling
- (b) High cohesion and low coupling
- (c) High cohesion and high coupling
- (d) Low cohesion and low coupling

Q12. The best (strongest) type of cohesion is:

- (a) Coincidental
- (b) Logical
- (c) Functional
- (d) Temporal

Q13. The design element that describes how software communicates with users and other systems is the:

- (a) Data design
- (b) Architectural design
- (c) Interface design
- (d) Deployment design

Q14. Hiding the internal details of a module from other modules is called:

- (a) Refactoring
- (b) Information hiding
- (c) Refinement
- (d) Aggregation

Q15. Singleton, Factory and Builder belong to which GoF pattern category?

- (a) Structural
- (b) Behavioral
- (c) Creational
- (d) Architectural

Answer Key

Q. No.	Answer	Explanation
1	c	The first principle is 'The Reason It All Exists' - value to users.
2	b	Inception establishes basic understanding of the problem.
3	b	The SRS is the requirements contract.
4	c	Expected requirements are implicit and basic.
5	b	Performance is a quality attribute, hence non-functional.
6	b	Cardinality = number of occurrences; modality = optional/mandatory.
7	c	Diamond denotes a relationship.
8	a	Level 0 DFD is the context diagram.
9	b	C stands for collaborator.
10	c	Sequence diagram shows time-ordered messages.
11	b	High cohesion, low coupling.
12	c	Functional cohesion is the best.
13	c	Interface design covers UI, external and internal interfaces.
14	b	Information hiding.
15	c	These are creational patterns.

Broad Questions

- 1 Explain Hooker's seven core principles of software engineering with a suitable example for each.
- 2 Describe the communication and planning principles of software engineering in detail.
- 3 Explain the modeling, construction and deployment principles of software engineering.
- 4 What is requirements engineering? Explain the seven requirements engineering tasks in detail.
- 5 Explain how the requirements process is initiated. Describe stakeholders, multiple viewpoints and the first set of questions. Also explain elicitation techniques and QFD.
- 6 What is an SRS? Explain its structure and the characteristics of a good SRS. How are requirements validated?
- 7 Explain data modeling with data objects, attributes, relationships, cardinality and modality. Draw an ER diagram for a library management system.
- 8 Explain flow-oriented modeling. Draw Level 0 and Level 1 DFDs for an online shopping system and explain the data dictionary notation.
- 9 Explain object-oriented analysis, class-based modeling (CRC cards) and behavioral modeling (state and sequence diagrams) with examples.
- 10 Explain the design concepts and the design model. Describe cohesion and coupling, and explain pattern-based design with the types of design patterns.

Tip: In exams, support each answer with neat diagrams, tables and examples to score better.