System Integration and Architecture (Finals) - Jaish Khan

---

1. Software Architecture

> High-level structure of a software system which defines organization of its components and relationships/interactions of its components.

It is a systematic arrangement of ideas or information.
Components: Clients, servers, databases, layers, etc.
Interactions: Procedure calls, shared variable access, etc.

Architecture Design - System requirement specification and analysis model

Who is responsible for developing the architecture design?

Software architects and designers.

Why is software architecture design so important?

A poor design may result in a bad product that: doesn't meet system requirements, not adaptive to future, not reusable, shows unpredictable behavior, or performs badly.

When is software architecture design conducted?

An early phase of the Software Development Life Cycle (SDLC).

What are the outcomes?

Overall representation of the software to be built, including elements, connectors, constraints, and runtime behaviors

Architecture is designed with the help of a UML (Unified Modeling Language) or an ADL (Architectural Design Language).

1.1. Architecture Business Cycle (ABC)

- *Influences* → System stakeholders, Developing organization, Architect's background and experience, Technical environment.
- *Precautionary measures* → Knowing constraints and Early stakeholder engagement

1.1.1. ABC Activities

- Creating the business case for the system (Why we need a new system, what will be its cost?)
- Understanding the requirements
- Creating/selecting the architecture
- Communicating the architecture (To all stakeholders)
- Analyzing or evaluating the architecture
- Implementation based on architecture
- Ensuring conformance to an architecture

1.1.2. Architect's Tasks

1. *Static Partition and Decomposition* → Dividing the system into subsystems and defining communication between them.
2. *Dynamic Control Relationships* → Establishing data flow, control flow, and message dispatching between subsystems.
3. *Architecture Style Evaluation* → Considering and evaluating styles suited to the problem domain.
4. *Trade-off Analysis* → Balancing quality attributes and non-functional requirements when choosing architecture styles.

> System Integration (Tightly Coupled) vs System Interoperability (Loosely Coupled)

---

2. Software Design

> The process of defining the architecture, components, interfaces, and other characteristics of a system or its parts.

It is economically important and affects our quality of life as well. Design bridges that gap between knowing what is needed (SRS) to entering the code that makes it work (the construction phase).

2.1. Design vs Architecture

Architecture focuses on high-level concepts and the overall structure of a system. It is about selecting elements like clients, servers, and databases, defining their interactions, and setting constraints. Think of it as the blueprint that guides the system's organization and evolution.

Design takes those high-level concepts and adds concrete details for implementation. It dives into the specifics: modularization, interfaces, algorithms, data types, and the procedures needed to support the architecture and meet requirements.

In essence, architecture lays the foundation, and design builds upon it.

System Design in the SDLC

- Analysis phase defines business needs.
- Design Phase determines how the system will operate.
  - Business requirements are converted into technical system requirements
  - Design documents and models communicate system requirements
  - System Specification is final deliverable of the design phase

2.2. System Acquisition Strategies

1. **Custom in-house Development** → Building a new system from scratch.
   - *Advantages*: Flexibility, creativity, leveraging current technologies, skill building
   - *Disadvantages*: Dedicated effort, skill requirements, high risks
2. **Packaged Software** → Buying pre-built software for common business needs.
   - *Advantages*: Efficiency, quick installation, pre-tested solutions
   - *Disadvantages*: Limited functionality, reliance on vendor customization
   - Customization Options: Parameter manipulation, workarounds

- **Systems Integration** → Combining packaged software, legacy systems, and new software.
  - Challenges: Data integration between different systems

3. **Outsourcing** → Hiring an external vendor to create or supply the system.
    - *ASP* (Application Service Providers) → Supply software applications and/or services over the Internet.
    - *SaaS* (Software as a Service) → An extension of ASP model.
    - Risks: Confidentiality breaches, loss of control, skill loss
    - **Outsourcing Contracts**
        1. *Time and arrangements*
        2. *Fixed-price contract*
        3. *Value-added contract*

Influences on Acquisition Strategy

- Business Need is:
    - Unique → Custom solutions.
    - Common → Packaged software.
    - Not Critical → Outsourcing.
- In-House experience:
    - Functional and Technical → Custom solutions.
    - Functional → Packaged software.
    - Neither → Outsourcing.
- Project Skills:
    - Strategic → Custom solutions.
    - Not Strategic → Packaged software.
- Project Management:
    - Excellent → Custom solutions.
    - Decent → Packaged software or Outsourcing.
- Time Frame:
    - Flexible → Custom solutions.
    - Short → Packaged software.

Selecting an Acquisition Strategy → To implement the strategies, additional information is needed like Tools and Technologies, Vendor Research and Service Provider Evaluation.

- *RFP* (Request for Proposal) → Describes the system/service needed, vendors respond with solutions.
- *RFI* (Request for Information) → Shorter, less detailed alternative to RFP for smaller projects
- *RFQ* (Request for Quote)

Alternative Matrix → Combines several feasibility analyses into one matrix. It is an activity of the design phase.

- *Weighted Alternative Matrix*: Incorporates weights and scores for subjective evaluation

---

3. Requirements Engineering

> Systematic process of determining software product requirements.

It has a formal starting and ending point in the overall Software Development Life Cycle.

- **Starts** → when it is recognized that a problem exists and requires a solution.
- **Ends** → when with a complete description of the external behavior of the software to be built.

It is a *continuous process* in which the related activities are repeated until requirements are of acceptable quality. It is also one of the most critical process in "system development". RE Processes are tailored based on individual project needs.

Inputs of the RE Process

1. *Existing system information* → Information about systems "to be replaced" or other systems which interact with the specified one.
2. *Stakeholder needs* → Description of what the system stakeholders need.
3. *Organizational standards* → Standards about development practices, quality management etc.
4. *Regulations* → External regulations like health and safety regulations.
5. *Domain information* → General info about the applications (of the specific domain).

Outputs of the RE Process

1. *Agreed requirements* → A description of the system requirements, which is understandable by stakeholders and which has been agreed upon by them.
2. *System specification* → A more detailed specification of the system.
3. *System models* → A set of models such as a data flow model, object model and a process model which describes the system from different perspectives.

Variability → RE Processes vary significantly between and within organizations.

- Variability Factors:
    1. *Technical maturity* (technologies and methods)
    2. *Disciplinary involvement* (engineering and managerial disciplines)
    3. *Organizational culture* (culture of the organization)
    4. *Application domain* (Different types of applications)

Types of Requirements → There are 5 types.

1. *Functional Requirements*

2. *Non-Functional Requirements*
3. *User Requirements*
4. *System Requirements*
5. *Business Requirements*

3.1. Requirements Engineering Activities

3.1.1. Requirements Elicitation

> Determining system requirements through stakeholder consultation, documents, domain knowledge, market studies.

Also known as, **Requirements Acquisition** or **Requirements Discovery**.

Elicitation Steps

1. Decide on elicitation scope and agenda.
2. Prepare resources.
3. Prepare questions.
4. Perform session.
5. Organize and share notes.
6. Document issues.

Elicitation Techniques

Interviews → Most commonly used technique where questions are asked.

- Steps:
    1. *Selecting interviewees* (various organizational levels)
    2. *Designing interview questions* (structured vs. unstructured)
        1. Close-Ended vs Open-Ended vs Probing questions.
    3. *Preparing for the interview* (plan, knowledge areas, priorities, interviewee preparation)
    4. *Conducting the interview* (professionalism, recording, understanding, fact separation)
    5. *Post-interview follow-up* (report, interviewee review)

Joint Application Development (JAD) → A collaborative information gathering technique. A structured process in which 10 to 20 users meet under a facilitator. It reduces scope creep by about 50%. They are used to resolve conflicts of requirements and are costly and optional.

- Steps:
    1. *Selecting participants* (similar to interviews, facilitator expertise crucial)

2. *Designing JAD session and preparation* (session 1/2 day to weeks long, plan, information focus)
3. *Conducting the session* (formal agenda, ground rules, facilitator roles, neutrality)
4. *Post-JAD follow-up* (report preparation and circulation)

Questionnaires → Written questions for information gathering.

- Steps:
  1. *Selecting participants* (representative sample)
  2. *Designing the questionnaire* (good practice guidelines)
  3. *Administering the questionnaire* (response rate improvement)

Document Analysis → Used to understand the "as-is" system.
Formal system that the organization uses is described using Forms, reports, manuals, charts.
Informal system (the real one) differs from the formal one, and reveals what needs to be changed.

Observation → The act of watching processes being performed. Allows us to gain insights into the "as-is" system and to validate information from other sources. This step is usually performed when all others are followed completely.

3.1.2. Requirements Analysis

1. **Necessity Checking** → Is the requirement necessary?
   - *Interaction Matrix* → Each requirement is compared with other requirements where
     1. For requirements which conflict, fill in a 1
     2. For requirements which overlap, fill in a 1000
     3. For requirements which are independent, fill in a 0
2. **Consistency and Completeness Checking** → Is the requirement consistent and compelete?
   1. *Consistency* → No requirements should be contradictory.
   2. *Completeness* → No services/constraints which are needed have been missed out.
3. **Feasibility Checking** → Is the requirement feasible in the context of the budget and schedule?

Requirements Negotiation

> Addressing conflicting, incomplete, or infeasible requirements.

- **Stages of Negotiation Meetings**:

- *Information stage*
- *Discussion stage*
- *Resolution stage*

3.1.3. Requirements Specification

> Creating a tangible model of requirements using natural language and diagrams.

Requirements Documentation

Requirements Document → Capturing detailed descriptions of software system requirements. Its main users are Designers, developers and testers.

- **Systems Requirements Specification**

3.1.4. Requirements Validation

> Reviewing the requirements model for consistency and completeness. Detecting problems in the requirements document before system development.

3.2. Requirements Management

> Identifying, controlling, and tracking requirements and their changes. Performed throughout the requirements engineering process

Not a part of the Requirements Development.

---

4. Design Patterns

> General, reusable solutions to recurring software design problems.

They are templates for solving problems and NOT finished designs. They are different from Architectural Patterns.

- **Design Pattern Template**
  - Name and Classification: Conveying the pattern's essence
  - Intent: Describing the pattern's purpose and addressed problem
  - Also Known As: Listing alternative names
  - Motivation: Illustrative scenario showcasing the problem and solution
  - Applicability: Identifying suitable situations and poor designs addressed
  - Structure: Graphical representation of classes (e.g., using OMT)
  - Participants: Classes and objects involved, along with their responsibilities
  - Collaborations: How participants work together
  - Consequences: Pattern benefits, trade-offs, and results
  - Implementation: Pitfalls, hints, language-specific considerations

4.1. Types of Design Patterns

1. **Creational Patterns** → Constructing objects of the appropriate class, especially useful with polymorphism and runtime class selection
   - Total *5* ⟶ Abstract Factory, Builder, Factory Method, Prototype, Singleton
2. **Structural Patterns** → Creating larger structures from individual parts of different classes, varying based on structure and purpose
   - Total *7* ⟶ Adapter, Bridge, Composite, Decorator, Façade, Flyweight, Proxy
3. **Behavioral Patterns** → Describing object interactions and responsibility division, focusing on communication and simplifying flow control
   - Total *11* ⟶ Chain of Responsibility, Command, Interpreter, Iterator, Mediator, Memento, Observer, State, Strategy, Template Method, Visitor

4.1.1. Singleton Design Pattern

A creational design pattern that ensures that a class has only a single instance, with a global access point.

Example: We create a class named "ClassicSingleton", we can create some methods and variables inside to enforce singleton design pattern:

```
public class ClassicSingleton {
    private static ClassicSingleton instance = null;
    private String singletonData;  // Example data field
```

```
    private ClassicSingleton() {
        // Exists only to defeat instantiation.
        singletonData = "Initial singleton data";
    }

    public static synchronized ClassicSingleton getInstance() {
        if(instance == null) {
            instance = new ClassicSingleton();
        }
        return instance;
    }

    public String getSingletonData() {
        return singletonData;
    }

    public void setSingletonData(String data) {
        this.singletonData = data;
    }
}
```

1. A `getSingletonData()` function inside this class provides a global access point.
2. A static variable `instance` holds the single instance.
3. A string `singletonData` represents the data inside the class.
4. A private constructor is created to make it impossible to create new objects of this class.
5. The `getInstance()` method uses an if statement to make it so this class only ever has a single instance and if doesn't have one then it created.

The Singleton pattern has several benefits, including controlled access to the sole instance, serving as an elegant replacement for global variables, and offering more flexibility than static member functions, as it allows overriding. It is also possible to extend the Singleton pattern to allow for multiple instances if needed.

- *Participants*: Singleton class provides the `instance()` operation for client access.
  - `instance()` is a class operation (static) and might handle instance creation
- *Collaboration*: Clients exclusively interact with the instance through instance()
- *Consequences*: Gives controlled access to the single instance and replaces the need for global variables.

---

5. Architecture Representation

5.1. UML (Unified Modeling Language)

> A graphical language for visualizing, specifying, constructing, and documenting the artifacts of a software-intensive system.

> **View Model** → A complete and simplified description of a system from a particular perspective.

There is no single view that can present all aspects of complex software to stakeholders.

5.1.1. 4+1 View Model

Introduced by Philippe Kruchten (Kruchten, 1995).

> A "multiple-view" model that addresses different aspects and concerns of the system. It standardizes the software design documents and makes the design easy to understand by all stakeholders.

This model provides four essential views plus a fifth one:

1. **Logical View** → The application domain entities necessary to implement the functional requirements.
    - The logical view specifies system decomposition into conceptual entities (such as objects) and connections between them (such as associations).
    - *Diagrams*: Static (Class, Object), Dynamic (Sequence, State, Activity).
2. **Process View** → The dynamic aspects of the system like its execution time behavior. This view maps functions, activities, and interactions onto runtime implementation.
    1. It takes care of the concurrency/synchronization issues between subsystems.
    2. *Diagrams*: Activity (UML), Interaction overview.
3. **Physical View** → Installation, Configuration and Deployment of the software.
    1. It concerns itself with how to deliver the deploy-able system and shows the mapping of software onto hardware. Used in distributed or parallel systems.
    2. *Diagrams*: Deployment (UML), other documentation.
4. **Development View** → Static organization of the system modules.
    1. Modules such as namespaces, class library, subsystem, or packages are building blocks that group classes for further development and implementation.
    2. *Diagrams*: Package, Component (UML).

5. **Scenario View** → Describes the functionality of the system, i.e., how the user employs the system and how the system provides services to the users.
    - It helps designers to discover architecture elements during the design process and to validate the architecture design afterward.
    - *Examples*: Use Case diagram (UML) and other verbal documents.

Extended View: **User Interface (UI) View** → Provides a clear user-computer interface view and hides implementation details. It may be provided as a series of screenshots or a dynamic, interactive prototype demo (like Figma).

- Any modification on this view will have direct impact on the scenarios view.

5.2. ADL (Architectural Description Language)

> A notation to support architecture-based development. It is used to define and model system architecture before detailed design and development.

Examples → Archimate, Darwin, C4 model, EAST-ADL, Acme, Wright etc.

Parts of an ADL

1. *Components* → primary building blocks of the system.
    - They are the main entities or functionalities of the system like a service, a database, or a processing unit.
2. *Connectors* → interactions between components.
    - They define how components communicate like the data flow, control flow, or protocol interactions.
3. *Interfaces* → inputs and outputs of components or connectors.
    - They define interaction points and the services a component provides or requires.
4. *Architectural Style* → Configuration, Constraints etc.

Tool Support → Tools that assist in creating, analyzing, or visualizing the architecture.

5.2.1. ACME

A simple, generic language for describing software architectures and families of architectures. Intended to be standard representation for tools.

ACME Studio → Tool for viewing and editing ACME descriptions.

---

6. Architecture Patterns

> A stylized description of good design practice, which has been tried and tested in different environments.

Examples → Layered, Pipe and Filter, Blackboard, Broker, Repository, MVC etc.

> **How to select a pattern?**
>
> Pattern should be selected based on the important quality concern that they address.

Heterogeneous Architectures → System usually apply mixture of more than one pattern. Use a single "dominant" pattern for the most important quality concern.

6.1. Layer Pattern

> Organizes the system into hierarchical layers where each layer has specific responsibilities. Each layer provides services to the layer above it and depends on the layer below it.

It enhances separation of concerns. Simplifies system maintenance and testing.

- **Structure** → Common *Layers* include Presentation, Business Logic, Data Access, and Database.
- **Example** → A web application with:
  - *Presentation Layer*: UI handling (HTML, CSS, JS).
  - *Business Logic Layer*: Core application logic (validation, processing).
  - *Data Layer*: Interacts with the database.
- **Pros** → Modular and reusable, Easy to replace or modify a layer.
- **Cons** → Performance overhead due to multiple layers.

6.2. Pipe and Filters

> Made of a series of filters (processing units) connected by pipes (data streams). Data flows through the pipeline, being processed step-by-step by the filters.

Best for systems where data processing can be divided into steps.

- **Structure** → *Filters* process and transform data and *Pipes* transfer data between filters.
- **Example** → Unix/Linux shell commands (`cat file.txt | grep "word" | sort`).
- **Pros** → Easy to add, replace, or reuse filters. Supports parallel processing.

- **Cons** → Overhead in managing intermediate data and pipes. Error handling across filters can be complex.

6.3. Blackboard Pattern

> Components collaborate by reading from and writing to a central shared repository (blackboard). Used when solutions emerge from the contributions of diverse and specialized components.

Best for problems where no deterministic solution exists upfront.

- **Structure** → It has 3 types of components:
  - *Blackboard* → Stores/Manages central data.
  - *Knowledge Sources* → Separate components with knowledge focus on solving specific parts of the overall problem and share their results by updating a common blackboard.
  - *Controller* → Monitors changes on blackboard, schedules knowledge source activations.
- **Example** → AI systems like voice recognition or weather prediction.
- **Pros** → Flexible and supports incremental solution building. Promotes modularity.
- **Cons** → Complex synchronization and control. Performance issues with a heavily accessed blackboard.

6.4. Client-Server Pattern

> Functionality of the system is organized into services, with each service delivered from a separate server. Clients are users of these services and access servers to make use of them.

Servers are often centralized, and clients are distributed. Communication occurs over a network.

- **Structure** → Divides a system into two main roles: *Client* (Requests and consumes services) and *Server* (Provides services or resources).
- **Example** → A web browser and web server:
  - *Client*: Sends an HTTP request.
  - *Server*: Responds with the requested web page.
- **Pros** → Centralized Management, Scalability, Resource Sharing
- **Cons** → Single Point of Failure, Network Dependency

6.5. Broker Pattern

> Used to mediate communication between distributed components or systems. A **broker** component acts as an intermediary to facilitate communication and

> coordination.

It decouples client and server components in a distributed system.

- **Structure** → *Client* (Requests services), *Broker* (Manages communication, requests, and responses) and *Server* (Provides services).
- **Example** → Middleware systems like CORBA or Message Brokers (RabbitMQ, Kafka).
- **Pros** → Decouples components, making the system flexible and scalable. Supports heterogeneous systems.
- **Cons** → Performance overhead due to the broker layer. Potential single point of failure.

6.6. Repository Pattern

> Provides a repository that handles storage, retrieval, and manipulation. Decouples the data source logic from the business logic.

Simplifies data access by providing a unified interface.

- **Structure** → Repository interacts with data sources, Business logic interacts with the repository instead of the data source directly.
- **Example** → A repository for managing user data in a database.
- **Pros** → Reduces code duplication and improves testability. Centralized management of data access logic.
- **Cons** → Can add an unnecessary abstraction layer if the application is simple.

6.7. Model-View-Controller (MVC)

> Separates the system into three interconnected components: **Model**, **View** and **Controller**.

Promotes separation of concerns and makes the system modular and maintainable.

- **Structure** → *Model* (Interacts with the database and updates the state), *View* (Renders the UI), *Controller* (Routes requests and processes input).
- **Example** → A full stack web application:
  - *Model*: Backend database interaction (e.g., SQL queries).
  - *View*: HTML/CSS files rendered for the user.
  - *Controller*: Handles HTTP requests and updates the model/view.
- **Pros** → Improves code organization and testability. Allows independent development of components.
- **Cons** → Overhead in smaller applications. Increased complexity due to component interactions.

7. Process Modeling

> **Process models** are graphical representations of how a business system should operate.

They can document the current ("as-is") system or the desired future ("to-be") system. They clarify requirements definitions and use cases.

- **Logical process models** describe processes without specifying how they are implemented, while **physical process models** provide the necessary information for system implementation.

7.1. Data Flow Diagrams (DFD)

> They show how data flows through a system. They illustrate the processes, data stores, external entities, and data flows, providing a high-level overview of the system's functionality.

7.1.1. Elements of DFDs

- **Process** → An activity or function performed for a specific business reason.
  - *Symbol*: Circle
- **Data Flow** → A single piece of data or a collection of information.
  - *Symbol*: Arrow
- **Data Store** → A repository for storing data.
  - *Symbol*: Open Rectangle
- **External Entity** → A person, organization, or system outside the system but interacting with it.
  - *Symbol*: Square

7.1.2. Levels of DFDs

1. **Context Diagram** → The highest-level DFD showing the entire system and its interactions with external entities. It depicts the overall business process as a single process.
2. **Level 0 DFD** → Shows all major high-level processes and their interrelationships. It includes processes, data stores, external entities, and data flows.
   - **Balancing:** Ensures that all information from a higher-level DFD is accurately represented in the next lower level DFD.
3. **Level 1 DFD** → Break down each Level 0 process into a more detailed diagram. There is one Level 1 diagram for each process on the Level 0 diagram. The parent process and children processes are numbered consistently.

4. **Level 2 DFD** → Break down Level 1 processes. They show all processes, data flows, and data stores within a single Level 1 process. It is important to ensure balance between Level 1 and Level 2 DFDs.

DFD Fragments: A portion of a DFD representing a single use case.

Alternative Data Flows: A process may produce different data flows depending on circumstances. These are represented on the DFD, with process descriptions explaining the conditions for each alternative flow.

7.1.3. Steps for Creating DFDs

1. Make the **Context diagram**.
2. Create **DFD fragments** for each *use case*.
3. Organize DFD fragments into a **Level 0** diagram.
4. Break down each Level 0 process and Create **Level 1** DFDs (and subsequent levels) based on the steps within each use case. *3-7 processes per DFD*.
5. Validate (Check) the DFDs for completeness, correctness, and consistency.
    1. *Syntax Errors*: Grammatical errors violating DFD rules.
    2. *Semantics Errors*: Misunderstandings in the analysis and representation of the system.

Key Terms

- **J2EE** → Java 2 Enterprise Edition
- **CASE** → Computer Aided Software Engineering
- **MDE** → Model Driven Engineering
- **MDA** → Model Driven Architecture

OCL → Object Constraint Language
Kermeta → A language that is used to convert design into code in any programming language.

MoSCoW Model → Must have, Should have, Could have, Wished have

Workaround → custom-built add-on program that interfaces with the packaged application

State Chart → shows the state of a process in execution.