Table of Contents

CH1: Software Design & Architecture

Executive Summary

Software engineering design is a systematic, disciplined, & quantifiable approach to creating high-quality software systems. Functioning as the foundation of the software development lifecycle (SDLC), effective design mitigates the risks of bugs, crashes, & unmaintainable code. The discipline is characterized by a dual nature, serving as both a **process** of organized activities & a **product** consisting of technical artifacts like UML diagrams.

Key takeaways from this analysis include:

- **The Foundation Analogy:** Building software is compared to constructing a house; a weak architectural foundation leads to eventual failure regardless of aesthetic appeal.
- **Problem-Solving Frameworks:** Successful design requires a holistic approach to problem-solving, moving through initial (interpretation), operational (brainstorming), & goal (implementation) states while overcoming "functional fixedness."
- **Architecture vs. Detailed Design:** Architecture focuses on the "macro" view & non-functional quality attributes (security, scalability), while detailed design focuses on "micro" internal logic & functional requirements.
- **Core Principles:** The "modularization engine" is driven by abstraction (hiding unnecessary details) & encapsulation (restricting access), aimed at achieving low coupling & high cohesion.
- **Persistent Challenges:** Designers must navigate requirements volatility, rapid technological evolution, conflicting stakeholder influences, & ethical responsibilities.

Foundations of Software Engineering

Software engineering emerged in 1968 to address the increasing complexity & failure rates of software projects in the 1960s. It adapts traditional engineering methods to the digital realm.

Defining Key Terms

- **Software:** A collection of instructions enabling users to interact with hardware to perform specific tasks (e.g., operating systems, web browsers).
- **Engineering:** A systematic, organized, & logical process used to generate & evaluate designs that meet client objectives & user needs within specified constraints.
- **Software Engineering (IEEE Definition):** The application of a systematic, disciplined, & quantifiable approach to the development, operation, & maintenance of software.

The Software Development Lifecycle (SDLC)

The process is categorized into five primary phases:

1. **Requirements:** Analyzing, specifying, & validating stakeholder needs.

2. **Design:** Using specifications to create software architecture & detailed models.
3. **Implementation:** Translating designs into executable code using programming languages.
4. **Testing:** Verifying that the software meets requirements & behaves correctly.
5. **Maintenance:** Modifying software post-delivery to correct faults or adapt to new environments.

Engineering Problem-Solving

Design is fundamentally a problem-solving activity. Engineers must identify, evaluate, & propose solutions to complex issues through a structured framework.

The Three States of Problem-Solving

State	Activities
Initial State	Problem formulation & interpretation. Underst&ing the problem is often a challenge in itself.
Operational State	Brainstorming, thinking about viable solutions, & synthesizing possibilities.
Goal State	Identifying, evaluating, & validating the final solution.

Overcoming Functional Fixedness

A significant barrier in the operational state is **functional fixedness**, a mental block where individuals limit an object's use to its traditional function. Designers are encouraged to "think outside the box" & consider problems from unusual angles to find innovative solutions.

Software Engineering Design: Process & Product

In the industry, "design" is used interchangeably to describe both the activities performed & the resulting artifacts.

- **Process Perspective:** Focuses on project management aspects, including planning, organizing, & assigning tasks to ensure resources are aligned before construction.
- **Product Perspective:** Focuses on technical aspects, resulting in artifacts such as class diagrams, sequence diagrams, & design documents.

Benefits of Studying Design

- **Management Benefits:** Minimizes the effects of requirements volatility, optimizes human resource allocation, & prevents "single-point-of-failure" where one individual owns the entire system knowledge.
- **Technical Benefits:** Maps requirements to conceptual models, identifies main components & interfaces, allows for the evaluation of quality attributes (usability, efficiency), & enables the reuse of solutions in future projects.

Major Software Design Challenges

Modern software engineers face five significant challenges that complicate the development of high-quality systems:

1. **Requirements Volatility:** Changes in requirements late in the lifecycle are costly, requiring extensive rework of design, verification, & deployment plans.
2. **Inconsistent Development Processes:** Teams within the same organization may follow different methodologies (e.g., one using Agile, another Waterfall), leading to integration delays & mismatched systems.
3. **Fast-Changing Technology:** The rapid evolution of technologies like AI & Machine Learning requires designers to learn & implement new concepts quickly while ensuring interoperability with legacy systems.
4. **Managing Design Influences:** Stakeholders often have conflicting priorities (e.g., executives wanting profit-driven features vs. security teams dem&ing protection), requiring difficult design trade-offs.
5. **Ethics & Professionalism:** Designers must consider how their decisions affect society. Guidelines like the **ACM Code of Ethics** emphasize privacy, honesty, & prioritizing the public good.

The Software Design Process

The design process is divided into major & supportive activities that bridge the gap between requirements & construction.

Architecture vs. Detailed Design

Aspect	Software Architecture	Detailed Design
Focus	Macro-design; major components & quality.	Micro-design; internal logic & behavior.
Viewpoint	**Black-box:** External properties & interactions.	**White-box:** Internal structure & algorithms.
Requirements	Primarily Non-functional (Security, Scalability).	Primarily Functional (Logic, Data h&ling).
Timing	Foundation for subsequent work.	Starts after architecture is approved.

Specialized Design Activities

- **Component Design:** Creating the internal structure (e.g., UML class diagrams) of components identified during the architecture phase.
- **Interface Design:** Specifying how components interact internally or externally with third-party systems (e.g., payment gateways).
- **Construction Design:** The lowest level of design, focusing on algorithm analysis & implementation details.

- **Human-Computer Interface (HCI) Design:** Optimizing the interaction between users & software to ensure usability & efficiency.

Core Design Principles (Fundamentals)

These principles guide the "modularization engine" to create manageable, high-quality systems.

1. **Modularization:** The process of decomposing a system into small, fine-grained components with focused responsibilities. This makes projects easier to build, test, & maintain.
2. **Abstraction:** Hiding unnecessary details to focus on essential characteristics.
    - *Procedural Abstraction:* Simplifying complex behavioral operations (e.g., a "SEND" comm& hiding TCP/IP connection steps).
    - *Data Abstraction:* Simplifying the structural composition of data objects.
3. **Encapsulation:** Protecting hidden parts of an abstraction. It ensures that entities only expose essential information through controlled interfaces (e.g., private variables accessed via public methods).
4. **Coupling:** The degree of interdependence between modules. Designers aim for **Low Coupling** to minimize dependencies.
    - *Content Coupling (Avoid):* One module modifies the internal data of another.
    - *Common Coupling (Avoid):* Multiple modules rely on global variables.
5. **Sufficiency:** Ensuring a unit provides *only* the services necessary for its intent (no more).
6. **Completeness:** Ensuring a unit provides *all* the services required to achieve its intent (no less).

Specialized Roles in Design

Effective software design involves various professional roles with distinct focuses:

- **Systems Engineer:** Designs the overall process for both hardware & software.
- **Software Architect:** Focuses on high-level structure & quality attributes (black-box approach).
- **Component Designer:** Designs & implements the internal logic of specific modules (white-box approach).
- **User Interface (UI) Designer:** Specializes in system usability & the user experience.

Glossary of Key Terms

Term	Definition
Abstraction	The principle of focusing on the essential characteristics of an entity while deferring or hiding unnecessary details.
ACM Code of Ethics	A set of general ethical principles & professional responsibilities, such as respecting privacy & prioritizing the public good, that guide software designers.
Black-Box View	A design perspective that focuses on what a component does & how it interacts with others, rather than its internal workings.
Cohesion	A measure of how strongly related & focused the responsibilities of a single software module are.
Component Design	An activity within detailed design that specifies the internal structure & behavior (e.g., class diagrams) of individual modules.
Construction Design	The lowest level of detailed design, dealing with the analysis & design of specific algorithms & logic for function implementations.
Coupling	The degree of interdependence between software modules; designers generally aim for "low" or "loose" coupling.
Encapsulation	The principle of hiding the internal implementation of an entity & restricting access to only essential information through a well-defined interface.
Engineering	A systematic, organized, & structured process used to generate & evaluate designs that meet client objectives while satisfying constraints.
Functional Abstraction	Also known as procedural abstraction; it simplifies reasoning by using a single term (like "SEND") to represent a complex sequence of behavioral steps.
HCI Design	Human-Computer Interface design; the activity of optimizing the interaction between users & software to ensure usability & efficiency.
IEEE	The Institute of Electrical & Electronics Engineers, a professional organization that sets st&ards for software engineering.
Modularization	The process of decomposing a software system into smaller, separate components (fine-grained modules) to make the system more manageable.
Software	A collection of instructions that enable a user to interact with computer hardware to perform specific tasks.
Software Architecture	The high-level "macro-design" of a system that provides the major structural components, interfaces, & foundation for all subsequent work.
Structured Design	A design strategy that breaks a system into independent, single-purpose modules using a top-down approach focused on functions.

Sufficiency	A design principle that measures whether a design unit provides only the services necessary to achieve its intent, & nothing more.
Systems Engineer	A role responsible for designing the overall development process for a system as a whole, including both hardware & software.
White-Box View	A design perspective that focuses on the internal structure, algorithms, & data h&ling of a component.

CH2: Fundamentals, Process, & Requirements Engineering

Executive Summary

Software architecture serves as the high-level structural blueprint of a system, focusing on major components, their interactions, & non-functional requirements. It is a foundational, non-optional activity that translates requirements into a graphical, design-oriented domain. Unlike detailed design, which h&les internal logic & class-level implementation, architecture provides the "big picture" necessary to meet quality goals such as performance, security, & scalability.

The success of an architectural effort depends heavily on **Requirements Engineering**, which involves eliciting & analyzing stakeholder concerns to drive design decisions. A critical aspect of this process is the identification of **quality attributes** & the implementation of specific **tactics** to achieve them. Furthermore, maintaining **design synchronicity**—the alignment between the architectural vision & the final code—is essential to prevent system failure & excessive costs during later development stages.

1. Fundamentals of Software Architecture

Definition & Dimensions

Software architecture is defined as the high-level structure of a software system, the discipline of creating those structures, & their documentation. It captures three primary dimensions:

- **Structure:** The "parts" or components of the system.
- **Communication (Behavior):** How the parts fit & talk together.
- **Nonfunctional Requirements:** The quality attributes the system must exhibit.

Process vs. Product

Software architecture is categorized as both an activity & a result:

- **The Process:** The foundational design activity that evaluates & translates functional & non-functional requirements into design elements.
- **The Product:** The resulting high-level diagrams, component models, & blueprints that support detailed design & construction.

Scope: What Architecture is Not

Architecture focuses on aspects of a system that are difficult to change once built. It is specifically **not** concerned with:

- Development/Coding
- Algorithms.
- Internal data structures.

2. Comparison: Architecture vs. Detailed Design

Architecture & design differ in their level of abstraction & interaction focus.

Feature	Software Architecture (Big Picture)	Software Design (Detailed Picture)
Focus	Major components (Frontend, Backend, Database).	Internal details (Classes, methods, logic).
Interaction	Component-to-component communication.	Object-to-object / class-to-class interaction.
Example	Deciding the system needs a Payment Service.	Designing the addItem() method in a ShoppingCart class.
Products	High-level blueprints & subsystem diagrams.	Class diagrams, sequence diagrams, & flowcharts.

3. The Software Architecture Process

The creation of software architecture is a large-scale, iterative process involving five core tasks:

1. **Underst& & Evaluate Requirements:** Engaging in requirements engineering to define the system's needs.
2. **Design the Architecture:** Identifying major components, views, styles, & patterns.
3. **Evaluate the Architecture:** A lengthy, iterative process to catch defects early, when they are cheaper to correct.
4. **Document the Architecture:** Capturing the design in a format reviewable by stakeholders.
5. **Monitor & Control Implementation:** Ensuring the system's construction aligns with the architectural vision.

Problem-Solving Framework

Architectural design is driven by problem-solving under constraints. The architect must interpret requirements & scenarios while evaluating constraints (schedule, cost, & quality) through collaborative brainstorming & design reviews.

4. Requirements Engineering for Architects

Architects must be proficient in requirements engineering, as architectural design begins with underst&ing the context & boundaries of the system.

The Four Activities of Requirements Engineering

- **Elicitation:** Identifying stakeholders & uncovering their needs through interviews, meetings, observations, & scenarios.
- **Analysis:** Addressing raw requirements that are contradictory, incomplete, or vague. This includes classification, prioritization, & negotiation.
- **Specification:** Formally capturing the results in an appropriate format (e.g., Software Requirements Specification).
- **Validation:** Reviewing requirements with stakeholders to ensure accuracy & feasibility.

Requirement Classification

Requirements are categorized to understand their nature:

- **Functional vs. Non-functional:** What the system does vs. how it behaves (quality goals).
- **Product vs. Process:** What the software must do vs. the methodology used to develop it (e.g., Agile).
- **Imposed vs. Derived:** Stakeholder-given requirements vs. those identified by the team to meet higher goals.

Desired Characteristics of Requirements

To be successful, requirements must be:

- **Specific:** Clear, concise, & exclusive (not open to interpretation).
- **Consistent:** They must not conflict with or prevent the implementation of other requirements.
- **Correct:** Accurately describing a desired system function.
- **Attainable:** Realistic within the constraints of time, technology, & budget.
- **Complete:** Thoroughly describing functions individually & as a collective set.
- **Verifiable:** Possible to measure or test to confirm the requirement is met.

5. Quality Attributes & Tactics

Architecture focuses on the qualities (performance, security, etc.) provided by major components.

Quality Tags

It is insufficient to simply identify a component; architects must "tag" components with quality properties.

- **Example:** A component providing a search algorithm should be tagged with its performance rating (e.g., $O(n^2)$) so clients understand its limitations.

Design Tactics

A **tactic** is a specific design decision made to achieve or control a quality attribute response.

Quality Attribute	Description	Example Tactics
Security	Protecting & defending information.	User authentication, limiting access on a "need-to-know" basis.
Performance	Capacity to work under time/resource constraints.	Caching, introducing concurrency, reducing overhead.
Availability	System uptime.	Redundancy, task monitors, watchdog timers.
Modifiability	Ease of changing the system for future needs.	Modularization, encapsulation, reducing coupling.

Testability	Ease of verifying & validating functions.	Event logging (allowing testers to trace operations).
Usability	Complexity involved in learning/using the system.	Supporting "undo" operations, providing a "cancel" option.

6. Key Architectural Tasks & Concepts

Architectural Views

No single diagram can describe an entire architecture. Multiple views are required to communicate with different stakeholders:

- **Logical View**
- **Development View**
- **Process View**
- **Physical (Deployment) View:** Shows servers, nodes, & network.
- **User View**

Styles & Patterns

- **Architectural Styles:** High-level templates for organizing systems (e.g., Distributed Systems).
- **Architectural Patterns:** Reusable solutions to recurring architectural problems, providing more specific guidance than styles.

Design Synchronicity

This is the measurement of how well the final implementation reflects the architectural & detailed design. High synchronicity is required for successful implementation, particularly in multi-year efforts where deviation is common. This requires a well-defined process of documentation, review, & approval for all changes.

Glossary of Key Terms

Term	Definition
Architectural Pattern	Reusable solutions to recurring architectural problems that guide detailed design decisions.
Architectural Style	A general template or approach for organizing a system's structure, providing a high-level solution shape (e.g., distributed systems).
Architectural View	A representation of a system from a particular perspective to help stakeholders evaluate specific properties like usability or reusability.
Attainable	A characteristic of requirements indicating they can be realistically achieved within constraints like time, budget, & technology.
Availability	A quality attribute referring to the system's uptime & its ability to remain operational.
Conceptual Modeling	The process of discovering system boundaries & how a system interacts with its environment to further identify requirements.
Design Synchronicity	The measurement of how closely a software implementation reflects its intended architectural & detailed design.
Elicitation	The activity of identifying stakeholders & uncovering their needs, wants, & non-functional requirements.
Functional Requirement	A requirement that describes a specific action or service the system must perform (what the system does).
Interoperability	The ability of different software or hardware systems to work together & share data seamlessly.
Modifiability	The degree of complexity involved in changing a system to fit current or future needs.
Non-functional Requirement	A global quality goal describing how well a system should perform (e.g., security, performance).
Portability	The ease with which a system can be adapted to different software or hardware environments.
Requirements Engineering	The systematic approach to requirements specification involving elicitation, analysis, specification, & validation.
Software Architecture	The high-level structures of a software system, the discipline of creating them, & the documentation of these structures.
Software Design	The detailed process of working out the internal logic of components, including classes, functions, & algorithms.
Stakeholder	Any person, group, or organization with a direct or indirect interest in a system, such as engineers, customers, or managers.
Tactic	A specific design decision made to control the response of a quality attribute (e.g., using a watchdog timer for availability).

Testability	The degree of complexity involved in verifying & validating that a system's functions meet their requirements.
Verifiable	A characteristic of a requirement meaning it can be measured or tested to confirm it has been met.

CH3: Software Architecture & Quality Attributes

Executive Summary

Software architecture serves as the critical foundation for achieving system quality. While functional requirements define specific behaviors (the "what"), non-functional requirements—also known as Quality Attributes (QA)—specify the criteria for judging the operation of the system as a whole (the "how"). Quality attributes are not achieved by architecture alone but require consistent attention throughout design, implementation, & deployment.

To address the challenges of unstable definitions & overlapping concerns among attributes, architects utilize **Quality Attribute Scenarios**. These scenarios provide a concrete, measurable framework for evaluating system performance & effectiveness. Furthermore, architects employ **Tactics**, which are primitive design techniques intended to control the system's response to specific stimuli. Systematic architectural design involves seven key categories of decisions, ranging from resource management to technology choice, all aimed at satisfying the diverse needs of stakeholders, including developers, customers, & end-users.

1. Fundamental Concepts of Architecture & Requirements

System requirements are categorized into two primary types:

- **Functional Requirements:** Define specific behaviors or functions, typically stated as "the system shall do [requirements]." These are implemented during the detailed design phase.
- **Non-functional Requirements (Quality Attributes):** Constraints on how a system implements & delivers functionality. These criteria judge the system's operation as a whole & are detailed within the system architecture.

The Role of Architecture

Architecture provides the necessary foundation for quality, but it is not a self-actualizing solution. Quality attributes must be considered through all phases of development. Architectural documents serve various stakeholders, each with differing priorities:

- **Development Teams:** Prioritize maintainability.
- **Quality Assurance Teams:** Focus on testability.
- **Customers/End-users:** Often focus on usability & availability.

2. Characterizing Quality via Scenarios

Quality attributes often have overlapping concerns. For example, a Denial-of-Service (DoS) attack affects **Availability** (system is down), **Performance** (system is slow), **Security** (breach occurs), & **Usability** (users cannot access services). To resolve this ambiguity, architects use concrete scenarios.

The Six Parts of a Quality Attribute Scenario

A st&ard scenario is defined by the following elements:

Part	Description
Source of Stimulus	The entity (human, computer system, attacker) that generates the event.
Stimulus	The specific condition or event that arrives at the system (e.g., a crash, a request).
Environment	The conditions under which the stimulus occurs (e.g., normal operation, overload, power outage).
Artifact	The specific part of the system stimulated (e.g., the UI, database, or communication channels).
Response	The activity undertaken by the system after the stimulus arrives.
Response Measure	Measurable criteria used to test the requirement (e.g., response time < 2 seconds, 99.99% uptime).

Template: "Some **(Source)** generates some events **(Stimulus)** that arrives at some **(Artifact)** under some conditions **(Environment)** & must be dealt with **(Response)** in a satisfactory way **(Response Measure)**."

3. Systematic Quality Design Decisions

Architectural design involves making systematic decisions across seven categories:

1. **Allocation of System Responsibilities:** Deciding which parts of the system h&le specific tasks (e.g., login vs. payment processing).
2. **Coordination Model:** Defining how different parts of the system communicate & work together.
3. **Data Model:** Identifying data abstractions, their operations, properties, & organization.
4. **Management of Resources:** Allocating memory, processor time, & network b&width (e.g., ensuring resources during high traffic).
5. **Mapping among Architectural Elements:** Planning how code modules correspond to runtime elements like processes.
6. **Binding Time Decisions:** Determining when architectural elements are bound (e.g., choosing a language setting at installation versus at runtime).
7. **Choice of Technology:** Selecting specific tools such as Python for logic, PostgreSQL for databases, or AWS for hosting.

4. Analysis of Major Quality Attributes

Availability

Availability is concerned with system uptime & the duration of failures.

Tactics:

- **Fault Detection:** Ping/Echo, Heartbeat, & Exception h&ling.

- **Recovery-Preparation & Repair:** Voting, Active Redundancy (duplicate systems running in parallel), & Passive Redundancy.
- **Recovery Reintroduction:** Shadow states (keeping a parallel copy of data), Resynchronization, & Rollback.
- **Prevention:** Removal from service, Transactions, & Process Monitors.

Modifiability

Modifiability focuses on the cost of change in terms of time & money.

Tactics:

- **Localize Changes:** Using semantic coherence & generalizing modules to ensure changes do not impact unrelated components.
- **Prevention of Ripple Effect:** Hiding information, maintaining existing interfaces, & using intermediaries.
- **Defer Binding Time:** Delaying decisions until runtime (e.g., using configuration files instead of hard-coding UI themes) to increase flexibility.

Performance

Performance relates to timeliness & the system's ability to respond to events within constraints.

- **Stimulus Types:** Periodic (regular), Sporadic (unpredictable), Bursty (sudden high activity), & Stochastic (r&om/probabilistic).
- **Response Measures:** Latency (delay), Deadlines, Throughput (work per second), Jitter (inconsistency), Miss Rate, & Data Loss.

Tactics:

- **Resource Dem&:** Increasing computation efficiency & managing event rates.
- **Resource Management:** Introducing concurrency & maintaining multiple copies of data.
- **Resource Arbitration:** Implementing scheduling policies to prioritize tasks.

Security

Security is the ability to resist unauthorized access while providing service to legitimate users.

Tactics:

- **Resisting Attacks:** Authenticating & authorizing users, maintaining confidentiality/integrity, & limiting access.
- **Detecting Attacks:** Utilizing intrusion detection systems.
- **Recovering from Attacks:** Restoring functionality, identifying threat sources, & maintaining audit trails.

Testability

Testability refers to the ease with which software can demonstrate faults.

Tactics:

- **Manage Input/Output:** Separating interfaces from implementation & using record/playback techniques.
- **Internal Monitoring:** Using built-in monitors to observe the system's internal state.

- **Sample Measure:** Achieving 85% path coverage within three hours of completing a code unit.

Usability

Usability measures how easily a user can learn & accomplish tasks.

Tactics:

- **Separate User Interface:** Keeping UI distinct from underlying system operations to allow design changes without altering code.
- **Support User Initiative:** Providing "Cancel," "Undo," & "Aggregate" features.
- **Support System Initiative:** Anticipating user needs via User Models (e.g., movie recommendations based on history) or Task Models.

5. Architectural Tactics Summary Table

Quality Attribute	Example Tactics
Security	Resisting Attacks, Detecting Attacks, Authenticating Users
Testability	Separate interface from implementation, Built-in monitors
Modifiability	Localize changes, Encapsulation, Hide information, Defer binding time
Performance	Concurrency, Increase available resources, Caching, Resource arbitration
Availability	Redundancy (Active/Passive), Heartbeat, Shadow State, Rollback

Glossary of Key Terms

Term	Definition
Availability	The degree to which a system is operational & accessible when required; concerned with system failure & the duration of those failures (uptime).
Binding Time	Decisions that determine when architectural elements are set or chosen (e.g., design time, compile time, or runtime).
Caching	A tactic to improve response time by storing frequently accessed data in a temporary storage area.
Functional Requirements	Requirements that define the specific behavior or functions the system must perform (the "what").
Interoperability	The system's ability to collaborate & exchange information with other software or hardware systems.
Jitter	The inconsistency in response times within a system.
Modifiability	The degree of complexity & cost (time/money) involved in changing the system to meet new requirements.
Performance	The system's capacity to accomplish work under time & resource constraints; focuses on timeliness.
Portability	The degree of complexity involved when adapting a system to different software or hardware environments.
Quality Attribute (QA)	Non-functional requirements that specify criteria for judging the operation of a system (the "how").
Redundancy	A tactic used to prevent system failure by implementing multiple instances of critical components (e.g., active or passive redundancy).
Reliability	A measure of the system's failure rate over time.
Response Measure	The measurable criteria used to judge the success of a system's response to a stimulus (e.g., response time < 2 seconds).
Security	The system's ability to resist unauthorized access while providing service to legitimate users; involves resisting, detecting, & recovering from attacks.
Stimulus	A condition or event that arrives at the system & requires a response.
Tactics	Primitive design techniques used by architects to achieve specific quality attribute responses.
Testability	The ease with which a system demonstrates its faults through verification & validation.
Usability	How easy it is for users to learn the system, accomplish tasks, & the support the system provides for those tasks.
User Neutrality	A principle ensuring the system & its documentation focus entirely on the content & data rather than the specific user.

CH4: Software Architecture Styles & Patterns

Executive Summary

Software architecture is the fundamental organization of a system, decomposed into logical components to satisfy functional & quality requirements. The transition from "scratch-built" designs to the use of established **Architectural Styles** & **Architectural Patterns** represents a shift toward professional reuse & efficiency.

Key takeaways include:

- **Definitions:** Architectural Styles define the high-level structure (the "big picture"), while Architectural Patterns provide detailed, reusable recipes for specific problems within that structure.
- **Historical Foundation:** Modern software patterns evolved from physical architecture (Christopher Alex&er) into a hierarchical classification of system-wide architectural patterns & medium-scale design patterns.
- **Classification:** Systems are categorized by their primary characteristics: Data-Centered, Data Flow, Distributed, Interactive, or Hierarchical.
- **Value of Reuse:** Implementing established patterns benefits from documented experience, highlighting solution approaches, benefits, & consequences. This leads to improved quality, consistency across projects, & increased design efficiency.

Defining Architectural Styles & Patterns

The architectural integrity of a software system depends on the strategic use of past experience. The following distinction is critical for architects:

- **Architectural Style:** A high-level template that defines the overall structure of a system. Example: Layered Style.
- **Architectural Pattern:** A specific, reusable solution to a common problem within a structure. Example: Model-View-Controller (MVC) within a layered style.

The Benefits of Pattern-Based Design

Using well-understood & tested patterns provides three primary advantages:

1. **Improved Quality:** Patterns are based on best practices refined over time, resulting in more robust & reliable software.
2. **Consistency:** Uniformity in architecture across different projects or application segments makes systems easier to maintain.
3. **Increased Efficiency:** Utilizing documented solutions speeds up the design & implementation phases.

The Evolution of Architectural Documentation

The history of software architecture is a progression from physical design principles to specific software-centric solutions.

Era	Figure/Entity	Contribution
1977	Christopher Alex&er	Created a catalogue of 253 patterns for buildings; established the language of patterns (problem + solution + context).
1990s	SE Community	Introduced **Architectural Styles** for high-level system structures (e.g., Client-Server).
1994	Gang of Four (GoF)	Published 23 **Design Patterns** focused on object-oriented solutions at the class/object level.
1996	Buschmann et al.	Unified the field by defining **Architectural Patterns** (system-wide) vs. **Design Patterns** (subsystem-level).

Classification of Architectural Systems

Architectural patterns are selected based on the system type, requirements, & desired quality attributes.

Type	Description	Common Patterns
Data-Centered	Centralized repository for data; clients access & perform work on that data.	Blackboard, Repository
Data Flow	Oriented around the transport & transformation of a data stream.	Pipes-&-Filters
Distributed	Interaction between independent processing units over a network.	Client-Server, Broker
Interactive	User-centric systems requiring heavy user-system interaction.	MVC
Hierarchical	Components structured in layers or hierarchies to reflect levels of abstraction.	Layered

Detailed Analysis: Data-Centered Systems

Data-centered systems are decomposed around a central data repository. Communication is characterized by bidirectional interaction between “worker” components & a central “data management” component. Workers do not interact directly with each other.

The Blackboard Architectural Pattern

This pattern is designed for complex, “opportunistic” problem-solving where there is no predetermined sequence of operations. It is commonly used in expert systems, speech recognition, & image recognition.

Core Components:

- **The Blackboard:** A shared knowledge base (central hub) where all current information is displayed & accessible via an API.

- **Agents:** Specialized problem solvers (independent contributors) with unique expertise. They monitor the blackboard & jump in to contribute partial solutions when their expertise is relevant.
- **Controller:** An orchestrator that manages which agent accesses the blackboard at any given time, ensuring an orderly progress toward a solution.

Workflow Example: Student Scheduling System

1. **Client** requests a schedule via a UI.
2. **Controller** (ScheduleManager) invokes the **StudentHistory Agent**.
3. **StudentHistory Agent** reads from the **Blackboard**, modifies the schedule based on degree requirements, & writes back to the Blackboard.
4. **Controller** invokes the **CourseOfferings Agent**, which reads the modified schedule, checks availability, & updates the Blackboard.
5. The process continues until the **Controller** retrieves the finalized schedule from the Blackboard & delivers it to the client.

Detailed Analysis: Data Flow Systems

Data flow systems focus on the sequential transformation of data. The most prominent pattern in this category is **Pipes-&-Filters**.

Pipes-&-Filters Pattern

This pattern involves a series of processing elements (Filters) connected by conduits (Pipes) that transmit data.

Core Components:

- **Data Source:** The producer of the data.
- **Filter:** A component that processes, enhances, or modifies data. It produces an output based on its input.
- **Pipes:** The connections that transmit data from the source to filters, or between filters.
- **Data Sink:** The final consumer of the data.

Example: The Compiler Process

- **Scanner (Filter):** Breaks source code into tokens.
- **Parser (Filter):** Organizes tokens into a parse tree (structural hierarchy).
- **Code Generator (Filter):** Translates the parse tree into machine code or assembly instructions.

Distributed & Interactive Systems

While these categories can overlap, their primary focus differs based on connectivity & machine locality.

Interactive (Local) Systems

These systems run fully on a single machine & are designed for direct user interaction.

- **Examples:** Desktop calculator, offline dictionary, offline games.
- **Logic:** All processing & data management occur locally.

Distributed (Networked) Systems

These involve interactions between independent units over a network, supporting multiple users simultaneously.

- **Client-Server Pattern:** Multiple clients request services from a central server (e.g., Web applications, Banking apps).
- **Broker Pattern:** A mediator component coordinates communication between different processes, often used in Service-Oriented Architectures (SOA) to h&le routing & message translation.

Hierarchical Systems

Hierarchical systems structure components in layers to reflect different levels of responsibility & abstraction.

- **Vertical Structure (Layered):** Each layer has a specific role (e.g., hardware interaction at the bottom, UI at the top).
- **Separation of Concerns:** Components are grouped based on responsibility, preventing high-level logic from being entangled with low-level hardware operations.
- **Example:** Operating systems & complex control systems in manufacturing or robotics.

Summary Checklist for Pattern Selection

- **Complex Problem/No Fixed Order:** Use **Blackboard**.
- **Sequential Transformation:** Use **Pipes-&-Filters**.
- **Networked Request/Response:** Use **Client-Server**.
- **Cross-Network Coordination:** Use **Broker**.
- **Abstraction Layers:** Use **Hierarchical/Layered**.

Glossary of Key Terms

Term	Definition
Architectural Style	A high-level template defining the overall structure of a system (e.g., Layered, Data Flow).
Architectural Pattern	A specific, reusable solution to a common problem within a system structure (e.g., MVC, Client-Server).
Agent	An independent, specialized problem-solver component in a Blackboard system that contributes expertise to a shared solution.
Blackboard	A central data repository or shared workspace where independent agents read, write, & update information to solve complex problems.
Broker	A pattern where a mediator component coordinates communication, routes requests, & translates messages between networked components.
Client-Server	A distributed pattern where multiple users (clients) request resources or services from a central provider (server).
Controller	The component in a Blackboard system that orchestrates agent activity & manages access to the central data store.
Data Integrity	The quality property ensuring that data is complete, accurate, & consistent throughout its lifecycle & across systems.
Data Sink	The final consumer of data in a Pipes-&-Filters architectural pattern.
Data Source	The component that produces the initial stream of data in a Data Flow system.
Design Pattern	Medium-scale solutions (e.g., Observer, Factory) that influence subsystem architecture without changing the system's overall structure.
Filter	A component in a Pipes-&-Filters system that transforms, modifies, or enhances data received from a pipe.
Gang of Four (GoF)	Authors Gamma, Helm, Johnson, & Vlissides, who published the influential 1994 catalogue of 23 object-oriented design patterns.
Hierarchical System	A system structured in layers to reflect different levels of abstraction & responsibility (e.g., Operating Systems).
Pipe	The connector in a Data Flow system that transmits a stream of data from one filter to another.

CH5: Principles & Practices of Software Detailed Design

Executive Summary

Detailed design serves as the essential bridge between high-level software architecture & the actual construction of code. It involves the decomposition & refinement of system components into fine-grained elements, including functions, data variables, & internal logical structures. The primary objective of this phase is to produce a design sufficiently complete for implementation while maintaining "synchronicity"—the alignment between architecture, design, & code. Success in detailed design is governed by the application of SOLID principles (particularly SRP, OCP, & LSP), rigorous technical reviews, & comprehensive documentation via the Software Design Document (SDD).

Overview of Detailed Design

Detailed design is the process of refining & exp&ing the preliminary software architecture to the extent that it is sufficiently complete to be implemented. This stage marks a transition from a macro-design approach to a micro-design approach.

The Design Hierarchy

The software development process follows a logical progression where detailed design occupies a central role:

1. **Requirements Phase:** Gathering & documenting system requirements (e.g., a banking app must allow balance checks).
2. **Architecture (High-Level Design):** Defining the overall structure & interactions (e.g., choosing a client-server model).
3. **Detailed Design (Refinement):** Breaking architecture into smaller components & functions (e.g., specifying password validation & session management).
4. **Construction Phase (Implementation):** Translating the blueprints into executable source code.

The Designer's Mental Model

Effective designers must operate in two directions:

- **Design to Code:** Moving from requirements & architecture toward implementation.
- **Reverse Engineering:** The ability to analyze existing code to recreate the underlying detailed design & architecture.

Key Tasks in Detailed Design

The detailed design activity is comprised of five major tasks:

1. **Understanding Architecture & Requirements:** Designers focus on the specific requirements allocated to their assigned components rather than the entire system.
2. **Creating Detailed Designs:** This involves both structural & behavioral designs, focusing on:

- Internal & External Interface Design.
- Graphical User Interface (GUI) Design.
- Internal Component Design (Structural & Behavioral).
- Data Design (Databases & data dictionaries).

3. **Evaluating Detailed Designs:** Primarily conducted through Technical Reviews.
4. **Documenting Software Design:** Capturing all design decisions in the Software Design Document (SDD).
5. **Monitoring & Controlling Implementation:** Ensuring the final code adheres to the design.

Component Design in Object-Oriented (OO) Systems

Component-level design defines the internal logical structure & behavior of the components identified during the architectural phase. In OO systems, this involves a shift from abstract UML components to concrete classes, interfaces, & types.

Structural vs. Behavioral Modeling

- **Structural Design:** Typically modeled using UML class diagrams to define internal data structures.
- **Behavioral Design:** Modeled using sequence diagrams to define how components react & interact in different scenarios.

Abstract Classes & Interfaces

Object-oriented designers must distinguish between these two mechanisms for abstraction:

Feature	Abstract Class	Interface
Methods	Can have both abstract (no body) & concrete (with body) methods.	Originally only abstract; (Java 8+ allows default/static methods).
Variables	Can have instance variables.	Only public static final (constants).
Inheritance	A class can extend only one abstract class.	A class can implement multiple interfaces.
Constructor	Can have a constructor.	Cannot have constructors.
Use Case	Used when classes share common state & behavior.	Used to define a “contract” or blueprint for behavior.
Coupling	Tighter; carries some implementation.	Looser; only defines method signatures.

The SOLID Principles of Internal Design

To manage complexity & ensure reusability, designers follow the SOLID principles. The provided context details the first three:

1. Single Responsibility Principle (SRP)

SRP states that a class should have only one reason to change, meaning it should perform only one job.

- **Core Concept:** A module should be responsible to only one user or stakeholder.
- **Benefits:** This keeps designs maintainable & flexible. If a class h&les storing data, calculating grades, & printing reports, a change in the database format forces a change in the class, potentially breaking unrelated printing logic. Separating these into StudentData, GradeCalculator, & ReportPrinter mitigates this risk.

2. Open-Closed Principle (OCP)

OCP dictates that software designs should be open to extension but closed for modification.

- **Core Concept:** New functionality should be added by writing new code (extensions) rather than changing existing, working code.
- **Mechanism:** This is achieved by identifying areas likely to vary & encapsulating them through polymorphism. For example, a GameEngine should depend on an abstract Character class rather than a concrete TerrestrialCharacter, allowing new character types to be added without modifying the engine.

3. Liskov Substitution Principle (LSP)

LSP requires that objects should be replaceable with instances of their subtypes without altering the correctness of the program.

- **The Signature Rule:** Method signatures in the subtype must match the base class to ensure type correctness.
- **The Methods Rule:** Overridden methods must follow the base class contract. Subtype methods can “weaken” pre-conditions (require less) but cannot “strengthen” them (require more).
- **Example:** A Square that inherits from Rectangle might violate LSP if the user expects to set height & width independently, as changing one in a square necessarily changes the other.

Evaluation & Documentation St&ards

Technical Reviews

A successful evaluation involves a formal review process with the following criteria:

- **Timing:** Provide review notices early to allow thorough preparation.
- **Team Composition:** Include technical experts, stakeholders, & members of the Quality Assurance/testing teams.
- **Focus:** Center the review on how the design meets functional & non-functional requirements.
- **Record-Keeping:** Document the process & assign action items for any identified issues.

The Software Design Document (SDD)

The SDD is the primary reference for programmers, testers, & maintainers. Its st&ard structure includes:

- **Introduction:** Date of issue, scope, authorship, & summary.
- **Software Architecture:** Overview, stakeholders, & various architectural viewpoints.
- **Detailed Design:** Component-specific design viewpoints & detailed internal views.
- **Glossary & References:** Definitions of terms & lists of applicable documents or technologies used.

Management & Implementation

A critical aspect of detailed design is **Synchronicity**, which measures how well the detailed design adheres to the architecture & how well the code adheres to the design. Low synchronicity is a significant indicator of potential project failure. Maintaining high synchronicity is especially vital during the maintenance phase or when new engineers join a project.

Fundamental Rules of Design

- **What before How:** Define functionalities at every level of abstraction before deciding on the structures to implement them.
- **Simple is Better:** Fancy designs are often buggier & harder to verify. Complex problems should be broken down into simpler, individual problems.
- **Design for Extensibility:** A good design is "open-ended" & avoids restricting irrelevant details or introducing immaterial elements.
- **External before Internal:** Consider the solution as a "black-box" to decide how it interacts with its environment before organizing its internal components.

Study Guide

This study guide provides a comprehensive overview of the principles, tasks, & methodologies associated with the detailed design phase of software development. It synthesizes information regarding the transition from high-level architecture to implementable code, the application of object-oriented design principles, & the documentation necessary for successful software construction.

Part 1: Comprehensive Review

Overview of Detailed Design

Detailed design is the process of refining & exp&ing the preliminary software architecture of a system or component until it is sufficiently complete for implementation. It serves as a critical bridge between high-level design (architecture) & the actual source code.

- **Placement in the Lifecycle:** It follows the requirements phase (defining what the system must do) & the architecture phase (defining overall structure). It precedes the construction phase (coding).
- **Focus:** Designers delve deep into each component to define internal structures & behavioral capabilities.

- **Macro to Micro:** Design efforts shift from a macro-level approach to a micro-level approach, decomposing system components into fine-grained elements, functions, & data variables.

Key Tasks in Detailed Design

The detailed design activity involves five major tasks:

1. **Understanding Architecture & Requirements:** Unlike architecture, which looks at the complete set of requirements, detailed design focuses on requirements allocated to specific components.
2. **Creating Detailed Designs:** This involves both structural & behavioral designs, including interface design (internal/external), GUI design, internal component design, & data design (databases & data dictionaries).
3. **Evaluating Detailed Designs:** Primarily conducted through Technical Reviews involving experts, stakeholders, & quality assurance teams.
4. **Documenting Software Design:** Capturing details in the Software Design Document (SDD).
5. **Monitoring & Controlling Implementation:** Ensuring the final code adheres to the design.

Documentation & Evaluation

The **Software Design Document (SDD)** is the primary artifact, used by programmers, testers, & maintainers. Its standard sections typically include:

- **Scope:** High-level overview & objectives.
- **References:** List of applicable documents & technologies.
- **Body:** The main section where the design is documented for construction.
- **Glossary:** Definitions of terms & acronyms.
- **Interface Control Document (ICD):** Describes system interfaces & component communication.
- **Version Control Document:** Contains release information, including files, scripts, & executables.

Technical Reviews are the preferred evaluation technique. They require early notice for participants, the inclusion of technical experts & stakeholders, & a focus on how the design meets functional & non-functional requirements.

Implementation Management & Synchronicity

Detailed Design Synchronicity refers to the degree to which:

- Detailed designs adhere to the software architecture.
- Software code adheres to the detailed design. Low synchronicity is a flaw that can lead to project failure, particularly during the maintenance phase or when new engineers join a project.

Component-Level Design

Component design defines the internal logical structure & behavior of the components identified during the architecture phase. In Object-Oriented (OO) systems, this is modeled using UML class diagrams for structure & sequence diagrams for behavior.

Object-Oriented Foundations

To succeed in component design, designers must be proficient in:

- **Classes & Objects:** The basic building blocks of OO design.
- **Abstract Classes:** Classes that cannot be instantiated & contain at least one abstract method (no body). They serve as blueprints for concrete subclasses & promote code reusability.
- **Interfaces:** Blueprints that contain only abstract methods (though Java 8+ allows default/static methods with bodies). They allow for loose coupling & multiple inheritance of type.
- **Inheritance & Polymorphism:** Mechanisms that allow for generalization & dynamic method resolution at runtime.

SOLID Principles (Initial Concepts)

Three core SOLID principles are highlighted for internal component design:

1. **Single Responsibility Principle (SRP):** A class should have only one reason to change, meaning it should have only one job or responsibility. This makes the system more maintainable & flexible.
2. **Open-Closed Principle (OCP):** Software designs should be open for extension but closed for modification. Enhancements should be made by adding new code (often through polymorphism) rather than changing existing, working code.
3. **Liskov Substitution Principle (LSP):** Subtypes must be substitutable for their base types without altering the correctness of the program. This requires maintaining method signatures & honoring the behavioral contract of the supertype.

Glossary of Key Terms

Term	Definition
Abstract Class	A class declared with the abstract keyword that cannot be instantiated & typically contains methods without bodies to be implemented by subclasses.
Component Design	The detailed design task of defining the internal logical structure (classes, types) & behavior (algorithms, communication) of software components.
Detailed Design	The process of refining preliminary software architecture into a complete specification ready for implementation.
Interface	A blueprint for a class that contains only abstract methods (contract), used to achieve abstraction & loose coupling.
Liskov Substitution Principle (LSP)	A SOLID principle stating that objects of a superclass should be replaceable with objects of its subclasses without breaking the application.
Open-Closed Principle (OCP)	A design principle asserting that software entities should be open for extension but closed for modification.
Polymorphism	The ability of different classes to be treated as instances of the same general class through inheritance, allowing for dynamic method resolution.
Reverse Engineering	The process of analyzing existing code to determine the system's structure & behavioral design.
Software Design Document (SDD)	The primary documentation describing a software design, used by developers, testers, & maintainers throughout the lifecycle.
Single Responsibility Principle (SRP)	A principle stating that a class should have one, & only one, reason to change.
Synchronicity	The degree of alignment between architecture & detailed design, & between detailed design & the actual code.
Technical Review	An evaluation technique where a design is scrutinized by a team of experts & stakeholders to ensure it meets requirements.

CH6: Detailed Analysis of Creational Design Patterns & the Abstract Factory

Executive Summary

Software design patterns are recurring, reusable solutions to common object-oriented design problems that provide templates for efficient & consistent development. Among these, **Creational Design Patterns** focus on abstracting & controlling the object creation process by specifying common creational interfaces.

The **Abstract Factory Pattern** is a primary example of an object-creational design pattern. Its central purpose is to provide an interface for creating families of related or dependent objects without specifying their concrete classes. By decoupling the client code from concrete implementations, the pattern promotes consistency within product families, facilitates the addition of new product groups through extension (obeying the Open/Closed Principle), & enhances system modifiability. While it significantly improves flexibility & encapsulation, it requires a larger number of classes, which can increase initial design complexity.

Foundations of Design Patterns

Design patterns are distinct from architectural patterns & styles; they are reusable solutions tailored to specific object-oriented contexts to help developers avoid "reinventing the wheel."

Classification Criteria

The source material classifies design patterns based on two primary criteria: **Purpose** & **Scope**.

Criterion	Type	Description
Purpose	Creational	Deals with the process of object creation.
	Structural	Deals with the composition of classes or objects to form larger structures.
	Behavioral	Deals with how classes & objects interact & distribute responsibility.
Scope	Class-level	Primarily applies to classes & their relationships during design time.
	Object-level	Primarily applies to objects & their relationships during run-time.

Documenting Patterns

Comprehensive documentation is essential for understanding a pattern's utility. Key documentation categories include:

- **Intent:** The core purpose of the pattern.
- **Motivation:** The scenario or problem the pattern addresses.
- **Applicability:** When the pattern should be used.
- **Structure:** A visual representation (e.g., UML class diagram).
- **Participants:** The classes & objects involved & their specific responsibilities.

- **Collaborations:** How participants work together.
- **Consequences:** The positive & negative effects on the software solution.
- **Implementation:** Techniques & information for successful deployment.
- **Known Uses:** Real-world examples of the pattern in use.

The Abstract Factory Pattern

The Abstract Factory is characterized as a "one-stop shop" for creating families of related objects while hiding the details of their concrete classes.

Core Intent & Structure

The pattern is intended to manage & encapsulate the creation of a set of objects that conceptually belong together as a specific "family." Like all creational patterns, it is composed of two primary types of classes:

1. **Creator Classes:** These define the interfaces & concrete implementations for the factories.
2. **Product Classes:** These define the interfaces & concrete implementations for the objects being created.

Key Benefits

- **Encapsulation:** The client is aware of abstract product names (e.g., "Button") but remains ignorant of concrete class names (e.g., "DarkButton").
- **Flexibility:** New families can be added by creating a new factory without modifying existing client code.
- **Consistency:** It ensures that products from the same family are used together.
- **Open/Closed Principle (OCP):** Additions are made through extension rather than modification of existing code.

Implementation Methodology

The source outlines a logical seven-step process for designing a system using the Abstract Factory pattern:

1. **Design Product Interfaces:** Define the abstract interfaces for each product type (e.g., CPU, Monitor).
2. **Identify Families:** Determine the different groups required (e.g., St&ard vs. Advanced computers).
3. **Design Concrete Products:** For each group, create products that realize the interfaces from Step 1.
4. **Create Factory Interface:** Design an interface containing methods for creating each product interface.
5. **Create Concrete Factories:** Implement a factory for each family identified in Step 2.

6. **Associate Factories & Products:** Link each concrete factory to its specific set of concrete products.
7. **Create the Client:** Develop the client code that interacts only with the product & factory interfaces.

Illustrative Case Studies

1. User Interface (UI) Themes

A program requiring "Dark" & "Light" themes uses an AbstractFactory (e.g., GUIFactory) to create buttons & text fields.

- **Products:** Button & TextField.
- **Concrete Implementations:** DarkButton/DarkTextField & LightButton/LightTextField.
- **Factories:** DarkThemeFactory & LightThemeFactory.
- **Outcome:** The client asks the factory for a Button & does not care if it receives a DarkButton or LightButton. Adding a "Blue Theme" only requires adding a BlueThemeFactory & corresponding products.

2. Computer Store System

A store sells "Advanced" & "St&ard" computers composed of different grades of CPUs, Monitors, & Keyboards.

- **Interface:** ComputerPartsFactory defines methods like createCPU(), createMonitor(), & createKeyboard().
- **Families:** AdvancedComputerPartsFactory produces AdvancedCPU, etc., while St&ardComputerPartsFactory produces St&ardCPU, etc.
- **Data Integration:** Each product implementation can contain specific logic for retrieving data (reviews, comments) from unique remote sources (e.g., Database A for advanced products vs. Database B for st&ard).

3. Automotive Branch Management (RiyadhHQ)

A central client (RiyadhHQ) requests cars from different city branches without needing to know which city produces them.

- **Abstract Factory:** CarFactory with methods makeMercedes() & makeBMW().
- **Concrete Factories:** JeddahCarFactory & DammamCarFactory.
- **Decoupling:** RiyadhHQ only works with the CarFactory interface, allowing the system to switch between city branches at runtime without changing the client's code.

Evaluative Analysis

Aspect	Details
Pros	**Isolation:** Concrete product classes are isolated, making them easier to reuse. **Consistency:** Promotes the use of compatible products. **Modifiability:** Enhances the ability to update the system. **OCP Compliance:** New families require no modification to existing code.
Cons	**Complexity:** A large number of classes are required to implement the pattern. **Rigidity in Product Types:** Adding a completely new *type* of product (e.g., adding a "Mouse" to an existing computer factory) requires updating the abstract factory interface & all concrete factory implementations.

Glossary of Key Terms

Term	Definition
Abstract Factory	An object-creational design pattern that provides an interface for creating families of related objects without specifying concrete classes.
Behavioral Patterns	Patterns that deal with class interaction, variation of behavior, & the assignment of responsibility between objects.
Class-level Patterns	Design patterns that primarily apply to relationships between classes during design time.
Creational Patterns	Patterns that abstract & control the way objects are created by specifying common creational interfaces.
Design Pattern	A recurring & reusable solution to a common object-oriented design problem in a particular context.
Encapsulation	In the context of Abstract Factory, the practice of hiding concrete product class names from the client, who only knows abstract product names.
Factory Interface	An interface (e.g., GUIFactory) that specifies the methods for creating various abstract products.
Object-level Patterns	Design patterns that primarily focus on the relationships & interactions between objects at runtime.
Open-Closed Principle (OCP)	A design principle stating that software entities should be open for extension but closed for modification; supported by Abstract Factory through the addition of new factories.
Product Classes	The concrete implementations of objects (e.g., DarkButton, St&ardCPU) created by the factories.
Structural Patterns	Patterns that deal with the creation of structures to form existing objects.

CH7.1: Structural Design Patterns: Adapter, Composite, & Facade

Executive Summary

Structural design patterns are essential architectural tools that facilitate the design of class & object structures at runtime. This briefing document examines three primary structural patterns: the **Adapter**, the **Composite**, & the **Facade**.

The Adapter pattern enables interoperability between classes with incompatible interfaces by acting as a translator. The Composite pattern allows for the creation of tree-like hierarchies, enabling clients to treat individual objects & collections of objects uniformly. Finally, the Facade pattern provides a simplified, unified interface to a complex subsystem, reducing coupling & complexity for the client. Collectively, these patterns prioritize software reusability, flexibility, & the reduction of client-side complexity through strategic abstraction.

The Adapter Design Pattern

The Adapter pattern is a structural design pattern used to adapt an existing interface into one that a client expects. It serves as a bridge between two incompatible interfaces, allowing classes to work together that otherwise could not.

Intent & Mechanism

The primary intent is to convert the interface of a class into another interface clients expect. The operational flow involves three steps:

1. **Request:** The client makes a request to the adapter by calling a method on it using the target interface.
2. **Translation:** The adapter translates that request into a call that the "Adaptee" (the legacy or incompatible class) understands using the adaptee's interface.
3. **Result:** The client receives the results of the call seamlessly.

Structural Variations

There are two primary implementations of this pattern:

Feature	Object Adapter	Class Adapter
Mechanism	Uses **Composition**: The adapter holds an instance of the Adaptee.	Uses **Inheritance**: The adapter class inherits from the Adaptee class.
Binding Time	Runtime delegation (adaptee.specificOperation()).	Compile-time implementation (specificOperation()).
Language Support	Broadly supported.	Requires multiple inheritance, which is not supported by many languages (e.g., Java).

Preference	Generally preferred due to fewer inheritance-related issues.	Less common due to language limitations.

Case Study: Bird to ToyDuck Adaptation

In a scenario where a system expects ToyDuck objects (Target) but only Bird objects (Adaptee) are available, an adapter is used:

- **Target Interface (ToyDuck):** Expects a squeak() method.
- **Adaptee (Bird/Sparrow):** Contains fly() & makeSound() methods.
- **Adapter (BirdAdapter):** Implements ToyDuck & wraps a Bird instance. When the client calls squeak(), the adapter internally triggers bird.makeSound().

Evaluation

- **Advantages:** Achieves high reusability & flexibility. It allows clients to use polymorphism to swap between different adapter implementations without complicating client code.
- **Disadvantages:** Introduces a slight increase in overhead due to request forwarding. In complex systems, "adapter chains" may be required to reach the desired type, increasing architectural depth.

The Composite Design Pattern

The Composite pattern is an object structural pattern used to represent whole-part hierarchies through tree-like structures. It is uniquely designed to allow clients to treat individual "leaf" objects & "composite" groups of objects uniformly.

Core Components

1. **Component:** The base interface or class that defines common operations for both simple & complex objects.
2. **Leaf:** A primitive object that represents a node with no children. It implements the component's operations directly.
3. **Composite:** A complex object that stores a collection of child components (leaves or other composites). It implements component operations by delegating to its children.

Implementation Requirements

To implement a Composite structure, a system must:

- Identify the tree-like structure required.
- Design a **Component base class** with overridable methods.
- Design a **Composite class** with an internal data structure (e.g., an ArrayList) to store nodes & methods to add or remove children.
- Design **Leaf classes** that override component methods to perform specific tasks.

Real-Life Applications

Example	Primitive (Leaf)	Composite
File System	File (cannot contain other objects).	Folder (contains files & other folders).
University	Single Course (e.g., Math 101).	Program (e.g., Computer Science degree).

Benefits

- **Uniformity:** Minimizes client complexity by shielding it from the operational differences between primitive & composite objects.
- **Scalability:** Makes it easy to add new component types without altering the existing client code.
- **Structure:** Provides a clear design for systems that naturally follow a whole-part relationship.

The Facade Design Pattern

The Facade pattern provides a simplified, higher-level interface to a set of interfaces in a complex subsystem. It acts as a single point of entry, making the subsystem easier to use.

Intent & Implementation

The Facade defines a unified interface that hides the "noise" & complexity of a subsystem.

Procedure for Application:

1. **Identification:** Determine all components & operations involved in a subsystem task.
2. **Design:** Create a Facade class (e.g., a ShopKeeper) that includes an interface method for the client.
3. **Encapsulation:** Implement the Facade's method by calling the necessary operations on the internal subsystem components (e.g., Iphone, Samsung, Blackberry).
4. **Access:** Clients interact only with the Facade type to execute subsystem operations.

Case Study: Mobile Shop Subsystem

A client wanting to interact with a mobile phone subsystem (consisting of various br&s & price models) uses a ShopKeeper Facade:

- Instead of the client dealing with Iphone.modelNo(), Iphone.price(), & similar calls for every br&, the client simply calls sk.iphoneSale().
- The ShopKeeper h&les the instantiation & specific method calls of the underlying mobile br& classes.

Evaluation of Benefits

- **Reduced Complexity:** Shields clients from the internal workings of complex subsystems.
- **Stability:** The Facade provides a stable interface; even if the internal subsystem changes, the client code remains unaffected.

- **Weak Coupling:** Promotes loose coupling, as clients depend on a single interface rather than multiple, interconnected subsystem components.

Conclusion

The structural patterns detailed in this document offer distinct solutions for managing object relationships. The **Adapter** resolves interface incompatibilities, the **Composite** manages hierarchical part-whole relationships, & the **Facade** simplifies access to complex systems. Implementing these patterns leads to more maintainable, decoupled, & reusable software architectures.

Glossary of Key Terms

Term	Definition
Adaptee	An existing class with an interface that needs adapting to be compatible with a client.
Adapter Pattern	A structural pattern that converts the interface of a class into another interface clients expect.
Class Adapter	A version of the Adapter pattern that uses inheritance to adapt the interface (requires multiple inheritance).
Component	In the Composite pattern, the base class or interface that defines operations common to simple & complex objects.
Composite Pattern	A pattern that composes objects into tree structures to represent whole-part hierarchies.
Facade Pattern	A pattern that provides a simplified, unified interface to a set of interfaces in a subsystem.
Leaf	A primitive object in the Composite pattern that does not have any children.
Object Adapter	A version of the Adapter pattern that uses composition & delegation at run-time to adapt an interface.
Structural Design Patterns	Patterns that deal with the design & composition of existing classes or objects at run-time.
Target Interface	The specific interface that a client requires & depends upon.
Weak Coupling	A design goal where a client depends on a single stable interface (like a Facade) rather than many complex internal components.
Whole-Part Relationship	A hierarchy where a “whole” object is composed of various “parts,” common in tree-like structures.

CH7.2: Behavioral Design Patterns: Iterator and Observer

Executive Summary

Behavioral design patterns are specialized architectural solutions responsible for the efficient and safe distribution of behaviors among program objects. This document focuses on two primary patterns: **Iterator** and **Observer**. The Iterator pattern provides a standardized mechanism for accessing and traversing elements within a collection without exposing its underlying structure. The Observer pattern establishes a one-to-many dependency, ensuring that when a central object changes state, all registered dependents are notified and updated automatically. Together, these patterns enhance system flexibility, decouple client code from internal data structures, and facilitate the compartmentalization of services for easier maintenance.

1. Overview of Behavioral Design Patterns

Behavioral patterns focus on the communication between objects and how they carry out their responsibilities. While the source context lists numerous behavioral patterns, the primary focus is on those that manage data traversal and state-dependent notifications.

Identified Behavioral Patterns

- **Primary Patterns:** Iterator, Observer.
- **Additional Patterns (not covered in detail):** Command, Chain of Responsibility, Interpreter, Mediator, Memento, State, Strategy, Template Method, and Visitor.

2. The Iterator Design Pattern

The Iterator design pattern is an object behavioral pattern that provides a standardized way to access and traverse objects in a collection data structure sequentially.

Core Intent and Functionality

The primary intent of the Iterator is to allow access to the elements of a collection object without the client needing to know the underlying representation (e.g., whether it is an array, a list, or another structure).

Structural Components

The pattern relies on a specific interaction between the client and the data structure interfaces:

- **Aggregate Interface:** Defines a method, such as `createIterator()`, to produce an iterator object.
- **Iterator Interface:** Provides the methods required for traversal, typically `next()` and `hasNext()`.
- **Concrete Classes:** These implement the respective interfaces. A concrete iterator class accesses the concrete aggregate class to perform the traversal.

Step-by-Step Design Process

1. **Interface Design:** Identify and design the primary Iterator interface.
2. **Concrete Iterator Creation:** For every collection data structure, design a concrete Iterator and implement its methods in terms of that specific structure.
3. **Collection Interface:** Create a Collection (or Aggregate) interface that includes a method to instantiate Iterators.
4. **Implementation:** Implement the Aggregate interface in collection classes to return the appropriate concrete Iterator.

Technical Benefits

Benefit	Description
Consistency	Provides a uniform way for clients to iterate through different types of collections.
Abstraction	Hides the internal complexity of collection objects; changes to the collection do not force changes in the client.
Extensibility	Allows for the creation of multiple iterators to support different traversal types (e.g., standard vs. reverse) from the same structure.

3. The Observer Design Pattern

The Observer design pattern standardizes operations between objects that interoperate via a one-to-many relationship.

Core Intent and Functionality

The pattern defines a dependency such that when one object (the **Subject**) changes its state, all its dependents (**Observers**) are notified and updated automatically.

Structural Components

- **Subject:** This class manages the state and maintains a list of observers. It provides methods to `attach()` or `detach()` observers and a `notifyAllObservers()` method to trigger updates.
- **Observer Interface:** An abstract interface or class that defines the `update()` method.
- **Concrete Observers:** These classes extend or implement the Observer interface and define specific behaviors that occur when the Subject's state changes.

Implementation Logic

In a typical implementation, such as a state conversion system, the Subject holds an integer state. When `setState()` is called, the Subject immediately executes `notifyAllObservers()`. This method iterates through the list of registered Observers and calls their respective `update()` methods. The Observers then pull the new state from the Subject and perform their specific logic (e.g., converting the state to Binary, Octal, or Hexadecimal strings).

Design and Application Steps

1. **Subject Interface:** Design the interface with code for attaching, detaching, and notifying (often using linked-list structures).
2. **Inheritance:** Classes managing "information of interest" must inherit from the Subject class.
3. **Observer Interface:** Design an interface featuring an abstract `update()` method.

4. **Concrete Observers:** Implement the `update()` method for all specific observer types in the system.
5. **Runtime Attachment:** At runtime, create observer instances and attach them to the subject.

Technical Benefits

- **System Flexibility:** Facilitates the addition of new services without modifying the core Subject.
- **Maintenance:** Compartmentalizes specific services, making it easier to modify or maintain existing system components.

4. Applied Scenarios and Exercises

The source context provides practical scenarios to illustrate the application of these patterns.

Iterator Scenario: Company Directory

A corporate directory stores employee records using different internal structures (e.g., an array and a list). The Iterator pattern is used to allow a client to traverse these different structures in a uniform way, potentially supporting both standard and reverse traversal methods.

Observer Scenario: Social Media Follower System

In this model, the **Influencer** acts as the Subject, and the **Followers** act as the Observers.

- **The Influencer:** Maintains a list of followers and uses `notifyFollowers()` to broadcast status updates.
- **The Followers:** Mobile app users and web browser users who implement a `Follower` interface.
- **The Outcome:** When the Influencer posts a new update, all followers automatically receive a notification (e.g., “Mobile User received notification: [Post Content]”) without the Influencer manually messaging each one.

Glossary of Key Terms

Term	Definition
Aggregate	An interface that defines a method for creating an Iterator object to traverse its elements.
Behavioral Design Pattern	A category of design patterns focused on the efficient distribution of responsibilities and behaviors among objects.
Concrete Iterator	A specific implementation of the Iterator interface that tracks the current position in a specific collection.
hasNext()	An Iterator method used to check if there are more elements remaining in the collection during traversal.
Iterator	A pattern and interface that provides a standardized way to access elements of a collection sequentially.
next()	An Iterator method used to retrieve the next element in a collection.
Observer	An interface defining the `update()` method used by a Subject to notify an object of state changes.
One-to-Many Dependency	A relationship where a single object (the Subject) is watched by multiple dependent objects (Observers).
Subject	The object in an Observer pattern that maintains a list of observers and notifies them of any state changes.
Update()	The method defined in the Observer interface that is called by the Subject to synchronize the observer's state.

CH8: Software Architecture Evaluation

Executive Summary

Software architecture serves as the critical foundation of any software system, defining its structure, component relationships, & external behaviors. Evaluating this architecture before the construction phase is essential to ensuring a high-quality product & minimizing the costs associated with late-stage design corrections. Evaluation is primarily driven by functional requirements, non-functional requirements, & business constraints.

The Architecture Tradeoff Analysis Method (ATAM) is the most prominent methodology for evaluating architectural decisions. Its primary objective is to assess how architectural choices impact quality attribute requirements & business goals, specifically identifying risks where different quality attributes (such as performance versus security) conflict. By utilizing scenarios & utility trees, ATAM provides a structured approach to prioritizing goals, uncovering cross-project reuse opportunities, & improving documentation. For smaller or less complex projects, a "Lightweight ATAM" offers a more efficient, albeit less rigorous, alternative for internal organizational reviews.

Fundamentals of Architecture Evaluation

Architecture design translates business goals & requirements into a structural blueprint. Because this phase is foundational, early validation is a critical success factor for any software project.

Benefits of Systematic Evaluation

- **Conflict Resolution:** Facilitates the prioritization of conflicting architectural goals.
- **Clarity & Documentation:** Forces a clear explanation of the architecture, leading to higher-quality documentation.
- **Knowledge Reuse:** Uncovers opportunities for reusing architectural patterns across different projects.
- **Process Improvement:** Results in more robust architectural practices over time.

Major Evaluation Methods

While several methods exist, they share a common reliance on scenarios & quality attributes. The three primary methods are:

1. **ATAM (Architecture Tradeoff Analysis Method):** Focuses on quality attributes & business goal trade-offs.
2. **SAAM (Software Architecture Analysis Method):** Used for analyzing architecture through scenarios.
3. **ARID (Active Reviews for Intermediate Designs):** Focused on evaluating design during the intermediate stages.

The Architecture Tradeoff Analysis Method (ATAM)

ATAM is designed to discover risks where architectural decisions affect quality attributes of interest. It recognizes that in complex systems, quality attributes cannot be achieved in isolation; improvements in one area (e.g., security) may negatively impact another (e.g., performance).

Quality Attribute Perspectives

Quality attributes are categorized into three distinct perspectives to ensure comprehensive evaluation:

Perspective	Key Attributes
End-User	Performance, availability, usability, security
Technical	Modifiability, portability, reusability, testability, interoperability
Business Community	Time to market, cost/benefits, projected lifetime, budget

Participant Groups & Roles

A successful ATAM execution requires cooperation between three specific groups:

- **Evaluation Team:** Three to five experienced architects who lead the process.
- **Project Decision Makers:** Individuals with authority to implement changes (e.g., project managers, customers).
- **Architecture Stakeholders:** Individuals whose work depends on the architecture (e.g., developers, testers, users, maintainers).

Specific Team Roles:

- **Team Leader:** Coordinates with the client, establishes contracts, & ensures report delivery.
- **Evaluation Leader:** Runs the evaluation & facilitates scenario elicitation & prioritization.
- **Scenario Scribe:** Captures the exact wording of scenarios on visible media (whiteboards).
- **Proceedings Scribe:** Records the raw scenarios, motivations, & resolutions in electronic form.
- **Questioner:** Raises architectural issues based on expertise in specific quality attributes.

The ATAM Process: Steps & Phases

The ATAM process is structured into four phases spanning nine logical steps.

Execution Phases

Phase	Activity	Participants	Typical Duration
0	Partnership/Preparation: Logistics, planning, & team formation.	Evaluation leadership & key decision-makers.	Informal; several weeks.
1	Evaluation: Steps 1–6.	Evaluation team & decision-makers.	1–2 days; 2–3 week hiatus.
2	Evaluation: Steps 7–9.	Team, decision-makers, & stakeholders.	2 days.
3	Follow-up: Report generation & delivery.	Evaluation team & client.	1 week.

The 9-Step Methodology

1. **Present the ATAM Method:** The evaluation team explains the process, techniques, & expected outputs (risks, sensitivity points, & tradeoffs).
2. **Present Business Drivers:** Decision-makers present business goals, major stakeholders, high-level functional requirements, & constraints (technical, economic, political).
3. **Present Architecture:** The lead architect describes the approach using models like the **4+1 Architectural View Model** (Contextual, Logical, Process, & Deployment views).
4. **Identify Architectural Approaches:** The team identifies patterns used (e.g., client-server, multi-layer, pipes-filters).
5. **Generate Quality Attribute Utility Tree:** A top-down approach to prioritize quality goals. The tree's leaves consist of quality scenarios evaluated by importance & difficulty.
6. **Analyze Architectural Approaches:** The team examines high-ranked scenarios to see how the architecture supports them, documenting rationale, risks, & non-risks.
7. **Brainstorm & Prioritize Scenarios:** Stakeholders brainstorm operationally meaningful scenarios. If these diverge significantly from the architect's utility tree, it identifies a major project risk.
8. **Analyze Architectural Approaches (Refined):** The architect demonstrates how the architecture h&les the high-ranked scenarios from the brainstorming session.
9. **Present Results:** The team delivers a report recapping the process, prioritized scenarios, utility tree, & all discovered risks & tradeoffs.

Analysis Artifacts: Risks, Sensitivities, & Tradeoffs

A core output of the ATAM is the identification of critical decision points:

- **Sensitivity Points:** Architectural decisions on which a particular quality attribute is highly dependent.
- **Tradeoffs:** Decisions that affect multiple quality attributes (often improving one while degrading another).
- **Risks & Non-risks:** Decisions that are either problematic or well-grounded in light of the requirements.

Lightweight ATAM

For smaller or less risky projects, a "Lightweight" version of ATAM is available. This version is often conducted by internal staff & can be completed in a single day or half-day.

Key Differences:

- **Efficiency:** Steps like scenario brainstorming (Step 7) & redundant analysis (Step 8) are often omitted or merged into the utility tree generation.

- **Internal Focus:** Participants are usually internal to the organization, which speeds up consensus but may reduce objective depth.
- **Condensed Timeline:** Presentation of business drivers & architecture is brief (15–30 minutes) as participants are already familiar with the project.
- **Main Focus:** The bulk of the time (2–3 hours) is dedicated to Step 6: mapping highly ranked scenarios onto the architecture.

Required Evaluation Outputs

At minimum, an ATAM evaluation must produce:

- **Software Architecture Document (SAD):** The primary artifact specifying the selected architecture.
- **Evaluation Report:** A comprehensive document that recaps the process, captures scenario analysis, explains c&idate architectures, & provides the rationale for the final architectural decisions.

Glossary of Key Terms

Term	Definition
ARID	Active Reviews for Intermediate Designs; an evaluation method using scenarios & quality attributes.
ATAM	Architecture Tradeoff Analysis Method; a specialized method to assess architectural decisions against quality attributes & business goals.
Business Drivers	The goals, context, constraints, & high-level requirements that motivate the creation of the software architecture.
Non-risk	An architectural decision that is deemed positive & supportive of the required quality attributes.
Quality Attribute	A property of a system (e.g., performance, security, or modifiability) used to judge its overall quality & effectiveness.
Quality Scenario	A specific description of a system's behavior used to characterize, prioritize, & evaluate quality attributes.
Risk	A potential problem identified during evaluation where an architectural decision may negatively affect a desired quality attribute.
SAAM	Software Architecture Analysis Method; an evaluation method sharing common scenario-based techniques with ATAM.
SAD	Software Architecture Document; the key artifact used to specify & document the selected system architecture.
Sensitivity Point	An architectural decision to which a particular quality attribute is highly sensitive.
Trade-off	A situation where achieving one quality attribute (e.g., high security) negatively impacts another (e.g., performance).
Utility Tree	A top-down hierarchical model used to identify & prioritize quality attribute goals & their associated scenarios.

CH9: Software Architecture Documentation

Executive Summary

Software architecture serves as the primary determinant of a system's success or failure. However, even the most robust architecture is ineffective if stakeholders cannot understand, apply, or modify it. Documentation is the essential bridge between architectural design & its practical implementation.

The core of architectural documentation lies in the selection & detail of specific "views," supplemented by information that spans the entire system. Effective documentation serves three primary purposes: **Education** (onboarding project members), **Communication** (aligning stakeholders), & **Analysis** (validating quality attributes). To succeed, documentation must be stakeholder-centric, utilizing standard templates—such as those from the Rational Unified Process (RUP) or the Software Engineering Institute (SEI)—to ensure clarity & completeness while maintaining a "minimum necessary information" threshold.

The Strategic Importance of Documentation

The architecture of a software system is its most critical foundation. Without adequate documentation that conveys both functional requirements & quality attributes, a project is prone to failure.

- **Necessity of Communication:** Creating an architecture is only half the task; communicating it to stakeholders is equally vital.
- **Consequences of Poor Documentation:** Architecture becomes useless if those responsible for its use, construction, or modification:
    - Do not know the architecture exists.
    - Cannot understand it well enough to implement it.
    - Misinterpret the design & apply it incorrectly.

Primary Utility:

- **Education:** Introducing new personnel to the project.
- **Communication:** Facilitating dialogue & agreement among stakeholders.
- **Analysis:** Providing a basis for assuring quality attributes & validating design decisions.

Frameworks for Architectural Views

Documentation is structured around "views," which represent specific aspects of the system. Several methodologies exist for selecting these views, with the **4+1 View Model** being the most commonly utilized.

Common Architectural View Approaches

Approach	Views Included	Primary Focus
Kruchten's 4+1	Logical, Process, Development, Physical + Use-Case	Behavioral requirements, concurrency, & mapping to physical nodes.
Siemens Four-Views	Conceptual, Module interconnection, Execution, Code	Structural & execution flow.
Herzum & Sims	Technical, Application, Project management, Functional	Management & technical integration.
Software Cost Reduction (SCR)	Module, Process, Uses	Encapsulation, performance, & incremental development.

The 4+1 View Model Detail

- **Logical View:** Focuses on behavioral requirements & key abstractions (objects/classes).
- **Process View:** Addresses concurrency, distribution, & the mapping of threads to objects.
- **Development View:** Outlines the organization of software modules, libraries, & units.
- **Physical View:** Maps software elements onto processing & communication nodes.
- **Use-Case View (The "+1"):** Relates all other views to important use cases to demonstrate system functionality.

The Documentation Process

A systematic approach to documentation involves four foundational steps:

1. **Define Stakeholders:** Identify all parties, including project managers, developers, testers, maintainers, customers, & end-users.
2. **Identify Concerns:** Record stakeholder concerns to prioritize which views are most relevant.
3. **Select Views:** Choose a set of views that is both minimal & adequate for the identified needs.
4. **Select Templates:** Use established st&ards (e.g., RUP or SEI) to define what & how to document.

Rational Unified Process (RUP) Template Structure

Based on Source Context the RUP template follows this hierarchy:

- **Introduction:** Purpose, Scope, & Overview.
- **Architectural Representation, Goals, & Constraints.**
- **Specific Views:** Use-Case, Logical (Design Packages), Process, Deployment, & Implementation (Layers).
- **Optional/Supplementary:** Data View, Size & Performance, & Quality.

Detailed Components of a Single View

Documenting a specific view requires more than a simple diagram. A comprehensive "view packet" includes:

1. **Primary Presentation:** Usually a graphical "cartoon" or a table showing elements & their relationships. It must include a key for notation.
2. **Element Catalog:** A detailed list explaining the elements in the primary presentation, their properties, interfaces, & relationship exceptions.
3. **Context Diagram:** A visual representation of how the system (or the specific portion of the view) relates to its external environment.
4. **Variability Guide:** Instructions on how to exercise variation points where decisions are left unbound until later development stages.
5. **Architecture Background:** The rationale for design decisions, including rejected alternatives, analysis results, & environmental assumptions.

Documentation Beyond the View

To provide a holistic understanding of the system, documentation must include information that applies to the entire architecture or links multiple views together.

- **Documentation Roadmap:** Guides stakeholders on how the documentation is organized & which scenarios require consulting specific parts.
- **System Overview:** A prose description of the system's purpose, users, & constraints to create a consistent mental model.
- **Mapping Between Views:** Tables or descriptions that capture the associations & correspondences between different views.
- **Rationale for System-Wide Decisions:** Documentation of organizational constraints, major requirements, & the selection of fundamental architectural patterns.
- **Directory:** Reference materials including an index of terms, glossary, & acronym list.

Documenting Quality Attributes

Architecture documentation must explicitly address quality attribute requirements (e.g., performance, modifiability) & the tradeoffs made to achieve them.

- **Rationale & Tradeoffs:** The document must explain the choice of design approach in the context of quality requirements.
- **Service Bounds:** Architectural elements providing services should have assigned quality attribute bounds.

- **Requirement Mapping:** A mapping that demonstrates how specific requirements—particularly quality attributes—are satisfied by the architecture.
- **Stakeholder Assurance:** Since every quality attribute has a constituency of interested stakeholders, these attributes must be clearly identified & validated through analysis results.

Glossary of Key Terms

Term	Definition
Architecture Background	The rationale for design decisions, including rejected alternatives, analysis results, & environmental assumptions.
Context Diagram	A diagram showing how the system, or a portion of it, relates to its external environment.
Documentation Roadmap	A guide describing the organization of documentation, including a list of views & usage scenarios for stakeholders.
Element Catalog	A detailed list explaining the elements depicted in a primary presentation, including their properties, interfaces, & relations.
Kruchten's 4+1 Views	A popular architectural model consisting of Logical, Process, Development, & Physical views, plus a Use Case view to tie them together.
Mapping between Views	Documentation, often captured in tables, that establishes the correspondences & associations between different architectural views.
Primary Presentation	The initial graphical or textual display of a view showing elements & their relationships; must include a key for notation.
Rationale	Documentation of architectural decisions that apply to multiple views, often including organizational constraints or major requirements.
Software Architecture	A crucial determinant of system success that defines function & quality attributes.
Stakeholders	Individuals or groups with an interest in the architecture, including project managers, developers, testers, maintainers, & end users.
Variability Guide	A guide explaining how to exercise variation points in an architecture where decisions are left unbound until later development stages.
View	A representation of a set of system elements & the relationships between them, chosen to address specific stakeholder concerns.