SREENIVASA INSTITUTE OF TECHNOLOGY AND MANAGEMENT STUDIES, CHITTOOR

(AUTONOMOUS)

Approved by AICTE, New Delhi, Affiliated to JNTUA, Ananthapuramu.

LECTURE NOTES

Subject Name : SOFTWARE ARCHITECTURE AND DESIGN PATTERNS
Year / Branch :4-1 SEMESTER /COMPUTER SCIENCE AND ENGINEERING
Regulation :R23
Prepared BY: M.L. MADHURI

Course Outcomes:

- To highlight the evolution of patterns.
- To learn how to add functionality to designs while minimizing complexity
- To learn what design patterns really are, and are not
- To know about specific design patterns.
- To learn how to use design patterns to keep code quality high without over design.

syllabus

UNIT – I

Envisioning Architecture: The Architecture Business Cycle, What is Software Architecture, Architectural patterns, reference models, reference architectures, architectural structures and views. Creating an Architecture: Quality Attributes, Achieving qualities, Architectural styles and patterns, designing the Architecture, Documenting software architectures, Reconstructing Software Architecture.

UNIT – II

Analyzing Architectures: Architecture Evaluation, Architecture design decision making, ATAM, CBAM. Moving from one system to many: Software Product Lines, Building systems from off the shelf components, Software architecture in future.

UNIT – III

Patterns: Pattern Description, Organizing catalogs, role in solving design problems, Selection and usage.Creational and Structural patterns: Abstract factory, builder, factory method, prototype, singleton, adapter, bridge, composite, façade, flyweight

UNIT – IV

Behavioural patterns: Chain of responsibility, command, Interpreter, iterator, mediator,memento, observer, state, strategy. template method, visitor.

UNIT – V

Case Studies: A-7E – A case study in utilizing architectural structures, The World Wide Web - a case study in interoperability, Air Traffic Control – a case study in designing for high availability, Celsius Tech – a case study in product line development

Prepared by M.L. Madhuri,
Assistant Professor SITAMS

UNIT-I

Envisioning Architecture: The Architecture Business Cycle, What is Software Architecture, Architectural patterns, reference models, reference architectures, architectural structures and views

The Architecture Business Cycle (ABC)

Definition

The Architecture Business Cycle (ABC) defines how software architecture is influenced by business goals, stakeholders, and technical environments, and how it, in turn, influences the organization and system development.

Key Terminologies

Term	Definition
Stakeholders	Individuals or groups with an interest in the system (users, developers, managers).
Business Drivers	Business goals that shape architectural decisions (cost, scalability, time-to-market).
Technical Environment	Tools, platforms, and technologies available.
Organizational Structure	Hierarchy and communication flow within the organization.

Conceptual Model

Stakeholders → Business Goals → Requirements → Architecture → System → Feedback → Stakeholders

Importance

- Aligns architecture with business strategy.
- Ensures stakeholder satisfaction.

Enables iterative improvement

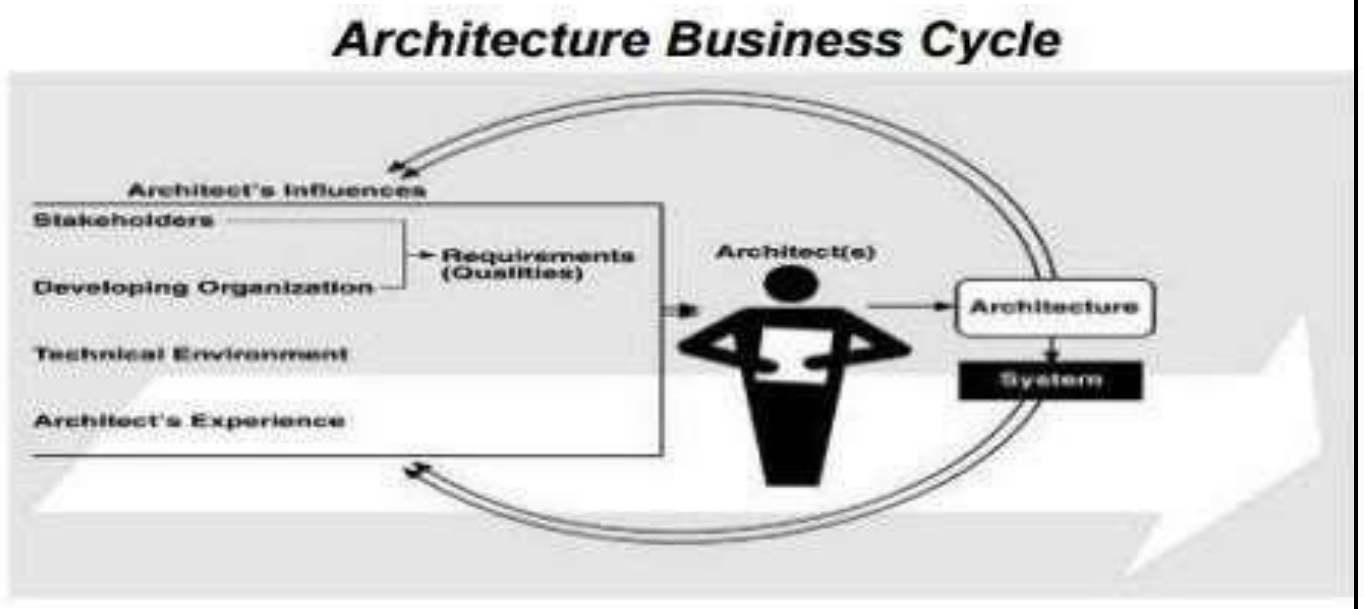


Figure 1: Architecture Business Cycle

Figure 1 shows the feedback loops. Some of the feedback comes from the architecture itself, and some comes from the system built from it.

The architecture affects the structure of the developing organization.

Architectures are influenced by:

- System stakeholders
- The Developing Organization
- The Background and Experience of the Architects
- The Technical Environment

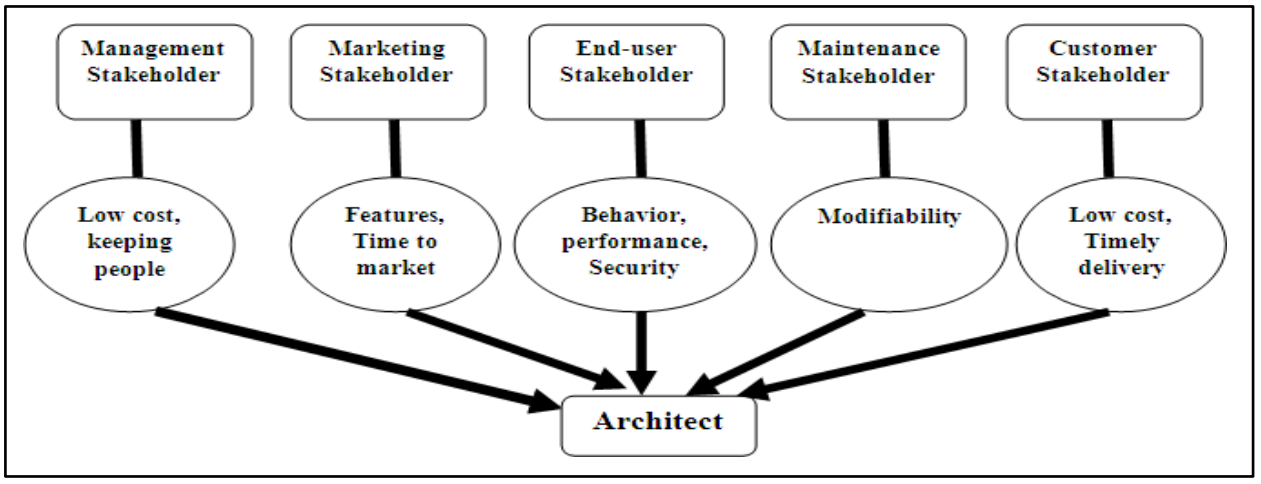


Figure 2:Influence of stakeholders on the architect

Figure 2 shows the architect receiving helpful stakeholder "suggestions."

Real-World Example

Amazon: Transitioned from monolithic to microservices architecture to improve scalability and deployment speed.

Advantages and Disadvantages

Advantages	Disadvantages
Business alignment	Complex stakeholder management
Stakeholder satisfaction	Communication overhead
Flexibility	Increased coordination cost

Applications

- Enterprise systems
- Cloud-native applications
- E-commerce platforms

Case Study

Netflix: Migrated from monolithic DVD rental system to microservices-based streaming platform, improving scalability and fault tolerance.

What is Software Architecture?

Definition

According to IEEE: Software architecture is the fundamental organization of a system, acting as a structural blueprint. It dictates how different components interact, communicate, and scale, balancing technical requirements (performance, security) with business goals.

Key Terminologies

Term	Definition
Components	Functional units of computation.
Connectors	Define interactions between components.
Configurations	Arrangements of components and connectors.

Architecture is high-level design

Architecture is the overall structure of the system

Architecture is the structure of the components of a program or system, their interrelationships, and the principles and guidelines governing their design and evolution over time.

Architecture is components and connectors

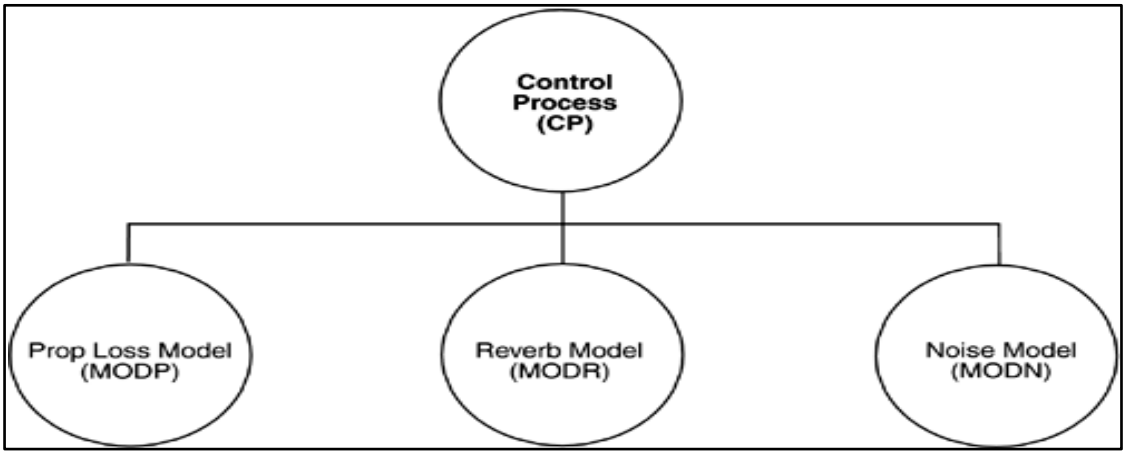


Figure 3: Typical, but uninformative, presentation of software architecture

The system consists of four elements which belongs to acoustics.

Three of the elements— Prop Loss Model (MODP), Reverb Model (MODR), and Noise Model (MODN)—might have more in common with each other than with the fourth—Control Process (CP)—because they are positioned next to each other.

All of the elements apparently have some sort of relationship with each other, since the diagram is fully connected.

WHY IS SOFTWARE ARCHITECTURE IMPORTANT?

There are fundamentally three reasons for software architecture's importance from a technical perspective.

- *Communication among stakeholders*
- *Early design decisions*
- Transferable abstraction of a system

Importance of Software Architecture:

- Architecture Is The Vehicle For Stakeholder Communication
- Architecture Manifests The Earliest Set Of Design Decisions
- The architecture defines constraints on implementation
- The architecture dictates organizational structure
- The architecture inhibits or enables a system's quality attributes
- Predicting system qualities by studying the architecture
- The architecture makes it easier to reason about and manage change
- The architecture helps in evolutionary prototyping
- The architecture enables more accurate cost and schedule estimates

ARCHITECTURAL STRUCTURES AND VIEWS

Architectural structures can by and large be divided into three groups, depending on the broad nature of the elements they show.

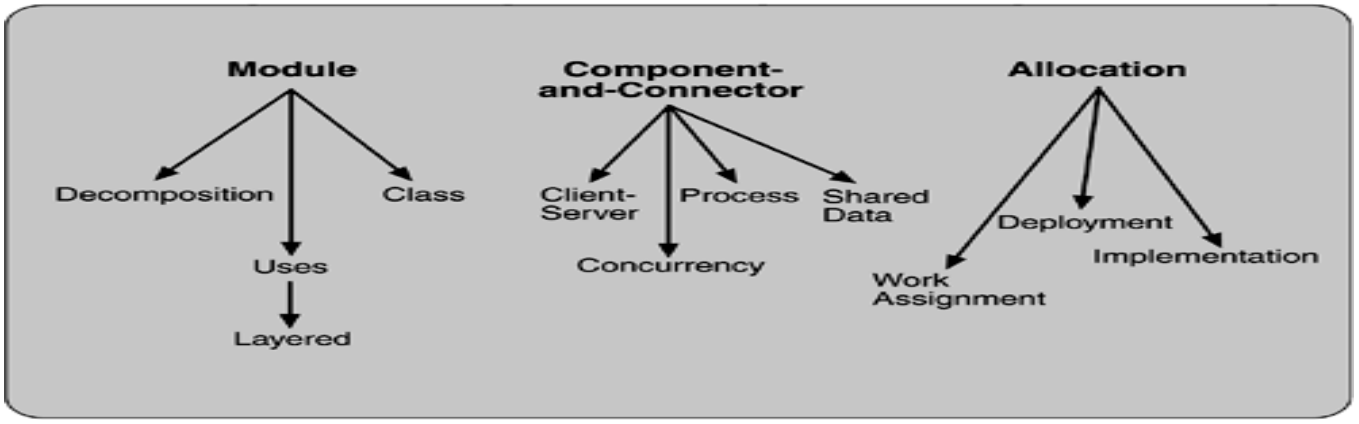


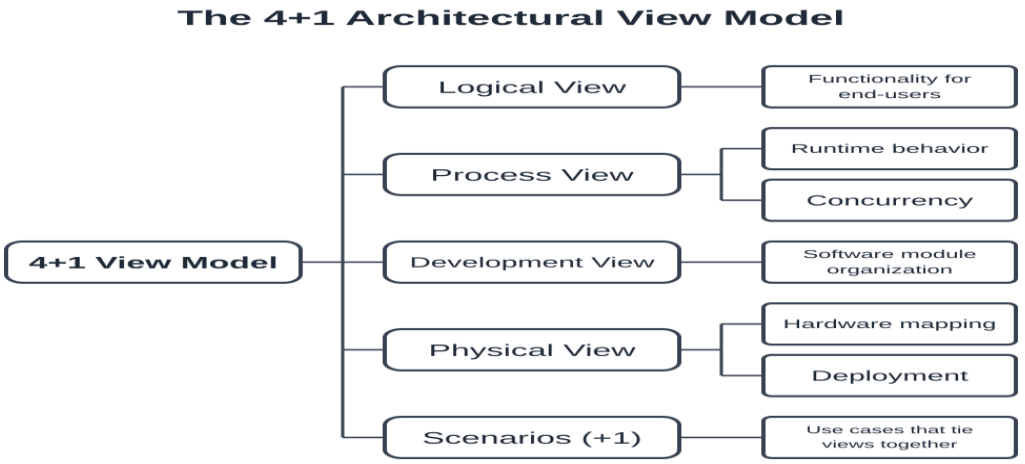
Figure 4: Architectural Views

Module	**Decomposition**	The units are modules related to each other by the "is a sub module of " relation, showing how larger modules are decomposed into smaller ones recursively until they are small enough to be easily understood
	Uses	The units of this important but overlooked structure are also modules, or procedures or resources on the interfaces of modules.
	Layered	When the uses relations in this structure are carefully controlled in a particular way, a system of layers emerges, in which a layer is a coherent set of related functionality.
	Class	The module units in this structure are called classes. The relation is "inherits-from" or "is-an-instance-of."
Component-and-Connector	**Client-server**	If the system is built as a group of cooperating clients and servers, this is a good component-and-connector structure to illuminate.
	Concurrency	This component-and-connector structure allows the architect to determine opportunities for parallelism and the locations where resource contention may occur.
	Process	Like all component-and-connector structures, this one is orthogonal to the module-based structures and deals with the dynamic aspects of a running system.
	Shared data	This structure comprises components and connectors that create, store, and access persistent data.
Allocation	**Work assignment**	This structure assigns responsibility for implementing and integrating the modules to the appropriate development teams.
	Deployment	The deployment structure shows how software is assigned to hardware-processing and communication elements.
	Implementation	This structure shows how software elements (usually modules) are mapped to the file structure(s) in the system's development, integration, or configuration control environments.

WHICH STRUCTURES TO CHOOSE?

Because software is complex, we cannot see everything at once. We use different **views** to look at different **structures**. A common standard is the **4+1 View Model**:

Kruchten's 4+1 View Model :



.

Figure 5: Kruchten's 4+1 views

Importance

- Reduces cost of change.
- Improves maintainability and scalability.
- Enables parallel development.

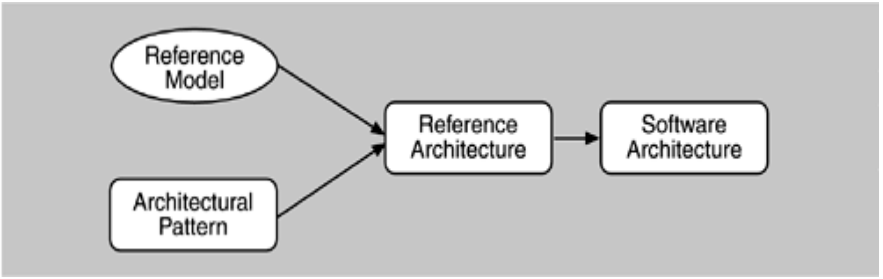
Real Examples

- **Banking System:** Layered architecture for security and modularity.
- **E-commerce Platform:** Microservices for scalability.

Architectural Patterns, Reference Models, and Reference Architectures

According to IEEE the definitions are as follows:

Concept	Definition



Architectural Pattern	Reusable solution to a recurring architectural problem.
Reference Model	Conceptual framework dividing functionality into layers.
Reference Architecture	Template solution for a specific domain.

Figure 6: The relationships of design elements

Pattern Catalog

Pattern	Description	Diagram Type
Layered Architecture	Separation of concerns	N-tier diagram
Client-Server	Distributed processing	Network diagram
Pipe and Filter	Data transformation pipeline	Data flow diagram
MVC	Separation of UI and logic	Interaction diagram
Microservices	Independent deployable services	Service mesh diagram
Event-Driven	Asynchronous communication	Event bus diagram

Reference Models

- **OSI Model:** 7-layer communication model.
- **TCP/IP Stack:** 4-layer practical model.

Reference Architectures

- **J2EE:** Enterprise Java applications.
- **.NET:** Microsoft's enterprise framework.
- **TOGAF:** Enterprise architecture methodology.

Advantages and Disadvantages

Pattern	Advantages	Disadvantages
Layered	Modularity	Performance overhead
Microservices	Scalability	Complex deployment

Decision Matrix

Use **Layered** for maintainability, **Microservices** for scalability, **Event-Driven** for responsiveness.

Creating Architecture: Quality Attributes, Achieving qualities, Architectural styles and patterns, designing the Architecture, Documenting software architectures, Reconstructing Software Architecture

Quality Attributes

- **Performance**
- **Security**
- **Modifiability**
- **Availability**
- **Scalability**
- **Usability**
- **Testability**
- **Deployment**

Quality Attribute Scenario

- **Source:** Who triggers the scenario
- **Stimulus:** What happens
- **Environment:** The stimulus occurs within certain conditions. The system may be in an overload condition or may be running when the stimulus occurs, or some other condition may be true.
- **Artifact:** This may be the whole system or some pieces of it.
- **Response:** System behavior
- **Response Measure:** Quantitative result

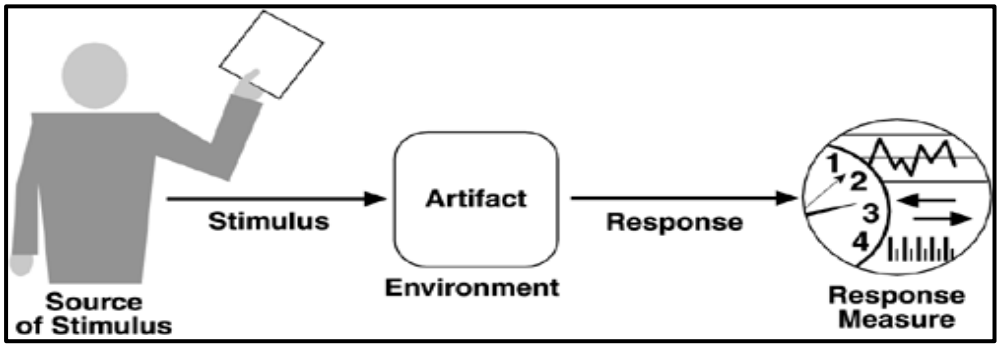


Figure 7: shows the parts of a quality attribute scenario.

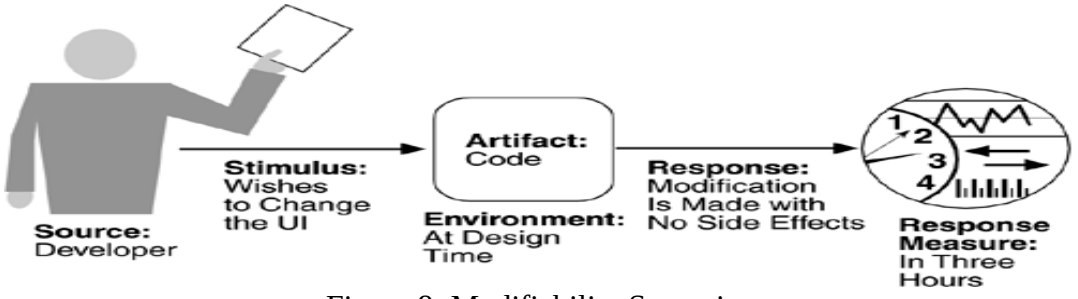


Figure 8: Modifiability Scenario

Achieving qualities

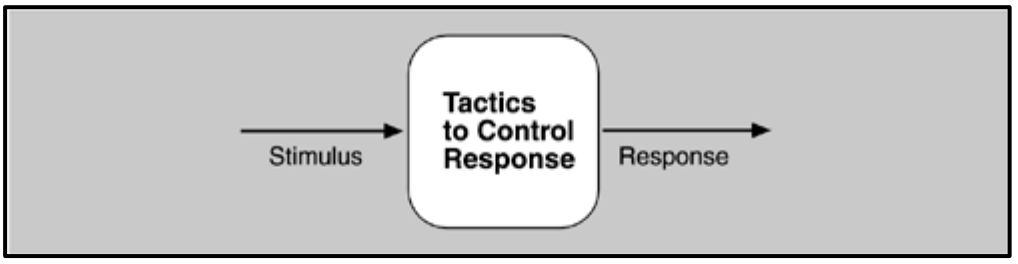


Figure 9: Achieving qualities

A tactic is a design decision that influences the control of a quality attribute response.

Quality Attribute	**Description**	**Tactics**
Performance	Response time, throughput	Caching, load balancing
Security	Protection from threats	Encryption, authentication
Modifiability	Ease of change	Layered design, abstraction
Availability	System uptime	Redundancy, failover
Usability	Ease of use	Consistent UI, feedback

Availability Tactics: A failure occurs when the system no longer delivers a service that is consistent with its specification; this failure is observable by the system's users.

- A fault (or combination of faults) has the potential to cause a failure.

Recall also that recovery or repair is an important aspect of availability. We first consider fault detection. We then consider fault recovery and finally, briefly, fault prevention.

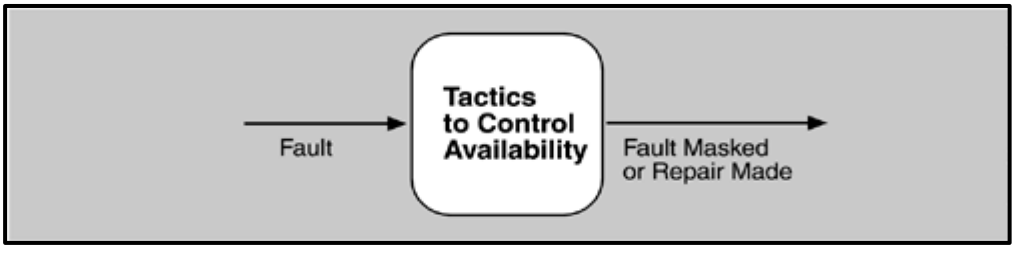


Figure 10: Goal of availability tactics

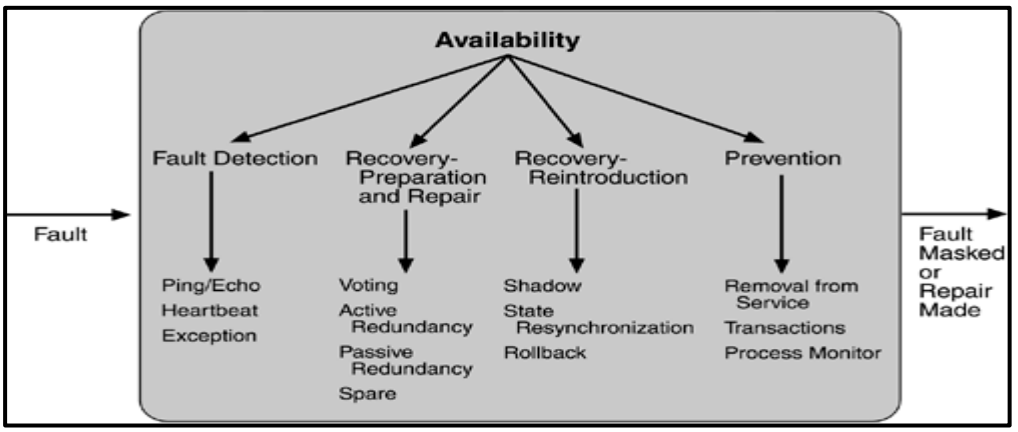


Figure 11: Summary of availability tactics

Modifiability Tactics: We organize the tactics for modifiability in sets according to their goals.

One set has as its goal reducing the number of modules that are directly affected by a change. We call this set **"localize modifications."**

A second set has as its goal limiting modifications to the localized modules. We use this set of tactics to **"prevent the ripple effect."**

A third set of tactics has as its goal controlling deployment time and cost. We call this set **"defer binding time."**

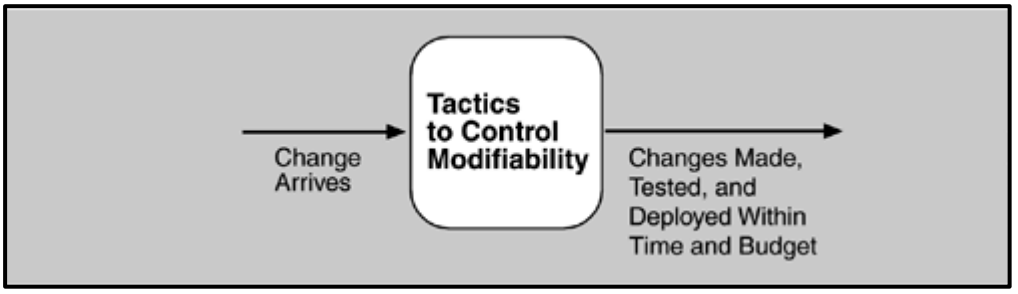


Figure 12: Goal of modifiability tactics

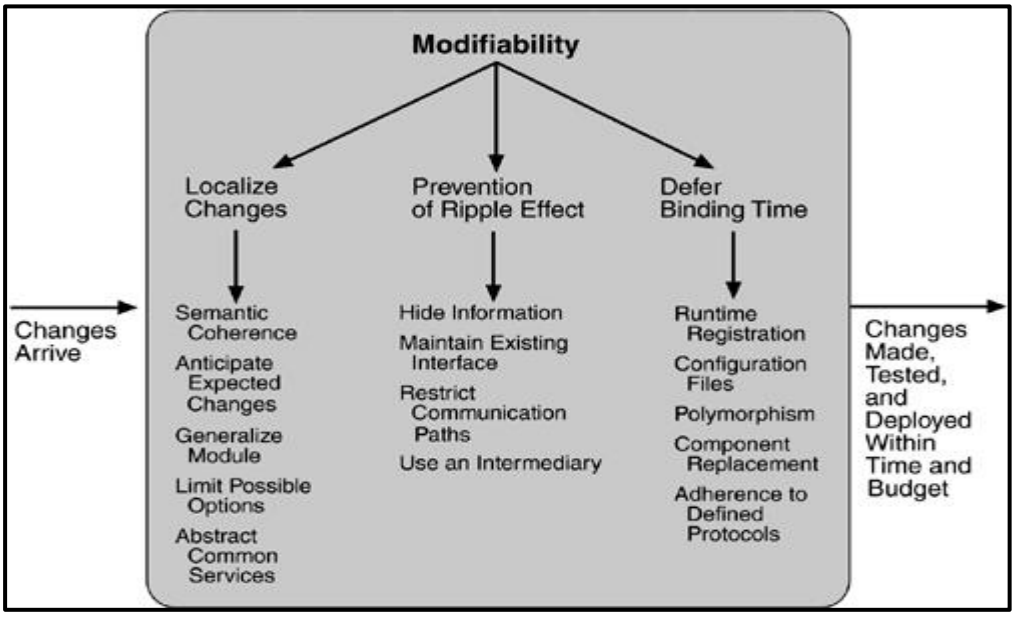


Figure 13: Summary of modifiability tactics

Performance Tactics: The event can be single or a stream and is the trigger for a request to perform computation.

It can be the arrival of a message, the expiration of a time interval, the detection of a significant change of state in the system's environment, and so forth.

The system processes the events and generates a response. Performance tactics control the time within which a response is generated.

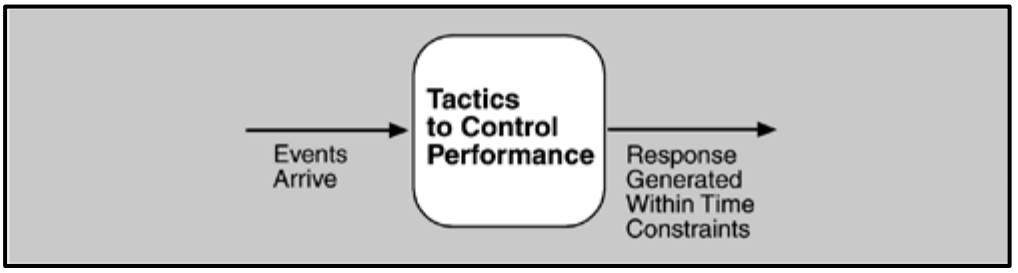


Figure 14: Goal of performance tactics

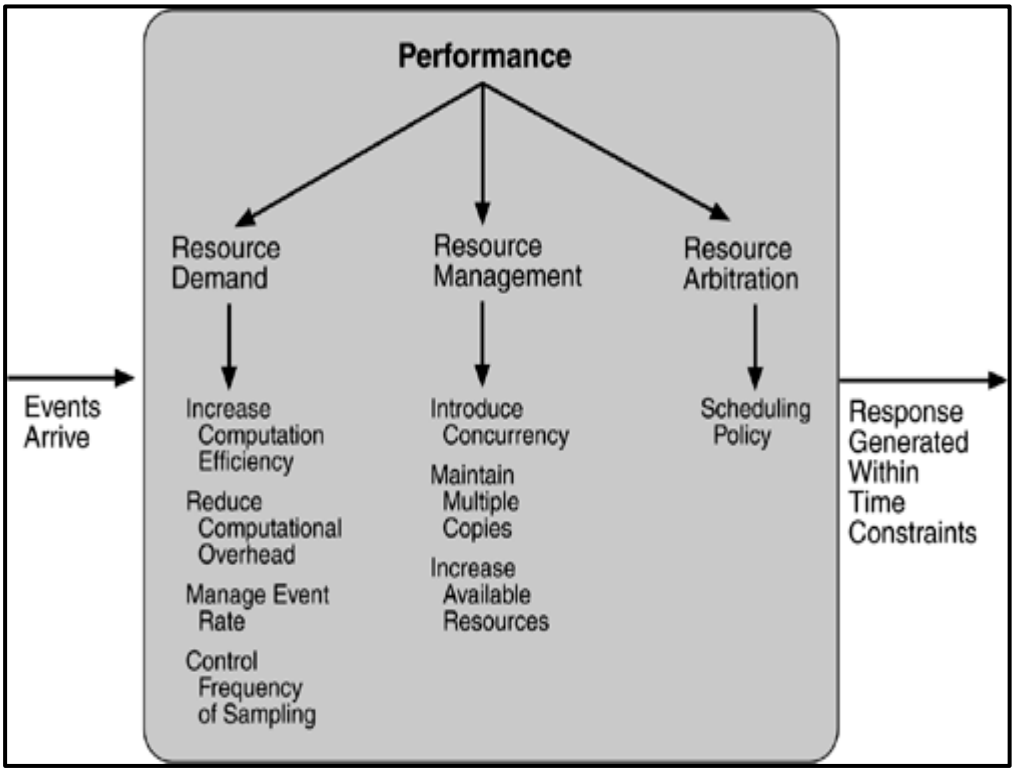


Figure 15: Summary of performance tactics

Security Tactics: Tactics for achieving security can be concerned with detecting attacks, and those concerned with recovering from attacks.

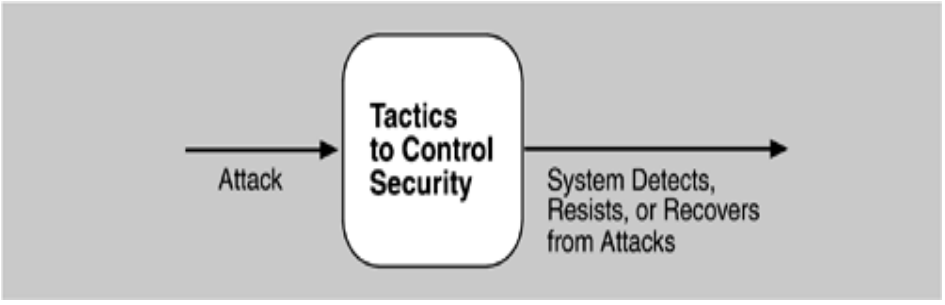


Figure 16: Goal of security tactics

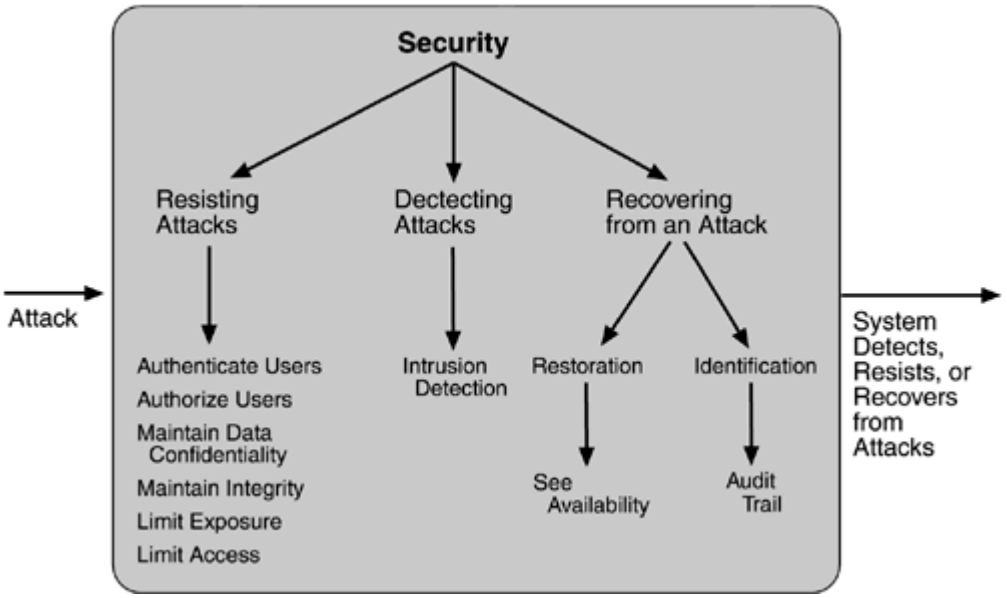


Figure 17: Summary of tactics for security

Testability Tactics: The goal of tactics for testability is to allow for easier testing when an increment of software development is completed.

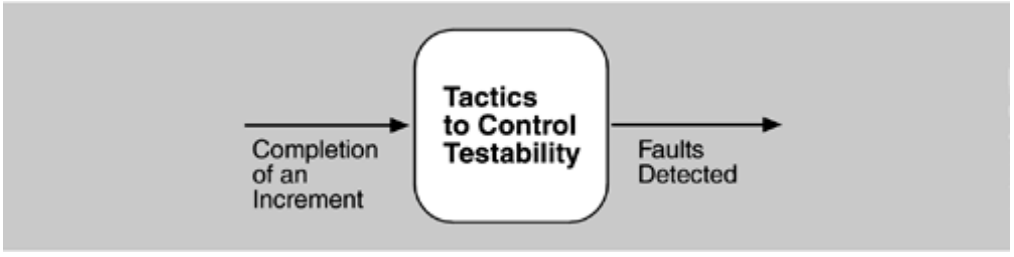


Figure 18: Goal of testability tactics

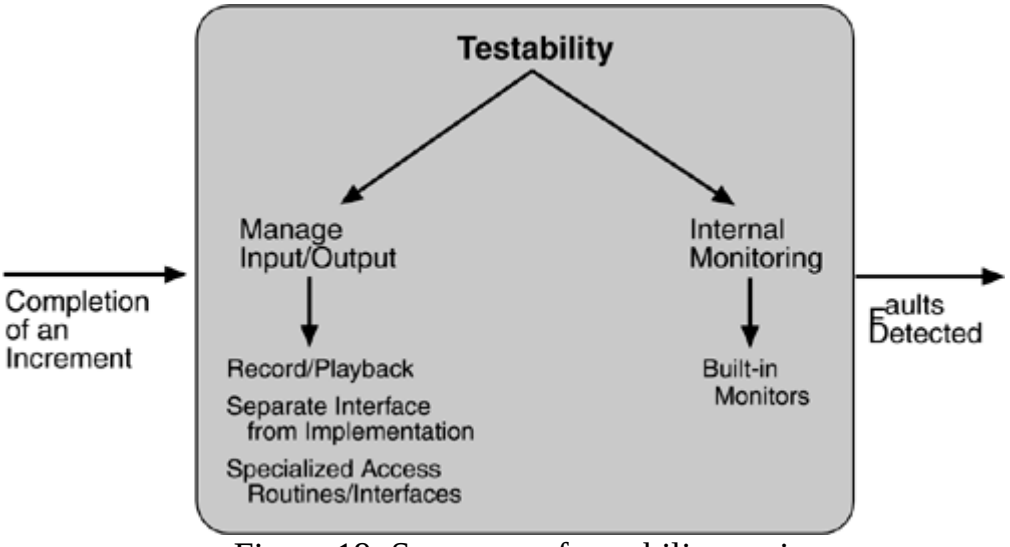


Figure 19: Summary of testability tactics

Usability Tactics: The first category is to support the user during system execution.

The second category is based on the iterative nature of user interface design and supports the interface developer at design time.

It is strongly related to the modifiability tactics already presented.

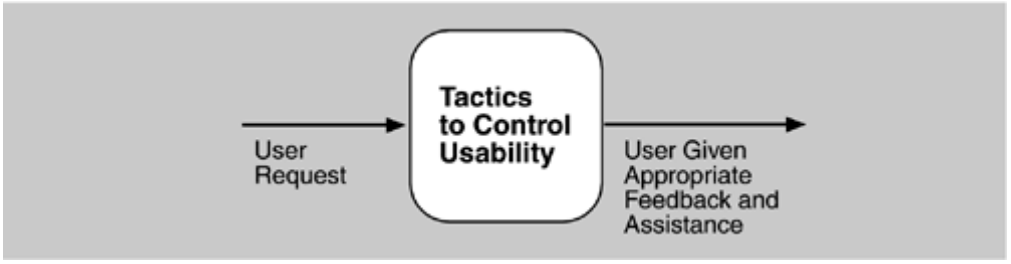


Figure 20: Goal of runtime usability tactics

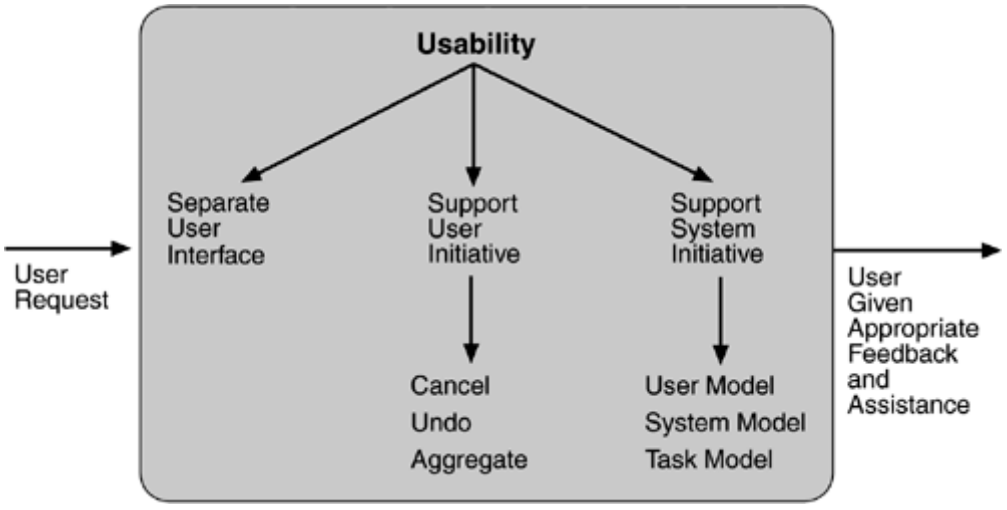


Figure 21: Summary of runtime usability tactics

Architectural Styles and Patterns

Catalog

- Layered, Client-Server, Microservices, Event-Driven, Pipe & Filter, Service-Oriented.

Comparison Table:

Style	Scalability	Modifiability	Complexity
Layered	Medium	High	Low

Microservices	High	High	High

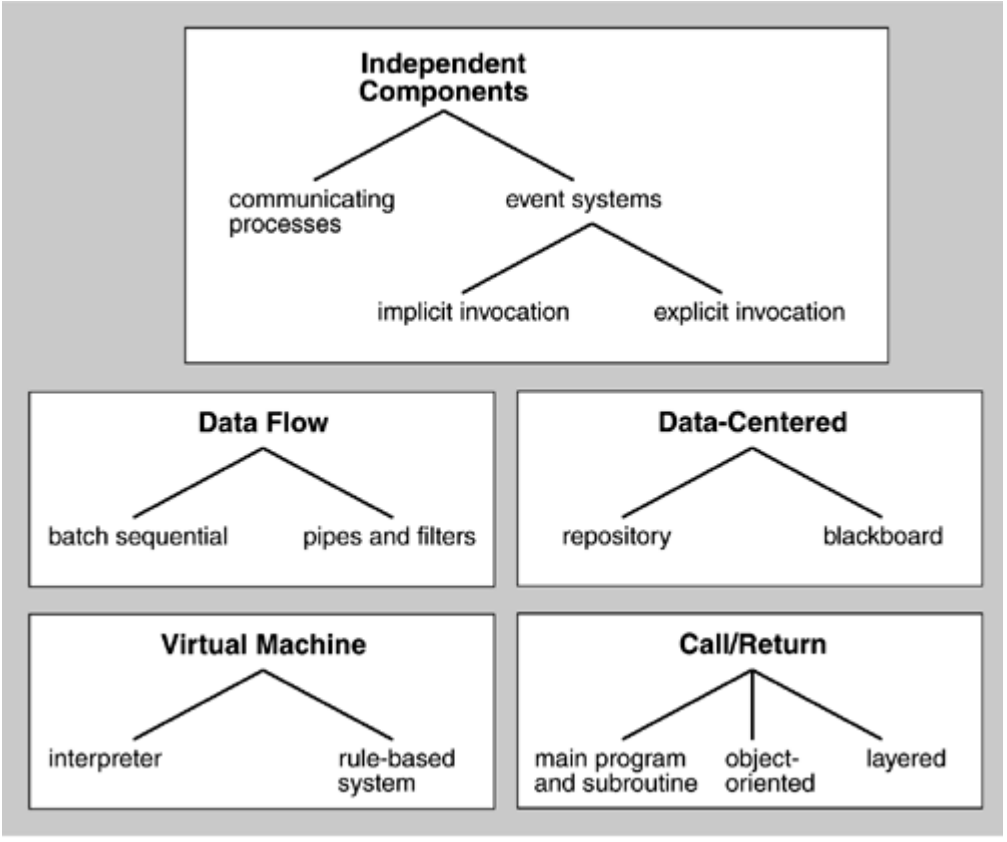


Figure 22: A small catalog of architectural patterns, organized by is-a relations

Designing the Architecture

Process

1. Define requirements.
2. Identify quality attributes.
3. Select patterns.
4. Evaluate alternatives.
5. Document decisions.

ADD Method

Attribute-Driven Design focuses on achieving quality goals through iterative refinement.

Example

Social Media Platform: Microservices for scalability, caching for performance.

Documenting Software Architectures

Standards

- **IEEE 1471:** Architecture description standard.
- **ADLs:** Architecture Description Languages (e.g., Acme, AADL).

UML Notations

- **Class Diagrams:** Structure.
- **Sequence Diagrams:** Behavior.
- **Deployment Diagrams:** Physical mapping.

There is no industry-standard template for documenting a view, but the seven-part standard organization is as follows:

1. Primary presentation
2. Element catalog
3. Context diagram
4. Variability guide
5. Architecture background
6. Glossary of terms
7. Other information

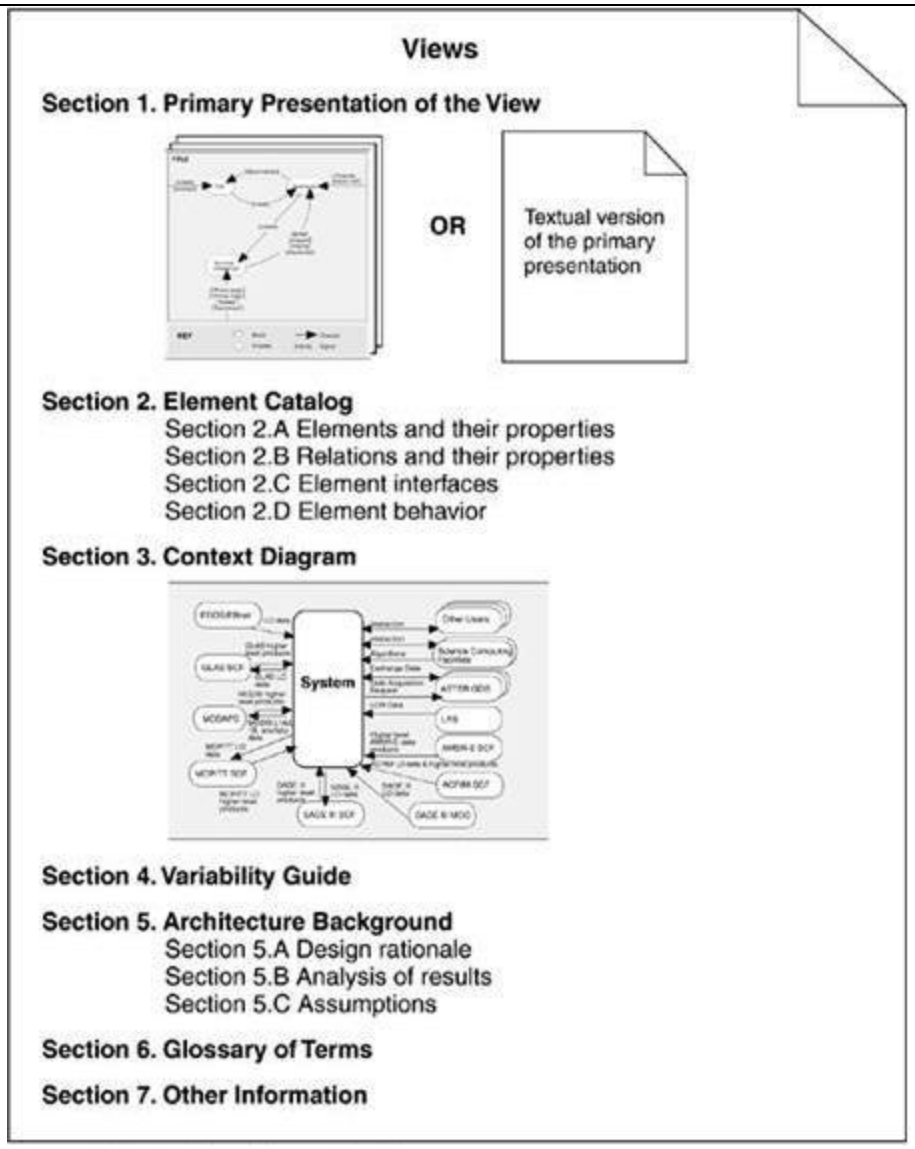


Figure 23: The seven parts of a documented view

Tools

ArchiMate

C4 Model

Reconstructing Software Architecture

Definition: Reverse engineering existing systems to recover architectural documentation.

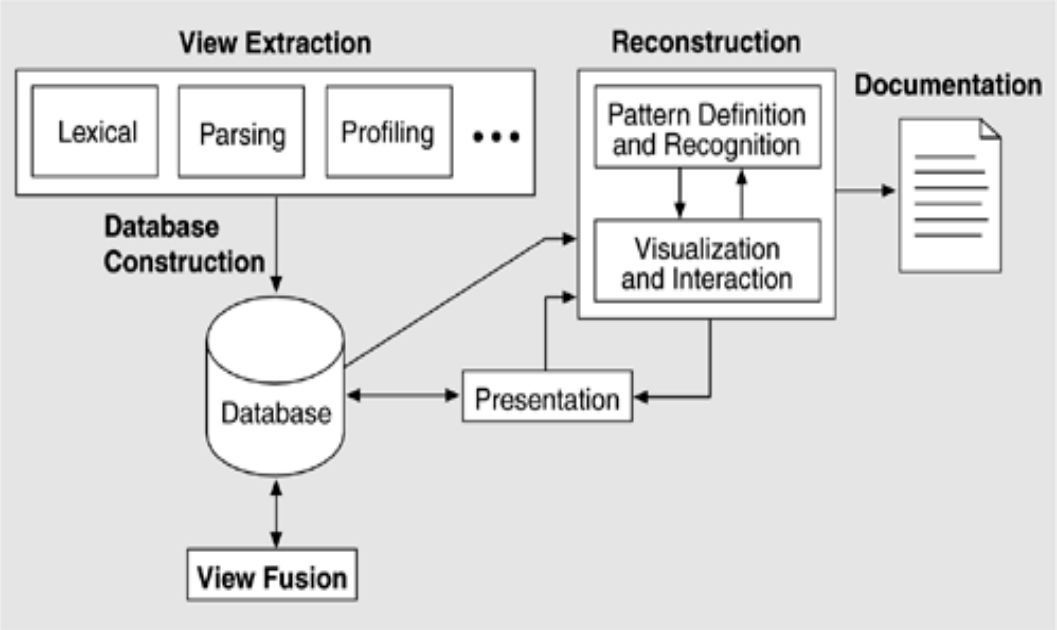


Figure 24: Architecture reconstruction activities.

Techniques

- Static and dynamic analysis.
- Visualization tools.

Case Study

Legacy Banking System: Reconstructed to migrate to cloud-native architecture.

a) Short Questions (2 Marks)

1. Define Software Architecture.
2. What is the Architecture Business Cycle (ABC)?
3. What is a Reference Model?
4. Define Quality Attribute.

5. What is Architectural Style?

b) Essay Questions (10 Marks)

1. Explain the Architecture Business Cycle with a neat diagram.
2. Discuss the concept of Software Architecture and its importance.
3. Explain Quality Attributes and techniques to achieve them.
4. Describe different Architectural Structures and Views with examples.
5. Explain the process of Documenting and Reconstructing Software Architecture.

c) Case Study-Based Questions

1. A college wants to develop an Online Student Management System. Explain how Software Architecture helps in designing the system.
2. An e-commerce company wants its website to be highly available and secure. Suggest suitable Quality Attributes and Architectural Styles.

Standard Quote:

"Think before you code. A good architecture today saves countless hours of maintenance tomorrow."

UNIT II – ANALYZING ARCHITECTURES

Architecture Evaluation Methods

Architecture Evaluation is a critical process used to determine if a software architecture will satisfy its required quality attributes (such as performance, availability, or modifiability).

Instead of waiting until a system is fully built to test it, evaluation allows stakeholders to identify risks and potential failures early in the lifecycle.

Steps:

Define Goals

Collect Scenarios

Analyze Tradeoffs

Report Findings

Methods:

ATAM (Architecture Tradeoff Analysis Method):

A comprehensive method used to understand the consequences of architectural decisions.

It helps identify 'trade-off points' where a decision affects multiple quality attributes and 'sensitivity points' where a small change in architecture significantly impacts a quality goal.

Architecture Evaluation is a critical process used to determine if a software architecture will satisfy its required quality attributes (such as performance, availability, or modifiability).

Instead of waiting until a system is fully built to test it, evaluation allows stakeholders to identify risks and potential failures early in the lifecycle.

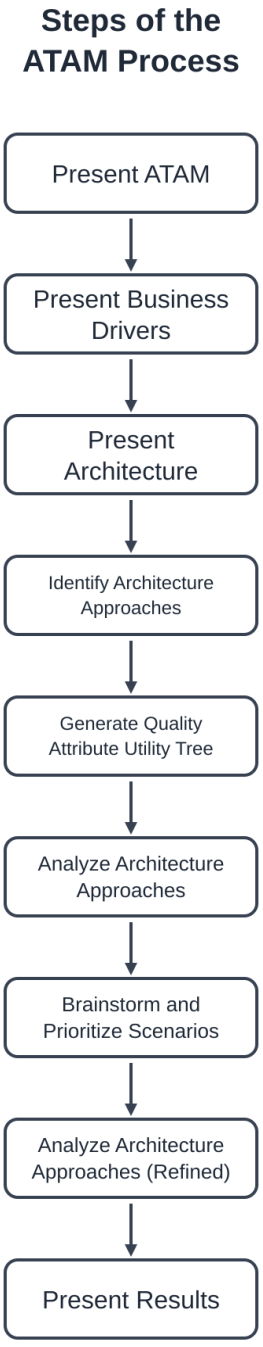


Figure 1. The nine steps of the Architecture Tradeoff Analysis Method (ATAM).

Example:

Evaluating trade-offs between performance and modifiability in a banking system.

CBAM (Cost Benefit Analysis Method):

An extension of ATAM that adds a financial layer to decision-making.

It helps organizations choose architectural changes based on their return on investment (ROI) by analyzing the cost of implementation against the quality benefits gained.

Purpose: Quantifies cost-benefit of architectural decisions.

Example Table:

Decision	Cost	Benefit	ROI
Add caching	3	8	2.67
Add redundancy	5	9	1.8

Economic Decision Making: The CBAM

While ATAM identifies technical risks, the Cost Benefit Analysis Method (CBAM) addresses the reality that organizations have limited budgets.

CBAM builds upon the results of an ATAM evaluation by quantifying the costs and benefits of different architectural strategies.

In a CBAM analysis, stakeholders assign a 'utility' value to various quality attribute scenarios. By calculating the cost of implementing a strategy versus the utility (benefit) it provides, architects can determine the Return on Investment (ROI) for different design decisions.

This ensures that the evolution of the architecture is driven by both technical excellence and business value.

Comparison: ATAM vs. CBAM		
Feature	**ATAM**	**CBAM**
Primary Focus	Technical risk and quality attributes	Economic costs and benefits
Key Output	Sensitivity points and tradeoffs	ROI of architectural strategies
Stakeholder Input	Quality attribute scenarios	Utility and cost estimates

Moving from One System to Many

Definition: Set of related systems sharing a common architecture.

Modern software engineering often involves creating multiple versions of a system for different markets or clients. This is achieved through Software Product Lines (SPL).

Instead of building each system from scratch, organizations develop a 'core asset base'—including architecture, components, and test plans—that can be reused across the entire product line.

However, moving toward an SPL approach requires a shift in architecture.

The architecture must be flexible enough to support 'variation points' where different products in the line can diverge in functionality.

This leads to significant long-term savings in development time and maintenance costs, but requires higher initial investment in architectural planning.

Advantages:

Reduced cost

Faster development

Disadvantages:

High initial investment.

Off-the-Shelf Components and the Future

Definition: Using Commercial Off-The-Shelf components.

Challenges: Integration, compatibility, customization.

Example: Using third-party payment gateways in e-commerce

Building systems from Off-The-Shelf (OTS) components presents unique architectural challenges. Because the architect does not control the internal design of OTS components, they must deal with 'architectural mismatch.'

This occurs when the assumptions made by the component (e.g., how it handles data or errors) conflict with the assumptions of the overall system architecture.

The Future of Architecture: As systems become more complex and interconnected, software architecture is moving toward more automated evaluation and self-healing structures.

The role of the architect is evolving from a pure designer to a curator of complex ecosystems, balancing rapid deployment with the rigorous quality standards.

AI-driven architecture design.

Cloud-native and server less architectures.

Model-driven engineering.

Quantum and edge computing integration

UNIT–II

a) Short Questions (2 Marks)

1. What is Architecture Evaluation?
2. Expand ATAM.
3. Expand CBAM.
4. Define Software Product Line.
5. What are Off-the-Shelf Components?

b) Essay Questions (10 Marks)

1. Explain Architecture Evaluation and its benefits.
2. Describe the ATAM method with suitable steps.
3. Explain CBAM and compare it with ATAM.
4. Discuss Software Product Lines with advantages.
5. Explain Building Systems using Off-the-Shelf Components.

c) Case Study-Based Questions

1. A software company wants to evaluate its hospital management system before deployment. Explain how ATAM can be applied.
2. A company develops similar software for schools, colleges, and universities. Explain how Software Product Lines help in reducing development time.

Standard Quote:

"Think before you code. A good architecture today saves countless hours of maintenance tomorrow."

Prepared by M.L. Madhuri,
Assistant Professor SITAMS

UNIT III – PATTERNS FUNDAMENTALS AND CREATIONAL/STRUCTURAL PATTERNS

What is a Design Pattern?

A design pattern is a general, reusable solution to a commonly occurring problem within a given context in software design.

It is not a finished design that can be transformed directly into code; rather, it is a description or template for how to solve a problem that can be used in many different situations.

Introduction to Pattern Catalogs

Design patterns are typically organized into catalogs to help architects and developers select the right tool for the job. These catalogs are categorized by intent: what the pattern does. The three primary categories are:

Organizing Pattern Catalogs

Categories:

Creational: Object creation.

Structural: Object composition.

Behavioral: Object interaction

- **Creational Patterns**: Deal with object creation mechanisms, trying to create objects in a manner suitable to the situation.

- **Structural Patterns**: Deal with object composition, simplifying the design by identifying a simple way to realize relationships between entities.
- **Behavioral Patterns**: Deal with communication between objects and the assignment of responsibilities.

Creational Patterns

Creational patterns abstract the instantiation process. They make a system independent of how its objects are created, composed, and represented.

Pattern	Problem & Definition	Real-World Analogy
Singleton	**Definition:** Ensures a class has only one instance and provides a global point of access to it. **Problem:** When you need to strictly limit a resource (like a database connection) so multiple parts of the program don't create conflicting copies.	A country can only have one official President at a time. No matter where you are in the country, 'The President' refers to the same specific person.
Factory Method	**Definition:** Defines an interface for creating an object but lets subclasses decide which class to instantiate. **Problem:** A system needs to create objects, but it doesn't know the exact type of object it will need until the program is running.	A toy factory has a standard assembly line. The line stays the same, but depending on the mold used, it can produce either a plastic car or a plastic doll.

Abstract Factory	**Definition:** Provides an interface for creating families of related or dependent objects without specifying their concrete classes. **Problem:** When your code needs to work with various families of related products (like Windows vs. Mac buttons) but shouldn't depend on the specific versions.	A furniture store that sells sets. If you buy a 'Victorian' set, the factory ensures the chair, sofa, and table all match that specific style.
Builder	**Definition:** Separates the construction of a complex object from its representation. **Problem:** When creating an object requires many steps or a long list of optional parameters, making the constructor messy.	Ordering a custom pizza. You start with the base, then choose to add cheese, pepperoni, or olives step-by-step until your specific pizza is built.
Prototype	**Definition:** Creates new objects by copying an existing object (a prototype). **Problem:** When creating a brand-new object from scratch is expensive or complicated, but you already have a similar one you can copy.	Cell division (mitosis). Instead of building a new cell from raw atoms, an existing cell makes an exact copy of itself.

Pattern	Intent	UML	Example	When to Use
Abstract Factory	Create families of related objects	Factory interface + concrete factories	GUI toolkit	When system should be independent of object creation
Builder	Construct complex objects step-	Director + Builder	StringBuilder	When object construction is complex

	by-step			
Factory Method	Define interface for creating objects	Creator + ConcreteCreator	DocumentFactory	When subclass decides which class to instantiate
Prototype	Clone existing objects	Prototype interface	clone()	When object creation is costly
Singleton	One instance globally accessible	Static instance	Runtime.getRuntime()	When only one instance is needed

Example (Singleton in Java):

```
class Singleton {
  private static Singleton instance;
  private Singleton() {}
  public static Singleton getInstance() {
    if (instance == null) instance = new Singleton();
    return instance;
  }
}
```

Structural Patterns

Structural patterns explain how to assemble objects and classes into larger structures while keeping these structures flexible and efficient.

Pattern	Problem & Definition	Real-World Analogy
Adapter	**Definition:** Allows objects with incompatible interfaces to work together. **Problem:** You have a useful class, but its interface doesn't match the code you are currently writing.	A power plug adapter. It allows a three-prong US plug to fit into a two-prong European outlet so the device can get power.
Bridge	**Definition:** Splits a large class or a set of closely related classes into two separate hierarchies—abstraction and implementation. **Problem:** When a class varies in two different ways (e.g., different Shapes and different Colors), leading to too many subclasses.	A light switch and a light bulb. The switch (abstraction) is separate from the bulb (implementation). You can change the bulb type without changing the switch.
Composite	**Definition:** Lets you compose objects into tree structures and then work with these structures as if they were individual objects. **Problem:** When you need to treat a single object and a group of objects the same way.	A file system. A folder can contain files or other folders. Whether you are 'moving' a single file or a whole folder, the action feels the same.
Façade	**Definition:** Provides a simplified interface to a complex library, framework, or system. **Problem:** A system is too complex to use easily because it has dozens of specialized classes that must be coordinated.	A customer service representative. You talk to one person (the façade), and they handle talking to the billing, shipping, and technical departments for you.

Flyweight	**Definition:** Lets you fit more objects into the available amount of RAM by sharing common parts of state between multiple objects. **Problem:** Your program creates so many similar objects that it runs out of memory.	A forest in a video game. Instead of storing data for 1,000 unique trees, the game stores one 'tree model' and just records the 1,000 different positions.

Structural Patterns:

Pattern	Intent	UML	Example	When to Use
Adapter	Convert interface	Adapter class	InputStreamReader	When interfaces are incompatible
Bridge	Separate abstraction from implementation	Abstraction + Implementor	Shape-Color	When abstraction and implementation vary independently
Composite	Tree structure of objects	Component + Leaf + Composite	FileSystem	When representing part-whole hierarchies
Façade	Simplify subsystem interface	Façade class	JDBC	When simplifying complex subsystems
Flyweight	Share common data	FlyweightFactory	Character objects	When many similar objects

				exist

UNIT–III

a) Short Questions (2 Marks)

1. What is a Design Pattern?
2. What is a Creational Pattern?
3. Define Singleton Pattern.
4. What is Factory Method Pattern?
5. What is Adapter Pattern?

b) Essay Questions (10 Marks)

1. Explain Design Patterns and their benefits.

2. Describe all Creational Design Patterns with suitable examples.
3. Explain Structural Design Patterns with neat diagrams.
4. Compare Singleton and Factory Method Patterns.
5. Explain Adapter, Bridge, Composite, and Facade Patterns with examples.

c) Case Study-Based Questions – 2

1. A banking application needs only one database connection throughout execution. Which Design Pattern is suitable? Explain.
2. A mobile application needs to support different payment gateways without changing existing code. Which Structural Pattern is suitable? Justify your answer.

Standard Quote:

"Think before you code. A good architecture today saves countless hours of maintenance tomorrow."

Unit 4: Behavioral Design Patterns

Behavioral patterns are concerned with algorithms and the assignment of responsibilities between objects.

Behavioral patterns use inheritance to distribute behavior between classes, or more commonly, use object composition to describe how a group of objects cooperate to perform a task that no single object can carry out alone.

Pattern	Problem & Definition	Real-World Analogy
Observer	**Definition:** Defines a subscription mechanism to notify multiple objects about any events that happen to the object they're observing. **Problem:** When one object changes state, and you need many other objects to update automatically without hard-coding them.	A newspaper subscription. Whenever the publisher prints a new edition, they automatically send it to everyone on their subscriber list.
Strategy	**Definition:** Defines a family of algorithms, puts each of them into a separate class, and makes their objects interchangeable. **Problem:** When you have many ways to do a task (like different sorting methods) and want to switch between them easily at runtime.	A GPS app. You can choose to get to your destination by car, walking, or public transit. The 'strategy' for the route changes, but the destination stays the same.
Iterator	**Definition:** Lets you traverse elements of a collection without exposing its underlying representation (list, stack, tree, etc.). **Problem:** You need to go through a list of items, but you don't want the code to care how that list is actually stored.	A TV remote 'Channel Up' button. You don't need to know the frequency or technology of the channel; you just want to go to the next one in the list.
Command	**Definition:** Turns a request into a stand-alone object that contains all information about the request. **Problem:** When you want to queue, delay, or undo actions.	A restaurant order slip. The waiter writes your request on a piece of paper. That paper (the command) is then passed to the chef, who executes it later.

State	**Definition:** Lets an object alter its behavior when its internal state changes. The object will appear to change its class. **Problem:** When an object has a massive 'if-else' or 'switch' statement that changes how it acts based on its current status.	A vending machine. If it's 'Out of Stock,' the button does nothing. If it's 'Waiting for Money,' the button asks for coins. If it has 'Credit,' the button drops a soda.

Pattern	Intent	UML	Example	When to Use
Chain of Responsibility	Pass request along chain	Handler chain	Logger chain	When multiple handlers may process a request
Command	Encapsulate request	Command + Invoker + Receiver	Undo/Redo	When requests need to be queued or logged
Interpreter	Define grammar	Expression tree	Expression evaluator	When language grammar is defined
Iterator	Sequential access	Iterator interface	Java Iterator	When traversing collections
Mediator	Centralize communication	Mediator + Colleagues	Chatroom mediator	When objects communicate indirectly
Memento	Capture state	Originator + Memento + Caretaker	Text editor undo	When restoring previous states
Observer	Notify dependents	Subject + Observer	GUI listeners	When one-to-many dependency

				exists
State	Change behavior by state	Context + State	TCP connection	When object behavior changes with state
Strategy	Select algorithm at runtime	Context + Strategy	Sorting strategies	When multiple algorithms exist
Template Method	Define algorithm skeleton	Abstract class template	Game framework	When steps of algorithm are fixed
Visitor	Add operations to structure	Visitor + Element	File system visitor	When new operations are needed without changing classes

Example (Observer in Java):

```
interface Observer { void update(); }
class Subject {
  List<Observer> observers = new ArrayList<>();
  void attach(Observer o) { observers.add(o); }
  void notifyAllObservers() { for (Observer o : observers) o.update(); }
}
```

Comparison: Distinguishing Similar Behavioral Patterns		
Pattern Pair	Primary Focus	Key Difference
Strategy vs. State	Both use composition to delegate behavior to a helper object.	In Strategy, the client usually chooses the algorithm. In State, the state transitions are often internal and automatic as the object's data changes.

Strategy vs. Template Method	Both allow for varying parts of an algorithm.	Strategy uses composition (object-based) to change the entire algorithm. Template Method uses inheritance (class-based) to change specific steps.
Mediator vs. Observer	Both address decoupling senders and receivers.	Mediator centralizes complex communication between many objects. Observer creates a distributed communication flow where subjects notify many listeners.

UNIT–IV

a) Short Questions (2 Marks)

1. What is a Behavioral Design Pattern?
2. Define Observer Pattern.
3. What is Strategy Pattern?
4. Define Iterator Pattern.
5. What is Command Pattern?

b) Essay Questions (10 Marks)

1. Explain Behavioral Design Patterns with suitable examples.
2. Describe Observer and Strategy Patterns with applications.
3. Explain Chain of Responsibility and Command Patterns.
4. Discuss State and Template Method Patterns.
5. Explain Visitor and Mediator Patterns with neat diagrams.

c) Case Study-Based Questions

1. A weather application automatically updates all subscribed users whenever weather information changes. Which Behavioral Pattern is suitable? Explain.
2. A navigation application allows users to choose Car, Bike, or Walking routes. Which Design Pattern can be used? Justify your answer.

Standard Quote:

"Think before you code. A good architecture today saves countless hours of maintenance tomorrow."

Prepared by M.L. Madhuri,
Assistant Professor SITAMS

Unit 5: Architectural Case Studies

1. A-7E Case Study

Focus: Utilizing architectural structures.

Diagram:

Control Module → Computation Module → Display Module

Lesson: Clear separation of control and computation improves maintainability.

2. World Wide Web

Architecture: Client-Server with HTTP protocol.

Diagram:

Browser ↔ Web Server ↔ Database

Focus: Interoperability via standard protocols.

3. Air Traffic Control

Goal: High availability and fault tolerance.

Techniques: Redundancy, failover, real-time monitoring.

Diagram:

Radar Systems → Control Servers → Backup Servers → Display Consoles

4. Celsius Tech

Focus: Product line development.

Architecture: Common core + variable modules.

Diagram:

Core Architecture → Naval System 1 / Naval System 2 / Naval System 3

Case Study	Architectural Focus and Solution
A-7E Avionics System	**Challenge:** Managing extreme complexity and ensuring the system could be easily modified as hardware or requirements changed. **Solution:** This case study focuses on the use of **architectural structures**. By using a strict module decomposition structure based on the principle of information hiding, developers were able to isolate changes. If the hardware changed, only the 'Hardware Hiding' module needed updates, leaving the rest of the software untouched.
The World Wide Web (WWW)	**Challenge:** Creating a system where millions of different devices, servers, and browsers can work together seamlessly across the globe. **Solution:** The primary challenge was **interoperability**. The solution was a client-server architecture built on standardized protocols (like HTTP) and data formats (like HTML). By enforcing a small set of constraints (Uniform Interface), the Web achieved massive scalability and allowed independent evolution of clients and servers.
Air Traffic Control (ATC)	**Challenge:** Preventing system failure at all costs, as a breakdown in air traffic software could lead to loss of life. **Solution:** The focus here was **high availability**. The architecture utilized redundancy and fault-tolerant patterns. By implementing 'shadow' or backup units that could take over instantly if a primary component failed, the system ensured that controllers would never lose their view of the airspace.

Celsius Tech	**Challenge:** Building multiple complex naval systems for different customers without starting from scratch every time. **Solution:** This is the definitive case study for **Software Product Lines**. Celsius Tech moved away from building one-off systems and instead created a common set of core assets. They developed a reusable architecture that could be configured to meet the specific needs of different ships, drastically reducing development time and costs.

UNIT–V

a) Short Questions (2 Marks)

1. What is interoperability?
2. What is high availability?
3. Name the case studies discussed in this unit.
4. What is Product Line Development?
5. What is Air Traffic Control System?

b) Essay Questions (10 Marks)

1. Explain the A-7E case study and its architectural significance.
2. Discuss the World Wide Web as a case study in interoperability.
3. Explain the Air Traffic Control System as a case study for high availability.
4. Describe CelsiusTech as a Product Line Development case study.
5. Compare the architectural lessons learned from all four case studies.

c) Case Study-Based Questions

1. An airline company wants to develop a highly reliable flight monitoring system. Which concepts from the Air Traffic Control case study can be applied? Explain.
2. A software company develops different versions of the same application for hospitals, banks, and schools. Explain how the CelsiusTech Product Line approach can help.

Standard Quote:

"Think before you code. A good architecture today saves countless hours of maintenance tomorrow."