Software Architecture

Lecture note for Seminar in Software Architecture (PIT9202)

Compiled by Gebremariam Assres

Note: This lecture note is structured based on the lecture materials and the book *Software Architecture in Practice* by Bass, L., Clements, P., & Kazman, R. (2022).

Table of Contents

1. Software architecture fundamentals

1.1 What is Software Architecture?

*The software architecture of a system is the **set of structures** needed to reason about the system, which comprise software elements, relations among them, and properties of both.*

This definition of architecture **contrasts** with others that emphasize **early or major decisions** in system design. While many decisions are indeed made **early**, not all of them qualify as architectural, and it is often difficult to determine whether a decision is truly major. In contrast, structures are more tangible and easier to identify, making them a powerful tool for designing systems. As such, software architecture is best understood as a set of software structures [1].

A **structure** itself is defined as **a collection of elements** bound together by specific relationships. Software systems consist of multiple structures, each contributing to the overall architecture, but no single structure can claim to fully represent the architecture on its own. Architectural structures fall into several categories, including **module** structures, **component-and-connector** structures, and **allocation** structures, each serving a unique purpose in system design and analysis.

Module Structures

Some architectural structures partition systems into implementation units known as **modules**. These modules are assigned specific **computational responsibilities**, forming the basis for work assignments among programming teams. In large projects, modules are further subdivided to allocate responsibilities effectively to programming sub-teams.

Module structures represent decisions on how to organize a system into distinct code or data units that need to be constructed or procured. The elements of these structures include modules of various types, such as **classes**, **layers**, or divisions of functionality, which serve as **units of implementation**. Modules are typically assigned specific areas of **functional responsibility**, with less emphasis on how the software operates at runtime. This approach ensures clarity and organization within the system, enabling efficient collaboration and scalability across teams.

Component-and-connector

Component-and-connector structures describe how system elements interact with each other. A **component** is a runtime entity that operates during execution to perform the system's functions. For instance, if a system is designed as a set of **services**, the services, their supporting infrastructure, and the synchronization and interaction relationships among them form a structure often used to describe the system.

These services are composed of programs within various implementation units, such as **modules**. The component-and-connector structure embodies decisions to organize a system as a collection of elements with **runtime behavior** (components) and their interactions (**connectors**). Components can include **services**, **peers**, **clients**, **servers**, **filters**, and other runtime elements. **Connectors** serve as communication mechanisms between components, utilizing methods such as **call-return**, **process synchronization operators**, and **pipes**.

Component-and-connector views enable architects to define and analyze runtime properties, including **performance**, **security**, **availability**, and other essential system qualities.

Allocation Structures

Allocation structures describe how software structures are mapped to the system's environments, including **organizational**, **developmental**, **installation**, and **execution** contexts. Modules are assigned to teams for development and integrated into specific locations in a file structure for implementation, testing, and deployment onto hardware. These mappings link software elements to the external environments where the software is created and executed.

A structure is considered **architectural** if it supports reasoning about the system and its properties. The reasoning should address attributes critical to stakeholders, such as **functionality**, **fault tolerance**, **ease of changes**, and **responsiveness** to user requests.

Architecture is an abstraction that focuses on software elements and their relationships. It deliberately omits details not relevant to system reasoning, such as private implementation specifics. This abstraction enables architects to concentrate on **arrangement**, **interaction**, **composition**, and **properties**, which are vital for managing architectural complexity.

Every system inherently has an **architecture**, comprising elements and their relationships. However, the architecture may not always be documented, especially if source code is lost or never delivered. Architecture can exist independently of its documentation, though proper **documentation** is critical for effective communication and reasoning.

The **behavior** of each element is integral to the architecture, as it illustrates how elements interact. While box-and-line diagrams provide a conceptual view, they must be complemented with detailed documentation of behavior and performance. Fine-grained behaviors below the architectural level may not be emphasized, but interactions influencing system acceptance are critical aspects of the architecture.

Physiological Structures

The **neurologist**, **orthopedist**, **hematologist**, and **dermatologist** each have distinct perspectives on the structure of the **same human body**. Similarly, **ophthalmologists**, **cardiologists**, and **podiatrists** focus on specific **subsystems** of the body.

The **kinesiologist** and **psychiatrist**, on the other hand, are concerned with different aspects of the body's **behavior**. Although these views may appear different and emphasize different properties, they are **interconnected** and together provide a comprehensive description of the **human body**. This analogy can be applied to **software**. Just as various medical specialists offer different viewpoints, **software** can be understood through different **structures and views**.

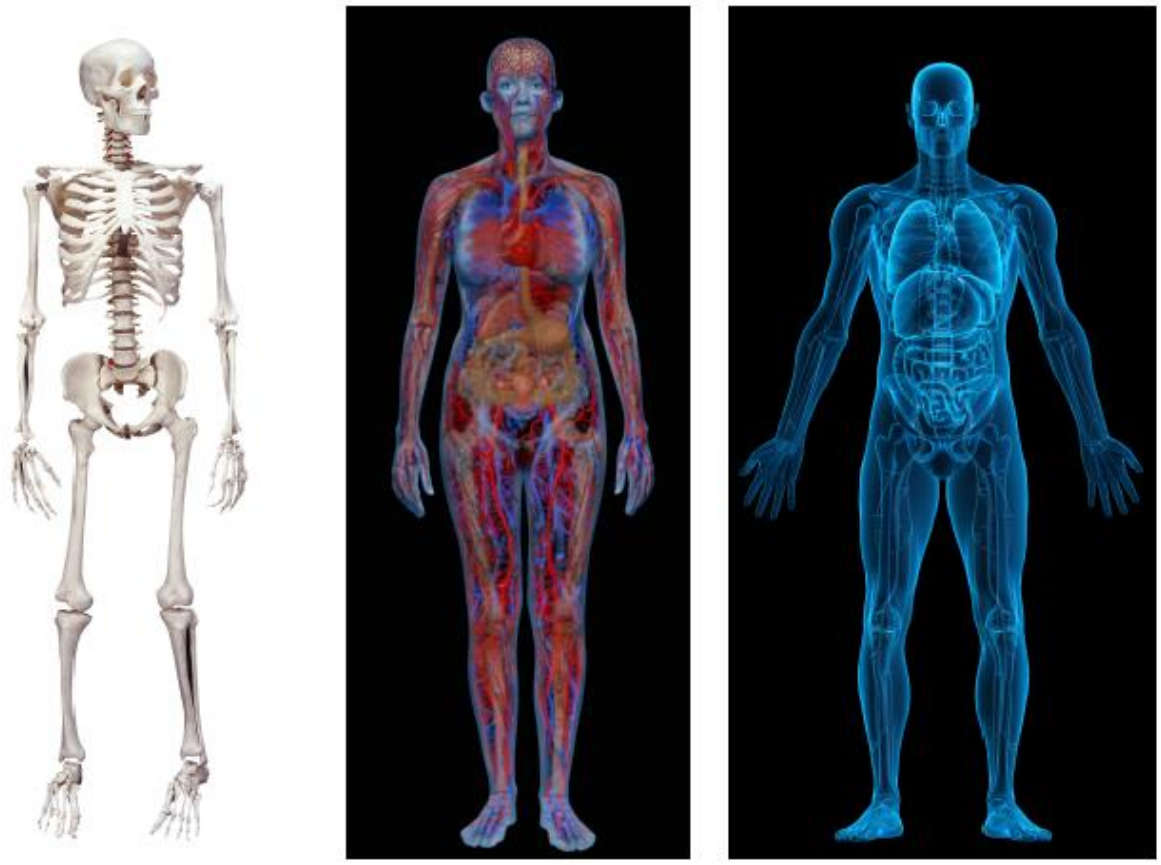

Fig. Physiological Structures

A **view** is a representation of a coherent set of **architectural elements** as perceived by various **stakeholders**. A **module structure** refers to the set of the system's **modules** and their **organization**. A **module view** is a representation of this structure, documented using a chosen **notation**, and is used by some **stakeholders**. **Architects** are responsible for designing these structures and documenting their views.

Each structure provides a unique perspective for reasoning about relevant **quality attributes**. For instance, the **module structure** is closely tied to how easily a system can be **extended** or **contracted**. The **concurrency structure** (parallelism) is essential for avoiding **deadlocks** and **performance bottlenecks**, while the **deployment structure** plays a crucial role in achieving **performance**, **availability**, and **security** goals.

Example of Module Structures

The **decomposition structure** focuses on how the system is broken down into **modules** that are related to each other through the **is-a-submodule-of** relationship. This structure illustrates how modules are recursively decomposed until they are small enough to be easily understood. Often, modules are associated with various products such as **interface specifications**, **code**, **test plans**, and more. The decomposition structure plays a vital role in determining the system's **modifiability**, ensuring that **likely changes** are localized. It is frequently used as the foundation for organizing development projects, including the structure of **documentation**, **integration**, and **test plans**. The units within this structure are typically named in a way that reflects the organization, using terms such as **segment** or **subsystem**.

In the **uses structure**, the units are also **modules**, or sometimes **classes**, which are related by the **uses relation**, a form of dependency. A unit uses another if the correctness of the first requires the presence of a correctly functioning version of the second, rather than a placeholder like a **stub**. This structure is particularly useful for engineering **systems** that can be extended to add functionality or from which useful **functional subsets** can be extracted. The ability to create subsets of a system facilitates **incremental development**, allowing for flexibility as the system evolves.

The **layer structure** consists of modules referred to as **layers**, which act as abstract "virtual machines" providing a cohesive set of services through a managed interface. **Layers** can use other layers, but only in a strictly controlled manner. In strictly layered systems, each layer can only use a single other layer. This structure contributes to the **portability** of a system, enabling it to change its underlying **computing platform** without significant disruption.

In the **class (or generalization) structure**, the modules are referred to as **classes**, with a relationship of **inherits from** or **is an instance of**. This view helps to reason about collections of similar **behavior** or **capability**. For instance, it can be used to examine the classes from which other classes inherit and the parameterized differences between them. The class structure is valuable for thinking about **reuse** and the incremental addition of functionality. If a project has undergone an **object-oriented analysis** and design process, this structure is typically reflected in the project's documentation.

The **data model** describes the system's static **information structure**, focusing on **data entities** and their relationships. For example, in a **banking system**, entities could include **Account**,

Customer, and **Loan**. An **Account** entity might have attributes such as **account number**, **type** (savings or checking), **status**, and **current balance**, providing a clear picture of how data is organized and interrelated within the system.

Example component-and-connector Structures

The **component-and-connector** structure illustrates how different components are connected and interact with each other within a system. These connectors can be familiar constructs, such as **"invokes"**, which represent how one component triggers actions in another. Several useful **component-and-connector structures** exist, each offering different ways of organizing and connecting system components.

One such structure is the **service structure**, where the units are **services** that interact with each other through service coordination mechanisms like **REST** or **SOAP**. This structure is particularly valuable when engineering a system composed of components that may have been developed **anonymously** and **independently**. It provides a framework for ensuring that these loosely coupled components can still work together effectively.

Another important structure is the **concurrency structure**, which helps identify opportunities for **parallelism** and pinpoint potential locations where **resource contention** may arise. In this structure, the units are the **components**, and the connectors are the **communication mechanisms** used to facilitate interaction between them. These components are organized into logical **threads**, enabling the system to handle multiple tasks simultaneously and efficiently.

Example Allocation Structures

The **deployment structure** illustrates how software is assigned to **hardware processing** and **communication elements** within a system. The key elements in this structure include **software elements** (typically processes from the **C&C view**), **hardware entities** like **processors**, and various **communication pathways**. The relationships in this structure are represented by the **allocated-to** relation, which indicates the physical units on which the software elements reside, and **migrates-to** when the allocation is dynamic. This structure is essential for reasoning about critical factors such as **performance**, **data integrity**, **security**, and **availability**. It is especially important in the context of **distributed** and **parallel systems**, where the allocation of resources can significantly impact system behavior.

The **implementation structure** shows how software elements, typically **modules**, are mapped to the **file structure(s)** in the system's development, **integration**, or **configuration control** environments. This mapping is crucial for managing **development activities** and **build processes**, ensuring that the system is properly organized and maintainable throughout its lifecycle.

The **work assignment structure** is responsible for assigning responsibility for implementing and integrating the modules to the respective teams. By incorporating this structure into the overall architecture, it becomes clear that decisions regarding work allocation have both **architectural** and **management** implications. The **architect** will understand the required **expertise** for each team, and this structure also plays a role in determining the major **communication pathways** among teams, such as **teleconferences**, **wikis**, **email lists**, and other communication tools.

Relating Structures to Each Other

Elements from one structure are often related to elements from other structures, and it is important to reason about these **relations** to understand how they interact. For example, a **module** in a **decomposition structure** may be represented as one, part of one, or several **components** in one of the **component-and-connector structures**. In general, the mappings between structures are **many-to-many**, meaning that a single element in one structure can correspond to multiple elements in another, reflecting the complexity and interconnectivity of the system's architecture.

1.2 Architectural Patterns

Architectural elements can be composed to address specific problems, and over time, certain **compositions** have proven useful across various **domains**. These compositions have been documented and widely disseminated as **architectural patterns**, which offer packaged strategies to solve common problems faced by a system. **Patterns** define the types of **elements** and the forms of **interaction** used to solve the problem at hand.

A common **module-type pattern** is the **layered pattern**, which arises when the **uses relation** among software elements is strictly **unidirectional**. In this case, a system of **layers** emerges, with each layer representing a coherent set of related functionalities. While this pattern is often applied with strict structural restrictions, many variations exist in practice that reduce these constraints.

There are also several common **component-and-connector type patterns**. One such pattern is the **shared-data** (or **repository**) pattern, which involves components and connectors that create, store, and access **persistent data**. Repositories in this pattern are typically implemented as a **database**, often a **commercial database**, with connectors being protocols for managing the data, such as **SQL**. Another widely used pattern is the **client-server** pattern, where components are categorized as **clients** and **servers**. The connectors in this pattern are the **protocols** and **messages** exchanged between clients and servers to carry out the system's work.

In terms of **allocation patterns**, the **multi-tier pattern** distributes and allocates components of a system into distinct subsets of **hardware** and **software**, linked by a communication medium. This pattern is a specialized form of the generic **deployment structure** (software-to-hardware

allocation). Additionally, the **competence center pattern** and **platform pattern** specialize in the **work assignment structure** of a software system.

In the **competence** center pattern, work is allocated to sites based on the **technical** or **domain expertise** available at each site. Meanwhile, the **platform** pattern focuses on one site developing reusable **core assets** for a software product line, while other sites develop applications that leverage these core assets.

1.3 What Makes a “Good” Architecture?

There is no such thing as an absolute **good** or **bad** architecture. Instead, architectures are evaluated based on how **fit** they are for a specific purpose or set of goals. They can be assessed, but only in the context of clearly defined and stated objectives. However, there are some generally accepted **rules of thumb** that guide the creation of effective architectures.

One such **rule of thumb** is that architecture should ideally be the product of a **single architect** or a small group of architects with an identified **technical leader**. This approach ensures that the architecture maintains its **conceptual integrity** and **technical consistency**. This recommendation holds true for various types of projects, including **Agile**, **open-source**, and **traditional** projects. Additionally, there should be a strong connection between the **architect(s)** and the **development team** to ensure effective communication and alignment.

The architecture should be built upon a **prioritized list** of well-specified **quality attribute requirements**. It should also be documented using **views** that address the concerns of the most important **stakeholders**, supporting the project’s **timeline**. The architecture must be evaluated for its ability to deliver the system’s key **quality attributes**, and this evaluation should be conducted early in the **life cycle** and repeated as needed. Moreover, the architecture should support **incremental implementation**, starting with the creation of a **"skeletal" system** in which the **communication paths** are exercised with minimal functionality before adding more complex features.

Structural “Rules of Thumb”

The **architecture** should be designed with well-defined **modules**, where functional responsibilities are assigned based on key principles. One such principle is **information hiding**, which means that modules should encapsulate elements that are likely to change, protecting the rest of the system from those changes. Another important principle is the **separation of concerns**, where modules have **well-defined interfaces** that separate changeable aspects from more stable components of the system. Achieving **quality attributes** should rely on well-known **patterns** and **tactics** that are specific to each attribute, ensuring consistency and reliability throughout the system.

The architecture should be flexible and not dependent on any particular version of a product, ensuring that it is structured in such a way that version changes can be made **easily** and at low cost. Additionally, **modules** responsible for producing data should be separated from those that consume data. This separation increases **modifiability**, as changes are confined to either the production or consumption side of the data flow. It's also important to note that there is **no one-to-one correspondence** between **modules** and **components**.

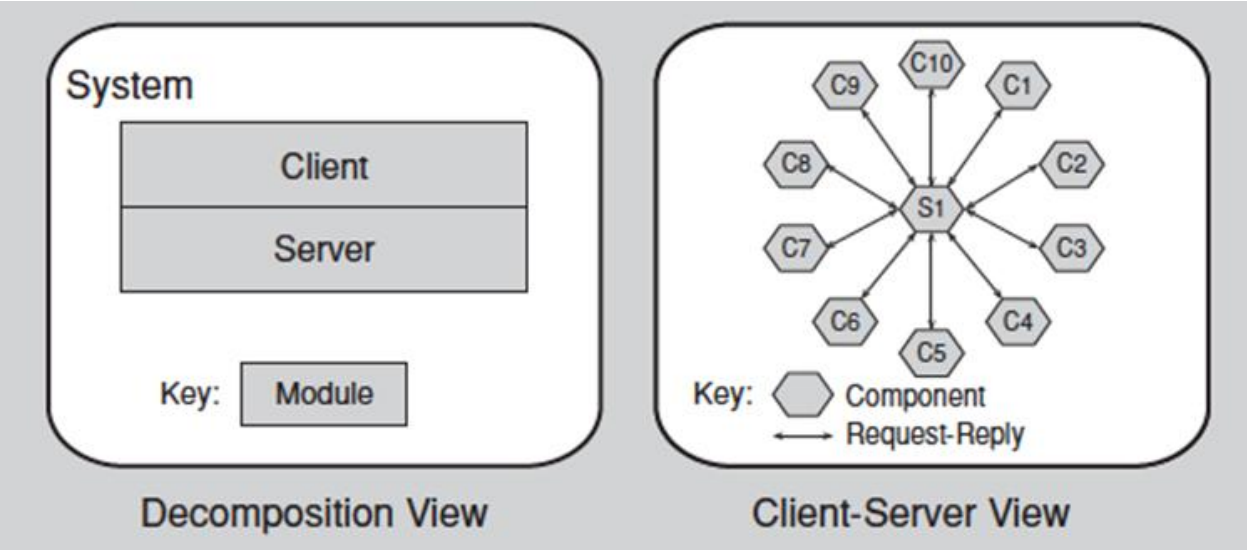

Fig. Component view vs. Module view

The **process** should be designed in such a way that its assignment to a specific **processor** can be easily changed, possibly even at **runtime**, enhancing flexibility. The architecture should also feature a small number of ways for **components** to interact, which helps the system maintain consistency. This design choice ensures that the system behaves in a consistent manner, performing the same functions in the same way throughout. Such consistency improves **understandability**, reduces **development time**, increases **reliability**, and enhances **modifiability**.

Finally, the architecture should contain a specific, limited set of **resource contention areas**, with clear resolutions that are well-defined and maintained, ensuring the system operates smoothly even under varying loads and configurations.

1.4 Why is Software Architecture Important?

The **architecture** of a system plays a crucial role in whether it can exhibit its required **quality attributes**, as these attributes are determined by the architecture itself. For **performance**, the architecture must manage time-based behavior, shared resources, and the frequency and volume of inter-element communication. In terms of **modifiability**, responsibilities should be

assigned to elements in a way that ensures most changes will only affect a small number of elements. **Security** concerns are addressed by managing and protecting inter-element communication, controlling access, and establishing authorization mechanisms. For **scalability**, the architecture should localize resource use to facilitate easy replacements and avoid hardcoding resource assumptions or limits. **Incremental subset delivery** is managed by controlling inter-component usage, while **reusability** is enhanced by restricting inter-element coupling, making it easier to extract elements without being burdened by excessive dependencies.

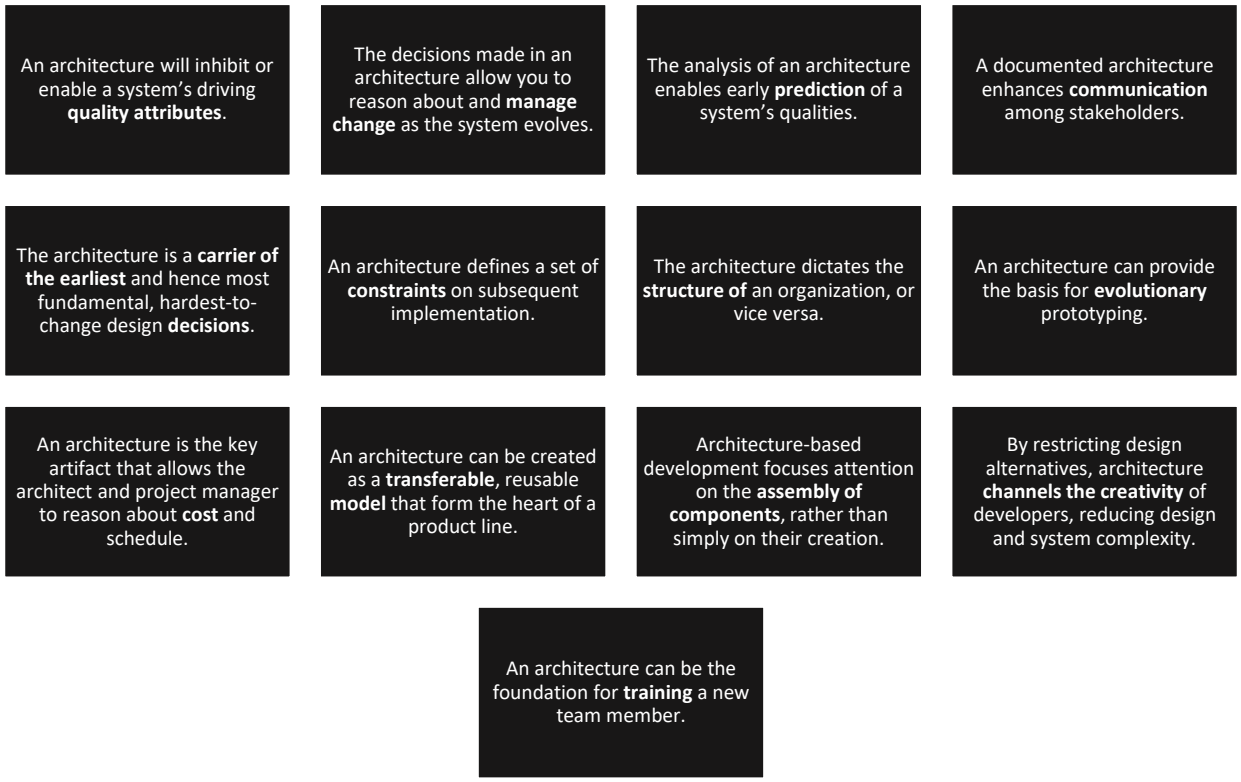

Fig. Why is Software Architecture Important?

When it comes to **reasoning about and managing change**, it's important to recognize that approximately **80% of software's total cost** occurs after deployment, as systems need to accommodate new features, adapt to new environments, and fix bugs. Architecture helps partition changes into three categories: **local changes**, which can be handled by modifying a single element; **nonlocal changes**, which require modifications across multiple elements while maintaining the overall architectural approach; and **architectural changes**, which affect how elements interact and often require system-wide modifications. **Local changes** are the most desirable as they are easier to implement, and a good architecture ensures that the most common changes are local.

Predicting system qualities is achievable because architectural decisions lead to specific quality attributes. By examining the architecture, we can confirm that these decisions have been made and predict that the system will exhibit the desired qualities. The earlier a **problem** is identified in the design, the cheaper, easier, and less disruptive it will be to fix. Architecture also facilitates **communication among stakeholders** by providing a common abstraction that can be understood by both technical and nontechnical people. Stakeholders like customers, users, managers, and architects each focus on different aspects of the system, such as performance, cost, schedule, and team coordination. **Architecture** helps bridge these concerns by offering a common language for negotiation and resolution, allowing for mutual understanding.

The **earliest design decisions** have a profound impact on the system's development, deployment, and maintenance. These decisions, such as whether the system will run on a single processor or be distributed, dictate critical aspects of the system. Changes to these early decisions can be challenging and costly. **Constraints** on an implementation ensure that the software conforms to the architectural design, with elements interacting in prescribed ways and fulfilling their responsibilities. The architect must manage these constraints to meet performance budgets and ensure that each unit stays within its limits.

Architecture also influences the **organizational structure**, as it prescribes the structure of the development project and even the organization itself. It forms the basis for the work-breakdown structure, which dictates planning, scheduling, budgeting, team communication, configuration control, and testing. If responsibilities are formalized, changing them could be costly. A well-defined architecture enables **evolutionary prototyping**, where a skeletal system is built early in the development cycle, allowing for early identification of potential **performance issues** and reducing project risks.

In terms of **cost and schedule estimates**, architects help project managers create accurate estimates by combining **top-down** and **bottom-up** approaches, with the latter being more accurate and fostering a sense of ownership among developers. **Reuse** is another key benefit of architecture, as it allows systems with similar requirements to leverage existing assets, including code, requirements, and design patterns. A **software product line** capitalizes on reusable architecture, which defines what is fixed for all members of the product line and what is variable.

Finally, **architecture-based development** often incorporates independently developed components, such as commercial off-the-shelf software, open-source software, and networked services. This approach can decrease time to market, increase reliability, lower costs, and provide flexibility. By **restricting the design vocabulary** to a small set of patterns, architects can reduce system complexity, enhance **reuse**, and create more regular, understandable, and communicable designs. **Patterns** guide architects to focus on quality attributes, as the properties of software design follow from the choice of an architectural pattern. Architecture

can also serve as a **training tool**, providing new project members with an introduction to the system, the team structure, and how the system is expected to function.

1.5 Discussion and Questions

- How do different architectural perspectives, such as modules, components, and allocation structures, contribute to the overall design and development of a software system?
- In what ways does a documented architecture enhance communication among stakeholders and aid in predicting system qualities?
- How can software architecture influence organizational structures and project management, particularly in terms of cost and schedule estimation?
- Why is focusing on the assembly of components, rather than their individual creation, considered a key benefit of architecture in reducing complexity and fostering creativity?
- How can reusable architectural models and frameworks serve as training foundations for new team members and support the development of product lines?

2. Quality attributes of software systems

2.1 Understanding Quality Attributes

Understanding Quality Attributes is essential for effective system design and architecture. **System requirements** can be classified into three categories: **functional requirements**, which define system tasks and reactions to runtime stimuli; **quality attribute requirements**, which annotate and qualify functional requirements; and **constraints**, which are decisions with no degree of freedom, such as screen size.

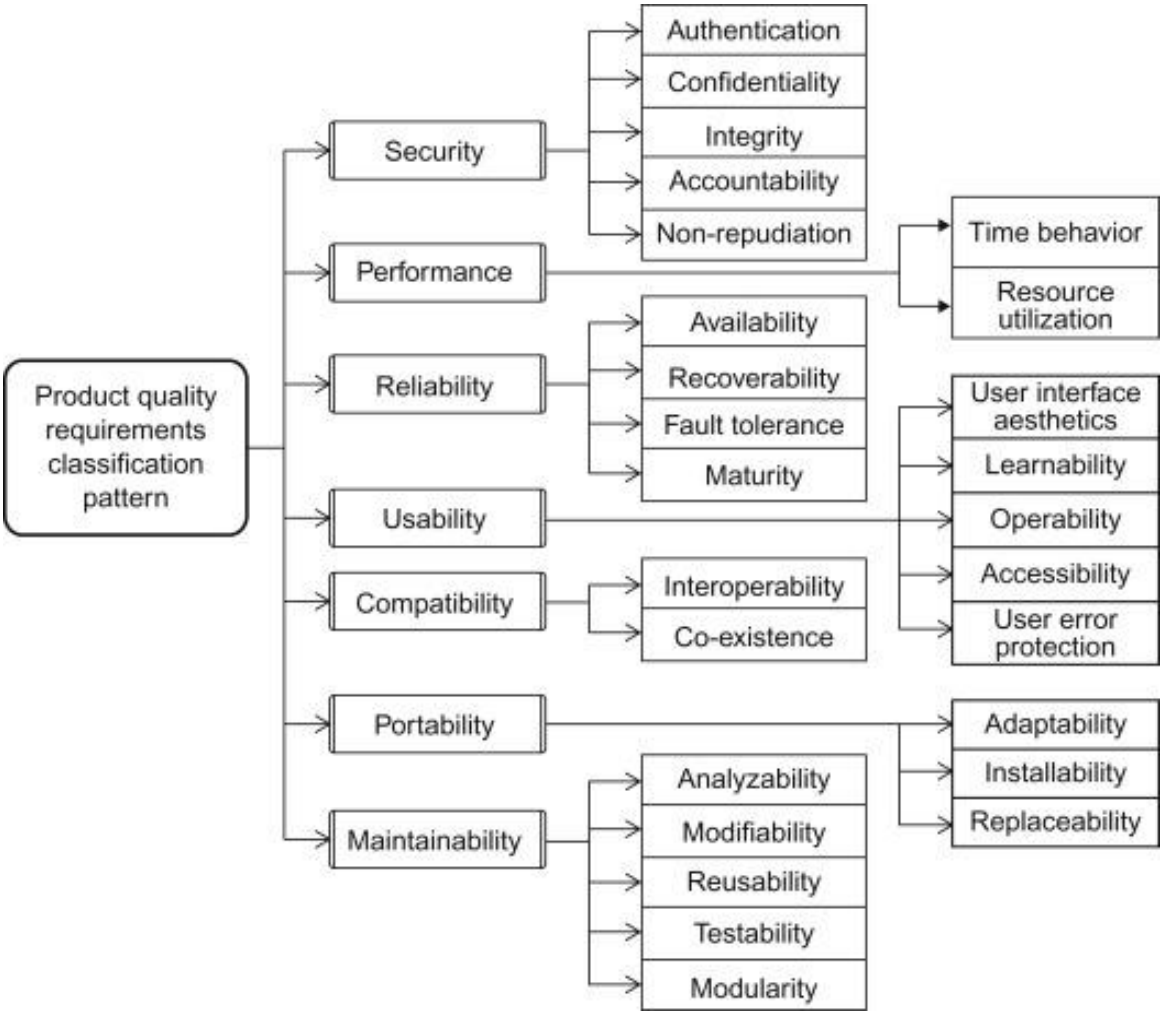

Fig. Categories system requirements

Functionality refers to the system's ability to perform its intended tasks. Interestingly, functionality has a **complex relationship** with architecture. While functionality does not dictate

architectural design, architectural choices can influence how functionality is achieved. Moreover, functionality and **quality attributes** are orthogonal to each other.

A **quality attribute (QA)** can be understood through annotations. For instance, a functional requirement like, "When the user presses the green button, the Options dialog appears," could be annotated with quality attributes such as **performance** (how quickly the dialog appears), **availability** (frequency of failure or speed of repair), or **usability** (ease of learning and using the function).

To **specify QA requirements**, scenarios are employed. These scenarios include six key parts:

1. **Source of stimulus**—the entity generating the stimulus, such as a human or a computer system.
2. **Stimulus**—a condition requiring a response when it arrives at the system.
3. **Environment**—the conditions under which the stimulus occurs, such as during normal operation or overload.
4. **Artifact**—the part of the system affected, which could range from the entire system to a specific component.
5. **Response**—the activity the system undertakes as a reaction.
6. **Response measure**—a measurable outcome ensuring the requirement can be tested.

Achieving QA through tactics involves using design primitives known as **tactics**, which architects have relied on for years. Tactics serve three main purposes:

1. They augment or adapt existing **design patterns** to meet QA goals.
2. When no suitable pattern exists, they enable architects to design solutions from first principles.
3. Cataloging tactics makes design more systematic, offering multiple options to improve a quality attribute. The choice of tactics depends on trade-offs, cost, and other influencing factors.

Guiding quality design decisions involves a systematic approach to key architectural considerations. These include:

1. **Allocation of responsibilities**, such as identifying system functions, infrastructure, and QA satisfaction, and assigning these to modules, components, and connectors.

2. **Coordination model**, which identifies system elements needing coordination, determines coordination properties like timeliness and consistency, and selects communication mechanisms (e.g., stateful vs. stateless, synchronous vs. asynchronous).
3. **Data model** decisions, which define data abstractions, metadata, and data organization (e.g., relational databases or object collections).
4. **Management of resources**, which involves identifying resources, their limits, and how they are shared or arbitrated during contention.
5. **Mapping among architectural elements**, such as linking modules to runtime elements, processors, and delivery units.
6. **Binding time decisions**, which determine when decisions are finalized, such as build-time module selection or runtime protocol negotiation.
7. **Choice of technology**, where architects evaluate available technologies, supporting tools, internal familiarity, and external support. They must also account for compatibility with existing technology stacks and side effects, such as resource constraints or required coordination models.

Incorporating these principles ensures an architecture that balances functionality, quality attributes, and adaptability while meeting stakeholder needs and system goals.

2.2 Availability

Availability is the property of software that ensures it is operational and ready to perform its tasks when required. This concept builds on **reliability** by incorporating the notion of **recovery** or **repair**, emphasizing the minimization of service outage time through fault mitigation. For instance, in a sample availability scenario, if a heartbeat monitor detects that a server is nonresponsive during normal operations, the system informs the operator and continues functioning without downtime.

The **goal of availability tactics** is to prevent faults from becoming observable failures or to limit the impact of faults until they can be repaired. While faults may have the potential to cause failures, availability tactics enable systems to endure and mitigate these faults effectively.

Recovering from faults involves preparation and repair through various mechanisms. **Active redundancy (hot spare)** ensures all nodes in a protection group process identical inputs in parallel, maintaining synchronous states. In **passive redundancy (warm spare)**, periodic state updates keep redundant spares ready, while **cold spares** remain idle until activated through a fail-over process. Techniques like **exception handling** address faults by reporting or masking them, and **rollback** reverts the system to a known good state. **Software upgrades** allow for non-disruptive updates, while **retry** resolves transient failures by reattempting operations.

Other tactics include **ignoring faulty behavior**, **degradation** (prioritizing critical functions over less critical ones during component failure), and **reconfiguration**, which reassigns responsibilities to functioning resources.

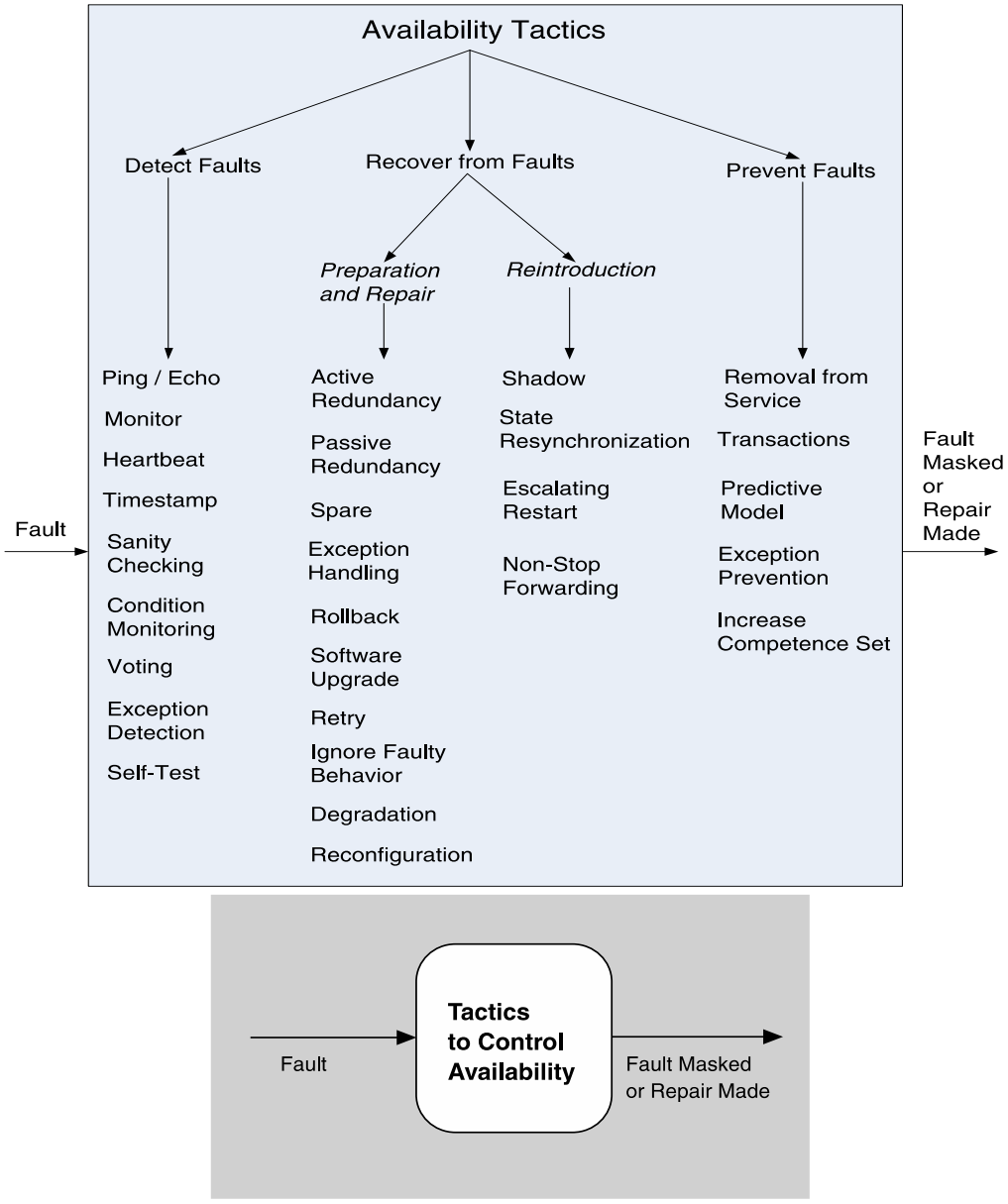

Fig. Goal of Availability Tactics

Reintroduction of components after fault recovery involves strategies like operating in **shadow mode**, where a previously failed or upgraded component runs in the background for validation before becoming active again. **State resynchronization** ensures standby components receive updated state information, complementing redundancy tactics. **Escalating restarts** progressively recover from faults by restarting components at varying granularities, minimizing

service disruption. **Non-stop forwarding** separates supervisory and data functions, allowing packet forwarding to continue even if the supervisory component fails.

Preventing faults focuses on proactive measures. For example, **removal from service** temporarily takes components offline to prevent potential failures. **Transactions** bundle state updates to ensure atomicity, consistency, isolation, and durability. A **predictive model** monitors system health and initiates corrective actions to avoid future issues. **Exception prevention** involves masking faults through techniques like smart pointers and wrappers. Finally, **increasing the competence set** of a component enables it to handle a broader range of scenarios, including faults, as part of its normal operation.

These availability tactics collectively ensure robust fault management, enabling systems to maintain functionality, recover efficiently, and prevent failures proactively.

2.3 Interoperability

Interoperability refers to the degree to which two or more systems can meaningfully exchange information. It is not a binary attribute but exists along a spectrum, offering varying levels of effectiveness in communication and integration. For instance, consider a vehicle information system that sends its current location to a traffic monitoring system. The traffic monitoring system processes this data, overlays it on a Google Map along with other relevant information, and broadcasts it. The accuracy of including the vehicle's location in the final broadcast is an example of interoperability, with a success rate of **99.9%** demonstrating a high level of effectiveness.

The **goal of interoperability tactics** is to ensure systems can exchange information effectively. Achieving this requires two primary conditions. First, systems must **know about each other**, which involves discovering and locating services. Second, systems must exchange information in a **semantically meaningful manner**, which includes providing services in the correct sequence and modifying information from one system into a format usable by another.

To establish **location**, systems may rely on a known **directory service** or use multiple levels of indirection, where one location points to another directory that can be searched for the desired service. Once systems are connected, the focus shifts to **managing interfaces** to facilitate effective communication. This can involve **orchestrating** interactions using a control mechanism to coordinate, manage, and sequence the invocation of services, particularly for complex tasks.

Additionally, systems may require **tailored interfaces**, where capabilities are added or removed to support specific needs. Tailoring may include translation, buffering, or data-smoothing to ensure seamless communication between systems.

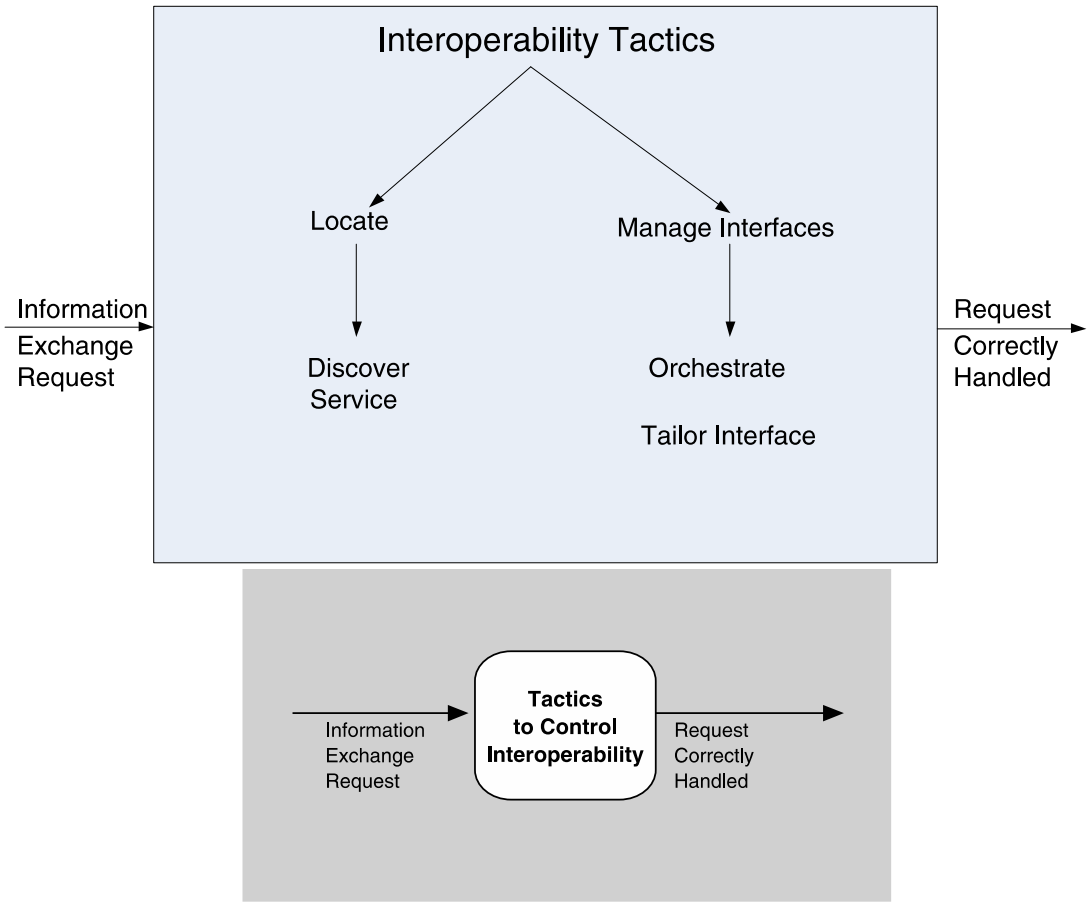

Fig. Goal of Interoperability Tactics

2.4 Modifiability

Modifiability refers to the ability to accommodate change, along with the associated costs and risks. An architect must address three critical questions: **what can change**, **what is the likelihood of the change**, and **when and by whom is the change made**. For example, consider a concrete modifiability scenario: a developer wishes to modify the user interface by changing the code during the design phase. These changes are completed without unintended side effects and within three hours.

The **goal of modifiability tactics** is to control the complexity, time, and cost involved in implementing changes. One approach is to **reduce the size of a module** by splitting it into smaller, more manageable parts. When a module has high capability, its modification costs are higher, so breaking it into smaller modules can lower the cost of change. Another tactic involves **increasing cohesion** by ensuring **semantic coherence**—if two responsibilities, A and B, within a module serve different purposes, they should be separated into distinct modules.

Deferring binding decisions until later in the lifecycle is another effective strategy. While artifacts designed with built-in flexibility are generally more cost-effective than coding for specific changes, implementing such mechanisms for late binding often incurs higher upfront costs.

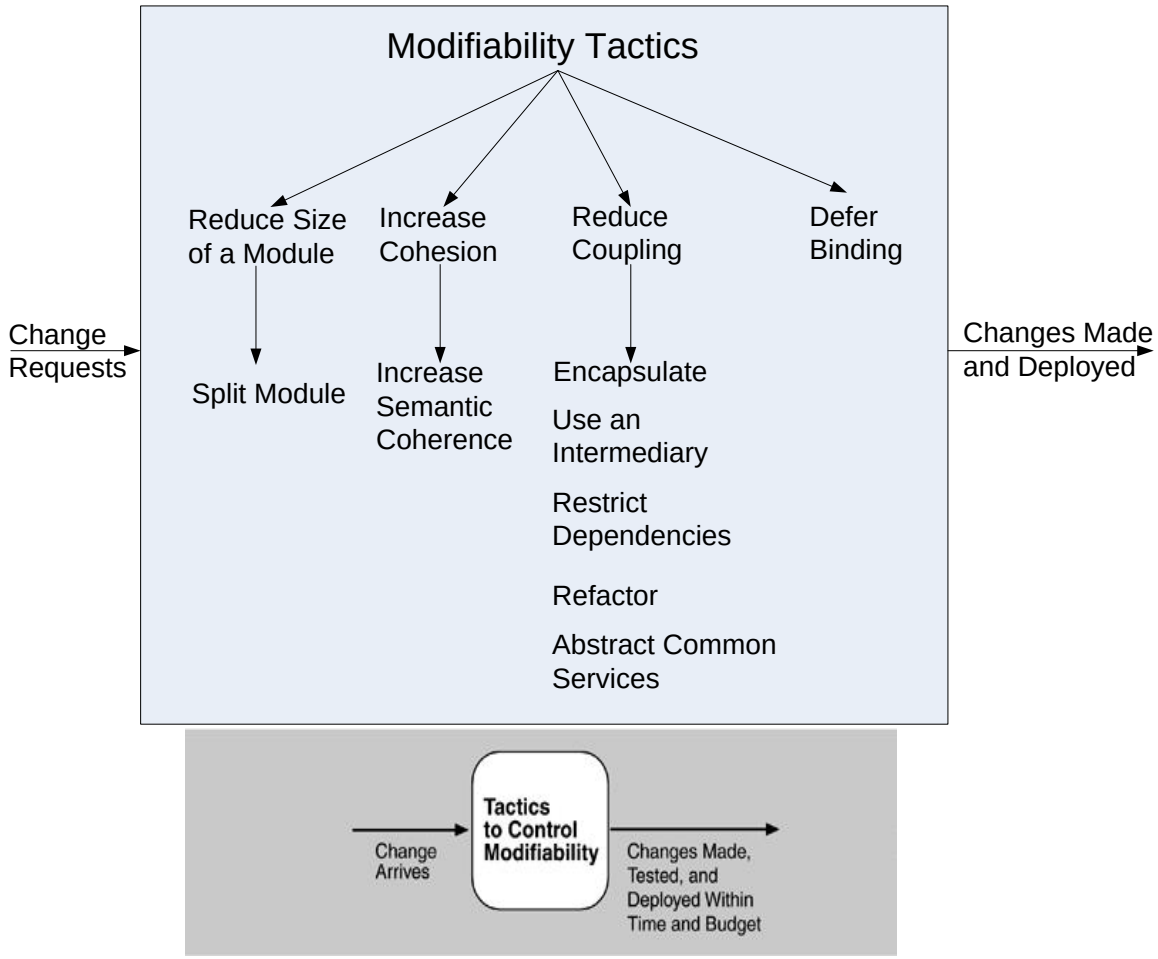

Fig. Goal of Modifiability Tactics

Reducing coupling is also essential for modifiability. **Encapsulation** introduces explicit interfaces for modules, such as an API that translates input parameters into internal representations. Using an **intermediary** can break dependencies, like when responsibility A depends on B. Architects can also **restrict dependencies** by limiting the modules a given module interacts with. **Refactoring** is useful when two modules are affected by the same change because they share redundant or duplicate code [10). Additionally, **abstracting common services** can reduce duplication and costs by consolidating similar but distinct services into a single, generic implementation.

2.5 Performance

Performance refers to a system's ability to meet **time requirements**. It focuses on the system's response to events like **interrupts** or **user requests**, ensuring it reacts within a specified time frame. The key aspect of discussing performance is characterizing the **timing** of events and the system's **time-based response**. For instance, in a concrete **performance scenario**, users may initiate transactions under normal conditions, and the system processes these transactions with an average latency of **two seconds**.

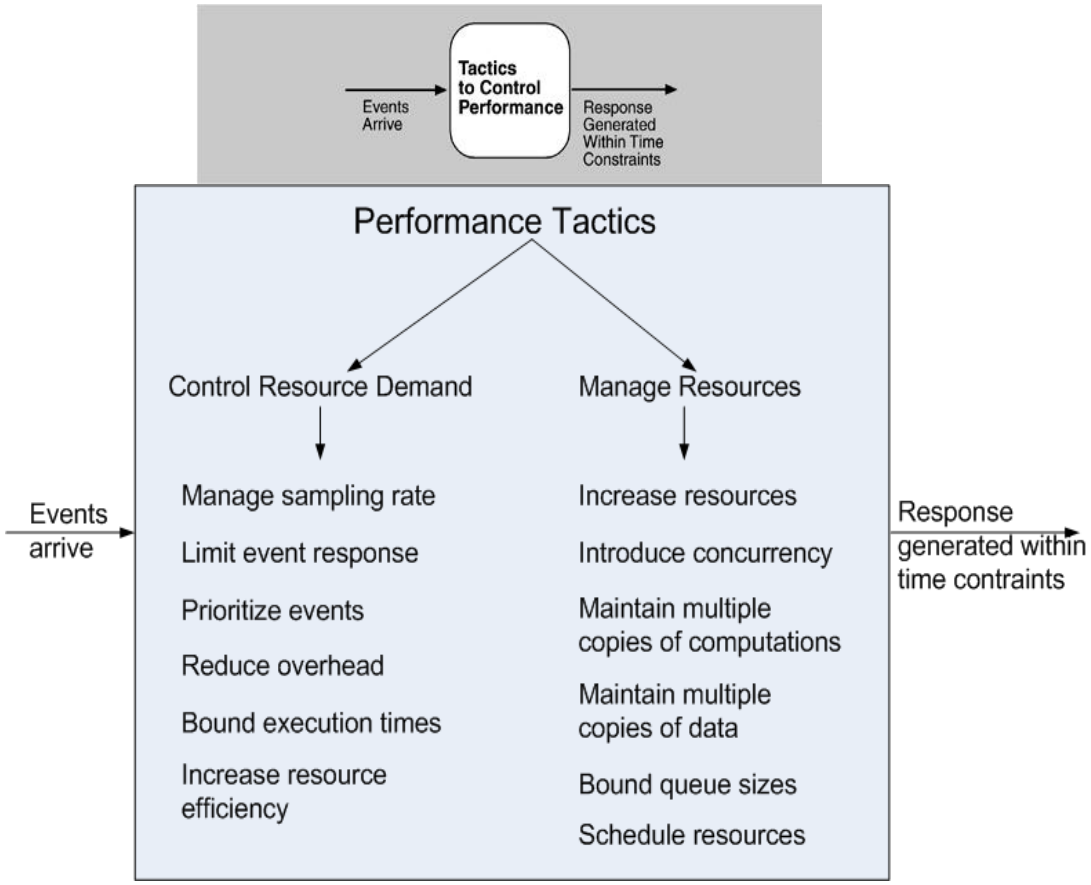

Fig. Goal of Performance Tactics

The goal of **performance tactics** is to ensure that the system responds to events within a specific **time-based constraint**. There are several tactics to control **performance**. One such tactic is to **control resource demand**, which can be done by reducing the **sampling rate** of a data stream, thereby decreasing the demand but with some **loss of fidelity**. **Limiting event response** ensures that events are processed only up to a certain maximum rate, ensuring **predictable processing**. **Prioritizing events** involves imposing a **priority scheme** to rank events by their importance, allowing the system to service higher-priority events first. **Reducing overhead** is another tactic, as intermediaries, while useful for **modifiability**, can increase the **resources consumed** and negatively impact **latency**. **Bounding execution times** involves

limiting the amount of time spent processing individual events, ensuring that the system does not exceed the time constraints. Additionally, improving the **efficiency of algorithms** in critical areas can help in **reducing latency**.

Another set of tactics involves **managing resources**. One way to improve performance is to **increase resources**, such as by adding **faster processors**, more **memory**, or **faster networks**, all of which can help reduce **latency**. **Increasing concurrency** enables the system to process multiple requests in parallel, thereby reducing the **blocked time**. **Maintaining multiple copies of computations** can reduce contention, especially when all computations are confined to a single server. Keeping **multiple copies of data** at different access speeds can also improve performance. Lastly, **bounding queue sizes** ensures that the number of **queued arrivals** is controlled, which in turn manages the **resources** used to process them. When contention arises for a resource, **scheduling resources** efficiently is crucial to avoid bottlenecks.

2.6 Security

Security is a measure of a system's ability to protect data and information from unauthorized access while still allowing access to authorized users and systems. An action taken against a system with harmful intent is called an **attack**, which can take various forms, including unauthorized attempts to access data or services, modify data, or deny services to legitimate users.

Security encompasses three main characteristics, commonly referred to as **CIA**: **Confidentiality**, **Integrity**, and **Availability**. **Confidentiality** ensures that data or services are protected from unauthorized access, for example, preventing a hacker from accessing your income tax returns on a government system. **Integrity** guarantees that data or services are not subject to unauthorized manipulation, ensuring that things like your grades remain unchanged after being assigned. **Availability** ensures that the system is available for legitimate use, like ensuring that a denial-of-service attack does not prevent you from accessing online services.

Additional characteristics that support these principles include **Authentication**, which verifies the identities of the parties involved in a transaction, and **Nonrepudiation**, which ensures that a sender cannot deny sending a message and that the recipient cannot deny receiving it. **Authorization** grants users the privileges needed to perform specific tasks, such as allowing a legitimate user to access their online banking account. For example, in a concrete security scenario, a disgruntled employee may attempt to modify the pay rate table remotely during normal operations, but the system maintains an **audit trail**, and the correct data is restored within a day.

The **goal of security tactics** is to ensure the protection of data and services from unauthorized access and attacks. **Physical security** is one method, which involves securing installations by

limiting access, detecting intruders, and having deterrence mechanisms like armed guards and recovery mechanisms such as off-site backups. Security tactics can be grouped into four categories: **detect**, **resist**, **react**, and **recover**. To **detect attacks**, tactics include comparing network traffic or request patterns to known malicious behavior signatures, detecting service denial by analyzing patterns of incoming network traffic for signs of denial-of-service attacks, verifying message integrity using techniques like checksums or hash values, and detecting message delays by analyzing suspicious timing behavior.

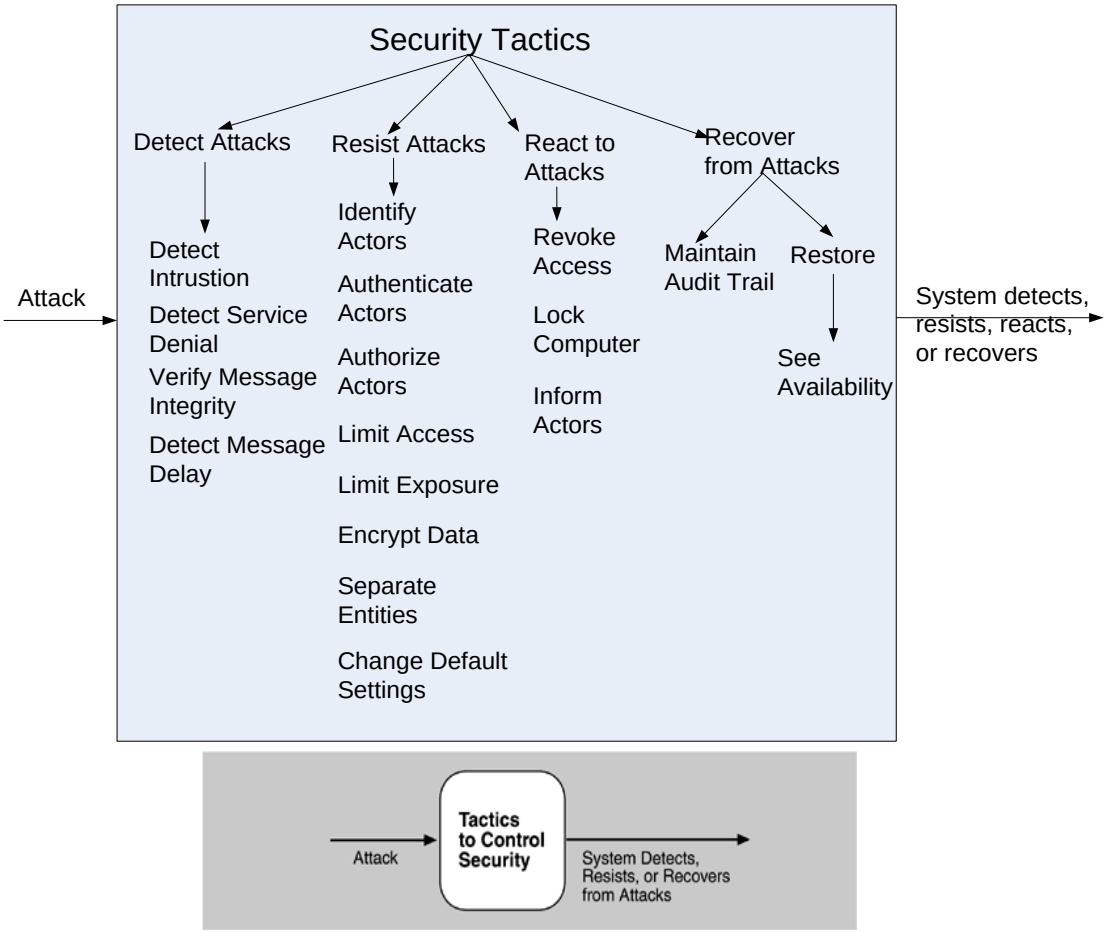

Fig. Goal of Security Tactics

To **resist attacks**, tactics include identifying actors (such as users or remote computers), **authenticating** them to ensure they are who they claim to be, and **authorizing** them to access or modify data and services as needed. Other tactics include **limiting access** to resources, minimizing the **attack surface** by reducing the number of access points, **encrypting data** to secure communication, separating entities through physical separation on different servers or virtual machines, and ensuring that users change **default settings** to enhance security.

In the event of an attack, the system must **react** by **revoking access** to sensitive resources, even for normally legitimate users, if an attack is suspected, and **locking computers** after repeated

failed access attempts. Additionally, it is essential to **inform actors** by notifying operators or cooperating systems when an attack is suspected or detected.

Finally, after an attack, the system must **recover**. This involves using **availability tactics** to recover failed resources, as well as conducting an **audit** to keep a record of actions taken by users and systems. The audit helps trace the actions of an attacker and identify the source of the breach.

2.7 Testability

Testability refers to the ease with which software can be tested to demonstrate its faults through **execution-based testing**. It is the probability that a system will fail during its next test execution. When a fault is present in a system, we want it to fail as quickly as possible during testing. For a system to be **testable**, it must be possible to control the inputs to each component and observe its outputs. A concrete example of **testability** would be when a unit tester completes a code unit during development and performs a test sequence that gives 85% path coverage within three hours of testing.

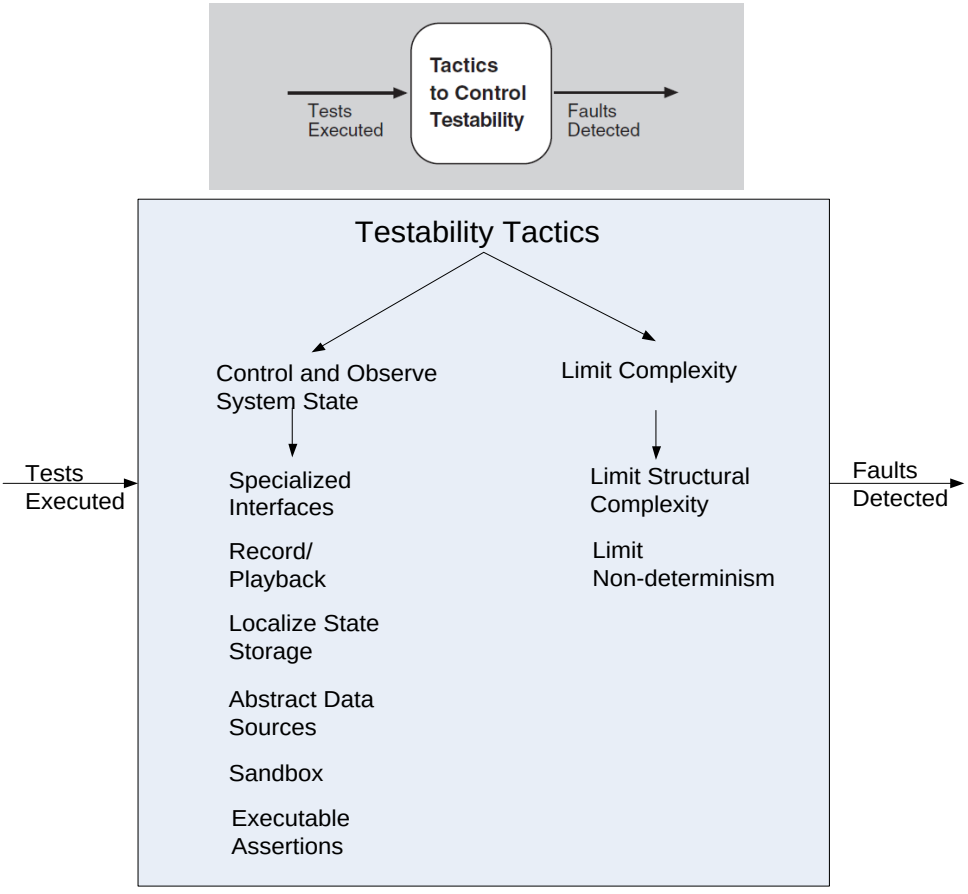

Fig. Goal of Testability Tactics

The goal of **testability tactics** is to make testing easier when each increment of software development is completed. Anything the architect can do to **reduce the high cost of testing** will offer significant benefits. There are two categories of tactics for improving testability. The first category focuses on adding **controllability** and **observability** to the system, while the second category addresses limiting the **complexity** in the system's design.

To control and observe a system's state, several tactics can be employed. One tactic is using **specialized interfaces** that allow control over variable values for a component, either through a **test harness** or normal execution. Another tactic involves **recording and playback**, where information crossing an interface is captured and used as input for further testing. **Localizing state storage** is also important, as it is convenient to store the state in a single place when starting a system in an arbitrary state for testing. **Abstracting data sources** allows for easier substitution of test data. A **sandbox** is another tactic where the system is isolated from the real world, allowing for experiments without worrying about the consequences. Finally, **executable assertions** involve hand-coding assertions and placing them in locations where the program might be in a faulty state, providing clear indicators of issues.

Limiting complexity is another critical tactic for improving testability. One way to limit **structural complexity** is by resolving cyclic dependencies between components, isolating them from external environments, and reducing dependencies between components. Another tactic is to **limit non-determinism** by identifying and eliminating sources of non-determinism, such as unconstrained parallelism.

2.8 Usability

Usability refers to how easy it is for users to accomplish their desired tasks and the level of support a system provides to assist them. It is one of the most cost-effective and easiest ways to enhance the **perceived quality** of a system. Usability encompasses several key areas, including learning how to use system features, performing tasks efficiently, minimizing the impact of errors, adapting the system to meet user needs, and increasing user confidence and satisfaction. For instance, in a typical usability scenario, a user might download a new application and, after just two minutes of experimentation, begin using it productively.

The primary goal of **usability tactics** is to enhance the ease with which users can interact with the system. In **Human-Computer Interaction (HCI)**, there are two main perspectives: **user initiative** (where the user takes action) and **system initiative** (where the system prompts the user). These perspectives are essential in creating scenarios that describe how the system supports user tasks. Usability tactics aim to support both types of initiatives.

To support **user initiative**, the system must allow actions like **canceling** a task, where the system must stop the task, free resources, and notify other components. It should also support

pause/resume actions, freeing resources temporarily to allocate them to other tasks. **Undo** functionality is vital, enabling users to revert the system to a previous state. Additionally, systems should allow for **aggregation**, where lower-level objects can be grouped, allowing users to apply actions to the entire group, thus reducing repetitive tasks.

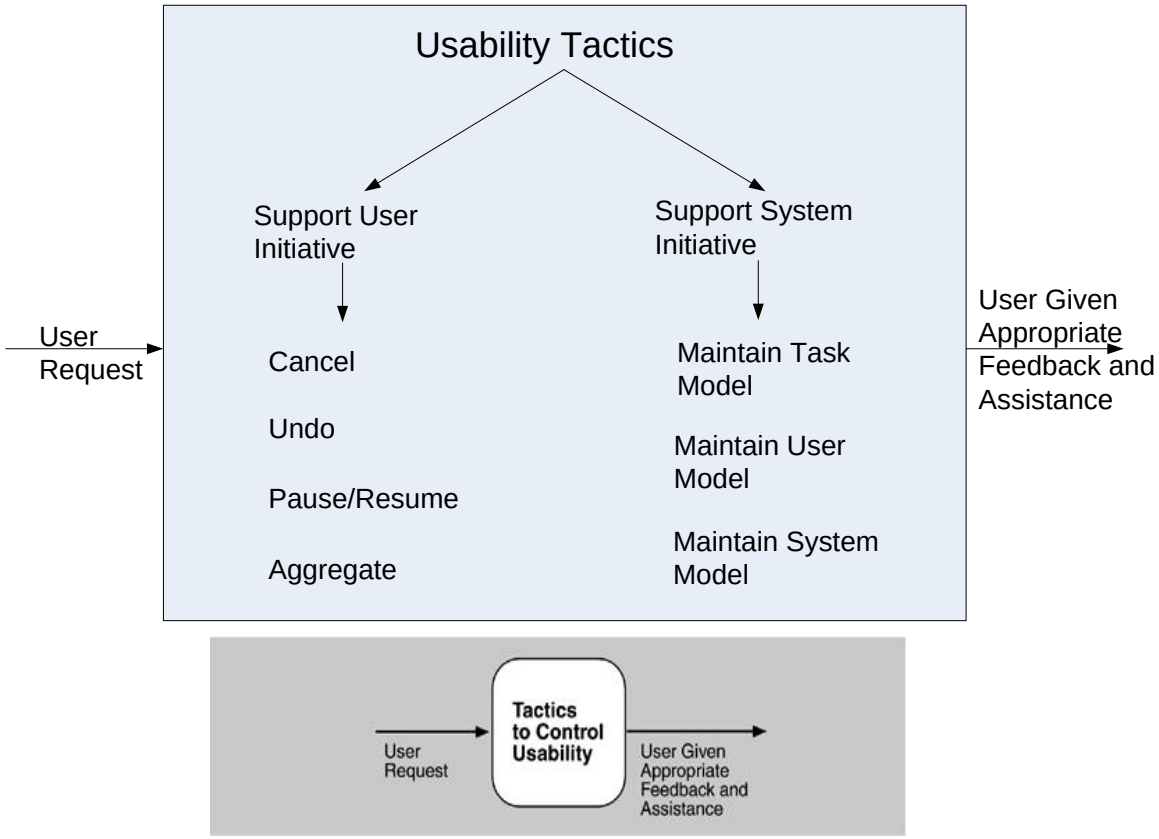

Fig. Goal of Usability Tactics

For **system initiative**, the system must maintain a **task model** to understand the context of the user's actions and provide helpful feedback. A **user model** is also important, as it represents the user's knowledge of the system and expected response times. Finally, the **system model** is essential for ensuring that the system can give feedback based on its own expected behavior, further enhancing usability.

2.9 Other Quality Attributes

Variability is a specialized form of **modifiability**, which refers to a system's ability to support the production of variants in a **preplanned fashion**. **Portability** is another form of modifiability, focusing on the **ease** with which software can be adapted to run on a different platform than the one it was originally built for. **Development Distributability** refers to the quality of

designing software to support **distributed development**, ensuring that teams in different locations can collaborate effectively.

Scalability can be categorized into two types: **horizontal scalability**, which involves adding more resources, such as additional servers to a cluster, and **vertical scalability**, which involves enhancing a physical unit by adding more resources like memory to a computer. **Deployability** addresses how the executable software arrives at a **host platform** and is **invoked**. **Mobility** deals with the challenges related to the movement of software across different platforms, considering factors like **size**, **display type**, **input devices**, **bandwidth**, and **battery life**.

Monitorability refers to the ability of **operations staff** to monitor the system while it is executing, ensuring proper functioning. **Safety** involves the system's ability to avoid entering **states** that could cause **damage**, **injury**, or **loss of life**, and includes mechanisms for recovering or limiting the damage. This is similar to **availability**, focusing on preventing, detecting, and recovering from failures.

Conceptual Integrity is about ensuring **consistency** in the design of the architecture, which enhances its **understandability**. It demands that similar tasks are carried out in the same way throughout the system. **Marketability** refers to the ability of an architecture to carry a **marketable meaning**, such as being **service-oriented** or **cloud-based**, independent of its underlying quality attributes.

Quality in Use encompasses the qualities that pertain to the system's use by stakeholders. It includes **effectiveness**, which measures whether the system is **correct**; **efficiency**, which refers to the effort and **time** required to develop the system; and **freedom from risk**, which concerns the system's impact on the economy, human health, and the environment.

Software and System Quality Attributes are essential not only for software but also for **physical systems**, such as **aircraft**, **kitchen appliances**, or **automobiles**. These systems rely on **embedded software** designed to meet various quality attributes, including **weight**, **size**, **electric consumption**, **power output**, **pollution output**, **weather resistance**, and **battery life**. The **software architecture** significantly affects these attributes, influencing overall system performance and functionality.

Using **Standard Lists of Product Quality** (like those from **ISO/IEC FCD 25010**) can be beneficial for **requirements gathering**, ensuring that no important needs are overlooked. These lists can also serve as the foundation for creating custom checklists tailored to specific **quality attributes** relevant to different domains. However, the limitations of these lists must be considered. No list is ever complete, and they can generate more **controversy** than understanding.

Furthermore, they may force architects to focus on quality attributes that may not be relevant to their system.

When dealing with **"X-ability"**—quality attributes where there is no established body of knowledge, such as **green computing**—the following steps can be taken: **model** the quality attribute, **assemble a set of tactics** for it, and construct **design checklists** to guide the development process.

2.10 Discussion and Questions

- How do architectural decisions impact the ability of a software system to achieve key quality attributes such as performance, modifiability, and security?
- In what ways can architectural patterns or tactics be used to optimize specific quality attributes, such as improving scalability or ensuring fault tolerance?
- How can architects balance competing quality attributes, such as maximizing performance while maintaining modifiability or security, during the design process?
- What role does documenting architecture play in ensuring that quality attributes like usability, availability, and reliability are communicated and met during the development lifecycle?

3. Architectural patterns and tactics

3.1 Architectural Patterns

Patterns and Tactics play a significant role in designing effective software architectures. The **Pattern Catalogue** is a comprehensive collection of various architectural patterns, including **Module Patterns**, which define how modules interact and organize within a system. Additionally, the **Component and Connector Patterns** are crucial as they define how components interact with each other through connectors.

The **Allocation Patterns** address the distribution of system components, and understanding the **relation between tactics and patterns** is key to leveraging them effectively together. To conclude, a summary of these concepts highlights the intertwined nature of these strategies in shaping robust software architectures.

An **architectural pattern** establishes a relationship between three key elements: the **context**, **problem**, and **solution**. The **context** refers to a recurring, common situation that gives rise to a particular problem. The **problem** is the specific issue that emerges in the given context, while the **solution** is the abstracted architectural resolution to address the problem.

The solution for a pattern is determined and described by several crucial components: a set of **element types**, such as **data repositories**, **processes**, and **objects**, a set of **interaction mechanisms** or **connectors**, like **method calls**, **events**, or a **message bus**, and a **topological layout** of the components.

Lastly, the solution is also defined by a set of **semantic constraints** that cover the **topology**, **element behavior**, and **interaction mechanisms**, ensuring consistency and functionality within the architectural framework.

3.2 Layer Pattern

There is often a need to develop and evolve portions of the system independently. This requires **clear and well-documented separation of concerns**, which enables **modules** to be developed and maintained independently. The challenge, or **problem**, arises when software needs to be segmented in a way that allows modules to evolve separately with minimal interaction between them. This segmentation supports key qualities like **portability**, **modifiability**, and **reuse**.

The **solution** to this problem is the **layered pattern**, which divides the software into distinct **units** called **layers**. Each layer groups a set of **modules** that offer a cohesive set of **services**, and the usage of these services must be **unidirectional**, meaning that higher layers cannot interact

with lower layers. The layers fully partition the software, with each partition exposed through a **public interface**.

The **overview** of the layered pattern revolves around defining these layers, where each layer is a grouping of modules that deliver a set of related services, and a unidirectional **allowed-to-use** relation exists among the layers. The **elements** involved in this pattern include the layer itself, which is essentially a type of module. The description of a layer should clearly define the modules it contains.

The **relations** in the layered pattern are governed by the **allowed-to-use** principle, which outlines the rules for how layers can interact with each other, including any permissible exceptions. There are specific **constraints** associated with this pattern: every piece of software is allocated to exactly one layer, there must be at least two layers (though typically three or more), and the **allowed-to-use** relations should not form a **circular** pattern—meaning that a lower layer cannot use a layer above it.

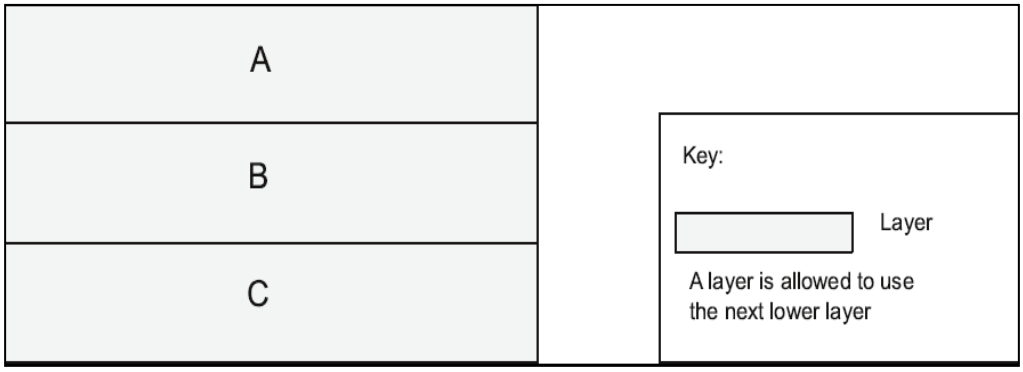

Fig. Example the Layer Architectural Pattern

However, the layered pattern does come with some **weaknesses**. The addition of layers can introduce up-front **cost** and **complexity** to the system. Furthermore, the use of layers may contribute to a **performance penalty** due to the added abstraction and interaction rules.

3.3 Broker Pattern

In many systems, especially those built from a collection of **services** distributed across multiple **servers**, implementing the system becomes complex. This complexity arises from the need to address how these systems **interoperate**, **connect** to each other, **exchange information**, and ensure the **availability** of component services. The challenge, or **problem**, lies in structuring distributed software so that **service users** do not need to know the nature and location of **service providers**. This makes it easier to dynamically change the **bindings** between users and providers without disrupting the system.

The **solution** to this problem is the **broker pattern**, which separates users of services (clients) from providers of services (servers) by inserting an intermediary called a **broker**. When a **client** needs a service, it queries the broker via a **service interface**. The broker then forwards the client's service request to the appropriate server, which processes the request and returns the results.

The **overview** of the broker pattern defines a runtime component, the broker, that mediates the communication between multiple clients and servers. The **elements** in this pattern include the **client**, which is the requester of services, and the **server**, which is the provider of services. The **broker** is the intermediary that locates the appropriate server to fulfill the client's request, forwards the request to the server, and returns the results. There are also **client-side proxies** and **server-side proxies**. The client-side proxy is responsible for managing communication with the broker, including **marshaling**, **sending**, and **unmarshaling** messages. On the other hand, the server-side proxy manages the actual communication with the broker, handling the marshaling, sending, and unmarshaling of messages.

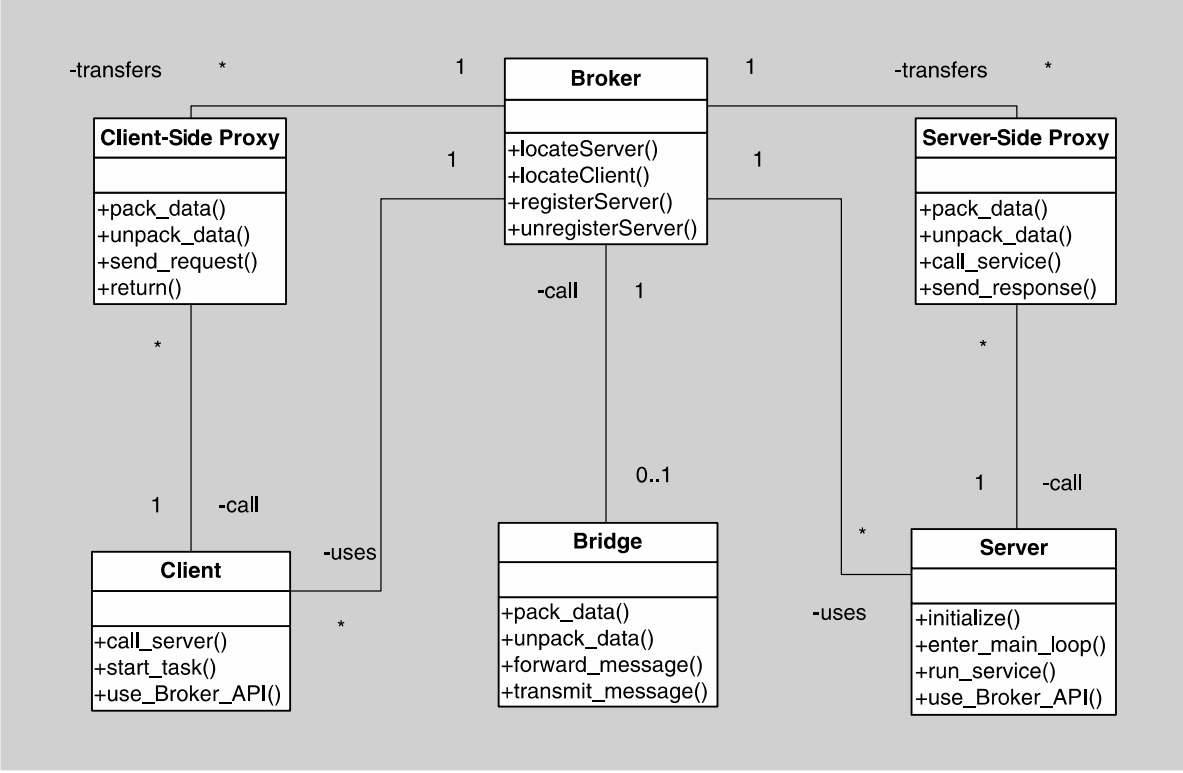

Fig. Example the Broker Architectural Pattern

The **relations** in this pattern define how clients (and, optionally, client-side proxies) and servers (optionally, server-side proxies) are attached to brokers. The **constraints** of the broker pattern

stipulate that the client can only attach to a broker via a **client-side proxy**, and the server can only attach to a broker via a **server-side proxy**.

Despite its benefits, the broker pattern has several **weaknesses**. First, brokers introduce a layer of **indirection**, which can add **latency** between clients and servers, potentially creating a **communication bottleneck**. Additionally, the broker can become a **single point of failure**, meaning that if the broker fails, the entire system might be impacted. The use of a broker also adds **up-front complexity**, and brokers may become a target for **security attacks**. Lastly, brokers can be difficult to **test** due to the added layers of communication and interaction.

3.4 MVC Pattern

The **MVC (Model-View-Controller)** pattern is commonly applied in the development of interactive applications, especially where user interfaces are frequently modified. In such systems, the **user interface** is often the most dynamic part, needing constant updates to reflect changes in data. Users typically wish to view data from different perspectives, such as a **bar graph** or a **pie chart**, and these representations must always reflect the current state of the underlying data.

The **problem** arises in how to keep **user interface functionality** separate from **application functionality** while still being responsive to user input and changes in the application's data. Additionally, there is the challenge of how to create, maintain, and coordinate multiple views of the user interface when the underlying application data changes.

The **solution** to this problem is the **MVC pattern**, which divides the application functionality into three distinct components: the **model**, the **view**, and the **controller**. The **model** contains the application data, the **view** displays a portion of this data and interacts with the user, and the **controller** mediates between the model and the view, managing notifications of any state changes in the system.

In the **overview** of the MVC pattern, system functionality is broken into three components: the **model**, which represents the application's data or state; the **view**, which is a **user interface** component displaying this data or allowing user input; and the **controller**, which manages interactions between the model and the view, translating user actions into changes in either the model or the view.

The **elements** of this pattern include the **model**, which holds or provides access to the application logic, the **view**, which is responsible for presenting the model to the user, and the **controller**, which handles user input and makes appropriate changes to the model or the view. The **relations** between these components are governed by the **notifies relation**, where the model, view, and controller notify each other of relevant state changes.

There are several **constraints** in the MVC pattern. For instance, there must always be at least one instance of each of the **model**, **view**, and **controller** components. Additionally, the **model** component should not interact directly with the **controller** to maintain a clean separation of concerns.

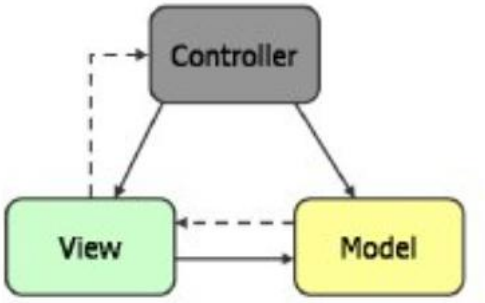

Fig. Example the MVC Architectural Pattern

However, the **weaknesses** of the MVC pattern include its potential complexity, which may not be justified for simple user interfaces. Moreover, the abstractions of model, view, and controller may not always be a good fit for certain **user interface toolkits**, making the pattern harder to implement effectively in some cases.

3.5 Pipe and Filter Pattern

Many systems are required to transform streams of **discrete data items** from **input** to **output**. These transformations often occur repeatedly, which makes it desirable to create them as independent, **reusable** parts. By doing so, systems can be more efficient and easier to maintain.

The **problem** arises in the need for such systems to be divided into **reusable**, **loosely coupled components** with simple, **generic interaction mechanisms**. This enables flexible combinations of components while ensuring that they are easily reused. Additionally, because the components are independent, they can be executed in **parallel**, enhancing the system's performance.

The **solution** to this problem is the **pipe-and-filter pattern**, which is characterized by successive transformations of data streams. In this pattern, data arrives at a filter's **input port(s)**, is **transformed**, and then passes through its **output port(s)** via a **pipe** to the next filter. A single filter can consume data from, or produce data to, one or more ports, facilitating flexible data flow and processing.

In the **overview** of the pattern, data is transformed from a system's external inputs to its external outputs through a series of **transformations** performed by its filters, which are connected by pipes. Each filter plays a role in processing the data, while pipes simply convey the data between filters.

The **elements** in this pattern include the **filter**, which is a component responsible for transforming data read from its input port(s) and writing the transformed data to its output port(s). The **pipe** is a connector that conveys data from a filter's output port(s) to another filter's input port(s). A pipe has a **single source** for its input and a **single target** for its output, ensuring a clear and unidirectional flow of data. Additionally, a pipe preserves the **sequence** of data items and does not alter the data as it passes through.

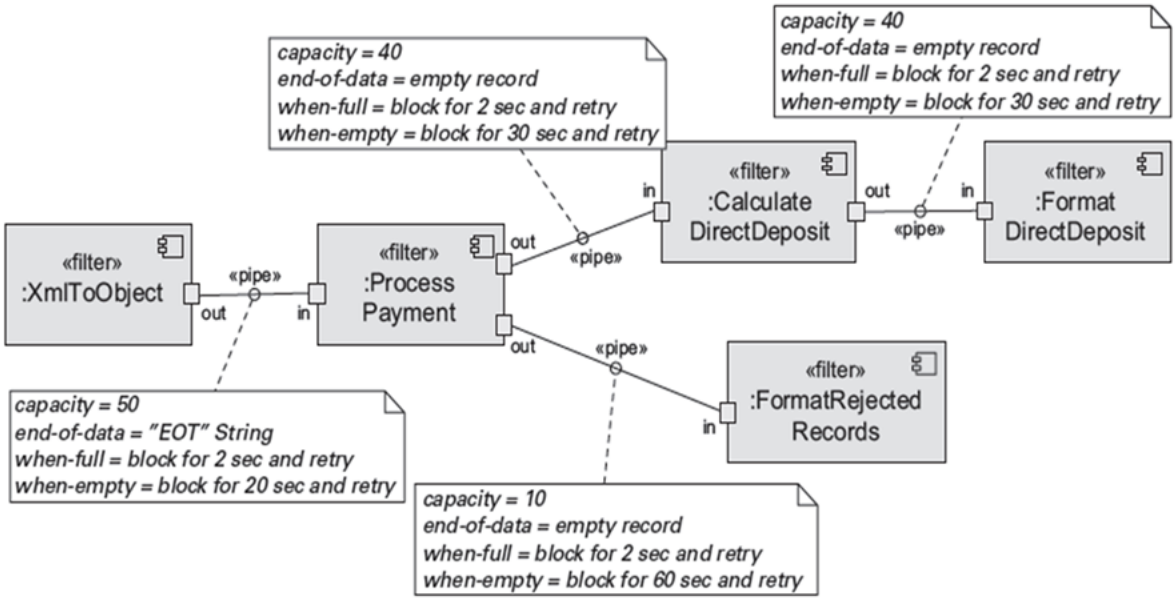

Fig. Example the Pipe and Filter Architectural Pattern

The **relations** in this pattern are governed by the **attachment relation**, which connects the output of filters to the input of pipes and vice versa. **Constraints** include the requirement that pipes must connect filter output ports to filter input ports, and the connected filters must agree on the **type of data** being passed along the connecting pipe to ensure proper data interpretation.

3.6 Client-Server Pattern

In many distributed systems, there are **shared resources** and **services** that large numbers of **clients** wish to access. Managing access to these resources and ensuring a consistent **quality of service** is a critical concern. The **problem** involves promoting **modifiability** and **reuse** by centralizing common services in a way that allows them to be updated in a single or a few locations. Additionally, scalability and **availability** need to be improved by centralizing control of resources while distributing the resources themselves across multiple physical servers.

The **solution** to this problem is to structure the system such that **clients** interact with **servers** by requesting services. These servers provide a set of services, and in some cases, certain

components may act as both **clients** and **servers**. The architecture may involve one central server or multiple distributed servers, depending on the system's needs.

In the **overview** of this model, **clients** initiate interactions with **servers** by invoking services as needed and then waiting for the results of those requests. This interaction is facilitated by various **elements**, including the **client**, which is a component that invokes services from a **server**. Clients have ports that describe the services they require, while the **server** is a component that provides services and has ports that describe the services it offers.

A key component of the interaction is the **request/reply connector**, which uses a **request/reply protocol** to allow clients to invoke services on a server. Important characteristics of this connector include whether the calls are **local** or **remote**, and whether the data being transmitted is **encrypted** to ensure security.

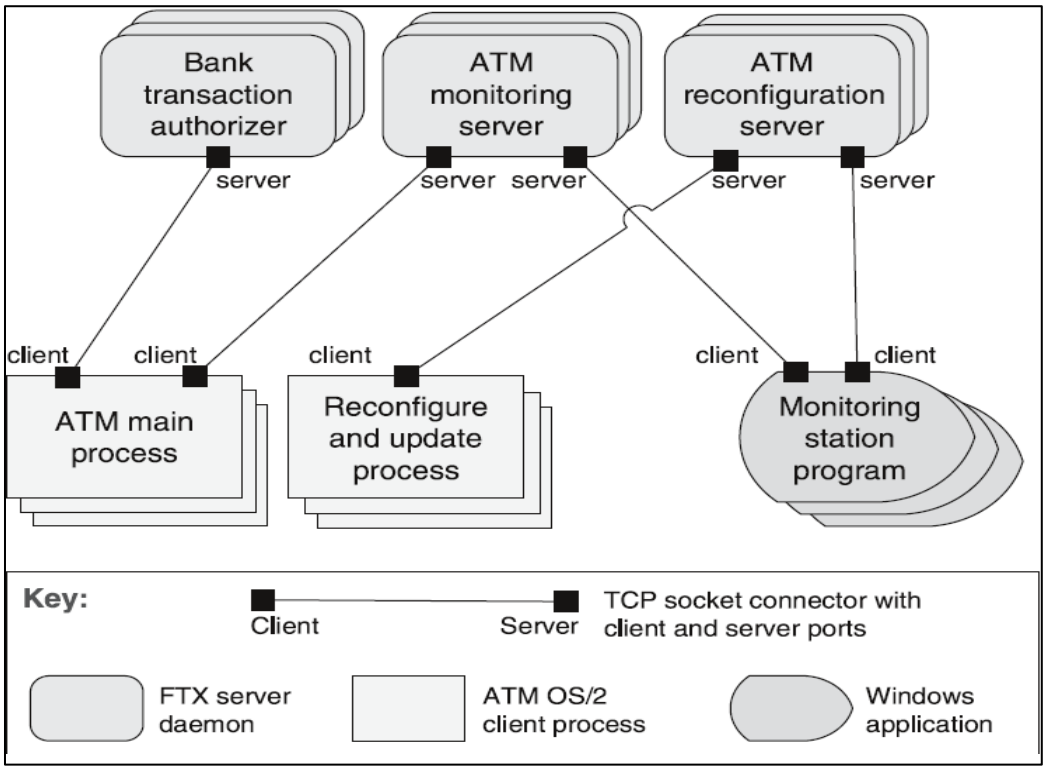

Fig. Example the Client-Server Architectural Pattern

The **relations** between components are defined by the **attachment relation**, which associates **clients** with **servers**, ensuring the proper connections are made. **Constraints** include the requirement that **clients** are connected to **servers** through **request/reply connectors**, and that **server components** can themselves act as **clients** to other servers.

Despite its advantages, this model has some **weaknesses**. A **server** can become a **performance bottleneck** if it is overwhelmed by too many requests. Additionally, the **server** is a potential

single point of failure, meaning that if it fails, the entire system could be impacted. Finally, decisions about whether functionality should reside in the **client** or the **server** can be complex and costly to change once the system has been built, making this an important consideration during the system's design phase.

3.7 Peer-to-Peer Pattern

In distributed systems, **computational entities** are often required to cooperate and collaborate to provide services to a distributed community of **users**. Each entity is considered **equally important** in terms of initiating interactions and providing its own resources. The **problem** arises in how these **distributed computational entities** can be connected via a common protocol to organize and share their services efficiently, ensuring both **high availability** and **scalability**.

The **solution** to this problem is the **peer-to-peer (P2P)** pattern, where components interact directly as **peers**. In this pattern, all peers are considered "equal"—no single peer or group of peers is critical for the health of the system. **Peer-to-peer communication** typically involves a **request/reply interaction**, but unlike the **client-server pattern**, there is no inherent asymmetry in the roles of the interacting components.

In the **overview** of the P2P pattern, **computation** is achieved through the cooperation of peers that request services from and provide services to one another across a **network**. Each peer operates independently, and special peer components may be used to provide **routing**, **indexing**, and **peer search** capabilities to facilitate interaction among peers.

The **elements** in this pattern include the **peer**, which is an independent component running on a **network node**. A **request/reply connector** is used to connect to the peer network, search for other peers, and invoke services from other peers. In some cases, the need for a reply may be eliminated, simplifying the communication process.

The **relations** in this pattern associate peers with their connectors. Attachments between peers and connectors may change dynamically at **runtime**, allowing flexibility in the system's operation. There are also specific **constraints** that may apply, such as limiting the **number of allowable attachments** to any given peer, restricting the number of **hops** used when searching for a peer, or managing which peers know about other peers. Some P2P networks may even be organized using **star topologies**, where peers only connect to **supernodes**.

Despite its advantages, the P2P pattern has several **weaknesses**. Managing **security**, **data consistency**, **data/service availability**, **backup**, and **recovery** becomes more complex in this decentralized environment. Additionally, smaller P2P systems may struggle to consistently achieve quality goals such as **performance** and **availability**, especially as the system scales.

3.8 SOA Pattern

Service-Oriented Architecture (SOA) is a design pattern where various services are offered by **service providers** and consumed by **service consumers**. These **services** are described and made available by providers, and consumers need to be able to use them without any detailed knowledge of their underlying **implementation**. This decouples the consumer from the complexity of the service's internal workings, enabling a more flexible architecture.

The **problem** arises from the need to support **interoperability** among distributed components that run on different platforms, are written in different implementation languages, and are provided by different organizations. These components are often distributed across the **Internet**, making it challenging to ensure smooth communication and data exchange across such diverse environments.

The **solution** to this problem is the **Service-Oriented Architecture (SOA)** pattern, which describes a collection of distributed components that either provide or consume services. In this architecture, computation is achieved through a set of cooperating components that interact over a **network**, enabling flexible, scalable, and loosely coupled interactions between systems.

The **overview** of SOA involves a network of **components** that provide and consume services. Key **elements** in this pattern include service providers, which offer one or more services through **published interfaces**, and service consumers, which invoke these services either directly or through an intermediary. **Service providers** can also act as **service consumers** in some cases. An **Enterprise Service Bus (ESB)** acts as an intermediary that routes and transforms messages between service providers and consumers, ensuring efficient communication. Additionally, a **registry of services** is used by providers to register their services and by consumers to discover available services at **runtime**. The **orchestration server** coordinates interactions between service consumers and providers, managing business processes and workflows.

The **connectors** in SOA include the **SOAP connector**, which uses the **SOAP protocol** for synchronous communication between web services, typically over HTTP. The **REST connector** relies on basic **request/reply** operations of the **HTTP protocol**. Another important connector is the **asynchronous messaging connector**, which uses a messaging system to facilitate **point-to-point** or **publish-subscribe** asynchronous message exchanges.

The **relations** in SOA are defined by the **attachment** of various components to their respective connectors. **Constraints** include the connection of service consumers to service providers, possibly through intermediary components such as the **ESB**, **registry**, or **orchestration server**.

Despite its advantages, **SOA-based systems** are typically complex to build. One of the main **weaknesses** of SOA is that you do not control the evolution of independent services, which can lead to difficulties in maintaining compatibility over time. Additionally, the **middleware** layer introduces **performance overhead**, and services themselves can become **performance bottlenecks**. Moreover, these services typically do not provide **performance guarantees**, which can further complicate optimization efforts.

3.9 Publish-Subscribe Pattern

In the **publish-subscribe pattern**, numerous independent producers and consumers of data must interact with one another. The exact number and nature of these **data producers** and **consumers** are not predetermined or fixed, and the **data** they share may vary. This pattern is designed to handle scenarios where the interaction between components is dynamic and flexible, without requiring them to have specific knowledge of each other.

The **problem** addressed by this pattern is how to create integration mechanisms that allow for the transmission of **messages** among producers and consumers while ensuring they remain unaware of each other's identity or, in some cases, even their existence. This allows for a more **decoupled** and scalable system.

The **solution** provided by the publish-subscribe pattern involves components interacting through announced **messages** or **events**. Components can **subscribe** to a set of events, and **publisher components** announce events by placing them on a **bus**. The **connector** then delivers these events to the **subscriber components** that have registered an interest in receiving them. This mechanism ensures that only those components that need the data are notified, without the need for direct interaction between producers and consumers.

In the **overview** of this pattern, components **publish** and **subscribe** to events. When a component announces an event, the **connector infrastructure** dispatches that event to all registered **subscribers**, facilitating efficient message delivery without requiring the components to have direct knowledge of each other.

The **elements** involved in the publish-subscribe pattern include any component with at least one **publish** or **subscribe port**. The **publish-subscribe connector** plays a central role by providing **announce** and **listen** roles for components that wish to publish or subscribe to events.

The **relations** in this pattern are defined by the **attachment relation**, which associates components with the publish-subscribe connector. This relation specifies which components **announce** events and which components are **registered** to receive those events.

There are certain **constraints** in this pattern, such as the requirement that all components are connected to an **event distributor**, which can be viewed either as a **bus** or a **connector**. **Publish ports** are attached to **announce roles**, while **subscribe ports** are attached to **listen roles**, ensuring proper event flow.

Despite its advantages, the publish-subscribe pattern has several **weaknesses**. It typically increases **latency** and can have a negative impact on **scalability** and **predictability** of message delivery times. Additionally, there is **less control** over the **ordering of messages**, and the delivery of messages is not always guaranteed, which can lead to potential inconsistencies in data delivery.

3.10 Shared-Data Pattern

In the **shared-data pattern**, various computational components need to **share** and **manipulate** large amounts of **data** that does not belong solely to any one of these components. This situation arises when multiple **independent components** need to access and work with the same set of data over time.

The **problem** this pattern addresses is how to store and manipulate **persistent data** that must be accessed by multiple, potentially independent, components. These components need a reliable way to access and modify the data without conflicts or loss.

The **solution** in the shared-data pattern involves the exchange of persistent data between multiple **data accessors** and at least one **shared-data store**. This exchange may be initiated by either the **accessors** or the **data store** itself, and the main mechanism for interaction is **data reading** and **writing**. Communication between the components is mediated by the shared data store, and control over the flow of data may be initiated by either the accessors or the data store. The **data store** ensures that the data remains **persistent** for the duration of its use.

The **elements** in this pattern include the **shared-data store**, which must address various concerns such as the **types of data** stored, performance-related properties, **data distribution**, and the number of **accessors** permitted. The **data accessor components** interact with the data store, and the **data reading and writing connectors** facilitate the exchange of data between the accessors and the data store.

The **relations** in this pattern are defined by the **attachment relation**, which specifies which **data accessors** are connected to which **data stores**. This ensures that only authorized components can access the shared data.

There are important **constraints** in this pattern, such as the rule that data accessors can only interact with the **data store(s)** and not directly with other components. This maintains a level of separation and control over how the data is accessed and manipulated.

While the shared-data pattern has its uses, it has certain **weaknesses**. The **shared-data store** can become a **performance bottleneck** if many accessors are competing for access to the data. Additionally, the **shared-data store** can act as a **single point of failure**, meaning if it becomes unavailable, all access to the data is lost. Another issue is that **producers** and **consumers** of the data may become tightly coupled, which can reduce the overall flexibility of the system.

3.11 Map-Reduce Pattern

In the context of modern businesses, there is an increasing need to quickly analyze enormous volumes of data, often at **petabyte scale**. For many applications dealing with ultra-large data sets, sorting the data and then analyzing the grouped data is sufficient. The **map-reduce pattern** addresses the challenge of efficiently performing a distributed and parallel sort of a large data set while providing a simple interface for programmers to specify the analysis to be conducted.

The **map-reduce pattern** requires three key components to function effectively: a specialized infrastructure, a **map** function, and a **reduce** function. The infrastructure is responsible for allocating software to hardware nodes in a **massively parallel computing environment** and handling the sorting of the data as needed. The **map** function, specified by the programmer, filters the data to retrieve the items to be combined, while the **reduce** function, also programmer-specified, combines the results of the map.

This pattern provides a framework for analyzing a large, distributed set of data that executes in parallel across multiple processors. The parallelization in **map-reduce** allows for **low latency** and **high availability**, with the **map** function handling the extract and transform portions of the analysis and the **reduce** function responsible for loading the results.

The **map** function is deployed across multiple processors, performing the extract and transformation portions of the analysis. The **reduce** function, which can be deployed as a single or multiple instances, performs the load portion of the extract-transform-load process. The **infrastructure** ensures the proper deployment of map and reduce instances, manages the data flow between them, and detects and recovers from failures.

The relations in **map-reduce** include the **"deploy on"** relation, which links map or reduce instances to the processors they are deployed on, and the **"instantiate, monitor, and control"** relation, which governs the interaction between the infrastructure and the map and reduce instances. Certain constraints include the requirement that the data to be analyzed exists as a set of files and that **map functions are stateless** and do not communicate with each other. The only communication between map-reduce instances is through the **<key, value>** pairs emitted by the map instances.

However, **map-reduce** has its weaknesses. If the data set is not large enough, the overhead of using the map-reduce model may not be justified. Additionally, if the data cannot be divided into similarly sized subsets, the parallelism advantages are lost. Furthermore, operations requiring multiple reduces are complex to orchestrate and may reduce the overall effectiveness of the pattern.

3.12 Multi-Tier Pattern

In a **distributed deployment**, there is often a need to divide a system's infrastructure into distinct subsets. This allows for better management, scalability, and distribution. The challenge arises in how to split the system into **computationally independent execution structures**, which are groups of **software** and **hardware** connected by some form of communication media.

The solution to this problem is to organize the execution structures of the system into a set of **logical groupings** of components, each called a **tier**. These groupings allow different sets of components to operate independently while still maintaining communication with one another. The **tiers** in a system represent logical divisions of the software components, which can be mapped to different hardware platforms as necessary.

Each **tier** is a logical grouping of software components that interact with other tiers. The relation between tiers is defined by **"is part of"**, which groups components into tiers, and **"communicates with"**, which shows how components in different tiers interact. Additionally, the **"allocated to"** relation specifies the mapping of tiers to physical computing platforms. A key constraint is that each **software component** must belong to exactly one **tier**, ensuring that the system is logically organized.

However, there are some **weaknesses** associated with this approach. The division of components into tiers often involves substantial **up-front cost** and **complexity**, which can be challenging, especially in large systems with many tiers.

3.13 Relationships Between Tactics and Patterns

Patterns are built from **tactics**. In this analogy, if a **pattern** is considered a molecule, then a **tactic** is like an atom. For example, the **MVC** pattern utilizes various tactics such as **increasing semantic coherence**, **encapsulation**, **using an intermediary**, and **using runtime binding**. These tactics work together to form the overall pattern.

While **patterns** solve specific problems, they may be neutral or have weaknesses concerning other qualities. For instance, the **broker pattern**, although useful, may present challenges such as potential **performance bottlenecks** or a **single point of failure**. To address these weaknesses, **tactics** can be employed to enhance the pattern. **Increasing resources** could help improve

performance, while **maintaining multiple copies** can increase **availability**. These tactics effectively augment the pattern's capabilities, ensuring it better meets system requirements.

3.14 Tactics and Interactions

Each tactic has **pluses** (its reason for being) and **minuses** – side effects. The use of tactics can help alleviate these minuses, but as with all design decisions, nothing comes without a cost. Tactics interact with one another in complex ways, and the challenge is managing these interactions effectively.

A common tactic for detecting faults is **Ping/Echo**, but it introduces common **side-effects** in the areas of **security**, **performance**, and **modifiability**. For instance, from a **security** perspective, there is the challenge of preventing a **ping flood attack**. From a **performance** standpoint, there's the concern of ensuring that the performance overhead of **ping/echo** remains minimal. In terms of **modifiability**, there's the question of how to add **ping/echo** to an existing architecture without introducing unnecessary complexity.

To address the **performance** side-effect of **Ping/Echo**, a tactic like **Increase Available Resources** can be used. However, this tactic introduces its own side effects, particularly around **cost** – more resources typically mean higher costs. In addition, there's the **performance** concern of how to utilize the increased resources efficiently.

A tactic to mitigate the efficient use of resources is the **Scheduling Policy**. While effective, this tactic also brings with it side effects in **modifiability**, such as how to add the **scheduling policy** to the existing architecture and how to change the policy in the future.

To address the concern of adding a scheduler to the system, the **Use an Intermediary** tactic can be applied. This tactic helps ensure that the scheduler is integrated into the system but introduces another **modifiability** concern: how to ensure that all communication passes through the **intermediary**.

To further address the concern of ensuring that all communication passes through the intermediary, the tactic of **Restrict Communication Paths** can be used. However, this tactic brings with it **performance** concerns – specifically, how to ensure that the performance overhead of the intermediary is not excessive.

This design problem is now recursive, as each tactic introduces new concerns that need to be addressed. As new concerns arise, additional tactics are added, potentially leading to an infinite progression. However, this process does eventually stabilize. Over time, the side effects of each tactic become small enough to be considered insignificant or ignorable, allowing the design process to come to a close.

3.15 Discussion and Questions

- How do architectural patterns enable the reuse of design decisions across different systems, and what are some examples of well-known architectural patterns used in practice?
- In what ways do tactics complement architectural patterns, and why are tactics necessary for tailoring patterns to meet specific system requirements?
- How can the underspecification of architectural patterns lead to challenges in their application to real-world systems, and what strategies can be used to address these challenges?
- What is the relationship between architectural patterns and system requirements, and how can you determine when the augmentation with tactics is complete?
- How do the simplicity of tactics and the complexity of architectural patterns interact to affect the overall design process in software engineering?

4. Documenting software architectures

4.1 Architecture Documentation

Even the best **architecture** will be ineffective if those who need it **do not know** what it is, cannot understand it well enough to use, build, or modify it, or **misunderstand** it and apply it incorrectly. Despite the **analysis**, hard work, and insightful design put in by the **architecture team**, all of this effort can be wasted if the architecture is not communicated effectively. **Architecture documentation** must be sufficiently **transparent** and **accessible** so that it can be quickly understood by new employees. It should also be concrete enough to serve as a **blueprint** for construction and provide enough detail to serve as a basis for **analysis**.

The documentation serves both **prescriptive** and **descriptive** purposes. For some audiences, it prescribes what should be true, placing constraints on decisions that are yet to be made. For others, it describes what is true, recounting decisions that have already been made about the system's design. Understanding the **stakeholders' uses** of the documentation is essential to determine the **information** to capture.

Architecture documentation serves three key purposes. First, it aids in **education**, introducing new people to the system, including new team members or external analysts and evaluators. Second, it acts as the primary **communication vehicle** among stakeholders, especially between architects and developers, and between current architects and future architects. Finally, it forms the basis for **system analysis** and construction by telling implementers what to implement. Each module in the system has **interfaces** that must be provided and used by other modules. Additionally, the documentation is used to register and communicate **unresolved issues** and serves as the foundation for **architecture evaluation**.

4.2 Documentation Notations

Informal notations are typically used to represent **views** with general-purpose diagramming and editing tools. These notations rely on **natural language** to describe the meaning of elements but lack formal analysis capabilities. They do not offer a precise semantic structure, making them useful for quick sketches or initial conceptualization, but they cannot be formally analyzed.

Semiformal notations, on the other hand, involve standardized graphical elements with specific rules of construction. While these notations do not provide a complete semantic treatment of the meaning behind their elements, they enable **rudimentary analysis**. **UML (Unified Modeling Language)** is an example of a semiformal notation, as it allows for structured visualization and some level of formal reasoning, though it still lacks full precision in its semantics.

Formal notations are the most structured, where views are described in a notation that has a **precise semantics**, often mathematical in nature. This enables **formal analysis** of both syntax and semantics. **Architecture Description Languages (ADLs)** are examples of formal notations, which also support **automation** through associated tools, providing a high degree of rigor and the ability to automate various aspects of the system design.

There are **tradeoffs** involved in choosing between these types of notations. More **formal notations** take more time and effort to create but offer **reduced ambiguity** and open up more opportunities for **analysis**. Conversely, **informal notations** are easier to create but offer fewer guarantees about the clarity and correctness of the design, which can lead to **misunderstandings** or inconsistencies.

Different notations are better suited for expressing different kinds of information. For example, **UML class diagrams** may not be helpful for reasoning about **schedulability**, while **sequence charts** might not tell much about a system's likelihood of **timely delivery**. It's important to select notations and **representation languages** based on the specific **issues** you need to capture and reason about in your architecture.

UML is a **general-purpose visual modeling language** that is commonly used to **specify**, **visualize**, **construct**, and **document** the artifacts of a system. It provides a standardized notation for describing **object-oriented systems** and has become a **de facto standard** for **object-oriented design**. While UML is essential in many software development careers, parts of it can also be applicable to other programming paradigms. UML 2.1, for instance, supports 13 different types of **diagrams**, offering a broad toolkit for various modeling tasks.

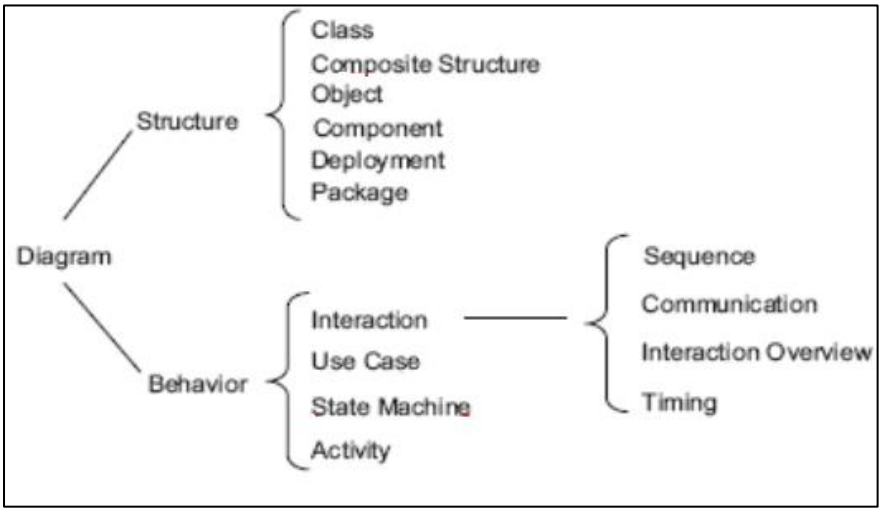

Fig. Types of Diagrams in UML 2.1

4.3 Views

Views allow us to break down software architecture into a number of manageable representations of the system. The principle of architecture documentation is to document the relevant views and the elements that apply across multiple views. Different views support different goals and uses, and the views you choose to document should depend on the specific uses you expect to make of the documentation. Each view comes with both a cost and a benefit, and it is essential to ensure that the benefits of maintaining a view outweigh its costs.

One important type of view is the **Module View**, which represents the implementation units of software that provide a coherent set of responsibilities. In the module view, **modules** have a **part/whole relationship**, where submodules form part of a larger module, and a **dependency relationship** is defined between different module views. There is also a **generalization/specialization** relationship between more specific (child) and more general (parent) modules. Module views may impose specific **topological constraints**, such as limiting visibility between modules. The **module view** is used as a blueprint for constructing code, performing **change-impact analysis**, planning **incremental development**, ensuring **requirements traceability**, and organizing **work assignments**. A **complete documentation** must include at least one module view.

The **Component-Connector (C&C) View** focuses on the interactions between **components** and **connectors**. Components represent the principal processing units and data stores, and they interact with other components through **ports** via **connectors**. **Connectors** act as pathways for interaction and have a set of **roles (interfaces)** indicating how components interact with them. The relationships in a C&C view include **attachments**, where component ports are linked to connector roles, and **interface delegation**, where component ports are associated with internal ports or subarchitectures. **Constraints** in C&C views include the requirement that components must be attached to connectors, and **interface delegation** can only happen between compatible ports or roles. C&C views help **guide development**, specify the **structure** and **behavior** of runtime elements, and help reason about **runtime system quality attributes** like **performance** and **availability**. While **UML components** are well-suited for C&C components, **UML connectors** have limitations, such as not representing substructure or behavior. Connectors can be represented using a **UML connector** or by annotating it with a tag to explain its complexity.

Allocation Views describe the mapping between **software elements** and **environmental elements**. A software element has certain required properties, while an environmental element provides the necessary properties to the software. The **allocation relationship** maps software elements to environmental elements, with specific properties dependent on the view. **Allocation views** are useful for reasoning about **performance**, **availability**, **security**, **safety**,

distributed development, concurrent access to software versions, and the form and mechanisms of system installation.

4.4 Quality Views

A **quality view** can be tailored to address specific stakeholder concerns by extracting pieces from various **structural views** and packaging them together. For example, a **security view** highlights components with security responsibilities, detailing how those components communicate and interact with **security-related data repositories**. It also shows any other **security measures** in the system's environment, such as physical security, and illustrates how security protocols operate. Additionally, the behavior part of this view captures how the system would respond to specific **threats** and **vulnerabilities**, including **human** interaction with security elements.

A **communications view** is particularly useful for systems that are **globally dispersed** and **heterogeneous**. It shows the channels of communication between components, network channels, quality-of-service parameters, and areas of **concurrency**. This view is instrumental in analyzing **performance** and **reliability**, helping detect issues such as **deadlocks** or **race conditions**. The behavior section of the view could demonstrate how **network bandwidth** is dynamically allocated to optimize performance.

An **exception or error-handling view** focuses on error detection, reporting, and resolution mechanisms. It outlines how components detect, report, and resolve faults, helping to identify the sources of errors and the appropriate **corrective actions** to be taken. Similarly, a **reliability view** models mechanisms like **replication** and **switchover**, also depicting timing issues and **transaction integrity**.

For performance analysis, the **performance view** highlights the aspects of the architecture that influence system performance. It shows **network traffic models**, **maximum latencies** for operations, and other relevant metrics that help assess the system's overall performance. Each of these views serves a specific function, offering valuable insight into how the system behaves under different conditions and how to address potential concerns.

4.5 Choosing the Views

When determining which views to include in an architecture, it is essential to consider several factors. First, you need to assess the **available people** and their **skills**, the **standards** to comply with, the **budget** on hand, and the **schedule**. Additionally, understanding the **information needs** of the **stakeholders** and the **driving quality attribute requirements** is crucial. The approximate **size** of the system must also be taken into account. In the architecture documentation, it is important to include at least one **module view**, one **C&C view**, and for larger systems, one **allocation view**.

The method for creating these views involves several steps. **Step 1** is to build a **stakeholder/view table**, where the rows list the stakeholders for the architecture documentation, and the columns enumerate the views that apply to the system. This table should also include **view sketches** resulting from the design work completed so far. The cells of the table describe the level of information detail required by each stakeholder for each view, which can range from none to high. **Step 2** involves combining views to reduce their number. This is done by identifying **marginal views** that only require an overview or serve very few stakeholders. These marginal views should be combined with others that have stronger constituencies, such as various **C&C views**, **deployment views** with **SOA**, and **decomposition** or **layered views**. Finally, **Step 3** focuses on prioritizing and staging. The **decomposition view** is particularly helpful to release early, and it is not necessary to fully satisfy the information needs of all stakeholders before starting another view. Additionally, it is not required to complete one view before beginning work on another.

4.6 Building the Documentation Package

A **documentation package** consists of various views as well as additional information that complements these views. **Documenting a view** involves several key sections. The first section is the **Primary Presentation**, which showcases the elements and relations of the view. This section should provide the necessary information about the system, using the specific vocabulary related to that view. It is often presented either graphically or textually. The second section is the **Element Catalog**, which details at least the elements depicted in the primary presentation. This catalog includes the elements and their properties, relations and their properties, element interfaces, and element behavior. The third section is the **Context Diagram**, used to depict the scope of a view, illustrating how entities in the environment interact with the system, such as humans, computers, sensors, etc. The fourth section, the **Variability Guide**, shows how to exercise any variation points that are part of the architecture. Finally, the fifth section is the **Rationale**, which explains the design choices made and provides a convincing argument for their validity, such as the choice of a particular pattern.

Beyond the views, **documentation information** must be included. This includes **document control information**, which lists the issuing organization, version number, date of issue, status, change history, and the procedure for submitting change requests. The first section of this additional information is the **Documentation Roadmap**, which tells the reader what information is included in the documentation and where to find it. This section also covers the scope, a summary of the documentation, an overview of the views, and how stakeholders can use the documentation. The second section, **How a View Is Documented**, explains the standard organization used to document views—either the one described in this chapter or a custom one. The third section, **System Overview**, provides a short prose description of the system's function, its users, and any important background or constraints. This section helps establish a consistent mental model of the system and its purpose and may reference the project's

concept-of-operations document. The fourth section, **Mapping Between Views**, helps readers understand the associations between views, offering insights into how the architecture functions as a unified conceptual whole. These associations can be captured as tables, as they are typically many-to-many. The fifth section, the **Rationale**, documents architectural decisions that apply to more than one view. It covers background or organizational constraints and decisions about which fundamental patterns were used. Finally, the **Directory** section serves as a reference material set, including an index of terms, a glossary, and an acronym list, to help readers quickly find information.

4.7 Documenting Behavior

Behavior documentation complements each view by describing how the architecture elements within that view interact with each other. It enables reasoning about critical system aspects such as the system's potential to **deadlock**, its ability to complete tasks within a **desired amount of time**, and the **maximum memory consumption**, among others. Behavior is documented in its own section within the view's **element catalog**.

There are several **notations** for documenting behavior. One common notation is **trace-oriented languages**, which capture sequences of activities or interactions between structural elements in response to specific stimuli or system states. Examples of this approach include **use cases**, **sequence diagrams**, **communication diagrams**, **activity diagrams**, **message sequence charts**, **timing diagrams**, and **Business Process Execution Language**. Another approach, **comprehensive languages**, models the complete behavior of structural elements and allows the inference of possible paths from the initial to the final state, such as with a **state machine**.

Use cases are a popular method for modeling behavior. They consist of **UML use case diagrams** and a textual description of a system. The primary goal of use case modeling is to elicit **stakeholder requirements** and provide a foundation for developers to analyze and implement the system. **Use case diagrams** represent scenarios where a use case is an instance or execution path of a system. For example, an **ATM withdrawal** use case could include scenarios such as **successful withdrawal**, **incorrect PIN**, **insufficient funds**, or **not enough notes in the ATM**. The **textual description** of these scenarios is tool-dependent and typically relies on **natural language**.

A **sequence diagram** is another tool used to show how objects collaborate to implement a specific behavior over time. It typically represents a single scenario and includes two dimensions: horizontal (objects collaborating) and vertical (message sequences over time). Some types of messages in sequence diagrams include **destroy** (where an instance causes another to cease existing), **synchronous** (where the caller waits for operation completion), and **create** (where the caller does not wait and proceeds immediately). Messages in a sequence diagram can be **numbered**, pass **parameters**, and even be **self-referential**.

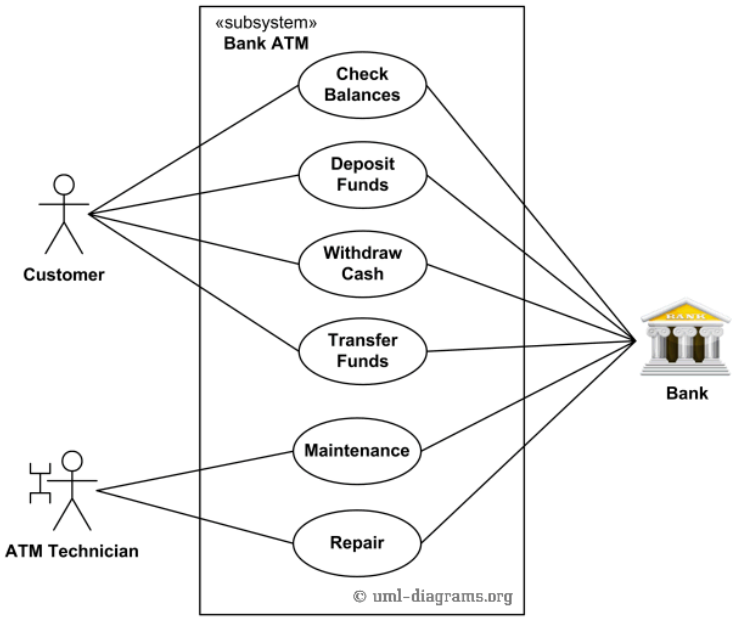

Fig. Use case Diagram for an ATM System

State machine descriptions complement the structural descriptions of elements with constraints on interactions and timed reactions to stimuli. A state machine shows how an object changes states throughout its life based on previous states and events. It includes a **start state** and at least one **end state**, with states representing a set of attribute values. A **transition** occurs when an object moves from one state to another based on specific conditions or events, which may trigger activities. Each state can have internal actions such as **entry**, which occurs when a state is entered, **exit**, which occurs when a state is left, and **do**, which represents ongoing processes while in the state. A state machine diagram shows that only one state can be active at any time, and if an event occurs that doesn't have a corresponding transition, the event is ignored—potentially signaling a design flaw.

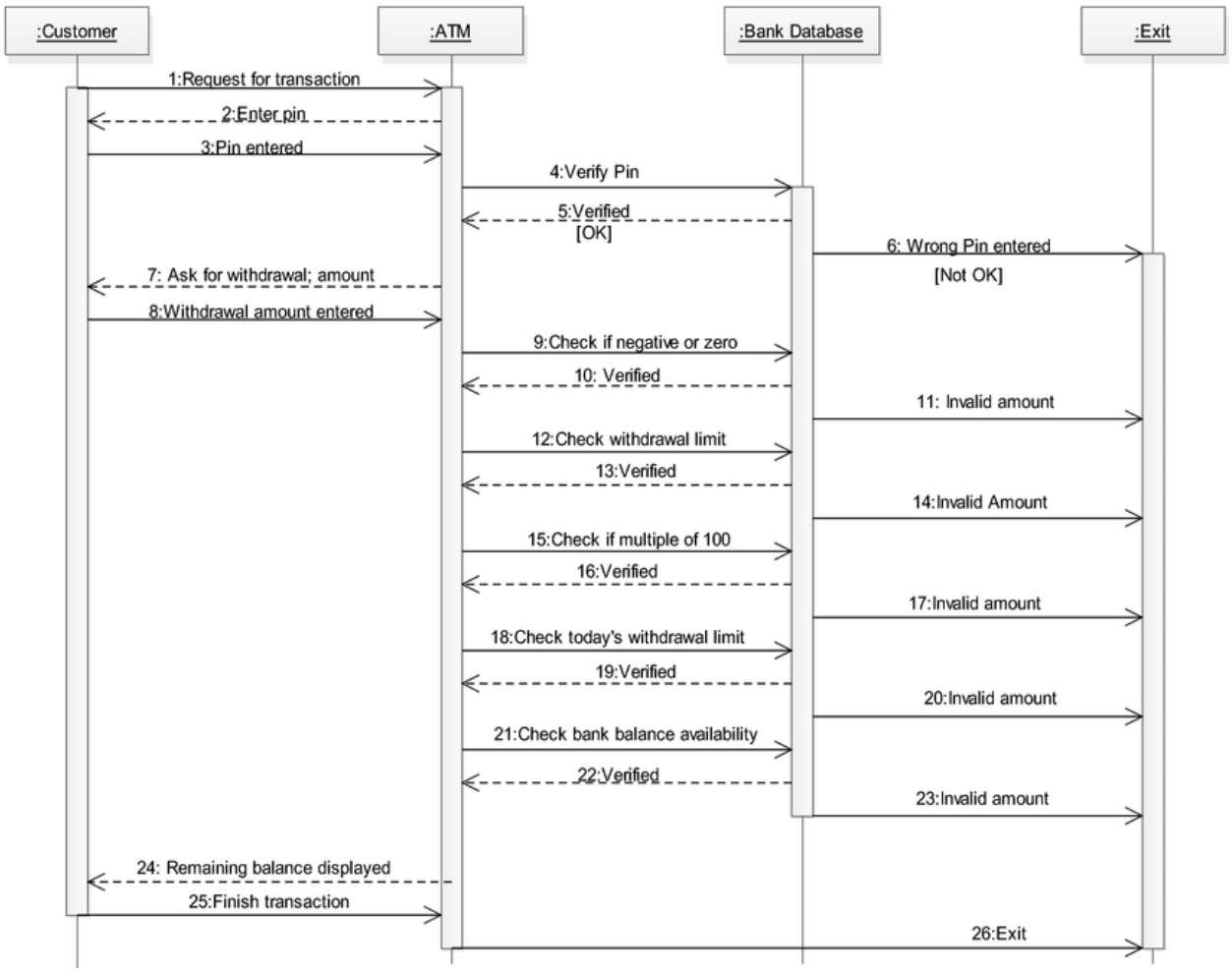

Fig. Sequence Diagram for ATM System

4.8 More on documenting architecture

To show **quality attributes** in architecture documentation, it is important to include a **rationale** explaining the choice of design approach, which should discuss the **quality attribute requirements** and any associated **trade-offs**. Architectural elements that provide services should have **quality bounds** assigned to them, which are documented in their **interfaces** or as **properties** of the elements. Being fluent in the "language" of a quality attribute allows for easier searching of architectural elements that meet the required attributes. The documentation should also have a **mapping** to requirements, illustrating how the **quality attributes** are satisfied. Additionally, each **quality attribute requirement** has stakeholders who need to know whether it is satisfied, and the documentation should include a **roadmap** to help identify where this information can be found.

When dealing with **frequently changing architectures**, especially in systems that change at **runtime** or have a high **release-and-deploy cycle**, it is crucial to address the challenge that the architecture evolves faster than the documentation. In such cases, it is important to **document what is true about all versions** of the system, recording **invariants** as you would for any architecture. Additionally, documenting how the architecture can change, including a **variability guide** showing how to add or replace components, is essential.

For **Agile projects**, adopting a **template** or standard organization to capture design decisions is recommended. Views should only be documented if there is a **strongly identified stakeholder** for that view. The sections of the template should be filled in as needed, and for information beyond views, it should only be recorded if it is relevant for downstream teams. The goal is not to produce a full architectural design document but to provide just enough design information to proceed with coding. Teams do not need to fill out every section of the template; they can mark sections as **N/A** if the information is not required. In Agile teams, models might be created during **brief discussions** or on a **whiteboard**, and a picture can be taken and used as the **primary presentation**.

4.9 Documenting Structure

UML **structural diagrams** are used to represent the elements of a system independent of time. These elements are typically depicted as **nouns** and are connected through various **structural** or **semantic** relationships. One of the most common structural diagrams is the **class diagram**, which describes the types of objects in a system and the relationships among them. A **class structure** in UML includes several key components: the class **name**, **fields**, and **methods/operations**. Some classes may be **abstract**, meaning they have an incomplete definition and serve as templates for other classes to inherit from. In UML, abstract class names and method names are often italicized, and the word **{Abstract}** is added to the name compartment.

An **interface** in UML defines a set of methods and fields that classes implement. The relationships between classes are key to understanding their interactions and include **generalization**, **association**, **aggregation**, and **composition**. **Generalization** represents a taxonomic relationship, where a more general class is extended by a more specific subclass. This is shown with an arrow pointing from the subclass to the superclass. It is often related to **inheritance**. **Name** is optional in generalization, but it typically describes a **verb** related to the problem domain. **Navigability** specifies the direction of the relationship and is denoted by a filled arrowhead. Relationships can be **bidirectional**, and **multiplicity** defines how many objects participate in the relation, with common values such as 0, 1, 0..1, 1..*, and *.

Association represents a connection between two classes, indicating that instances of both classes are related. It is denoted by a solid line in UML. An example of an association is a **Person**

who owns **zero or more Pets**. **Aggregation** is a specialized case of association, representing a **whole-part** relationship where the aggregate (whole) can exist independently of its parts (constituents). An example of aggregation is a **university**, which is comprised of many **colleges**. **Composition**, a stronger form of aggregation, also represents a whole-part relationship but with stronger ownership, meaning the **whole** cannot exist without its **parts**. An example of composition is a **human body**, which is composed of **1 head, 2 arms**, and **2 legs**. In composition, the parts tend to be of different classes, and the composite object cannot exist without its components.

Object diagrams provide snapshots of the objects in a system at a specific moment in time. These diagrams are particularly useful for illustrating complex relationships between objects. When **mapping documentation to code**, it is important to note that there is no one-to-one mapping from UML to any object-oriented language, as UML was designed to be language-independent. For example, UML mappings specific to **Java** do not represent a generalization and should be treated as unique cases. Additionally, when discussing association, it is crucial to specify **cardinality** and implement a **runtime mechanism** to enforce it, ensuring that the relationships between objects are properly managed at runtime.

4.11 Discussion and Questions

- How does understanding the target audience of an architecture document influence the way it is written and organized for effective communication among stakeholders?
- What factors should be considered when choosing which views to document in an architecture document, and how does the level of detail differ for different stakeholders such as management versus developers?
- How can the complexity of an architecture be effectively communicated through different views, and why is it important to focus on specific views rather than attempting to document every aspect?
- In what ways can documenting both the structure and behavior of the architecture provide a more comprehensive understanding of the system, and what challenges might arise in balancing these two aspects?

5. The role of architecture in software projects

5.1 The roles of architecture in software development:

The **system structure** serves as a blueprint, outlining how the **components, modules**, and **services** of the software system are organized and interact with each other. **Functional clarity** is achieved when functionalities are well-defined and distributed across different components, promoting **functional cohesion** and **separation of concerns**. The **fulfillment of quality attributes** is critical, as **design decisions** influence how attributes like **performance**, **security**, and **maintainability** are achieved. A strong architectural approach also aids in **risk mitigation**, addressing potential challenges upfront to prevent costly issues later in the project.

The architecture provides **guidance for development** by giving developers a clear vision of the system's structure and component interactions, making the development process more **organized** and less prone to errors. It fosters **collaboration** and **communication** by ensuring that stakeholders—such as **developers**, **testers**, and **clients**—share a common understanding of the system's design. Additionally, the architecture emphasizes **modifiability** and **extensibility**, allowing for changes to be made to specific components without impacting the entire system.

An effective **resource allocation** strategy is essential, as it defines how different parts of the system are deployed and interact to achieve **optimal performance** and **scalability**. The **integration strategy** outlines how various system components will integrate, especially in complex systems involving multiple subsystems or **third-party services**. **Technology selection** is another critical aspect, guiding the choice of the appropriate **frameworks** and **tools** that best meet the project's requirements and constraints.

Lastly, the architecture ensures **alignment with business goals**, making sure that the software system supports the overarching business objectives of the project. **Documentation** plays a vital role, providing a reference point for understanding the system's design and rationale, which is crucial for ongoing **maintenance** and future development.

5.2 Agility and architecture

Agile processes were developed in response to the need for projects to become more **responsive** to their stakeholders, deliver **functionality** that users care about more quickly, show **earlier progress** in the project's life cycle, and reduce the burden of documenting aspects of a project that are likely to change. These needs align with the goals of architecture, and agile projects must determine the right balance between **architecture planning** and flexibility. Specifically, it involves deciding how much **architecture** should be done up front versus

deferred until the **requirements** are more defined, and how much **documentation** is necessary and when it should be completed.

Initially, **Agile processes** were employed primarily for small- to medium-sized projects with short time frames, where they achieved considerable success. However, they were not often applied to larger projects, particularly those involving **distributed development**.

5.3 How Much Architecture?

There are two activities that can add time to a project schedule: **up-front design work** on the architecture and **up-front risk identification, planning, and resolution work**, and **rework** due to fixing defects and addressing modification requests. Intuitively, these two factors trade off against each other, with one potentially reducing the need for the other. Boehm and Turner plotted these two elements against each other for three hypothetical projects, each varying in size: one project with **10 KSLOC**, another with **100 KSLOC**, and one with **1,000 KSLOC**.

The results show that each project has a **sweet spot** where the balance between up-front work and rework is optimal. For the **10 KSLOC project**, the sweet spot is at the far left, meaning that devoting too much time to up-front work is unnecessary for a smaller project. For the **100 KSLOC project**, the sweet spot is around **20 percent** of the project schedule, and for the **1,000 KSLOC project**, it is around **40 percent** of the schedule. A project with **a million lines of code** is enormously complex, and it becomes difficult to imagine how **Agile principles alone** can cope with this complexity if there is no **architecture** to guide and organize the effort.

When it comes to **Agility and Documentation**, the key is to **write for the reader**. If the reader doesn’t need the information, it should not be written. However, it is essential to remember that the reader may be a **maintainer** or another newcomer not yet involved in the project, in which case, comprehensive documentation becomes necessary.

5.4 Agility and Architecture Evaluation

Architecture evaluation can certainly work as part of an **Agile** process. A core principle of Agile is meeting **stakeholders'** important concerns, and architecture evaluation plays a significant role in addressing these concerns. One of the key approaches to architecture evaluation is the **Architecture Tradeoff Analysis Method (ATAM)**. ATAM doesn't aim to analyze every aspect of an architecture, but instead focuses on a specific set of **quality attribute scenarios** that represent the most critical concerns of stakeholders. This makes it easy to tailor a **lightweight architecture evaluation** that is both practical and effective.

An example of Agile architecting can be seen in the development of the **WebArrow Web-conferencing system**. In this case, the architect and developers discovered that they needed to think and work in two different modes simultaneously. The first mode was **top-down**, where

they designed and analyzed architectural structures to meet the demanding **quality attribute requirements** and to balance various tradeoffs. The second mode was **bottom-up**, where they focused on a wide range of **implementation-specific** and **environment-specific constraints**, and worked to fashion solutions that addressed these challenges. This dual approach allowed the team to align both strategic architectural decisions and practical implementation needs.

5.5 Experiments to Make Tradeoffs

To analyze architectural tradeoffs, the team adopted an **agile architecture discipline** combined with a rigorous program of experiments, known as "spikes" in **Agile terminology**. These **experiments** were designed to answer critical questions regarding the system's architecture. For instance, the team needed to determine whether transitioning from **local flat files** to a **distributed database** would negatively impact user feedback time (**latency**). Additionally, they explored the scalability improvements, if any, from using **mod_perl** instead of standard **Perl**, and assessed how challenging it would be for the development and **quality assurance efforts** to convert to **mod_perl**. The team also investigated the capacity of the system, such as how many participants could be hosted by a single **meeting server**, and the correct **ratio** between **database servers** and **meeting servers**.

5.6 Best practices

If you are building a **large, complex system** with relatively **stable** and **well-understood requirements**, or if the project involves **distributed development**, investing time in a large amount of **up-front architecture work** will likely pay off. However, for **larger projects** with **unstable requirements**, it is advisable to start by quickly designing a **candidate architecture**, even if it lacks many details. This initial design should be **flexible**, and you should be prepared to **change** and **elaborate** on it as circumstances evolve. This includes making adjustments as you conduct **spikes** and **experiments**, and as both **functional** and **quality attribute requirements** emerge and become clearer. On **smaller projects** with uncertain requirements, it is essential to reach an agreement on the **major patterns** to be used, but avoid spending excessive time on **architecture design**, **documentation**, or **analysis** upfront.

5.7 Discussion and Questions

- How does the Agile Manifesto's emphasis on close-knit teams and frequent delivery of working software influence the way software architecture is approached in Agile projects?
- What challenges arise when attempting to apply Agile methodologies to large-scale projects, especially those with distributed development teams, and how can these challenges be addressed?
- Why is it important to combine Agile practices with architectural planning for large-scale projects, and how can this blend contribute to the success of the project?

- How do Agile architects strike a balance between delivering software iteratively and maintaining a sustainable architecture, and what strategies can be used to manage technical debt?

6. Evaluating software architectures

There are three primary forms of **evaluating software architectures**. The first is **evaluation by the designer** within the **design process**, where the designer assesses the architecture as it is being developed. The second form involves **evaluation by peers** during the design process, allowing colleagues to provide feedback and suggestions for improvement. The third form is **analysis by outsiders** once the architecture has been fully designed, where individuals not directly involved in the design process review and assess the architecture for its effectiveness and quality.

6.1 Evaluation by the Designer

Every time the **designer** makes a key **design decision** or completes a **design milestone**, the chosen alternatives and competing options should be carefully evaluated. **Evaluation by the designer** is a critical part of the **"generate-and-test" approach** to architecture design, where decisions are tested through evaluation to ensure they align with the overall goals.

The extent of the analysis needed depends on the **importance of the decision** and involves several factors. Firstly, the **importance of the decision** dictates the level of attention required; the more significant the decision, the more care should be taken to ensure it's well-made. Secondly, the **number of potential alternatives** plays a role.

The more alternatives there are, the more time might be needed for evaluation. However, it's crucial to quickly eliminate less promising alternatives so that only a small set of **viable options** remains.

Lastly, it's important to balance between **"good enough"** and **perfect**. Often, two alternatives may not differ significantly in their consequences. In such cases, it’s more effective to make a choice and proceed with the design process rather than spending excessive time seeking the absolute best option. Ultimately, the goal is not to overanalyze but to make timely decisions that move the design forward.

6.2 Peer Review

Architectural designs, much like code, can undergo **peer review** at various stages of the design process. A peer review can be conducted whenever a **candidate architecture** or at least a coherent, reviewable part of it exists. Typically, reviewers should allocate several hours, potentially up to half a day, for this process.

The **peer review steps** begin with the reviewers determining a set of **quality attribute scenarios** that will guide the review. These scenarios may be developed either by the review team or by

additional stakeholders. Next, the **architect** presents the portion of the architecture to be evaluated, ensuring that the reviewers individually understand it. At this stage, questions focus solely on clarifying the architecture.

For each scenario, the designer walks through the architecture, explaining how it satisfies the scenario. Reviewers then ask questions to confirm two key points: first, that the scenario is indeed satisfied, and second, whether any of the other scenarios might not be met. Potential problems identified during the review are captured, with **real problems** requiring action. These problems must either be fixed, or the **designers** and **project manager** must make an explicit decision to accept the problem along with its probability of occurrence.

6.3 Analysis by Outsiders

Outside evaluators can provide an **objective perspective** on an architecture, offering valuable insights that those directly involved in the project may overlook. The term "outside" is relative and can refer to various levels of separation. It may mean outside the **development project**, outside the **business unit** where the project resides but within the same **company**, or even outside the **company** altogether.

Outsiders are typically chosen for their **specialized knowledge** or significant experience in successfully evaluating **architectures**. Managers are often more inclined to pay attention to problems identified by an outside team, as they are perceived to be more impartial. An outside team is generally brought in to evaluate **complete architectures**, providing a thorough assessment that can identify potential issues or areas for improvement.

6.4 Contextual Factors for Evaluation

When performing an **architectural evaluation**, it is essential to have an **artifact** that clearly describes the architecture. This artifact serves as the foundation for the evaluation process. The next consideration is **who sees the results** of the evaluation. In some cases, evaluations are conducted with the full knowledge and participation of all relevant **stakeholders**, while in others, the evaluation may be more private.

The evaluation itself can be carried out by an **individual** or a **team**, depending on the scope and complexity of the assessment. Additionally, it is crucial to determine which **stakeholders** will participate in the evaluation. The process should include a method for gathering the **goals** and **concerns** of the key stakeholders regarding the system, ensuring their active involvement. Ultimately, the evaluation should address whether the system will meet the **business goals**, providing clarity on whether the architecture supports the broader objectives of the organization.

6.5 The Architecture Tradeoff Analysis Method (ATAM)

The **Architecture Tradeoff Analysis Method** (**ATAM**) has been used for over a decade to evaluate **software architectures** across a wide range of domains, including **automotive**, **financial**, and **defense** sectors. One of the strengths of the **ATAM** is that it is designed to be effective even when evaluators are not familiar with the architecture or its **business goals**.

Additionally, the system does not need to be constructed yet, and the evaluation process can accommodate a **large number of stakeholders**. This flexibility makes **ATAM** a valuable tool in diverse contexts and stages of the software development lifecycle.

Participants in the ATAM

In the **ATAM** process, participants are divided into several key groups. The **evaluation team** is external to the project whose architecture is being evaluated and typically consists of three to five people. In some cases, a single person may adopt several roles within the **ATAM**. It is essential that the evaluation team is composed of individuals who are recognized as **competent**, unbiased outsiders. The **project decision makers** are empowered to speak for the development project and have the authority to mandate changes. This group often includes the **project manager**, and if there is an identifiable **customer** funding the project, they may also be present or represented. The **architect** is always included in the evaluation, and their participation must be **voluntary**.

The **architecture stakeholders** are those with a vested interest in the architecture performing as expected. These stakeholders include **developers**, **testers**, **integrators**, **maintainers**, **performance engineers**, **users**, and **builders of systems** that interact with the one under consideration, among others. Their role is to articulate the specific **quality attribute goals** that the architecture should meet. For a large, **enterprise-critical architecture**, it is typical to enlist around 12 to 15 stakeholders to ensure comprehensive evaluation and input.

Outputs of the ATAM

The **outputs** of the **ATAM** process are varied and comprehensive. First, a **concise presentation** of the architecture is given, typically lasting about an hour. This presentation is followed by an **articulation of the business goals**, which are often seen by some participants for the first time and are carefully captured in the outputs. Next, **prioritized quality attribute requirements** are expressed through **quality attribute scenarios**, which help to clarify the desired outcomes for the system's performance.

Another key output is a **set of risks and non-risks**. A **risk** is an architectural decision that may lead to undesirable consequences based on quality attribute requirements, while a **non-risk** is a decision that, upon analysis, is deemed safe. The identified risks are used to form the basis of an **architectural risk mitigation plan**. Additionally, a **set of risk themes** is compiled. After completing the analysis, the evaluation team looks for overarching themes in the discovered

risks, which may indicate systemic weaknesses in the architecture or the architecture process and team. If left unaddressed, these risk themes could jeopardize the project's **business goals**.

The **mapping of architectural decisions** to quality requirements is another important output. For each quality attribute scenario reviewed, the architectural decisions that help achieve it are identified and documented. Furthermore, a **set of sensitivity and tradeoff points** is outlined. These are architectural decisions that significantly affect one or more quality attributes.

In addition to these tangible outputs, there are also **intangible results** from an ATAM-based evaluation. These include fostering a sense of **community** among the stakeholders, creating **open communication channels** between the architect and the stakeholders, and enhancing the overall understanding of the architecture, its strengths, and its weaknesses. While these intangible results are challenging to measure, they are just as important as the tangible ones and often have the longest-lasting impact.

Phases of the ATAM

Step 1: Present the ATAM
The first step in the **ATAM** process involves the **evaluation leader** presenting the ATAM to the assembled project representatives. This time is used to explain the process that everyone will be following, address any questions, and set the context and expectations for the remainder of the activities. The leader uses a standard presentation to describe the steps of the ATAM in brief and outlines the expected outputs of the evaluation.

Step 2: Present Business Drivers
In this step, everyone involved in the evaluation must understand the context for the system and the primary **business drivers** motivating its development. A **project decision maker** (ideally the **project manager** or the system's **customer**) presents a system overview from a business perspective. This presentation should cover the system's most important functions, relevant technical, managerial, economic, or political constraints, the business goals and context as they relate to the project, the major stakeholders, and the **architectural drivers** (i.e., the architecturally significant requirements).

Step 3: Present the Architecture
The **lead architect** or the architecture team then presents the architecture. The architect describes any **technical constraints**, such as operating system, hardware, or middleware prescribed for use, as well as other systems with which the system must interact. The architect outlines the **architectural approaches** (patterns or tactics) used to meet the requirements. This presentation should focus on conveying the essence of the architecture, avoiding ancillary details, and highlighting the views that were most important in creating the architecture and addressing the key **quality attribute concerns**.

Step 4: Identify Architectural Approaches
The **ATAM** focuses on analyzing the architecture by understanding its **architectural approaches**, particularly patterns and tactics. By now, the evaluation team will have a good idea of the patterns and tactics the architect used in designing the system, having reviewed the architecture documentation and the architect's presentation. The team may also spot approaches that were not explicitly mentioned. The evaluation team simply catalogs the identified patterns and tactics, which will be publicly captured and serve as the basis for later analysis.

Step 5: Generate Utility Tree
In this step, the **quality attribute goals** for the system are articulated in detail through a **quality attribute utility tree**. The utility tree helps make the requirements concrete by defining the relevant quality attribute requirements that the architects aimed to fulfill. The important quality attribute goals were named in Step 2, and now the evaluation team works with the project decision makers to identify, prioritize, and refine the system's most important quality attribute goals. These goals are expressed as **scenarios**, which populate the leaves of the utility tree and are assigned a rank of importance (High, Medium, Low).

Step 6: Analyze Architectural Approaches
The evaluation team then examines the highest-ranked scenarios one at a time, with the architect explaining how the architecture supports each one. The team, particularly the questioners, probes for the **architectural approaches** used to implement the scenario. As they work through the analysis, the team documents relevant **architectural decisions**, identifying and cataloging **risks**, **non-risks**, **sensitivity points**, and **tradeoffs**. The goal is not to conduct a comprehensive analysis but to establish a link between the architectural decisions and the quality attribute requirements that need to be satisfied.

Step 7: Brainstorm and Prioritize Scenarios
In this step, the stakeholders brainstorm scenarios that are operationally meaningful with respect to their individual roles. For example, a maintainer might propose a modifiability scenario, while a user may present a scenario concerning functionality or ease of operation. The purpose is to gauge the larger stakeholder community's perspective on what system success looks like for them. Once the scenarios are collected, they are prioritized by voting. The list of prioritized scenarios is compared with those from the utility tree exercise, and if discrepancies arise, they may indicate that there is a misalignment between what the architect had in mind and what the stakeholders wanted.

Step 8: Analyze Architectural Approaches
In this step, the evaluation team repeats the activities of Step 6 using the highest-ranked, newly generated scenarios. The team guides the architect in explaining how the relevant **architectural**

decisions contribute to realizing each of the new scenarios. This step might cover the top 5-10 scenarios, time permitting.

Step 9: Present Results
Finally, the evaluation team convenes privately to group the identified risks into **risk themes**, based on underlying concerns or systemic deficiencies. For example, risks related to inadequate or outdated documentation might be grouped into a risk theme about insufficient documentation. The team also identifies which **business drivers** are affected by each risk theme, elevating these issues to the attention of management. The results of the evaluation are summarized and presented to the stakeholders, including the documented **architectural approaches**, the prioritized list of **scenarios**, the **utility tree**, the identified **risks**, **nonrisks**, **sensitivity points**, and **tradeoff points**, as well as the **risk themes** and the business drivers they threaten.

6.6 Lightweight Architectural Evaluation

An **ATAM** is a substantial undertaking, requiring about **20 to 30 person-days** of effort from the evaluation team, with additional time commitments from the architect and stakeholders. This investment of time is justified for **large and costly projects**, where the risks of making a major mistake in the architecture are deemed unacceptable.

However, for smaller and less risky projects, a **Lightweight Architecture Evaluation** method, based on the ATAM, has been developed. This method can be completed in a **single day** or even a **half-day meeting**, and it is typically carried out entirely by members internal to the organization. While this lower level of scrutiny and objectivity may not probe the architecture as deeply, it is sufficient for less complex projects.

Since the participants are all internal and fewer in number compared to the full ATAM evaluation, giving everyone a chance to voice their opinion and achieving a **shared understanding** takes much less time. As a result, the steps and phases of a **Lightweight Architecture Evaluation** can be carried out more efficiently. A typical agenda for this process would span around **4-6 hours**.

6.11 Discussion and Questions

- Why is it crucial to evaluate the architecture of a system if it is important enough to design, and how does this evaluation contribute to the success of the project?
- What are the benefits and challenges of conducting evaluations at different stages of a project, such as during critical decision-making or through lightweight peer reviews?
- How does the ATAM (Architecture Tradeoff Analysis Method) facilitate comprehensive architecture evaluations, and why is it effective in aligning stakeholders' quality attribute requirements with architectural decisions?

- What is the role of risk identification in the ATAM evaluation process, and how can architects address these risks to improve the overall system design?
- How can a lightweight architecture evaluation, based on the ATAM, be conducted efficiently in an afternoon, and in what scenarios might this approach be more appropriate than a comprehensive evaluation?

7. Software architecture for the cloud

7.1 Basic Cloud Definitions (from NIST)

The **National Institute of Standards and Technology (NIST)** defines several key characteristics of cloud computing. **On-demand self-service** allows resource consumers to independently provision computing services, such as server time and network storage, as needed. This process is automated and requires no direct interaction with the service provider. **Ubiquitous network access** ensures that cloud services and resources are accessible over the network using standard mechanisms, enabling use by a diverse range of clients.

Resource pooling is another essential feature, where the cloud provider's computing resources are pooled to serve multiple consumers efficiently. This includes **location independence**, which means that consumers do not need to know or control the exact location of the resources they use. **Rapid elasticity** allows cloud capabilities to be provisioned and scaled quickly and elastically, accommodating varying demands.

Additionally, **measured service** ensures that resource usage is monitored, controlled, and reported, allowing consumers to be billed accurately based on their usage. Finally, **multi-tenancy** enables multiple consumers to share applications and resources without being aware of each other, providing cost efficiency and scalability.

7.2 Basic Service Models

Cloud computing offers three **basic service models** that cater to different types of consumers and needs.

Software as a Service (SaaS) targets end users, allowing them to use applications that run on the cloud infrastructure. Examples include email services and data storage platforms, where users interact with applications without managing the underlying infrastructure or platforms.

Platform as a Service (PaaS) is designed for developers or system administrators, providing them with the tools and environment necessary to deploy applications onto the cloud infrastructure. This model supports various programming languages and tools, enabling the development and management of applications without the need to manage the underlying hardware or software layers.

Infrastructure as a Service (IaaS) offers more fundamental computing resources, such as processing power, storage, and networks. This model is ideal for developers or system administrators who need the capability to provision and control these resources. Consumers

can deploy and run arbitrary software, including operating systems and applications, on the cloud infrastructure, providing maximum flexibility and control.

7.3 Deployment Models

Cloud computing supports several **deployment models**, each tailored to specific organizational and operational needs. A **private cloud** is an infrastructure that is owned and operated solely by a single organization. It is exclusively used for the organization's applications, ensuring greater control, security, and customization while remaining isolated from external entities.

A **public cloud**, on the other hand, is an infrastructure that is made available to the general public or a large industry group. It is owned and managed by a third-party organization that sells cloud services, offering scalability and cost-efficiency for a wide range of users.

The **community cloud** serves a specific group of organizations that share common concerns or objectives, such as compliance or security requirements. This infrastructure is shared among participating organizations to support their shared goals while maintaining some level of exclusivity.

Finally, a **hybrid cloud** combines two or more cloud models—such as private, public, or community clouds—into a unified system. While these components remain distinct entities, they are integrated to enable data and application portability, offering the flexibility to optimize resources and balance workloads effectively.

7.4 Economic Justification

Economic Justification for cloud computing is grounded in concepts like **economies of scale**, **equipment utilization**, and **multi-tenancy**. Large-scale data centers are inherently more cost-effective to operate than smaller ones. A **large data center**, comprising 100,000+ servers, benefits from reduced per-unit costs compared to a **small data center** with fewer than 10,000 servers.

Economies of Scale arise for several reasons. **Power costs**, which constitute 15–20% of total operating costs, are significantly reduced in large data centers. These centers achieve lower per-server power costs through **shared resources** like racks and switches, **negotiated power prices**, **strategic geographic location** in low-cost areas, and use of **alternative energy sources** such as wind farms or rooftop solar. **Infrastructure labor costs** also decrease, as system administrators in large data centers can manage over 1,000 servers, compared to around 150 in smaller facilities. Furthermore, **security and reliability** investments, including redundancy and disaster recovery, are more cost-efficient when amortized over a larger server base. Lastly, large data centers enjoy up to **30% discounts on hardware** due to bulk purchasing.

Equipment utilization is another key factor, driven by **virtualization technology**, which allows multiple applications and their operating systems to run on the same server. This improves server utilization by accommodating **variations in workload**. Random user access patterns tend to generate a **uniform server load**, while **time-of-day differences** allow workplace and consumer services to be co-located for efficiency. **Geographical time differences** and **seasonal fluctuations** (e.g., holidays or tax season) can also be leveraged to balance workloads. Additionally, resource usage can be optimized by pairing CPU-intensive services with I/O-heavy services and preparing for **usage spikes** during unexpected events like news or sporting events.

Finally, **multi-tenancy** is a cost-saving architecture where a single application serves multiple consumers, as seen in platforms like Salesforce. Multi-tenancy reduces costs by streamlining **help desk support**, enabling **simultaneous upgrades** for all consumers, and simplifying development and maintenance with a **single software version**. These combined factors underscore the financial advantages of cloud computing for organizations.

7.5 Basic Mechanisms of cloud implementation

The **basic mechanisms of cloud implementation** include several key components: **Hypervisor**, **Virtual Machine**, **File System**, and **Network**. The **hypervisor** is a software tool that enables the running of multiple virtual machines on a single physical machine. It manages the virtualization process, ensuring that each virtual machine operates independently and securely. A **virtual machine** has an isolated **address space** from other virtual machines, making it appear like a **bare-metal machine** from the application's perspective. It is also assigned an **IP address** and has **networking** capabilities, allowing it to connect to other systems. Additionally, a virtual machine can be loaded with any **operating system** or **application** that can run on the host machine's processor.

Each virtual machine has access to a **file system**. One widely used open-source cloud file system is the **Hadoop Distributed File System (HDFS)**. **HDFS** ensures **data availability** by using redundancy mechanisms. In a typical **HDFS write** scenario, an application writes data as it would to any file system. The client buffers the data until it reaches 64KB, then notifies the **NameNode** of its intention to write a new block. The **NameNode** returns a list of three **DataNodes** to store the block. The client sends the block to the first DataNode and informs it of the other two replicas. This first DataNode writes the block and forwards it to the second DataNode, which in turn does the same with the last DataNode. Once all DataNodes have written the block, they report to the client, which then commits the write to the **NameNode**.

In cases of failure, **HDFS** has mechanisms to ensure data integrity. If the client fails, the application detects the issue and retries the operation, ensuring that the write is not complete until committed by the client. If the **NameNode** fails, the **Backup NameNode** takes over, and a **log file** is maintained to prevent data loss. The **DataNodes** keep a true list of the blocks they

store, and the client retries the process. If a **DataNode** fails, the client or an earlier DataNode in the pipeline detects the failure and requests a different DataNode from the **NameNode**. Since each block is replicated three times, data is not lost even if a DataNode fails.

Regarding the **network**, every virtual machine is assigned an **IP address**, and each **TCP/IP message** includes the IP address in its header. The cloud's **gateway** can adjust the IP address for various purposes, facilitating proper routing and communication across the system.

7.6 Cloud Technologies

IaaS, **PaaS**, and **Databases** represent different layers of cloud technology, each offering distinct functionalities. **IaaS (Infrastructure as a Service)** involves a setup of servers that manage the foundational technologies of the cloud system. These servers are typically arranged in **clusters**, which may consist of thousands of servers. Some of these servers act as the infrastructure for the IaaS environment, while each server is equipped with a **hypervisor** as its base, enabling the virtualization process. The **IaaS architecture** is composed of several key components, including the **Cluster Manager**, responsible for overseeing each cluster, the **Persistent Object Manager** for managing data persistence, and the **Virtual Resource Manager**, which functions as a gateway for messages and manages additional resources. There is also a **File System Manager**, which operates similarly to **HDFS** and manages the network-wide file system. IaaS provides various services, such as the **automatic reallocation of IP addresses** in the event of a failure of a virtual machine instance, as well as **automatic scaling**, where new virtual machines are created or deleted based on system load.

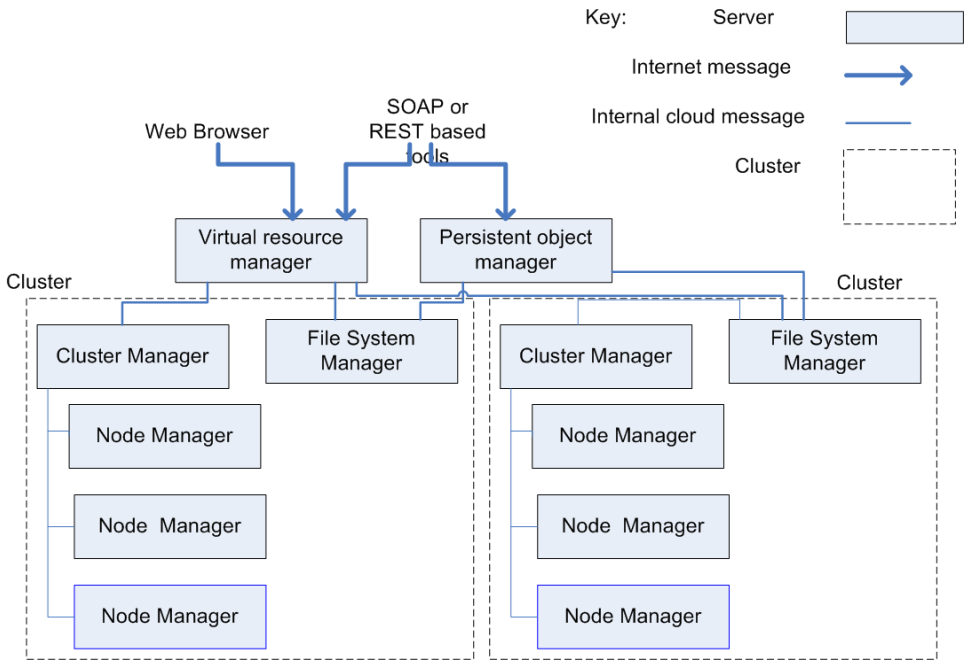

Fig. IaaS Architecture (an example)

On the other hand, **PaaS (Platform as a Service)** provides an integrated stack tailored for developers. For example, the **LAMP stack**—comprising **Linux**, **Apache**, **MySQL**, and **Python**—enables developers to write code in **Python**, while PaaS takes care of managing its deployment across the underlying layers of the stack.

When considering **databases**, the emergence of new data models was driven by the limitations of **relational databases**. The massive amounts of data collected from web systems, which is often processed sequentially, introduced overhead in **RDBMS** during creation and maintenance. According to the **CAP Theorem**, it is not possible to achieve **consistency**, **availability**, and **partition tolerance** simultaneously, leading to the introduction of alternative data models. The **relational model** is not optimal for all applications, which is why new models such as **key-value** and **document-centric** databases were developed. **Key-value** stores like **HBase** organize data with one column designated as a **key**, and the others as **values**. The data does not follow a schema, allowing flexibility with the keys and values, which can also be time-stamped to detect collisions after the fact, though **HBase** does not support transactions. In contrast, **document-centric databases** like **MongoDB** store data as objects rather than simple data entries, allowing for the storage of objects containing links to other objects. These databases do not enforce primary or secondary indexes; a field is either indexed or not, offering more flexibility in data retrieval.

However, certain aspects are omitted from these databases. **Transactions** are not supported, meaning no locking occurs, and the application must handle interference and consistency issues. Additionally, there are no predefined **schemas**, so applications must ensure correct naming conventions. Regarding **consistency**, the **CAP Theorem** suggests that consistency is often replaced by **eventual consistency** in these databases. Finally, **normalization** and **joins** are not supported in these systems, as performing a join requires an indexed field, and since indexing is not guaranteed, joins cannot be performed. This also means that **normalization** of tables is not a feature in these newer database models.

7.7 Architecting in a Cloud Environment

Architecting in a **cloud environment** requires careful consideration of several **quality attributes** that differ from traditional, non-cloud environments. These include **security**, **performance**, and **availability**.

Security in the cloud is particularly challenging due to **multi-tenancy**, which introduces additional risks not commonly seen in non-cloud setups. One major concern is **inadvertent information sharing**, where data may be unintentionally exposed due to shared resources. For example, information on a **disk** may remain accessible if the disk is reallocated to another user. Another security risk is the potential for a **virtual machine "escape"**, where one user could break out of the **hypervisor** and gain access to the underlying system. Although this scenario

has been purely academic so far, it remains a concern. Additionally, **side-channel attacks** could allow one user to detect information from another user by monitoring resources like the **cache**. Again, this threat is still mostly theoretical, but it highlights the importance of robust security measures. Finally, **denial of service (DoS)** attacks are a significant threat, where one user could consume excessive resources, denying others the resources they need. Given these risks, organizations must carefully assess potential security issues before deciding which applications to host in the cloud.

On the **performance** side, **auto-scaling** offers a significant advantage by automatically provisioning additional resources as the load increases. However, the **response time** for scaling may not always be quick enough to meet demand. As a result, architects need to understand the **resource requirements** of their applications and incorporate that knowledge into the design. It may also be beneficial to make applications **self-aware**, so they can proactively adjust to changing resource needs before performance issues arise.

When it comes to **availability**, failure is an expected occurrence in the cloud. With thousands of servers operating in parallel, some degree of failure is inevitable. Cloud providers typically guarantee that the cloud infrastructure will remain available, with some **notable exceptions**. However, **application developers** must assume that **instances** will fail at some point and should build in **detection** and **correction** mechanisms to handle failures when they occur. This ensures that the application remains resilient and continues to function even when parts of the underlying infrastructure are compromised.

7.8 Discussion and Questions

- How do the unique characteristics of cloud platforms, such as scalability and resource elasticity, impact the design and architecture of cloud-based applications?
- What are the critical architectural considerations an architect must account for when designing applications for the cloud, particularly in terms of security, performance, and availability?
- How does understanding the underlying infrastructure of a cloud cluster (e.g., distributed resources, load balancing) influence the decisions made in designing a cloud application?
- What strategies can architects use to ensure the security of cloud-based applications, and how can they mitigate potential risks such as data breaches or unauthorized access?
- In what ways can an architect ensure optimal performance and availability of a cloud application, and what tools or techniques might be used to monitor and adjust these factors during the application's lifecycle?

8. Emerging trends and future challenges in software architecture

8.1 Emerging Trends in Software Architecture

Microservices and Modular Architectures

Microservices involve breaking down applications into smaller, independent services that can be developed, deployed, and scaled independently. This modular approach enhances flexibility and allows teams to work on different services simultaneously.

Scalability is a key benefit, as each service can be scaled individually based on its demand. Fault isolation ensures that failures in one service do not impact the entire system, improving overall reliability.

Tools like **Kubernetes** and Docker play a pivotal role in managing and deploying microservices. Additionally, the adoption of domain-driven design and event-driven microservices is gaining momentum, enabling more efficient and context-aware application development.

Cloud-Native Architectures

Cloud-native architectures leverage the full potential of cloud platforms, allowing applications to be highly elastic, resilient, and **dynamically scalable**. By design, these applications can handle rapid changes in user demand without compromising performance.

Serverless computing, exemplified by platforms like AWS Lambda and Azure Functions, eliminates the need to manage underlying infrastructure, freeing developers to focus on functionality.

Hybrid cloud solutions and multi-cloud strategies are emerging trends, providing organizations with the flexibility to distribute workloads across multiple cloud providers while ensuring redundancy and cost optimization.

AI and Machine Learning Integration

The integration of AI and machine learning into software architecture is transforming how systems operate. Data pipelines are essential for ingesting and processing large volumes of data, while model deployment frameworks ensure **seamless integration of AI models** into production environments. Scaling inference workloads requires architectures that can handle the computational demands of real-time predictions.

Emerging patterns such as AI-first designs prioritize AI as a core component of systems, optimizing resource usage and decision-making processes. However, balancing these computational demands with scalability remains a significant challenge.

Edge Computing and IoT Architectures

Edge computing **processes data closer to its source**, such as IoT devices, reducing latency and saving bandwidth. This approach is particularly beneficial for applications requiring real-time responses, like **autonomous** vehicles and **smart cities**.

By processing data locally, edge computing minimizes the load on central servers and **reduces the risk of network bottlenecks**. Modern architectures often combine edge and cloud computing, enabling data aggregation and advanced analytics in the cloud while maintaining real-time processing at the edge.

Event-Driven Architectures (EDA)

Event-driven architectures enable systems to react to **real-time events**, making them highly suitable for applications like e-commerce, real-time analytics, and IoT. By **decoupling** event producers from consumers, EDAs enhance scalability and flexibility.

Platforms like **Apache Kafka** are commonly used to implement these architectures, allowing for high-throughput event streaming and processing. This approach ensures that systems remain responsive and adaptable to dynamic workloads.

Low-Code and No-Code Platforms

Low-code and no-code platforms empower users to develop applications with **minimal coding**, democratizing software development and enabling non-developers to contribute. These platforms simplify the development process, making it accessible to a broader audience.

However, architects must ensure that applications built on these platforms remain **scalable**, secure, and robust. The trend of using such platforms is driving innovation and reducing development time, particularly for prototyping and simple applications.

Resilience and Chaos Engineering

Resilience in software architecture focuses on building systems capable of **withstanding failures**. Chaos engineering, popularized by tools like Netflix's Chaos Monkey, involves **deliberately introducing failures** to test system behavior under adverse conditions.

This proactive approach helps identify vulnerabilities and improve system reliability. Integrating resilience testing into CI/CD pipelines ensures continuous validation of system robustness, preparing systems to handle unexpected disruptions gracefully.

8.2 Future Challenges in Software Architecture

Managing Complexity

As software systems grow in size and functionality, their **complexity increases**. **Distributed** architectures, in particular, introduce challenges in understanding and managing interdependencies.

Clear and comprehensive **documentation** is essential to mitigate complexity, providing a shared understanding for all stakeholders. Standardized **design patterns** and **automation tools** can streamline development and maintenance processes, reducing the cognitive load on architects and developers.

*Ensuring **Security** and Privacy*

The **dynamic** nature of software systems exposes them to evolving cybersecurity **threats**. **Architects must prioritize security** from the outset by adopting secure-by-design principles.

Compliance with regulations such as GDPR is critical to safeguarding user data and ensuring legal compliance. AI-driven threat detection systems can enhance security by identifying and responding to potential breaches in real-time. By integrating these measures, architects can build systems that are both secure and privacy-conscious.

Balancing Performance and Cost

Optimizing system **performance** while managing costs is a persistent challenge in software **architecture**. High-performance systems often require significant computational resources, which can drive up operational expenses.

Cost-aware architectures leverage techniques like autoscaling to allocate resources dynamically based on demand. Utilizing spot instances in cloud computing can further reduce costs by taking advantage of discounted and unused capacity..

Data Management and Governance

The exponential growth of data presents challenges in storage, processing, and ensuring data quality. **Real-time processing architectures** are essential for applications requiring immediate insights, such as financial transactions and monitoring systems.

Adherence to data governance frameworks helps maintain data integrity, ensuring that data is accurate, consistent, and compliant with regulations. Effective data management strategies are crucial for deriving actionable insights from large datasets.

Addressing Technical Debt

Technical debt accumulates when short-term solutions are prioritized over long-term maintainability. Over time, this can hinder system performance and scalability. Regular **refactoring and architectural reviews** [10] are essential to address technical debt, ensuring that systems remain agile and adaptable.

Sustainability in Architecture

With growing concerns about environmental impact, sustainability has become a key consideration in software architecture. **Energy-efficient architectures** prioritize resource optimization, reducing the carbon footprint of software systems. Green computing practices,

such as using renewable energy sources and **optimizing algorithms** for efficiency, contribute to sustainable development. **Architects** play a crucial role in designing systems that balance functionality with environmental responsibility.

8.3 Discussion and Questions

- How do you see AI impacting architectural decision-making?
- What strategies would you adopt to manage technical debt in large systems?
- Can we balance innovation with sustainability in software design?

References

1. Bass, L., Clements, P., & Kazman, R. (2022). *Software architecture in practice*. Addison-Wesley.
2. Richards, M. (2015). *Software architecture patterns*. O'Reilly Media, Inc.
3. Clements, P., Kazman, R., & Klein, M. (2002). *Evaluating software architectures: methods and case studies*. Addison-Wesley.
4. Hofmeister, C., Nord, R., & Soni, D. (2000). *Applied software architecture*. Addison-Wesley Professional.
5. Medvidovic, N., & Taylor, R. N. (2010, May). *Software architecture: foundations, theory, and practice*. In 2010 ACM/IEEE 32nd International Conference on Software Engineering (Vol. 2, pp. 471-472). IEEE.
6. Hasselbring, W. (2018). Software architecture: Past, present, future. In *The Essence of Software Engineering* (pp. 169-184). Springer, Cham.
7. Muccini, H., & Moghaddam, M. T. (2018, September). Iot architectural styles. In *European Conference on Software Architecture* (pp. 68-85). Springer, Cham.
8. Patwardhan, A., & Patwardhan, R. (2016). XML Entity Architecture for Efficient Software Integration. *arXiv preprint arXiv:1606.07941*.
9. Seifermann, S., Taspolatoglu, E., Reussner, R. H., & Heinrich, R. (2016). Challenges in secure software evolution-the role of software architecture. In *3rd Collaborative Workshop on Evolution and Maintenance of Long-Living Software Systems, ser. Softwaretechnik-Trends* (Vol. 36, No. 1, pp. 8-11).
10. Samarthyam, G., Suryanarayana, G., & Sharma, T. (2016, September). Refactoring for software architecture smells. In *Proceedings of the 1st International Workshop on Software Refactoring* (pp. 1-4).
11. Woods, E. (2016). Software architecture in a changing world. *IEEE Software*, *33*(6), 94-97.
12. Wolff, E. (2016). Microservices: flexible software architecture. Addison-Wesley Professional.
13. Malavolta, I., & Capilla, R. (2017, April). Current research topics and trends in the software architecture community: Icsa 2017 workshops summary. In *2017 IEEE International Conference on Software Architecture Workshops (ICSAW)* (pp. 1-4). IEEE.