Software Architecture

Systems Analysis and Design
(40418)

Sharif University of Technology

Software Architecture

- The software architecture of a program or computing system is the structure or structures of the system, which comprise
  1. software elements,
  2. the externally visible properties of those elements,
     - those assumptions other elements can make of an element
  3. and the relationships among them.

- Software architecture constitutes a relatively small, intellectually graspable model for how a system is structured and how its elements work together.

- The architecture is not the operational software.

Why Architecture?

from a technical perspective

1. **Communication among stakeholders.** Software architecture represents a common abstraction of a system that all the stakeholders can use as a basis for mutual *understanding*, *negotiation*, *consensus*, and *communication*.

Why Architecture?

from a technical perspective

- **2 Early design decisions.** The early decisions carry weight far out of proportion to their individual gravity with respect to the system's remaining development, its deployment, and its maintenance life. It is also the earliest point at which design decisions governing the system to be built can be analyzed.

- defines constraints on Implementation
- inhibits or enables a system's quality attributes
- predicting system qualities by studying the Architecture
- makes It easier to reason about and manage change
- enables more accurate cost and schedule estimates

Why Architecture?

from a technical perspective

3. **Transferable abstraction of a system.** The architecture can be applied to other systems exhibiting similar quality attribute and functional requirements and can promote large-scale re-use.

- The earlier in the life cycle re-use is applied, the greater the benefit that can be achieved.

- Software Product Lines Share a Common Architecture
- Systems Can Be Built Using Large, Externally Developed Elements
- Less Is More: It Pays to Restrict the Vocabulary of Design Alternatives
- An Architecture Permits Template-Based Development
- An Architecture Can Be the Basis for Training

Architectural Views

- Modern systems are more than complex enough to make it difficult to grasp them all at once.
- We restrict our attention at any one moment to one (or a small number) of the software system's structures.

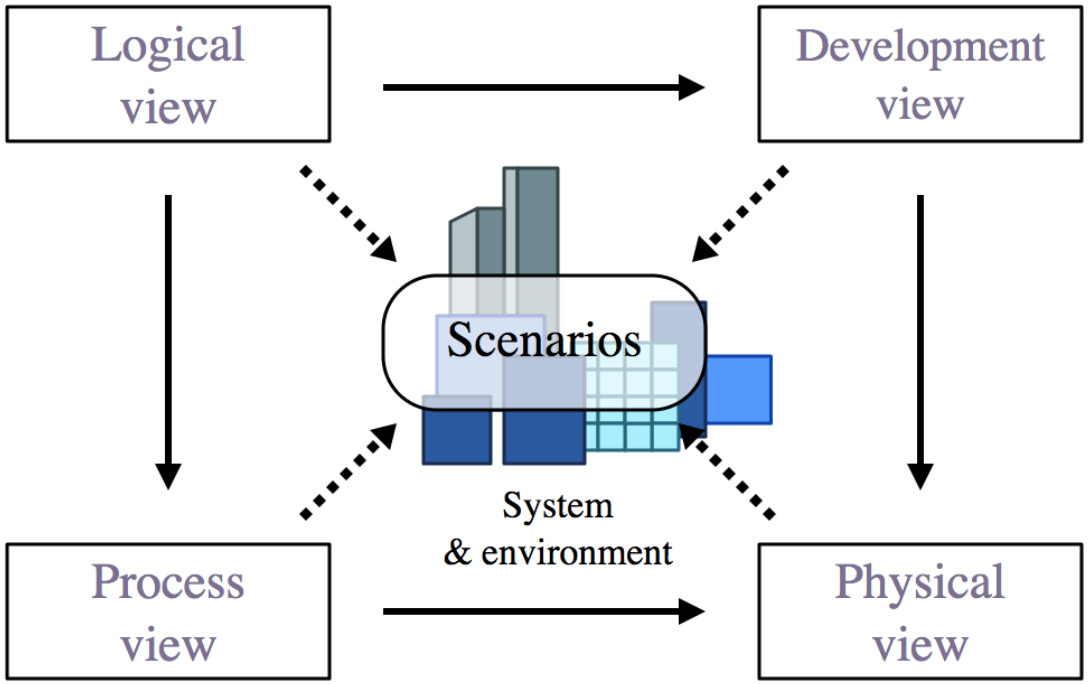
- A *view* is a representation of a coherent set of architectural elements, and their relations, from the perspective of a related set of concerns.

- A set of *Views* are used to describe the system from the viewpoint of different stakeholders, such as end-users, developers, system engineer, and project managers.

Architectural Views

"4+1" view model by Philippe Kruchten

- "describing the architecture of software-intensive systems, based on the use of multiple, concurrent views".



Logical View

- The logical view is concerned with the object model of the design (when an object-oriented design method is used)

- UML models
- Class diagram, Communication diagram

Development View

- The development view illustrates a system from a programmer's perspective;
  - also known as the Implementation view;
- Describing the static organization of the software in its development environment; mapping software to the development environment.

- It is concerned with software management.

- UML models
- Component diagram to describe system components.
- Package diagram to represent the development view.

Process View

- The process view deals with the dynamic aspects of the system.
- Explains the system processes and how they communicate, and focuses on the runtime behavior of the system.

- The process view addresses concurrency, distribution, integrators, performance, and scalability, etc.

- UML models
  - Activity diagram, Sequence diagram

Physical View

- The physical view depicts the system from a system engineer's point-of-view;
- mapping(s) of the software onto the hardware and reflects its distributed aspect.

- It is concerned with the topology of software components on the physical layer, as well as the physical connections between these components.

- This view is also known as the deployment view.

- UML models
- Deployment diagram

Scenarios View

- Selected use cases or scenarios are used to illustrate the architecture serving as the 'plus one' view.

- The scenarios describe sequences of interactions between objects, and between processes.

- They are used
  - to identify and illustrate architectural elements,
  - to validate that the structures were not in conflict with each other,
  - to validate if the models in fact describe a system meeting its requirements, and
  - as a starting point for tests of an architecture prototype.

Modelling Architecture

- Architecture is typically expressed as a collection of models.
  - documentation must be created to show relation between models.

- Collection of models helps to answer whether the proposed architecture meets the specified requirements.
  - to understand the system
  - to determine amount of reuse from other systems and the reusability of the system being designed
  - to provide blueprint for system construction
  - to reason about system evolution
  - to analyse dependencies
  - to support management decisions and understand risks

Quality Attributes

- Business considerations determine qualities that must be accommodated in a system's architecture.
  - Availability, Modifiability, Performance, Security, Testability, Usability
  - Time to market, Cost and benefit, Targeted market
- Achievement of quality attributes is critical to the success of a system.
- Architecture is the first stage in software creation in which quality requirements could be addressed.

Quality Attributes

- Functionality and quality attributes are orthogonal.
  - it is possible to independently choose a desired level of each.
  - an architect's choice will determine the relative level of quality.
  - BUT, any level of any quality attribute is not achievable with any function!

- Software architecture constrains the of allocation of functionality to structure as other quality attributes are important.

Quality Attributes

- Achieving quality attributes must be considered throughout design, implementation, and deployment.
  - No quality attribute is entirely dependent on design, nor is it entirely dependent on implementation or deployment.
  - Satisfactory results are a matter of getting the big picture (architecture) AS WELL AS the details (implementation) correct.

- To ensure quality, a good architecture is necessary, but NOT sufficient.
  - Architecture is critical to the realization of many qualities, but by itself, is unable to achieve qualities.

Quality Attributes

1. **Qualities of the system** that focus on availability, modifiability, performance, security, testability, and usability.

2. **Business qualities** (such as time to market) that are affected by the architecture.

3. **Qualities**, such as conceptual integrity, that are **about the architecture itself** although they indirectly affect other qualities, such as modifiability.

System Quality Attributes

- Six most common and important system quality attributes
  - Availability
  - Modifiability
  - Performance
  - Security
  - Testability
  - Usability

- other attributes can be found in the research literature and in standard software engineering textbooks
  - Scalability
  - Reliability
  - Interoperability
  - …

Business Qualities

- Goals center on *cost*, *schedule*, *market*, and *marketing* considerations.

- Time to market
- Cost and benefit
- Projected lifetime of the system
  - long lifetime => modifiability, scalability, and portability
- Targeted market
  - platforms => the size of the potential market, portability and functionality
- Rollout schedule
  - flexibility and customizability of the architecture
- Integration with legacy systems

Architecture Qualities

- Qualities directly related to the architecture itself that are important to achieve.
- **Conceptual integrity** is the underlying theme or vision that unifies the design of the system at all levels.
- **Correctness and completeness** are essential for the architecture to allow for all of the system's requirements and runtime resource constraints to be met.
- **Buildability** allows the system to be completed by the available team in a timely manner and to be open to certain changes as development progresses.

System Quality Attributes

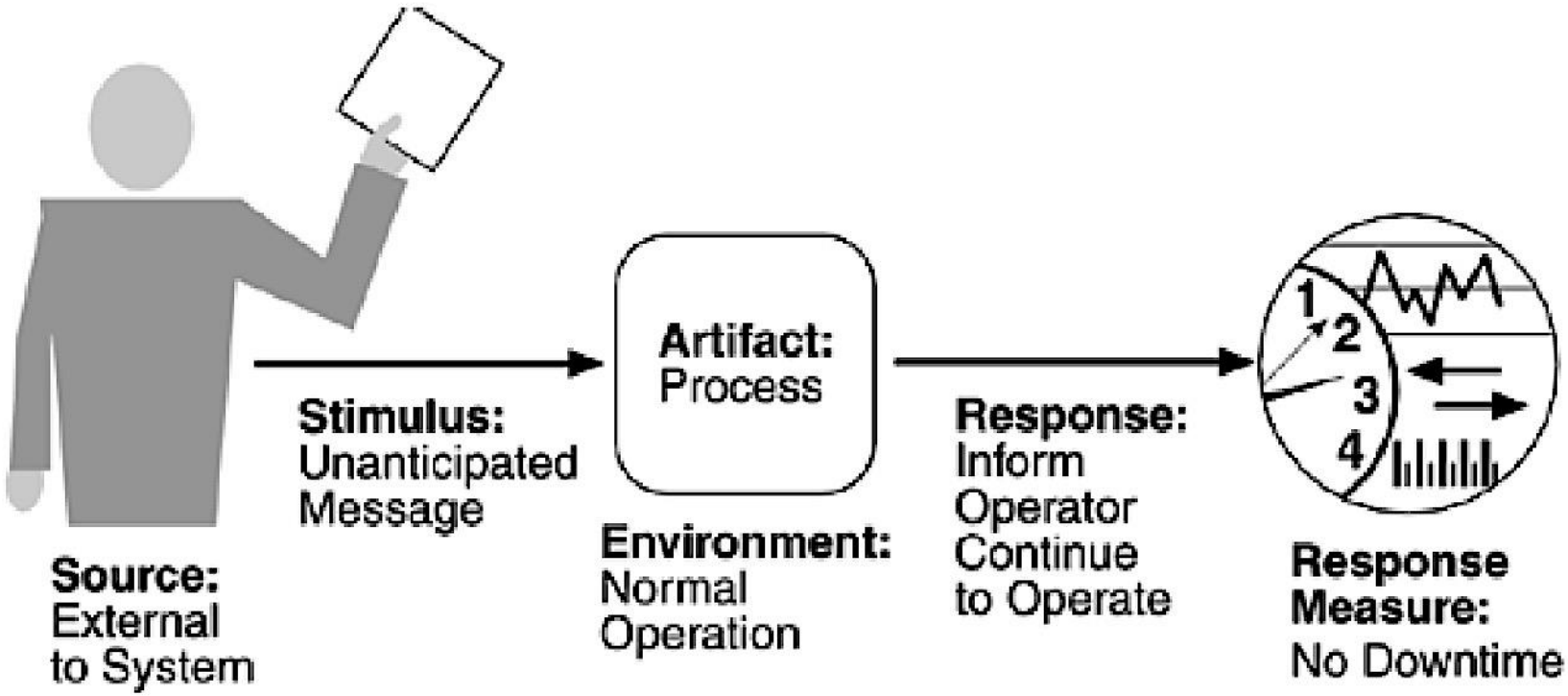
How specified and measured

- **Quality Attribute Scenario-** a quality-attribute-specific requirement
  1. **Source of stimulus.** This is some entity (a human, a computer system, or any other actuator) that generated the stimulus.
  2. **Stimulus**. The stimulus is a condition that needs to be considered when it arrives at a system.
  3. **Environment**. The stimulus occurs within certain conditions. The system may be in an overload condition or may be running when the stimulus occurs, or some other condition may be true.
  4. **Artifact**. Some artifact is stimulated. This may be the whole system or some pieces of it.
  5. **Response**. The response is the activity undertaken after the arrival of the stimulus.
  6. **Response measure.** When the response occurs, it should be measurable in some fashion so that the requirement can be tested.

Read more: Software Architecture in Practice, By Len Bass, Paul Clements, Rick Kazman, Addison Wesley.

System Quality Attributes

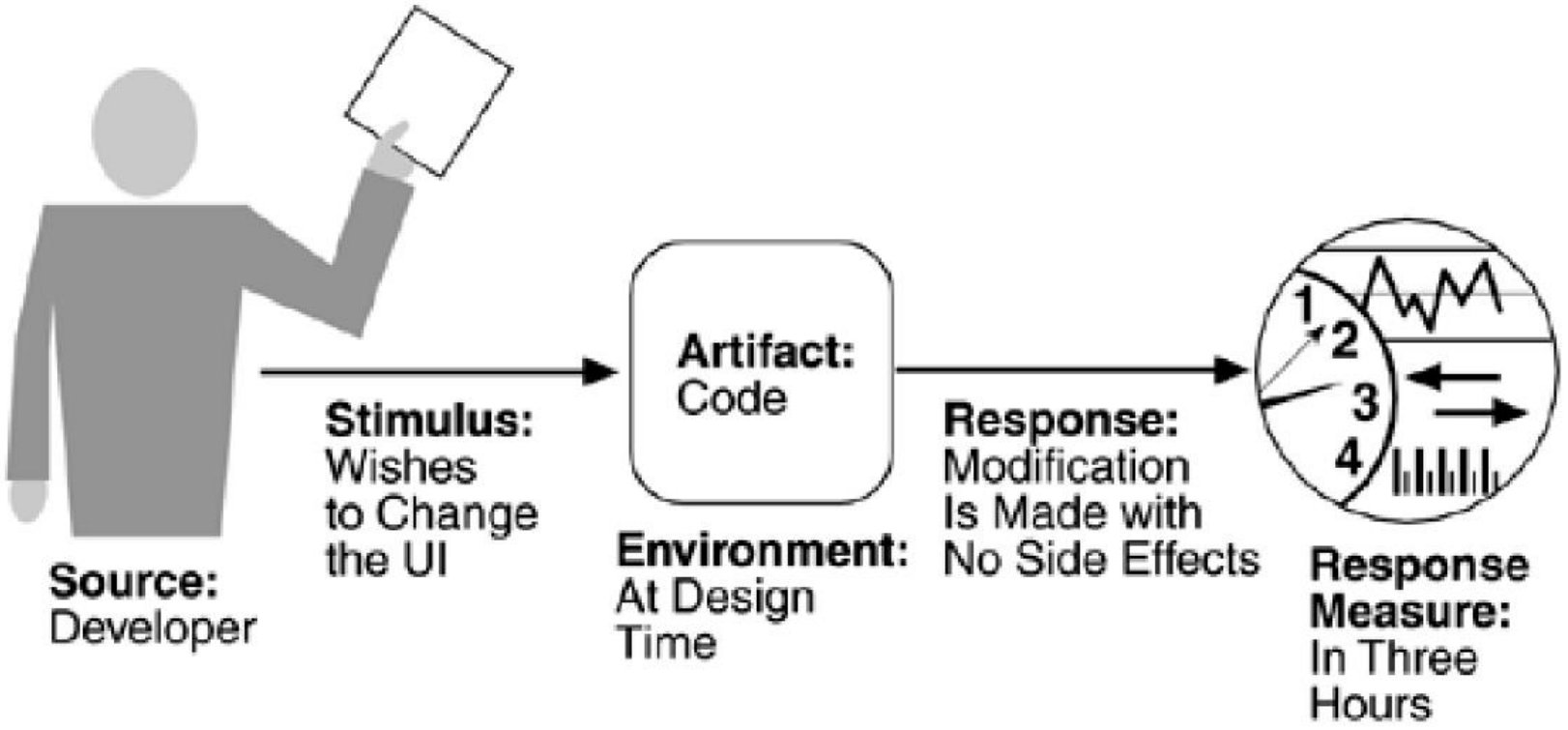
Quality Attribute Scenario- Availability



Read more: Software Architecture in Practice, By Len Bass, Paul Clements, Rick Kazman, Addison Wesley.

System Quality Attributes

Quality Attribute Scenario- Modifiability



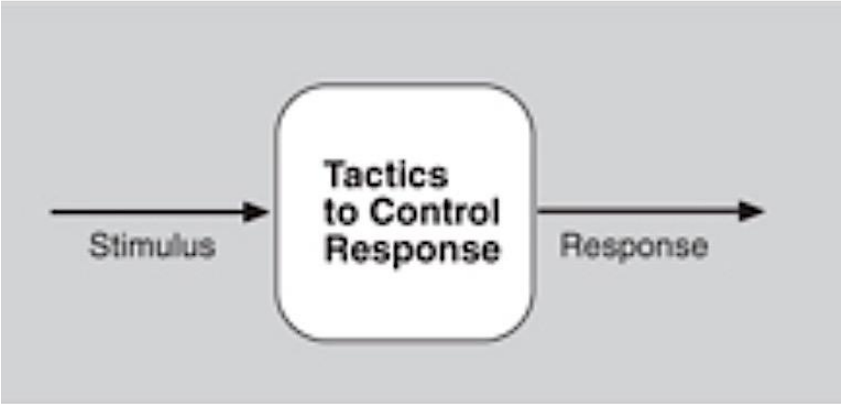
Read more: Software Architecture in Practice, By Len Bass, Paul Clements, Rick Kazman, Addison Wesley.

Achieving Qualities

Architectural Tactics

- The achievement of quality attributes relies on fundamental design decisions.

- A *tactic* is a design decision that influences the achievement of a quality attribute response—tactics directly affect the system's response to some stimulus.



Architectural Tactics

- The focus of a tactic is on a single quality attribute response. Within a tactic, there is no consideration of tradeoffs.
- A collection of tactics is an architectural **strategy**.
- The tactics, similar to design patterns, are design techniques that architects have been using for years.
- The tactics needs to be refined by designers to make each tactic concrete.
  - The application of a tactic depends on the context.
    - *Manage sampling rate* is relevant in some real-time systems but not in all real-time systems and certainly not in database systems.

Availability

- Concerned with system failure and its associated consequences.
  - A system failure occurs when the system no longer delivers a service consistent with its specification.
  - Such a failure is observable by the system's users

- Availability tactics
  1. will keep faults from becoming failures, or
  2. at least bounds the effects of the fault and make repair possible.

1. Fault Detection; recognizing fault
2. Fault Recovery; preparing for recovery and making the system repair
3. Fault Prevention

Modifiability

- Design must be easy to change.

- Two classifications of affected software units
  1. Directly affected
  2. Indirectly affected

- **Directly** affected units' responsibilities change to accommodate a system modification.

- **Indirectly** affected units' responsibilities do not change, but implementations must be revised.

Modifiability

Tactics

- Tactics for minimizing the number of software units affected by a change focus on clustering the *anticipated* changes

- **Anticipate expected changes**: Identify design decisions that are most likely to change, and encapsulate each in its own software unit

- **Cohesion:** Keeping software units highly cohesive increases the chances that a change to the system's responsibilities is confined to the few units that are assigned those responsibilities

- **Generality**: The more general the software units, the more likely change can be accommodated by modifying a unit's inputs rather than modifying the unit itself

Modifiability

Tactics

- Tactics for minimizing the impact on indirectly affected units focus on reducing dependencies

- **Coupling**: Lowering coupling reduces the likelihood that a change to one unit will ripple to other units

- **Interfaces**: If a unit interacts with other units only through their interfaces, changes to one unit will not spread beyond the unit's boundary unless its interface changes

- **Multiple interfaces**: A unit modified to provide new data or services can offer them using a new interface to the unit without changing any of the unit's existing interfaces

Performance

- Performance attributes describe constraints on system speed and capacity.
- **Response time**: How fast does our software respond to requests?
- **Throughput**: How many requests can it process per minute?
- **Load**: How many users can it support before response time and throughput start to suffer?

Performance
Tactics

- Improve utilization of resources

- Manage resource allocation more effectively
  - First-come/first-served: Requests are processed in the order in which they are received
  - Explicit priority: Requests are processed in order of their assigned priorities
  - Earliest deadline first: Requests are processed in order of their impending deadlines

- Reduce demand for resources

Security

- Two key architectural characteristics particularly relevant to security.

1. **Immunity**: ability to thwart an attempted attack
    - The architecture encourages immunity by:
        - Ensuring all security features are included in the design
        - Minimizing exploitable security weaknesses

2. **Resilience**: ability to recover quickly and easily from an attack
    - The architecture encourages resilience by:
        - Segmenting functionality to contain attack
        - Enabling the system to quickly restore functionality

Testability

- Easier testing when an increment of software development is completed

- Two categories of tactics

1. Providing input and capturing output
   - **Record/playback**; capturing information crossing an interface and recording output
   - **Separate interface from implementation**; allows substitution of implementations for testing purposes
   - **Specialize access routes/interfaces**; specialized testing interfaces allows the capturing or specification of variable

2. Internal monitoring: tactics based on internal state to support the testing process
   - **Built-in monitors**. The component can maintain state, performance load, capacity, security, or other information accessible through an interface.

Usability

- Usability reflects the ease in which a user is able to operate the system; how easy it is for the user to accomplish a desired task and the kind of support the system provides to the user.
  - Some user-initiated commands require architectural support
  - There are some system-initiated activities for which the system should maintain a model of its environment

- Two categories of tactics

  1. **Runtime**; supports the user during system execution

  2. **Design-time**; supports the interface developer at design time
     - Separate the user interface from the rest of the application; User interface should reside in its own software unit

Architectural Tactics

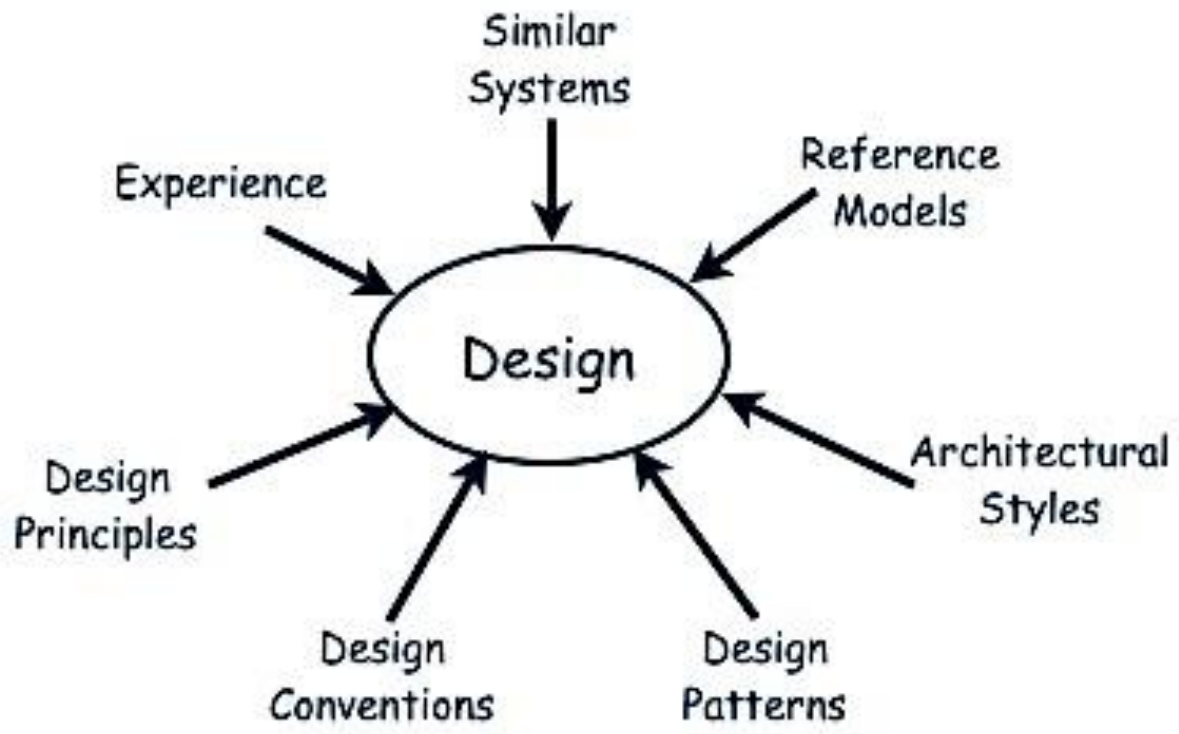
why

- Design patterns are *complex*; they are often difficult to apply as is; architects need to modify and adapt them. By understanding the role of tactics, an architect can more easily assess the options for augmenting an existing pattern to achieve a quality attribute goal.

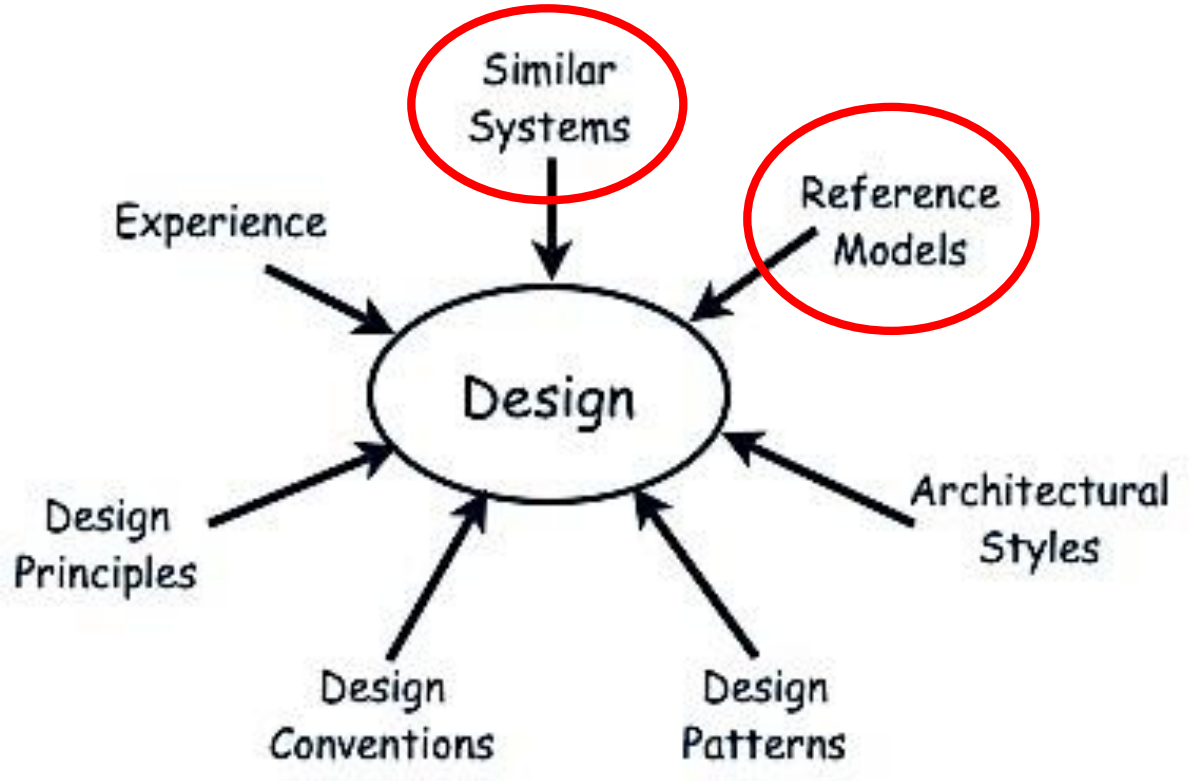
- If *no* pattern exists to realize the architect's design goal, tactics allow the architect to construct a design fragment from "first principles." Tactics give the architect insight into the properties of the resulting design fragment.

- By *cataloging* tactics, we provide a way of making design more systematic within some limitations. The choice of which tactic to use depends on factors such as tradeoffs among other quality attributes and the cost to implement

How to Design



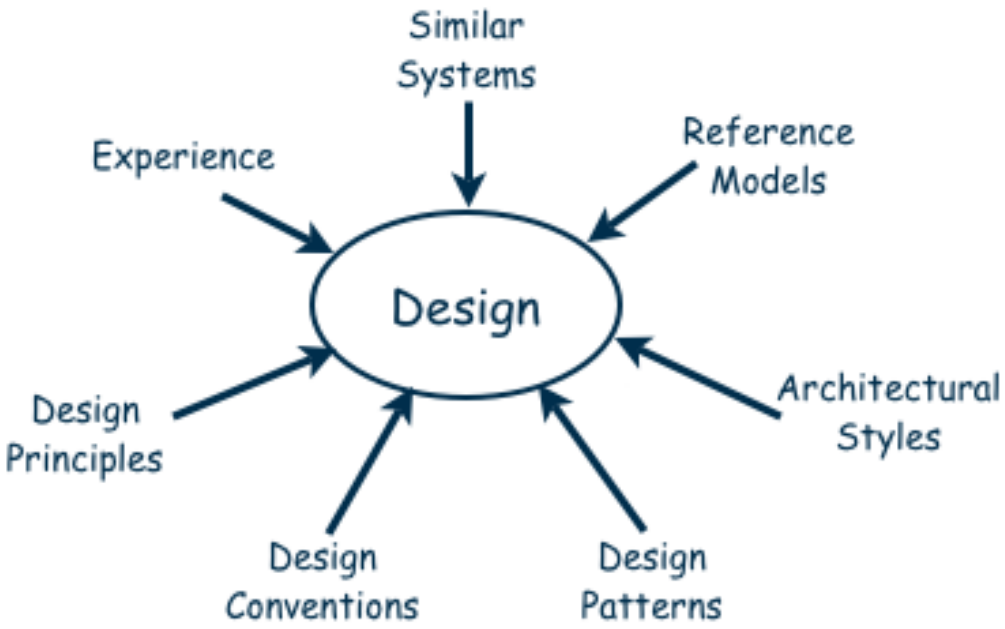
How to Design



- We may solve problems by reusing and adapting solutions from similar problems/systems or reference models.

How to Design

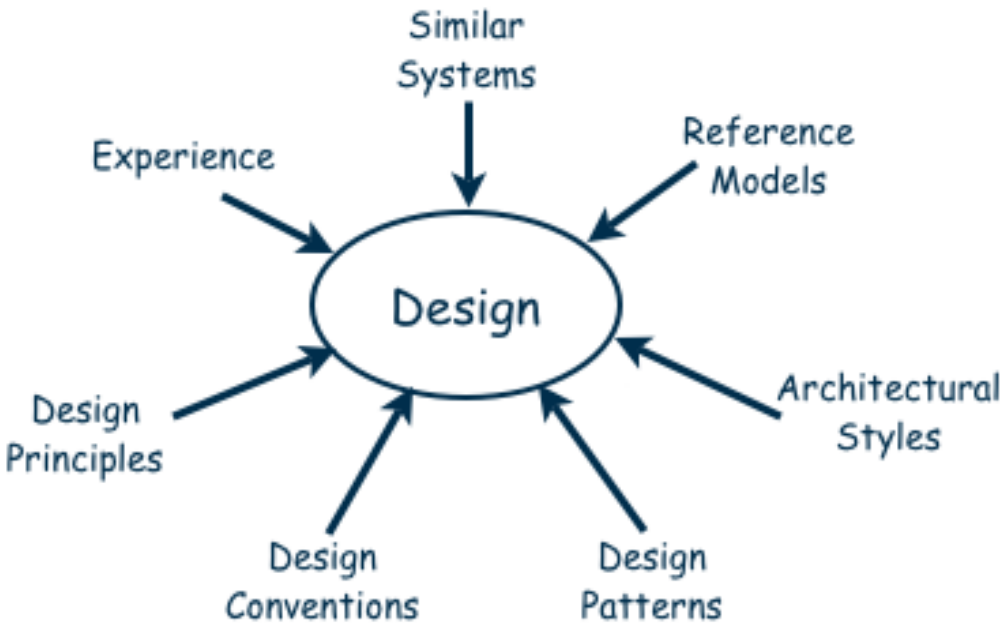
- Examples of good design.
- Design principles: descriptive characteristics of good design.
- Architectural styles/patterns: generic solutions.
- Design patterns: generic solutions for making lower-level design decisions.



- Design convention or idiom: collection of design decisions and advice that, taken together, promotes certain design qualities.

How to Design

- Examples of good design.
- **Design principles: descriptive characteristics of good design.**
- Architectural styles/patterns: generic solutions.
- Design patterns: generic solutions for making lower-level design decisions.



- Design convention or idiom: collection of design decisions and advice that, taken together, promotes certain design qualities.

Design Principles

- Descriptive characteristics of good design.

- Abstraction
- Modularity: coupling and cohesion
- Information hiding
- Limit complexity
- Hierarchical structure

Modularity

- A design is modular when each activity of the system is performed by exactly one software unit, and when the inputs and outputs of each software unit are well-defined.

- A software unit is well-defined if its interface accurately and precisely specifies the unit's externally visible behaviour.

- Structural criteria which tell us something about individual modules and their interconnections
  - Coupling
  - Cohesion

Coupling

- The strength of the connection between modules

- content coupling
- common coupling
- external coupling
- control coupling
- stamp coupling
- data coupling

http://pages.cpsc.ucalgary.ca/~eberly/Courses/CPSC333/Lectures/Design/coupling.html

Cohesion

- The glue that keeps a module together

- coincidental cohesion
- logical cohesion
- temporal cohesion
- procedural cohesion
- communicational cohesion
- sequential cohesion
- functional cohesion

- data cohesion (to cater for abstract data types)

http://pages.cpsc.ucalgary.ca/~eberly/Courses/CPSC333/Lectures/Design/cohesion.html

Cohesion & Coupling

strong cohesion & weak coupling

$\Rightarrow$simple interfaces

$\Rightarrow$

- simpler communication
- simpler correctness proofs
- changes influence other modules less often
- reusability increases
- comprehensibility improves

Information hiding

- Each module has a secret

- Design involves a series of decision: for each such decision
  1. who needs to know and
  2. who can be kept in the dark

- Information hiding is strongly related to
  - abstraction: if you hide something, the user may abstract from that fact
  - coupling: the secret decreases coupling between a module and its environment
  - cohesion: the secret is what binds the parts of the module together

Complexity

- Measure certain aspects of the software (lines of code, # of if-statements, depth of nesting, …)

- Use these numbers as a criterion to assess a design, or to guide the design

higher value $\Rightarrow$ higher complexity $\Rightarrow$ more effort required (= worse design)

- Two kinds:
  - **intra-modular:** inside one module
  - **inter-modular:** between modules

Intra-modular Complexity

- Attributes of a single module

1. measures based on size
   - e.g., counting lines of code, counting operators and operands, size of vocabulary, program length

2. measures based on structure (e.g., control structures, data structures, or both)
   - e.g., average number of instructions
   - McCabe's cyclomatic complexity: number of linearly independent paths through the program

Inter-modular Complexity

- Looks at the complexity of the dependencies between modules

- Draw modules and their dependencies in a graph
- then the arrows connecting modules may denote several relations, such as:
  - A contains B
  - A precedes B
  - **A uses B (mostly important)**

The *uses* relation

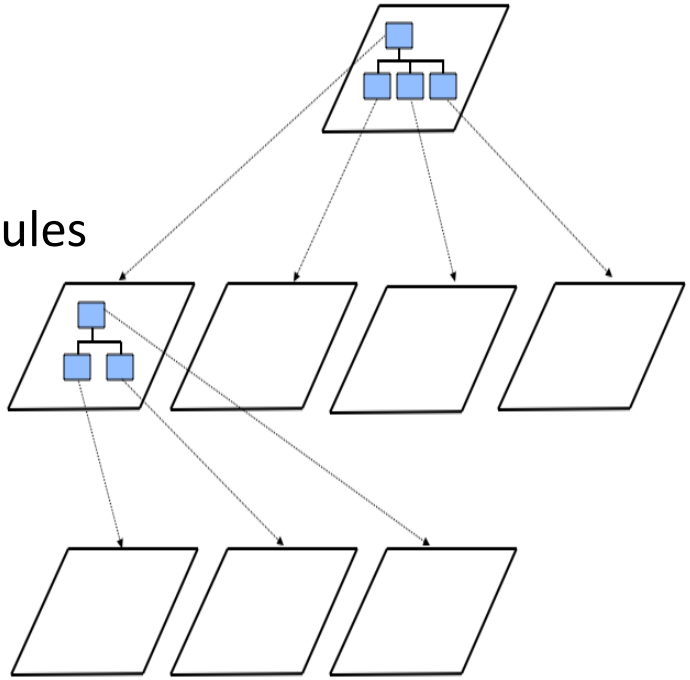
- In a well-structured piece of software, the dependencies show up as **procedure calls**
  - *call-graph*

- possible shapes of this graph:
  - chaos (directed graph)
  - hierarchy (acyclic graph)
  - strict hierarchy (layers)
  - tree

Hierarchical structure

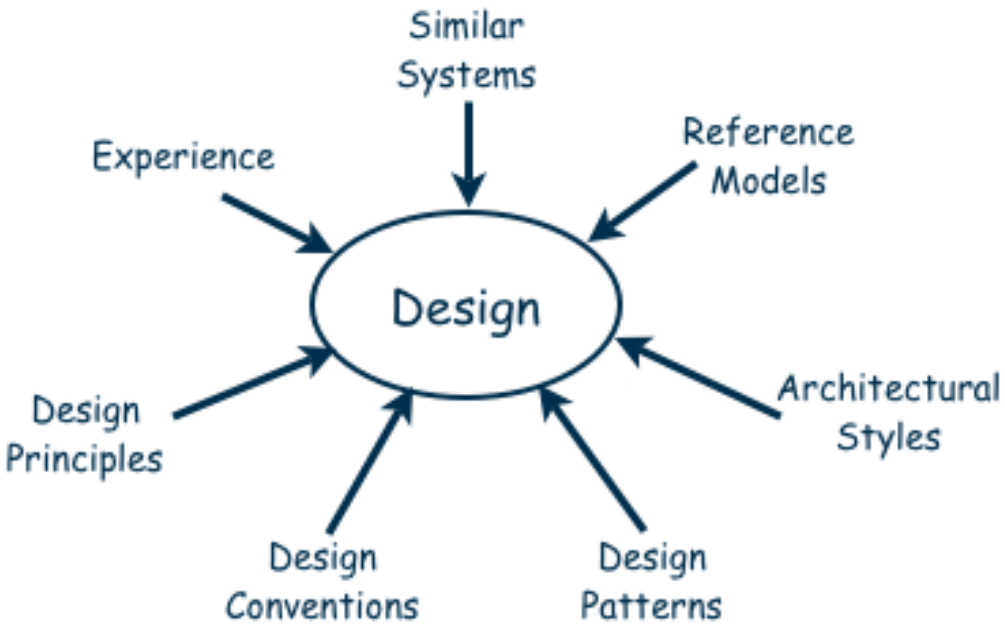
- Creating a hierarchy of information with increasing details
- Designers must decompose to isolate key problems

- Functional decomposition
  - assign functions or requirements to modules
- Feature-oriented decomposition
  - assigns features to modules
- Object-oriented design
  - assigns objects to modules



How to Design

- Examples of good design.
- Design principles: descriptive characteristics of good design.
- **Architectural styles/patterns: generic solutions.**
- Design patterns: generic solutions for making lower-level design decisions.



- Design convention or idiom: collection of design decisions and advice that, taken together, promotes certain design qualities.

Architectural Styles and Patterns

- There are many recognized architectural patterns and styles, such as
  - Blackboard
  - Client-server (2-tier, 3-tier, n-tier, cloud computing exhibit this style)
  - Component-based
  - Layered (or multilayered architecture)
  - Peer-to-peer (P2P)
  - Pipes and filters
  - SOA, Microservices architecture, …
  - MVC?
  - Even-driven
  - Asynchronous Messaging
  - ….

Architectural Styles and Patterns

- An architectural **Style** is a specialization of element and relation types, together with a set of constraints on how they can be used.

- An architectural **pattern** is a general, reusable solution to a commonly occurring problem in software architecture within a given context.
  - It expresses a fundamental structural organization schema for software systems.
  - It provides a set of predefined subsystems, specifies their responsibilities, and includes rules and guidelines for organizing the relationships between them.

Styles vs. Patterns

- **Pattern** is a context-problem-solution triple; **Style** is simply a condensation that focuses most heavily on the solution part.

- An essential part of an architecture **Pattern** is its focus on the problem and context as well as how to solve the problem in that context.
- An architecture **Style** focuses on the architecture approach, with more lightweight guidance on when a particular style may or may not be useful.

- There may be more than one pattern for each style.

Architectural Styles and Patterns

- No fixed set of views is appropriate for every system; broad guidelines can help us gain a footing.

- Architects need to think about their software in three ways simultaneously
  1. How it is structured as a set of implementation units;
  2. How it is structured as a set of elements that have runtime behavior and interactions;
  3. How it relates to non-software structures in its environment.

- Architectural styles/patterns provide general beneficial properties, supporting specific quality attribute tactics utilized.

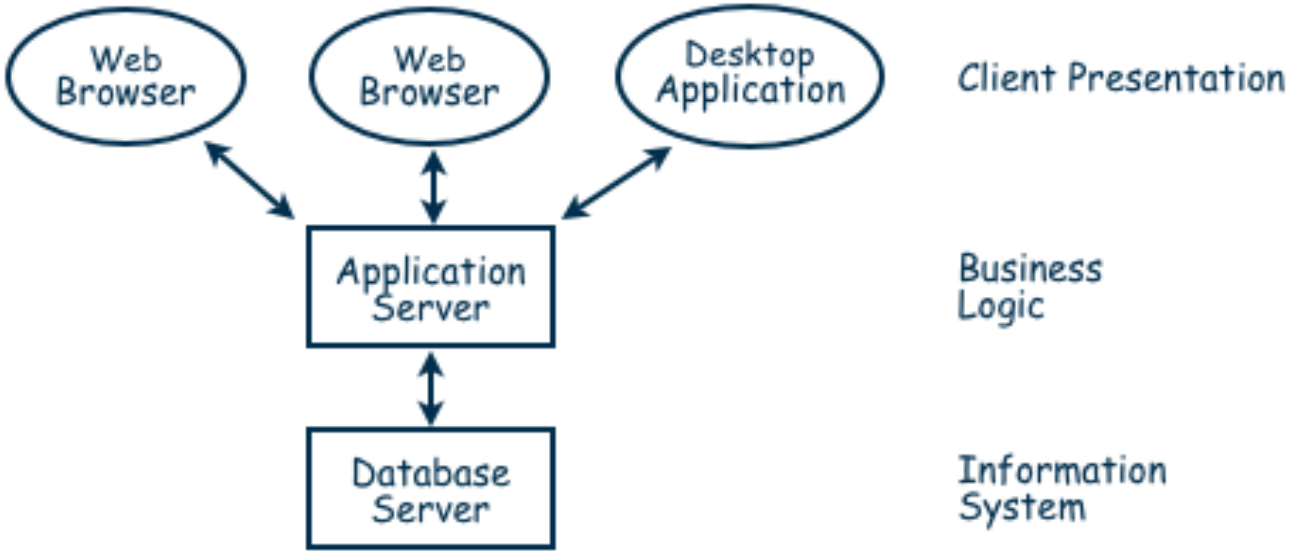
Architectural Styles

- Each style describes a system category that encompasses:

1. a set of components (e.g., a database, computational modules) that perform a function required by a system,
2. a set of connectors that enable “communication, coordination and cooperation” among components,
3. constraints that define how components can be integrated to form the system, and
4. semantic models that enable a designer to understand the overall properties of a system by analyzing the known properties of its constituent parts.

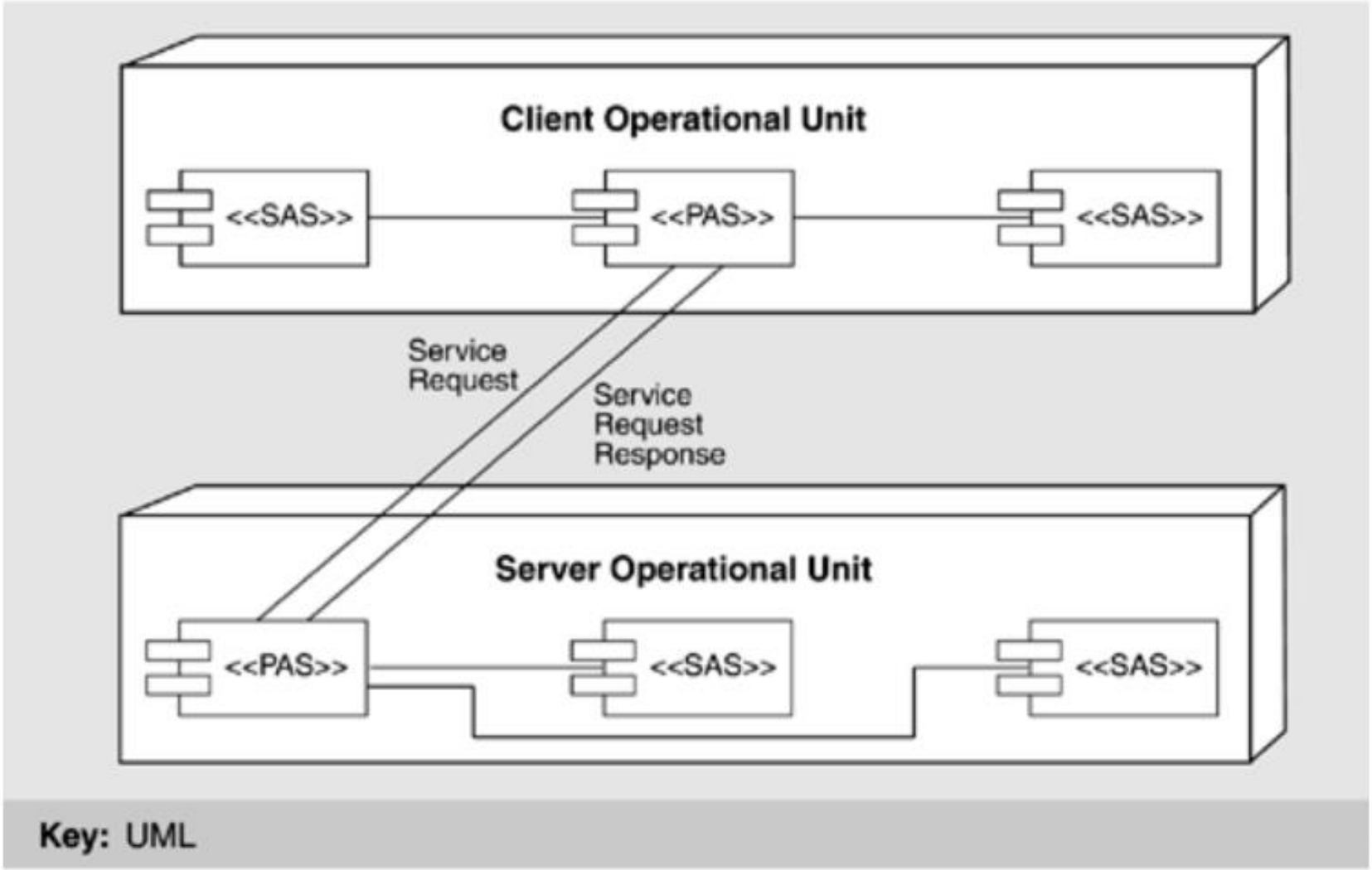
Client/Server

1. Server components offer services
2. Clients access them using a request/reply protocol



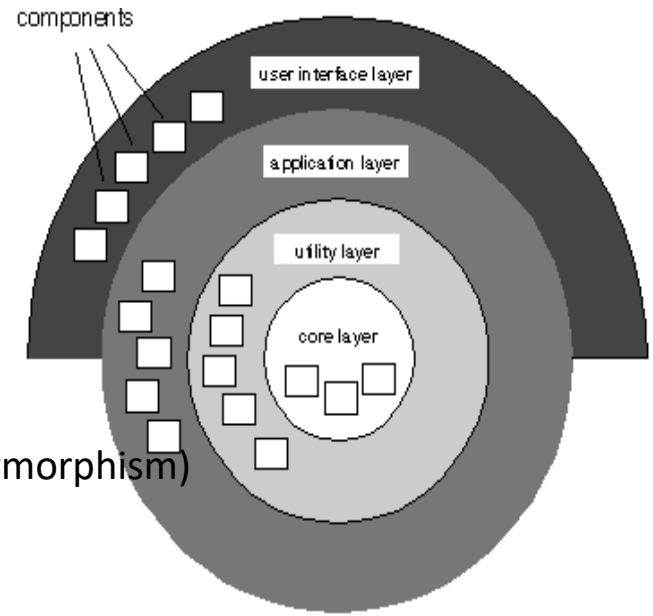
Client/Server

Air Traffic Control,
Clements et al, 2011



Layered

- Use layers to represent and separate elements of the software architecture
  - Easier to understand a complex system
  - Model-view-controller (MVC) architecture separates application logic from user interface



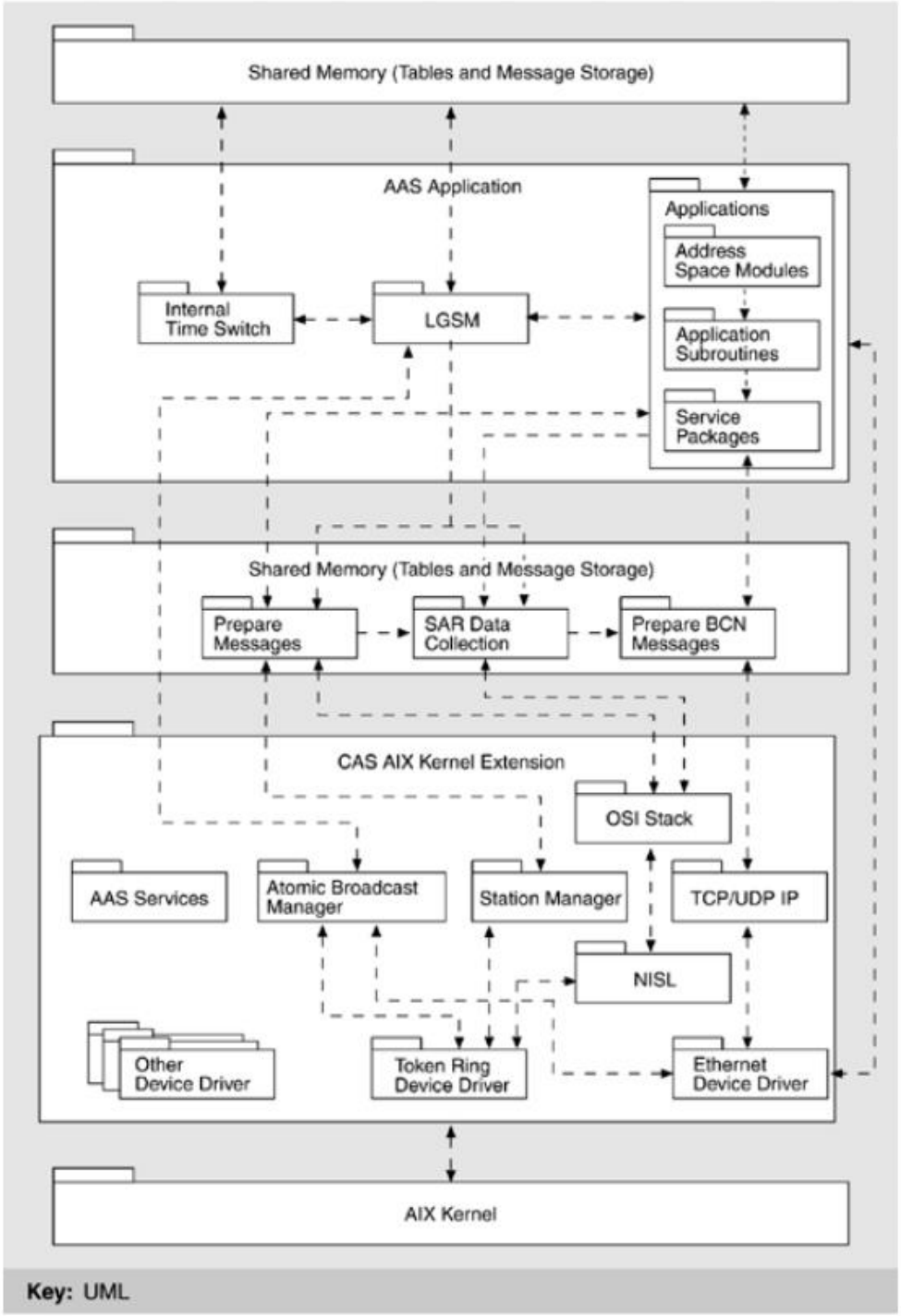
- Foundation (e.g., container classes)
- Problem domain (e.g., encapsulation, inheritance, polymorphism)
- Data management (e.g., data storage and retrieval)
- User interface (e.g., data input forms)
- Physical architecture (e.g., specific computers and networks)

Computing Layers

- **Presentation layer**—the user interface
- **Presentation logic layer**—processing that must be done to generate the presentation, such as editing input data or formatting output data.
- **Application logic layer**—the logic and processing to support business rules, policies, and procedures
- **Data manipulation layer**—to store and retrieve data to and from the database
- **Data layer**—the actual business data

Layered

Air Traffic Control, Clements et al, 2011



Service-Oriented Architecture (SOA)

- Service-Oriented Architecture (SOA) consist of a collection of distributed components that provide and/or consume services.
- Service provider components and service consumer components can use different implementation languages and platforms.
- Services are largely standalone.
- Computation is achieved by a set of cooperating components that provide and/or consume services over a network.

Service-Oriented Architecture (SOA)

- **Service providers** provide one or more services through published interfaces. Properties will vary with the implementation technology (such as EJB or ASP.NET) but may include performance, authorization constraints, availability, and cost.

- **Service consumers** invoke services directly or through an intermediary.

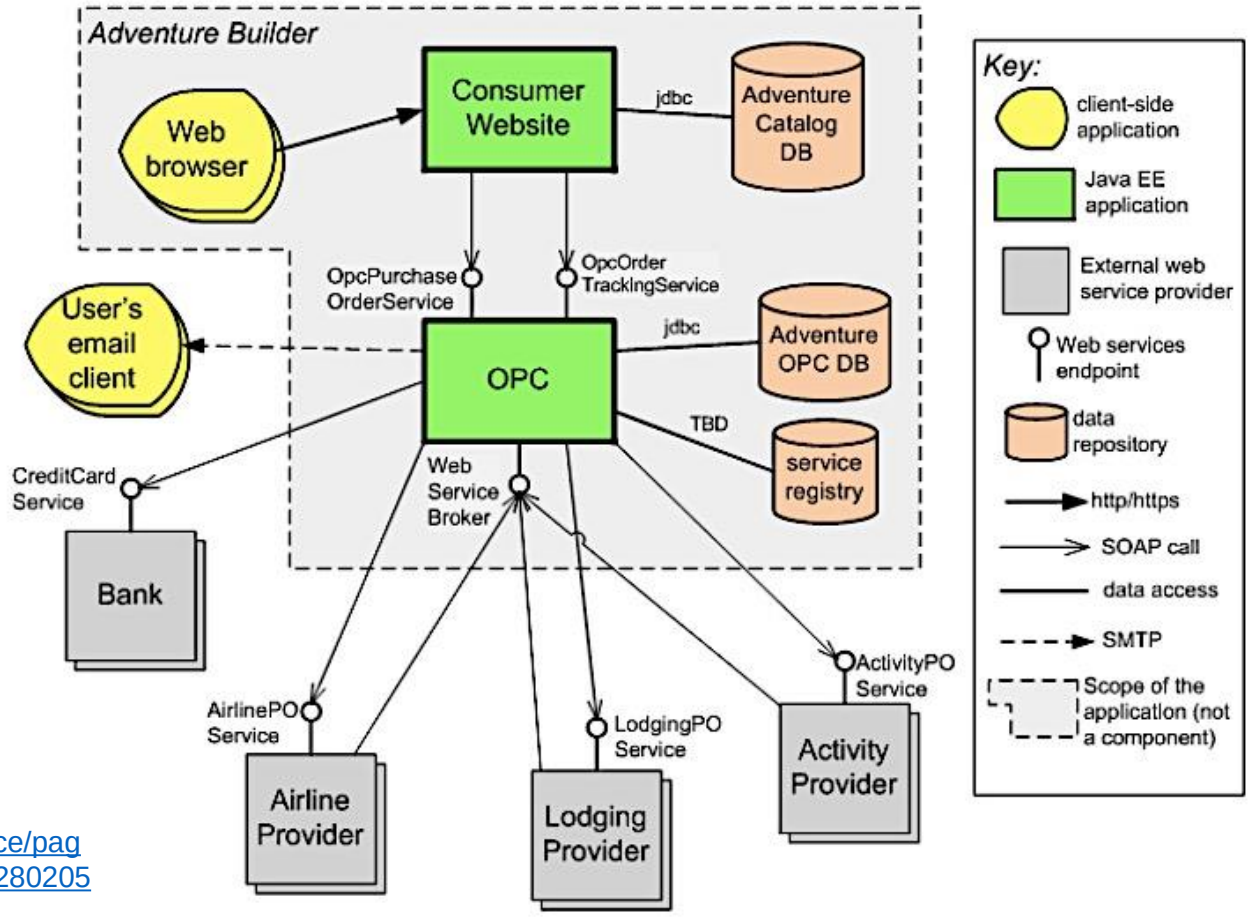
- **Simple Object Access Protocol (SOAP) connector** uses the SOAP protocol for synchronous communication between Web services, typically over HTTP. Ports of components that use SOAP are often described in WSDL.

- REpresentational State Transfer **(REST) connector** relies on the basic request/response operations of the HTTP protocol.

Service-Oriented Architecture (SOA)

Adventure Builder 2010, Clements et al, 2011

The Adventure Builder system interacts via SOAP Web services with several other external service providers (e.g., mainframe, Java, or .NET); the SOAP connector provides the necessary interoperability.



https://wiki.sei.cmu.edu/confluence/pages/viewpage.action?pageId=146280205

REST

- The REpresentational State Transfer (REST) is the software architectural style of the World Wide Web.

- REST is an architectural style for distributed hypermedia systems, describing the software engineering principles guiding REST and the interaction constraints chosen to retain those principles, while contrasting them to the constraints of other architectural styles.

REST- RESTful

- **RESTful** applications conform to the REST constraints.

- RESTful systems typically communicate over HTTP with the same Methods (GET, POST, PUT, DELETE etc) that browsers use to retrieve web pages and to send data to remote servers.

- REST systems interface with external systems as web resources identified by URI, for example, which can be operated upon using standard methods such as `GET /people/1` or `DELETE /people/1`.

Microservices

- A variant of the service-oriented architecture that structures an application as a collection of loosely coupled services
  - Used to build *flexible*, *independently* deployable software systems.

- **Microservice** = an atomic and self-sufficient piece of software.
  - a self-contained piece of business functionality with clear interfaces, and may, through its own internal components, implement a layered architecture.

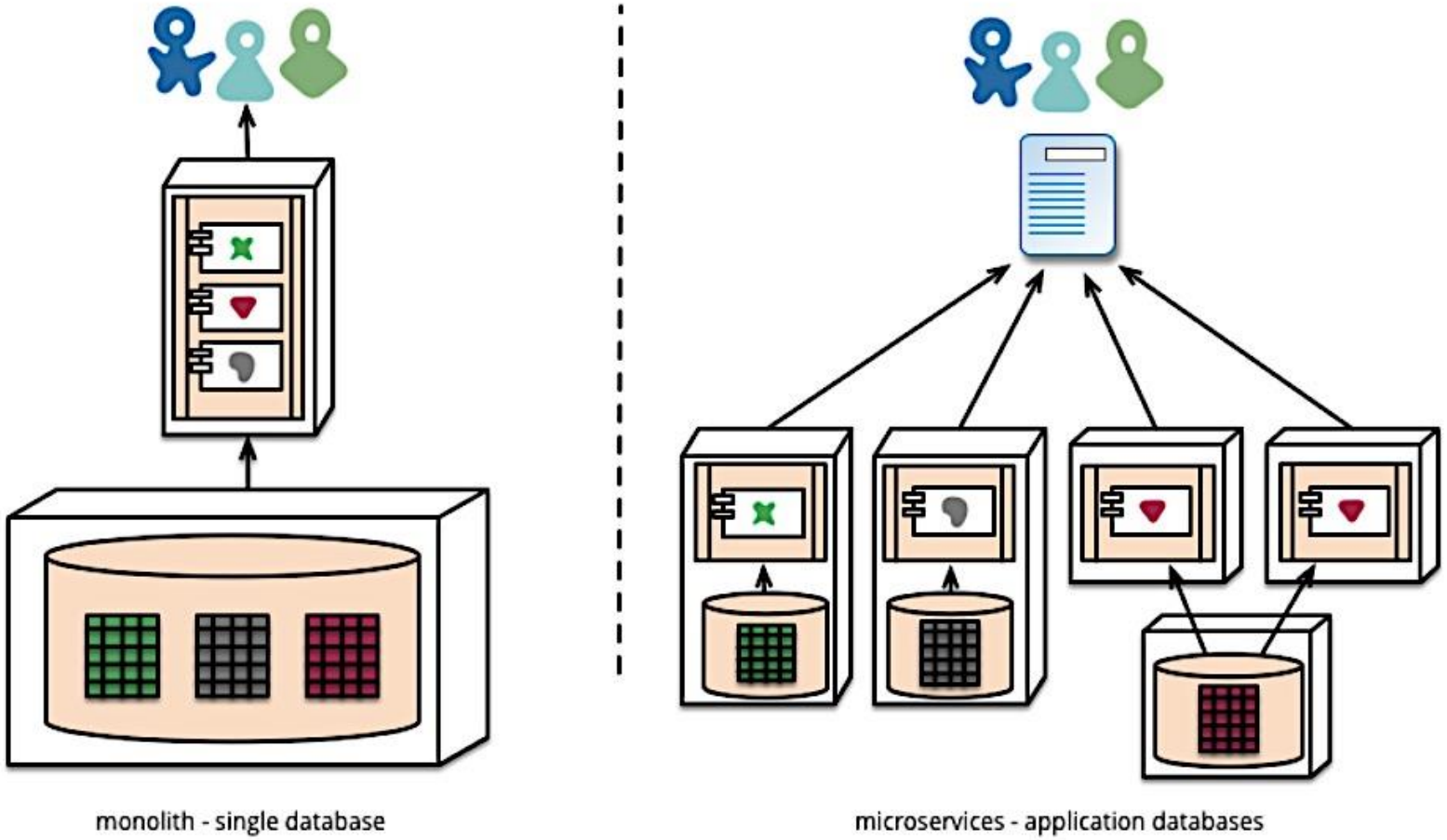
- **Service components** contain one or more modules (e.g., Java classes) that represent either a single purpose function (e.g., providing the weather for a specific city or town) or an independent portion of a large business application (e.g., stock trade placement or determining auto insurance rates).

Microservices

- Separately deployed units
  - Applications built using the microservices architecture pattern are generally more robust, provide better scalability, and can more easily support continuous delivery

- Distributed architecture
  - The microservices architecture pattern is a distributed architecture, meaning that all the components within the architecture are fully decoupled from one another and accessed through some sort of remote access protocol (e.g., JMS, AMQP, REST, SOAP, RMI, etc.).
  - The distributed nature of this architecture pattern is how it achieves some of its superior scalability and deployment characteristics.

Microservices



https://martinfowler.com/articles/microservices.html

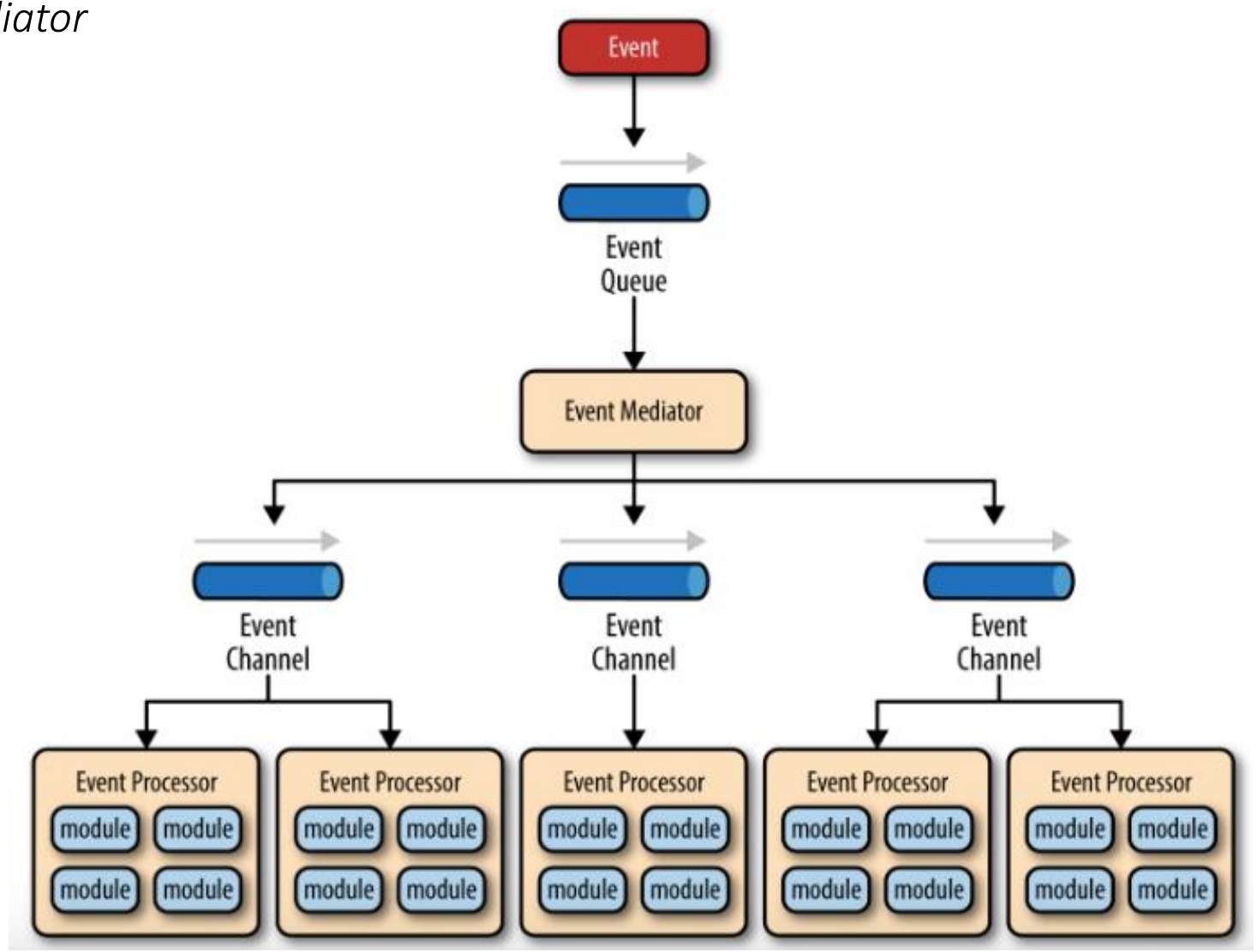
Event-Driven Driven

- Distributed asynchronous architecture pattern used to produce highly scalable applications.
- Made up of highly decoupled, single-purpose event processing components that asynchronously receive and process events.

- Two main topologies
  1. Mediator; commonly used when you need to orchestrate multiple steps within an event through a central mediator
  2. Broker; used when you want to chain events together without the use of a central mediator.

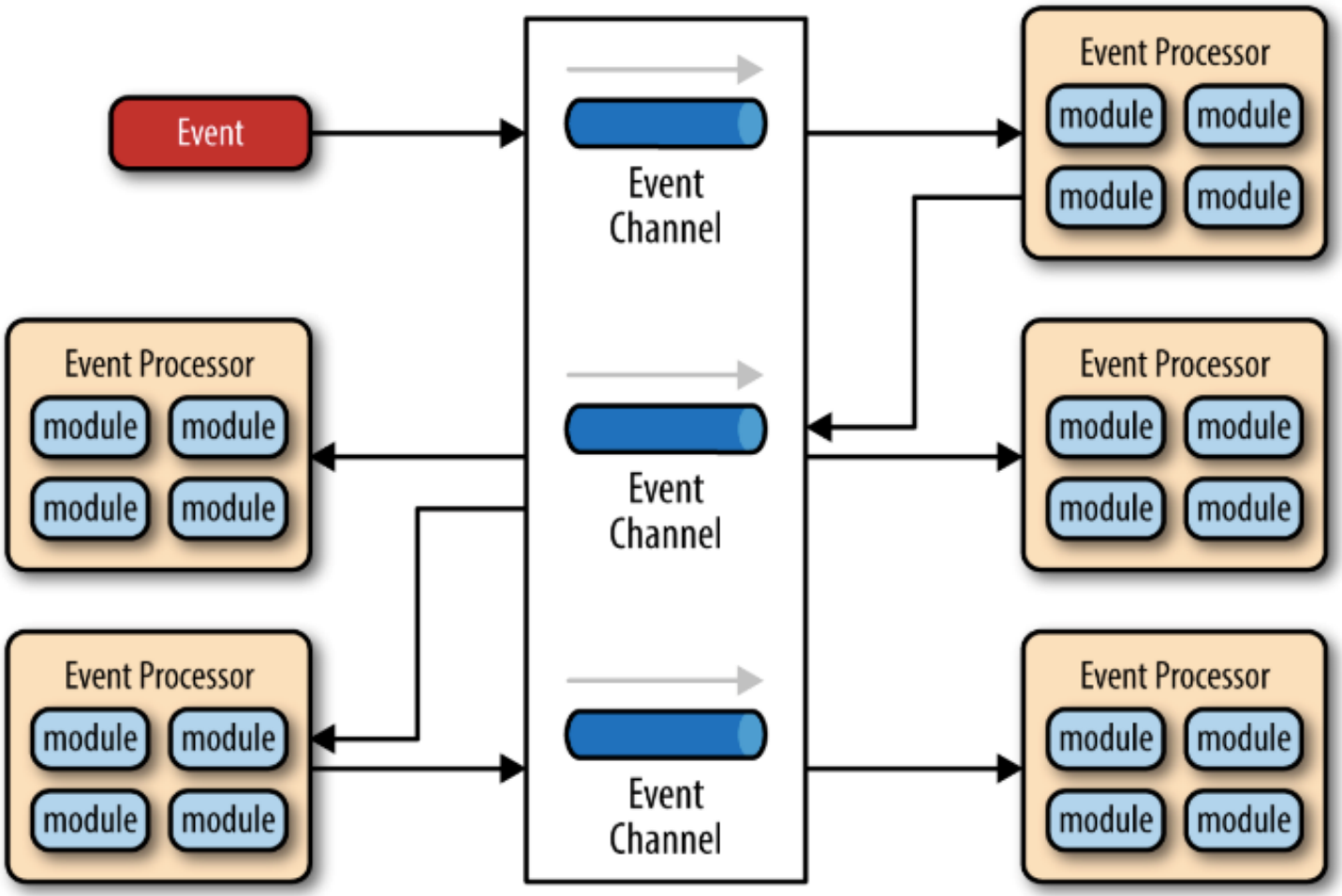
Event-Driven Architecture

Mediator



Event-Driven Architecture

Broker



Asynchronous Messaging

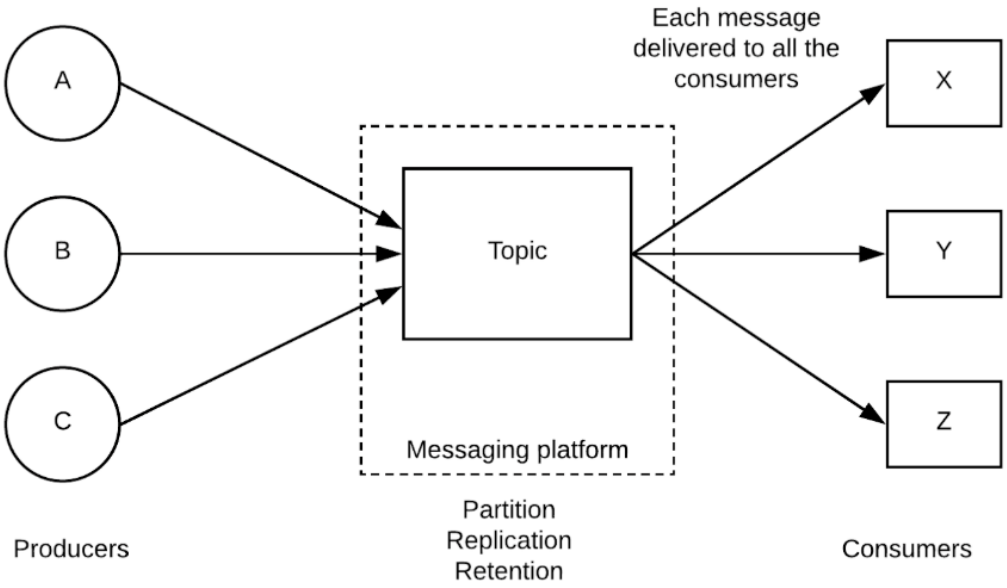
- High-rate data flow impossible to respond in a synchronous manner.
- Not expect an immediate response or even a response at all.

  - Guaranteed delivery
  - Extensive processing
  - Correlation and time series analysis
  - Decoupling of source and target systems

- Topic based publish-subscribe
- Queue based publish-subscribe
- Event based real-time processing
- Batch processing
- Store and Forward

Asynchronous Messaging

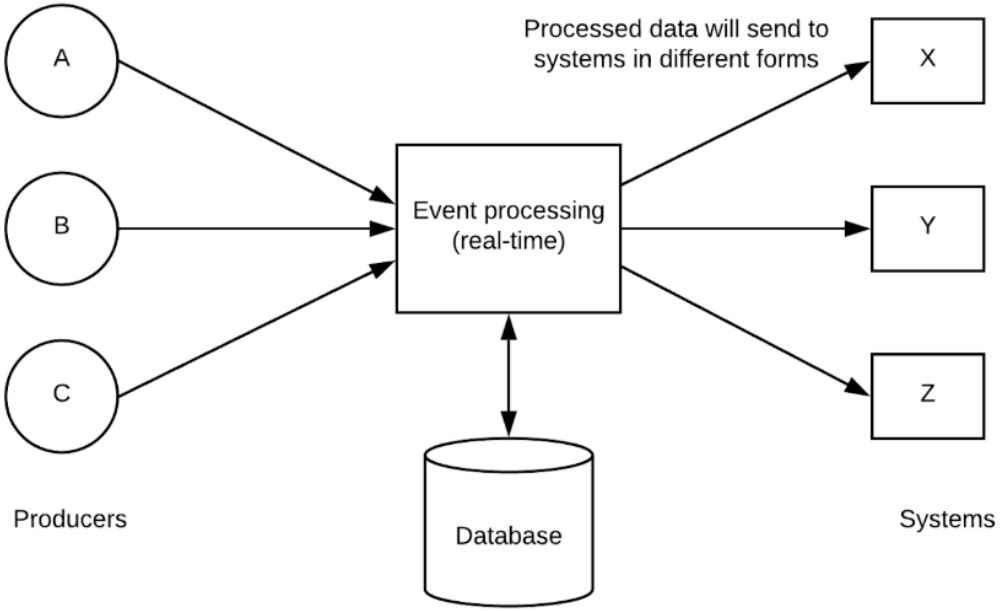
Topic-based publish-subscribe



- e.g., Apache Kafka

Asynchronous Messaging

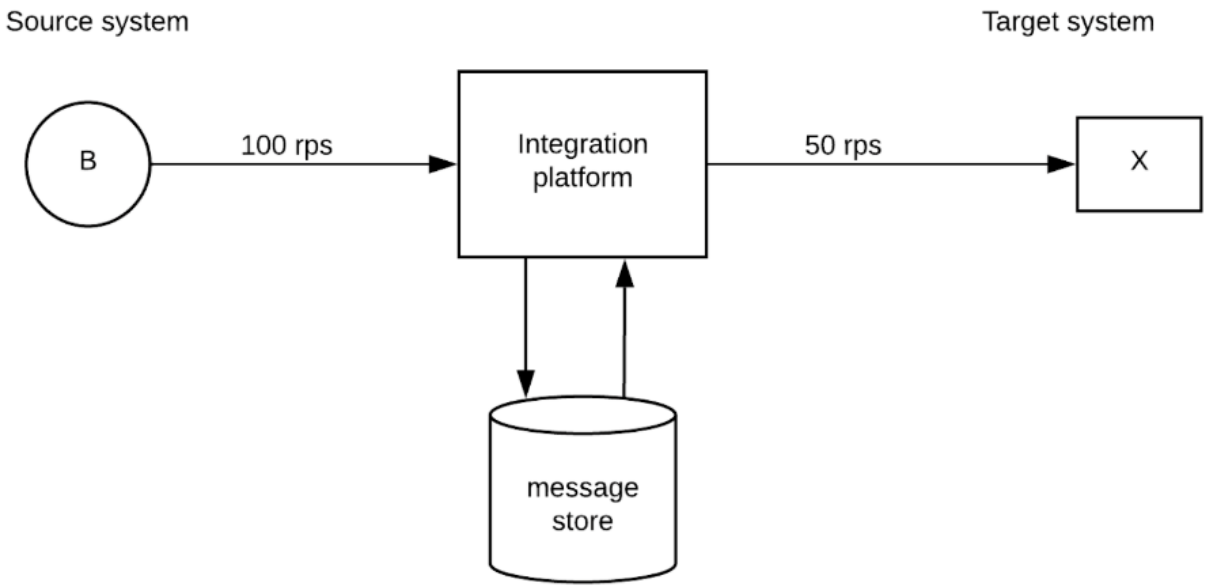
Event-based real-time processing



- e.g., WSO2 Stream Processor, Kafka streams, Apache Flink, Apache storm
- supports integration with AI as well as ML

Asynchronous Messaging

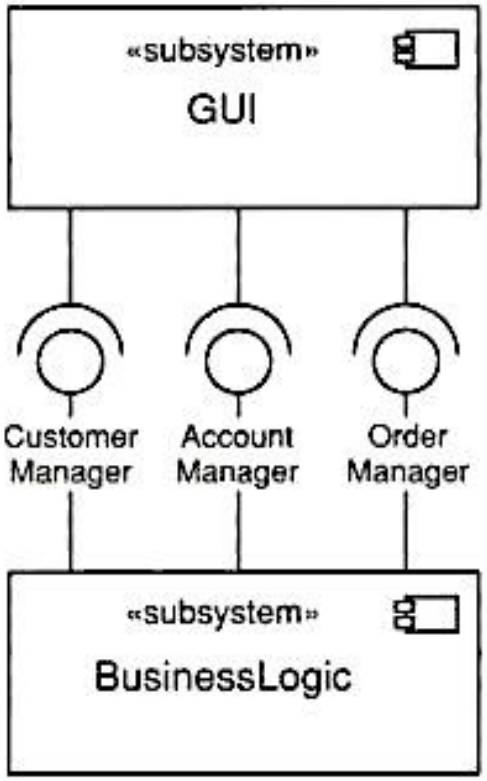
Store and Forward



- messages needs to be processed as soon as possible.
- e.g., WSO2 Enterprise Integrator

Component-Based

- Component-Based Development (CBD) is about constructing software from plug-in parts
  - use interfaces to make components "**pluggable**"
- A reuse-based approach to defining, implementing and composing loosely coupled independent components into systems.



- An individual software component is a *software package*, a *web service*, a *web resource*, or a *module* that encapsulates a set of related functions (or data).

- by designing to an interface, you allow the possibility of many different realizations by many different components.

Model-View-Controller (MVC)

- What is MVC?
- Style or pattern?
- Architectural or design level?
- Active?

Exercise ☺

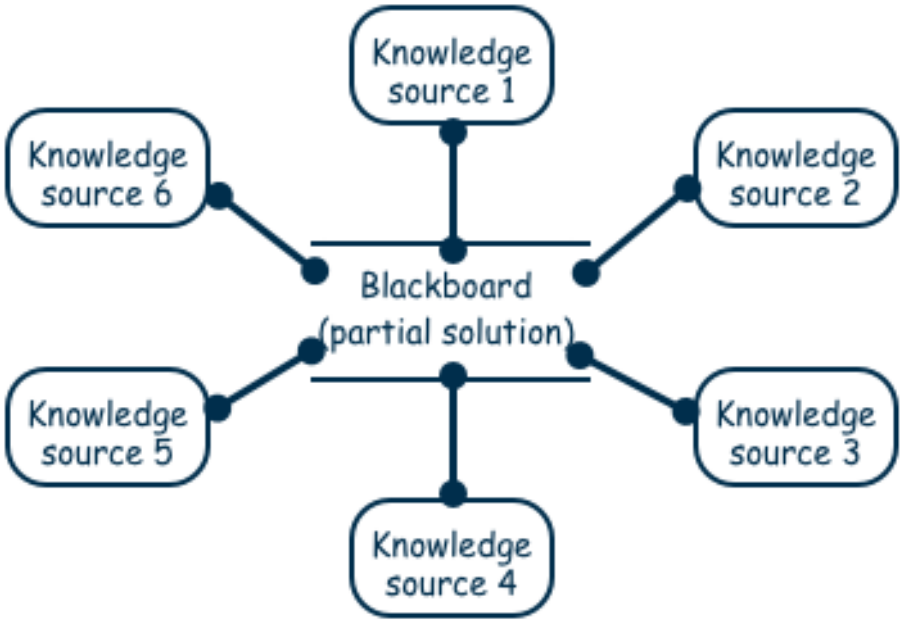
- Read More …
  - Erich Gamma, Richard Helm, Ralph Johnson, and John Vlissides, *Design Patterns: Elements of Reusable Object-Oriented Software*, Addison-Wesley, 1994
  - Martin Fowler, *GUI Architectures*

  https://martinfowler.com/eaaDev/uiArchs.html
  - Alex Moldovan, *Is Model-View-Controller dead on the front end?*

  https://www.freecodecamp.org/news/is-mvc-dead-for-the-frontend-35b4d1fe39ec/

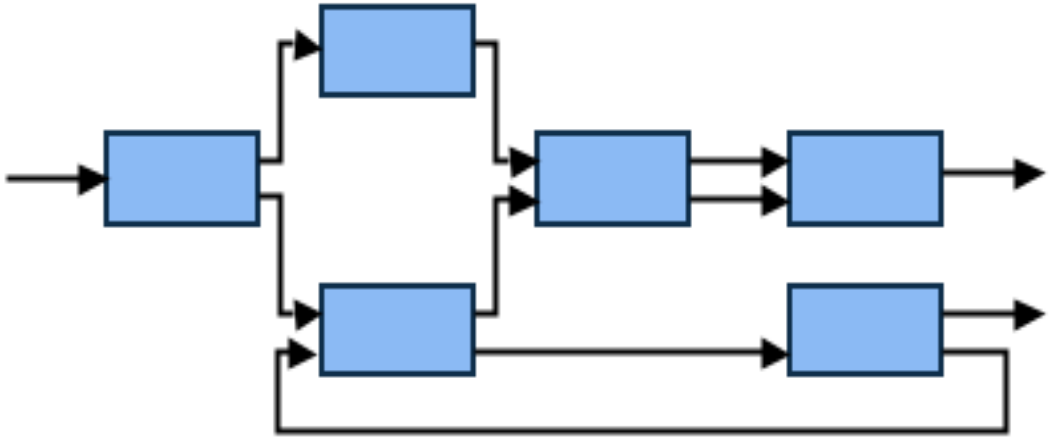
Blackboard

1. A central data store
2. A collection of components that operate on it to store, retrieve, and update information



Pipes and Filters

- The system has
  - Streams of data (pipe) for input and output
  - Transformation of the data (filter)
  - The designer can understand the entire system's effect on input and output as the composition of the filters

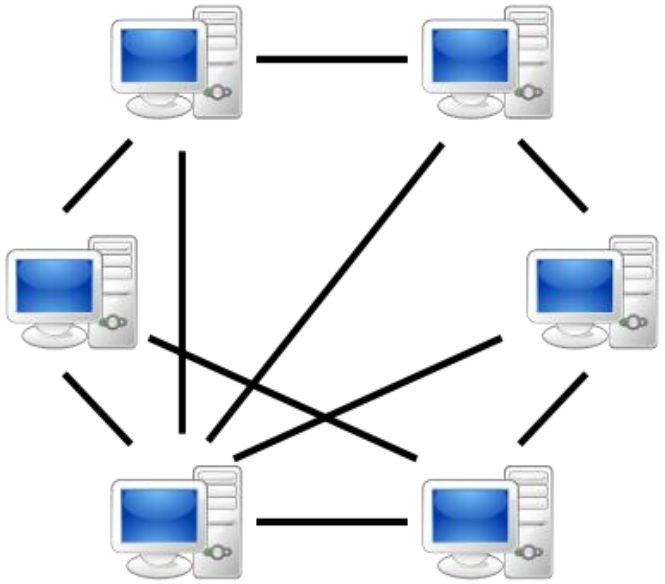


Peer-to-peer (P2P)

- Each component acts as its own process and acts as both a client and a server to other peer components.

- Any component can initiate a request to any other peer component.

- Characteristics
  - Scale up well
  - Increased system capabilities
  - Highly tolerant of failures



Architectural Styles

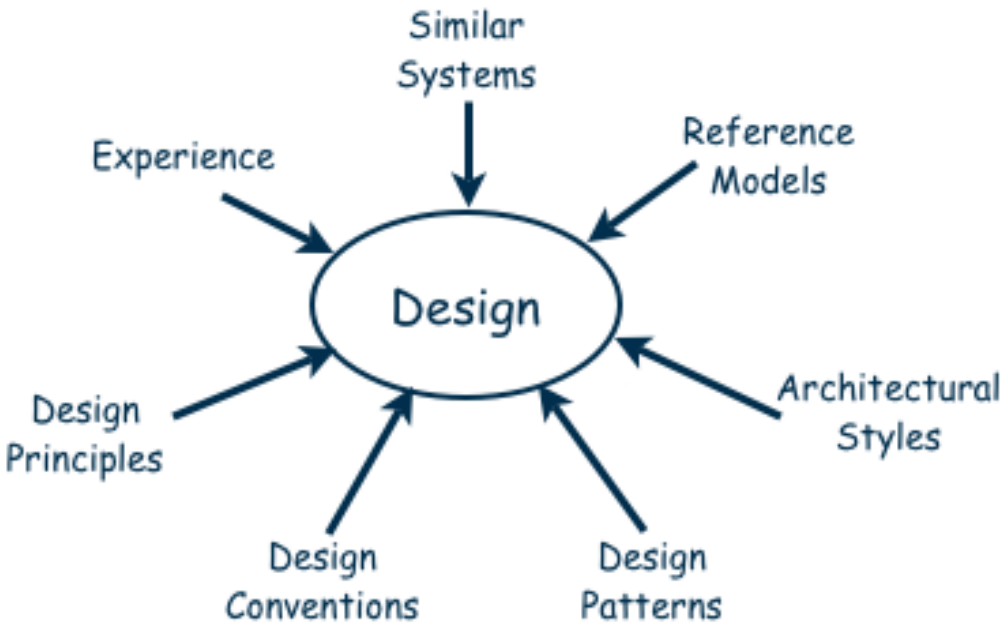
- Actual software architectures rarely based on purely one style.

- Good design is about selecting, adapting, and integrating **several** architectural design styles to produce the desired result.

- Architectural styles can be combined in several ways
  - Use different styles at different layers (e.g., overall client-server architecture with server component decomposed into layers)
  - Use mixture of styles to model different components or types of interaction (e.g., client components interact with one another using publish-subscribe communications

How to Design

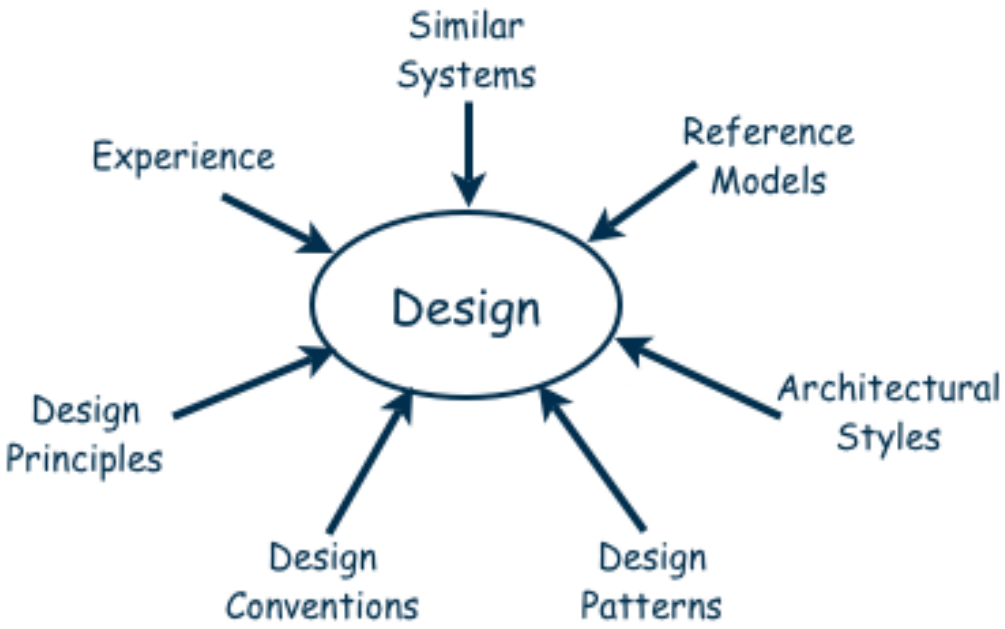
- Examples of good design.
- Design principles: descriptive characteristics of good design.
- Architectural styles/patterns: generic solutions.
- **Design patterns: generic solutions for making lower-level design decisions.**



- Design convention or idiom: collection of design decisions and advice that, taken together, promotes certain design qualities.

How to Design

- Examples of good design.
- Design principles: descriptive characteristics of good design.
- Architectural styles/patterns: generic solutions.
- Design patterns: generic solutions for making lower-level design decisions.



- **Design convention or idiom: collection of design decisions and advice that, taken together, promotes certain design qualities.**

For Further Reading!

- Len Bass, Paul Clements, Rick Kasman, *Software Architecture in Practice*, 3rd Edition, Addison-Wesley Professional, 2012.
- Clements, Bachmann, Bass , Garlan, Ivers, Little, Merson, Nord, and Stafford, *Documenting Software Architectures: Views and Beyond*, Addison-Wesley, Boston, MA, 2011.
- Robert Hanmer, *Pattern-Oriented Software Architecture FOR DUMMIES*, Wiley, 2013.
- Martin Fowler, *Microservices: A definition of this new architectural term*. 2014. Available in: http://martinfowler.com/articles/microservices.html.
- Mark Richards, *Software Architecture Patterns*, O'Reilly Media Inc, 2015. https://www.oreilly.com/library/view/software-architecture-patterns/9781491971437/
- Software Architecture in Practice: http://etutorials.org/Programming/Software+architecture+in+practice,+second+edition/