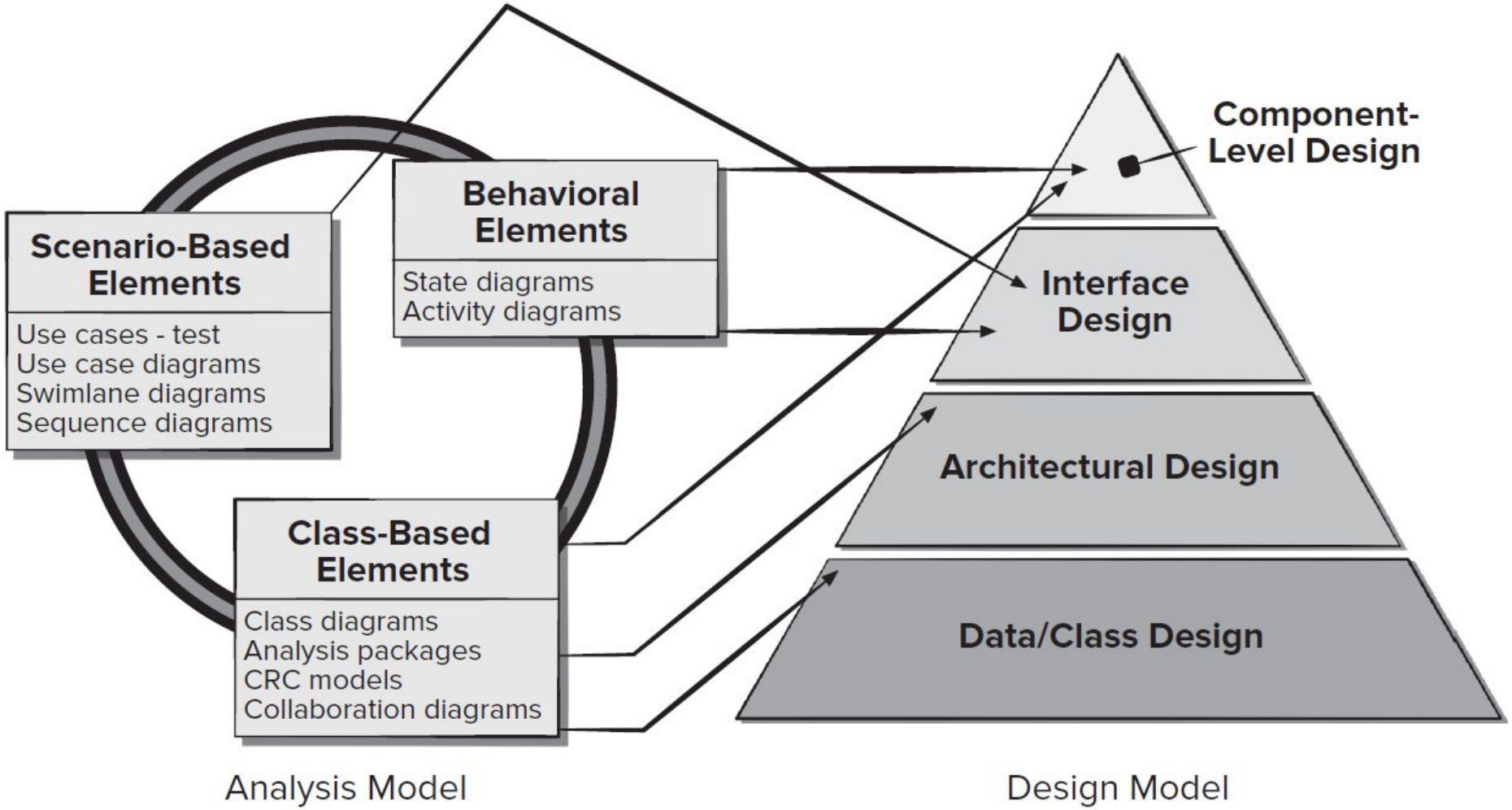
Architecture Design

Instructor: Mehroze Khan

The Design Process

- **Design** is the creative process of figuring out how to implement all the customer's requirements; the resulting plan is also called the **design**.
- Early design decisions address the system's architecture, explaining how to:
  - Decompose the system into units
  - How the units relate to one another
  - Describe any externally visible properties of the units
- Later design decisions address how to implement the individual units.

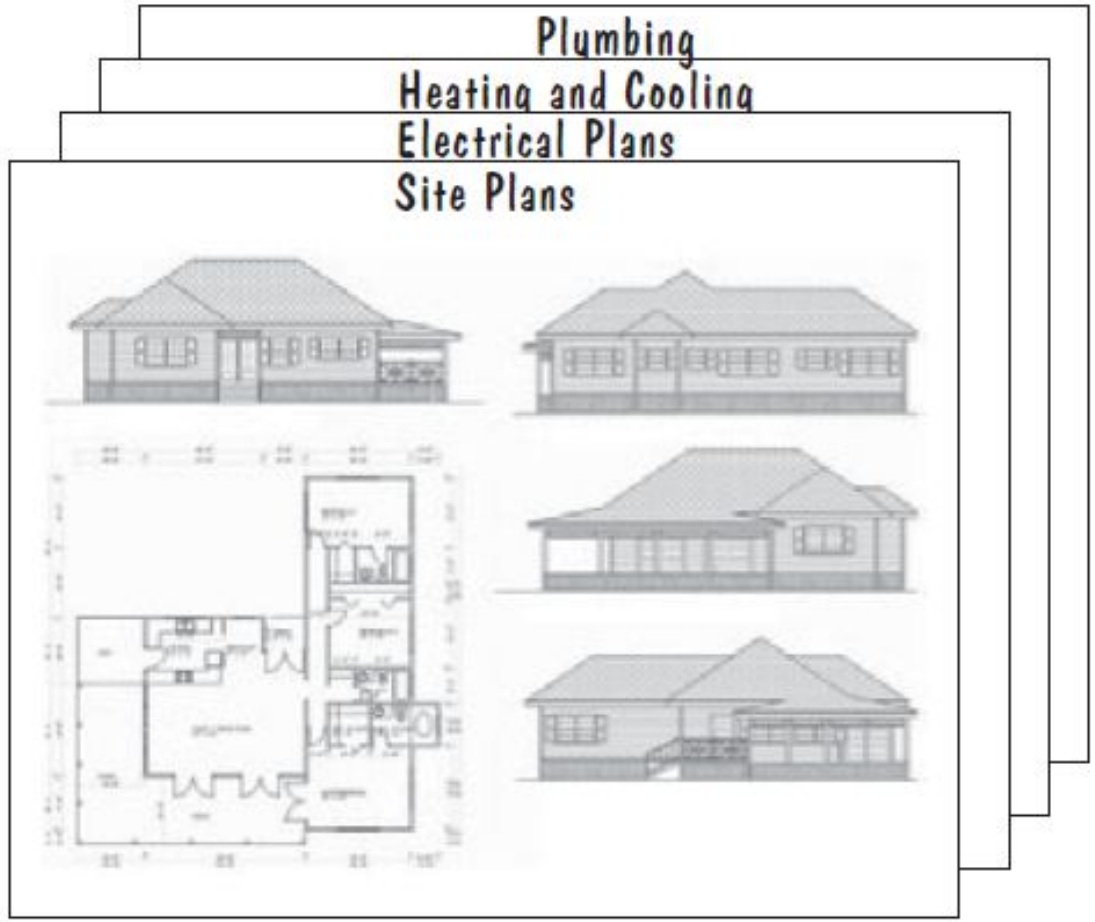
Translating Requirements to Design



Requirements and Design

- To see how architecture relates to both design and requirements, consider the **example** to build a new house.
- The requirements include:
  - Rooms for parents and three children
  - A place for the children to play
  - A kitchen and a large dining room that will hold an extendable table
  - Storage for bicycles, lawn mower, ladder, barbecue.
  - Heating and air conditioning

Architectural Plans



Multiple Designs

- Maximize playing area
- Minimize playing area
- Large bedrooms
- Two story house
- Single story house

Which is the best design?

Will a proposed solution result in modified requirements?

The Design Process

Design is a Creative Process

- Design is an intellectually challenging task
  - Numerous possibilities the system must accommodate
  - Non-functional design goals (e.g., ease of use, ease to maintain)
  - External factors (e.g., standard data formats, government regulations)
- We can improve our design by studying examples of good design
- Most design work is **routine design**, solve problem by reusing and adapting solutions from similar problems

The Design Process

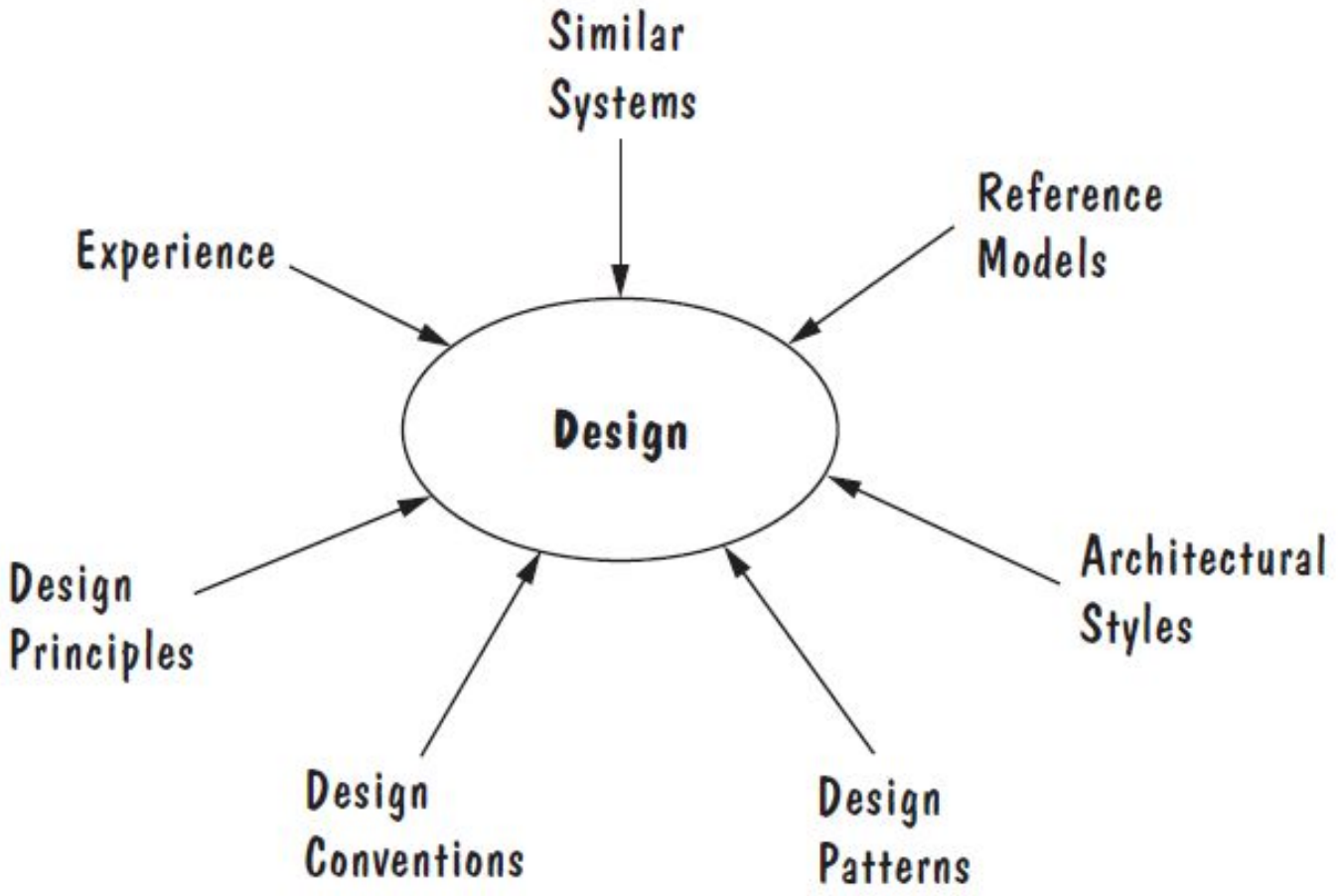
Design is a Creative Process

Using already available designs and solutions provides:

- ***Efficiency*** *(*in quickly settling on a plan)
- ***Predictability*** *(*in knowing that the resulting product should be similar in quality to existing one*)*

The Design Process

Design is a Creative Process



Sources of Design Advise

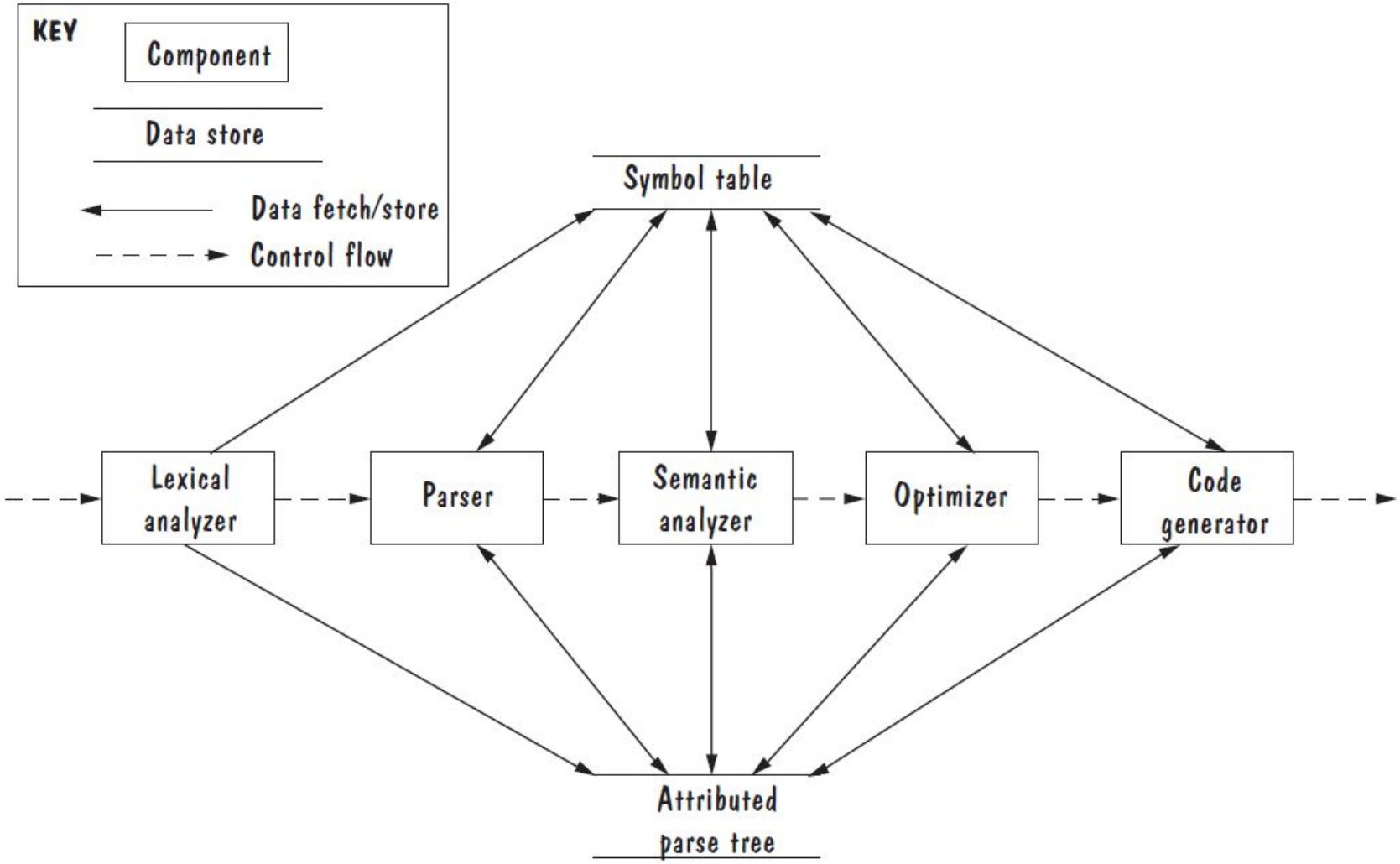
The Design Process

Design is a Creative Process

- Ways to Leverage Existing Solutions:

- **Cloning**: Borrow a whole design/code, make minor adjustments to fit the problem.
- **Reference Model**: A standard generic architecture that suggests how to decompose a system into its major components and how those components interact with each other.
- Software architectures have generic solutions too, called **architectural styles**.
  - Focusing on one architectural style can create problems
  - Good software architectural design is about selecting, adapting, and integrating several architectural styles in ways that best produce the desired result.

Reference Model of a Compiler



The Design Process

Design is a Creative Process

- Ways to Leverage Existing Solutions(Continued):

- **Design patterns** are generic solutions for making lower-level design decisions about individual software modules or small collections of modules.
- A **design convention** or **idiom** is a collection of design decisions and advice that, taken together, promotes certain design qualities.
- The guidance we can use in innovative design is a set of basic **design principles** that are descriptive characteristics of good design, rather than prescriptive advice about how to design.

Design Process Model

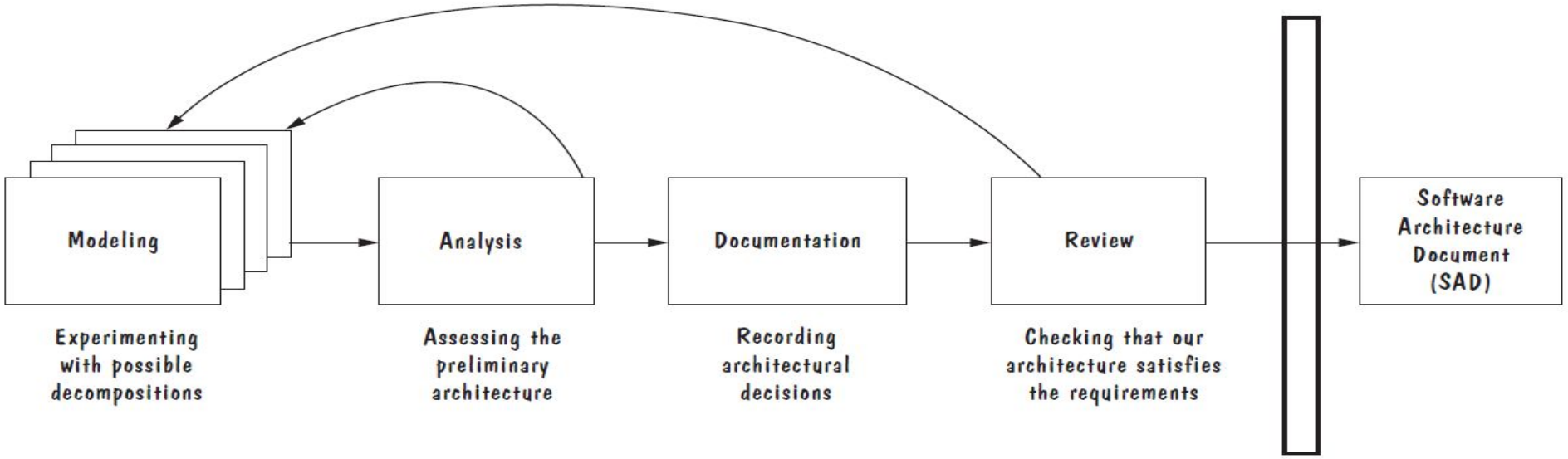
Design is an Iterative Process

- During the first part of the design phase, we iterate among three activities:
  - Drawing architectural plans
  - Analyzing how well the proposed architecture promotes desired properties
  - Using the analysis results to improve and optimize the architectural plans
- At the architectural stage, we focus on system-level decisions
  - Communication
  - Coordination
  - Synchronization
  - Sharing
- As the architecture starts to stabilize
  - Document models.
  - Each of the models is an architectural view, and the views are interconnected
  - A change to one view may have an impact on other views.

Design Process Model

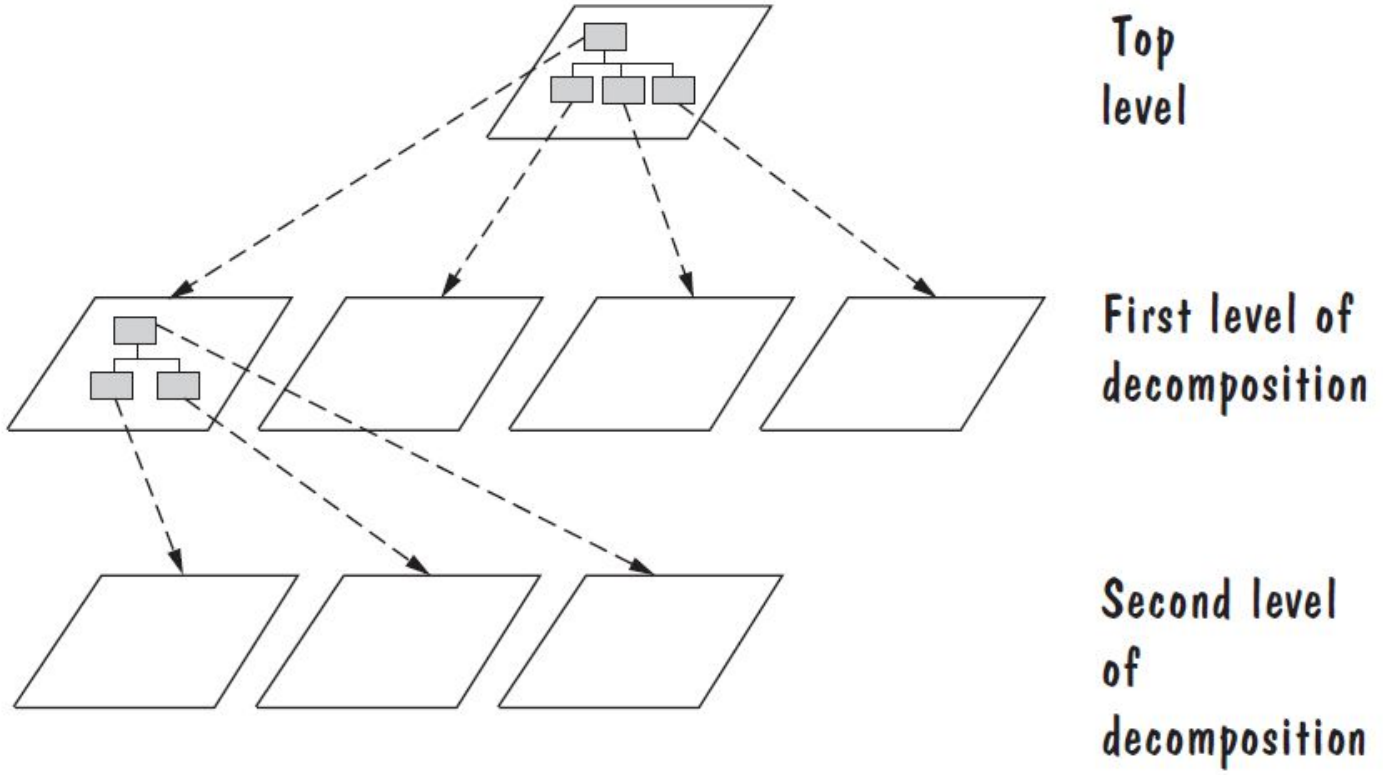
Design is an Iterative Process

- Once the architecture is documented
  - Conduct a formal design review, in which the project team checks that the architecture meets all the system's requirements and is of high quality.
  - If problems are identified during the design review, we may have to revise our design yet again to address these concerns.
- The outcome of the software architecture process is the **SAD**, used to communicate system-level design decisions to the rest of the development team.



Decomposition and Views

- Design by decomposition starts with a high-level description of the **system's key elements**.
- Iteratively refine the design by dividing system's elements into its constituent pieces and describing their interfaces.



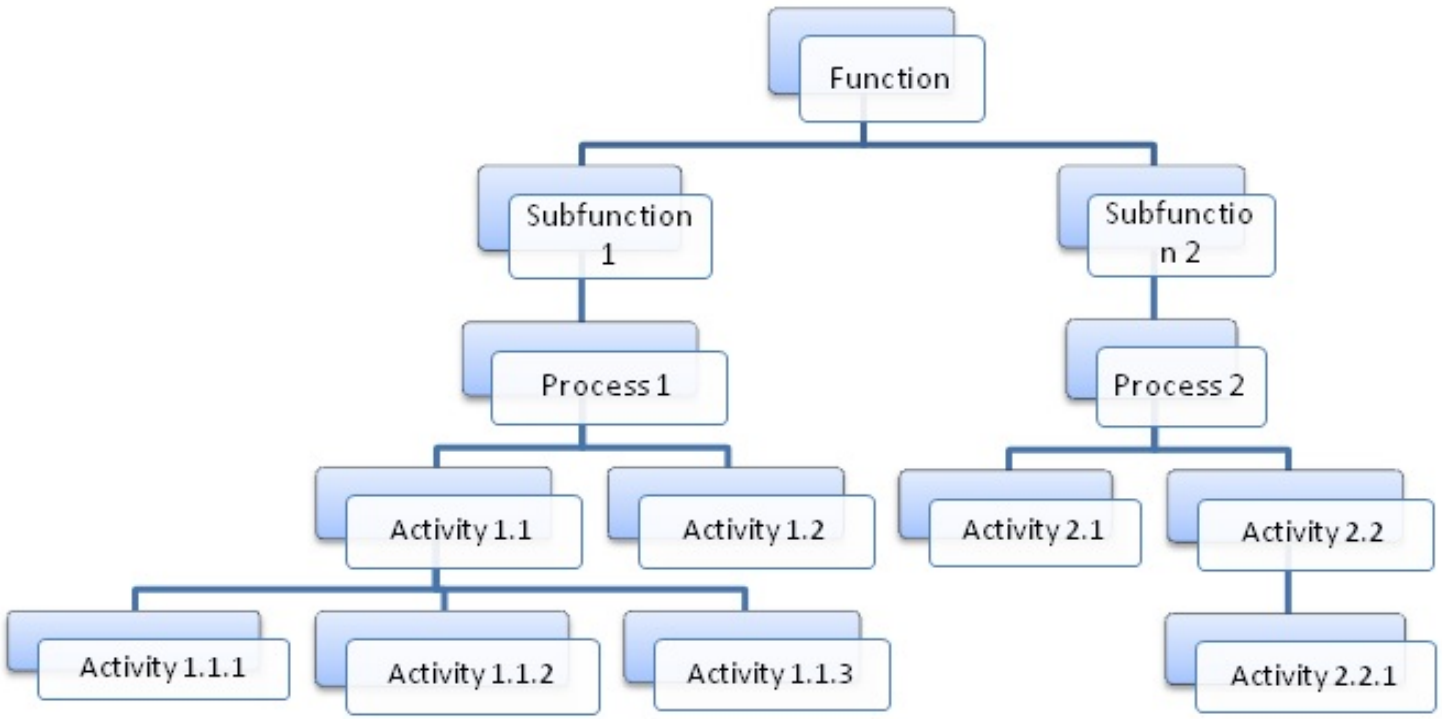
Design Methods

- Some popular design methods:
  - Functional decomposition
  - Feature-oriented decomposition
  - Data-oriented decomposition
  - Process-oriented decomposition
  - Event-oriented decomposition
  - Object-oriented design

Design Methods

- **Functional decomposition:**
  - Partitions functions or requirements into modules.
  - Lower-level designs divide these functions into subfunctions, which are then assigned to smaller modules.
  - The design also describes which modules (subfunctions) call each other.

Functional Decomposition



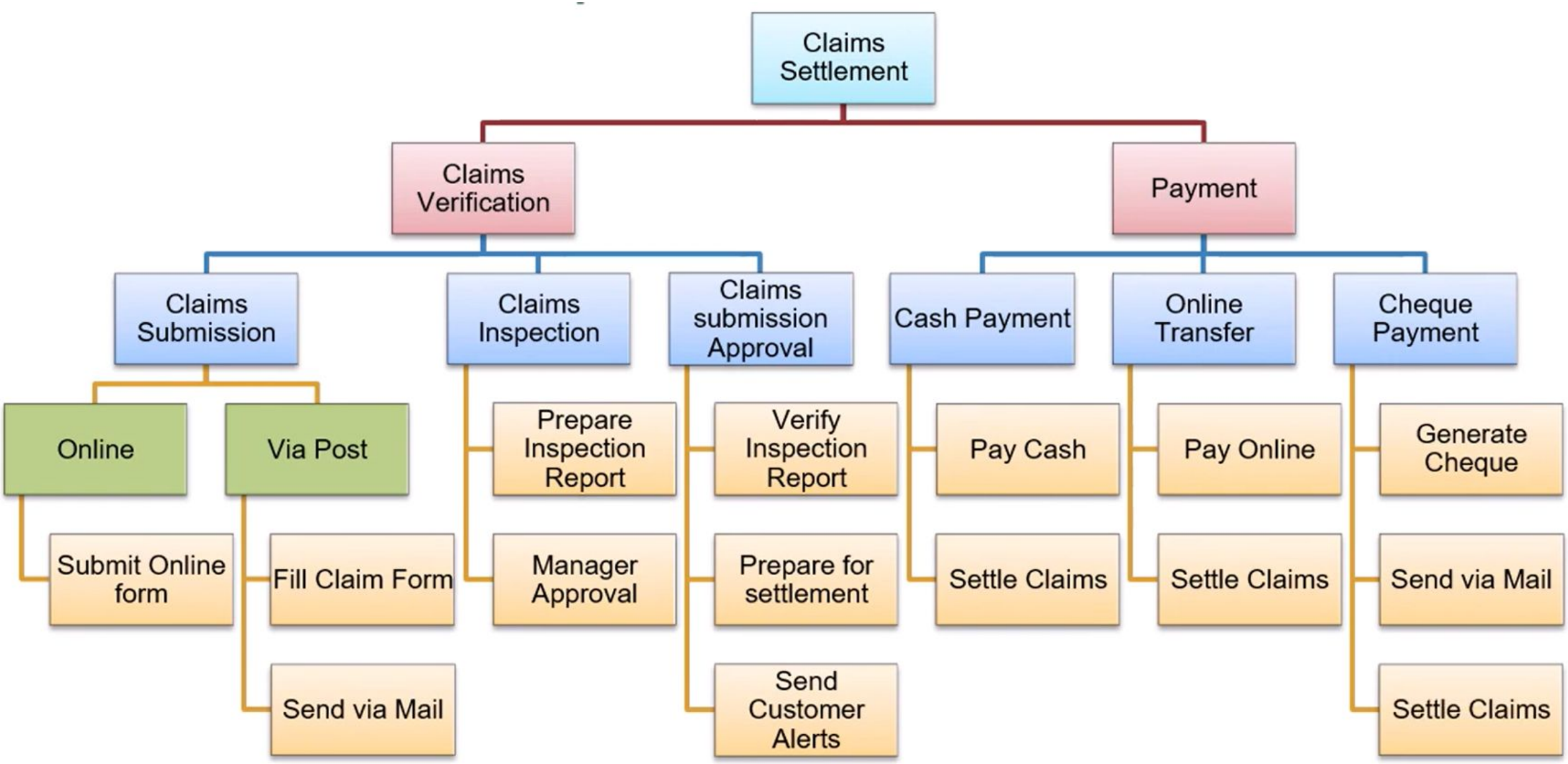
Functional Decomposition Example

Roy is a Business Analyst working for WQA Cargos International. The company specializes in providing freight services to ship the goods across the world. Recently there has been a lot of complaints from the customers for the delay in the post shipment insurance settlements against the claims raised. The organization has asked Roy to understand the full process of insurance settlements in the company so that later the issue of delayed settlements can be addressed.

- Roy decides to use the Functional decomposition technique to understand the full process.
- What to decompose: *Business Process*

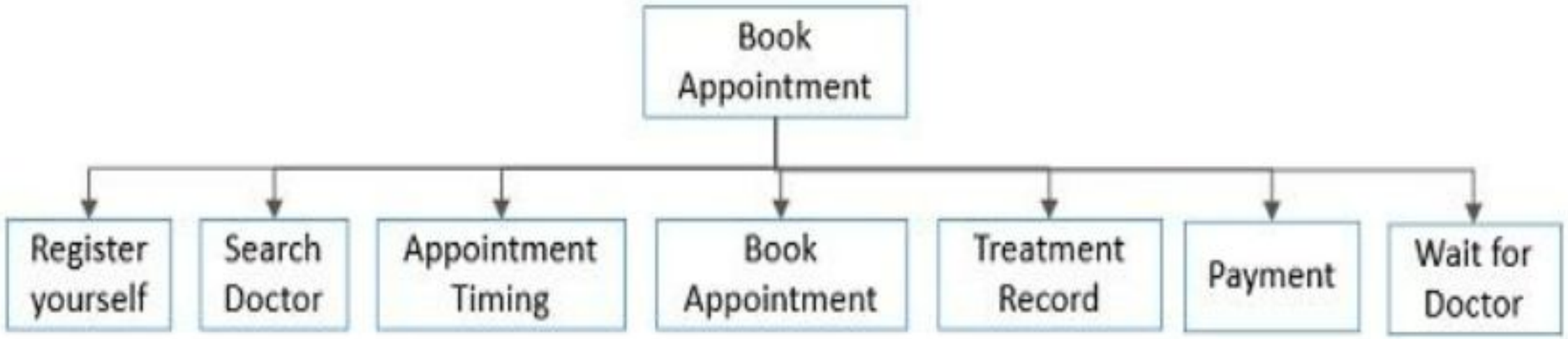
https://www.youtube.com/watch?v=AovfJJRZ1EA&ab_channel=Techcanvass

Functional Decomposition Example

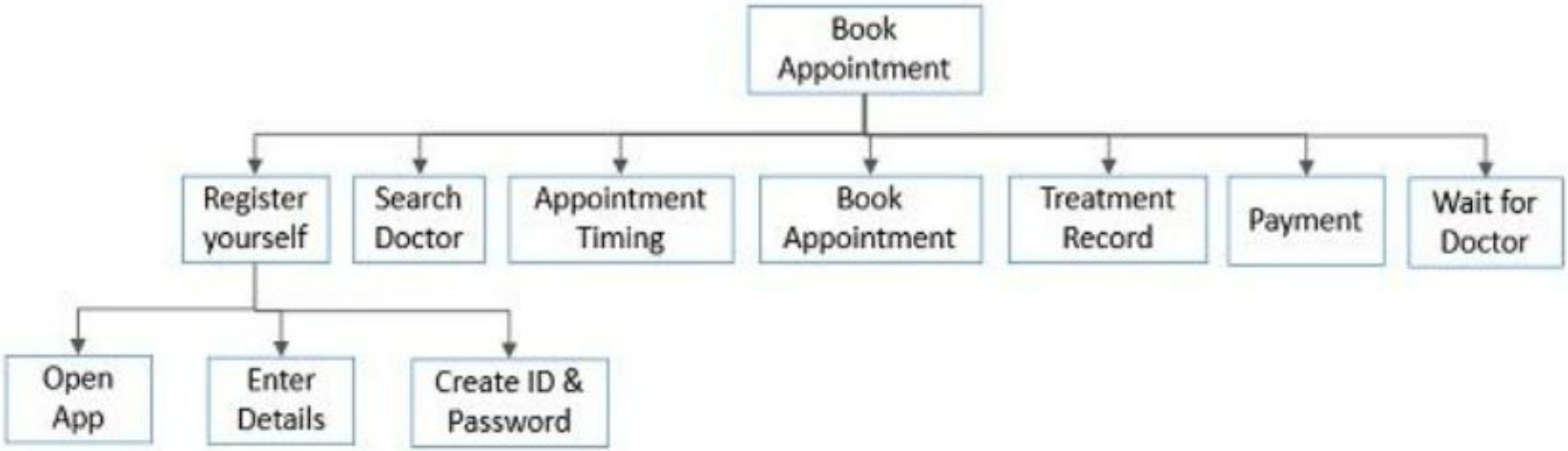


Functional Decomposition Example

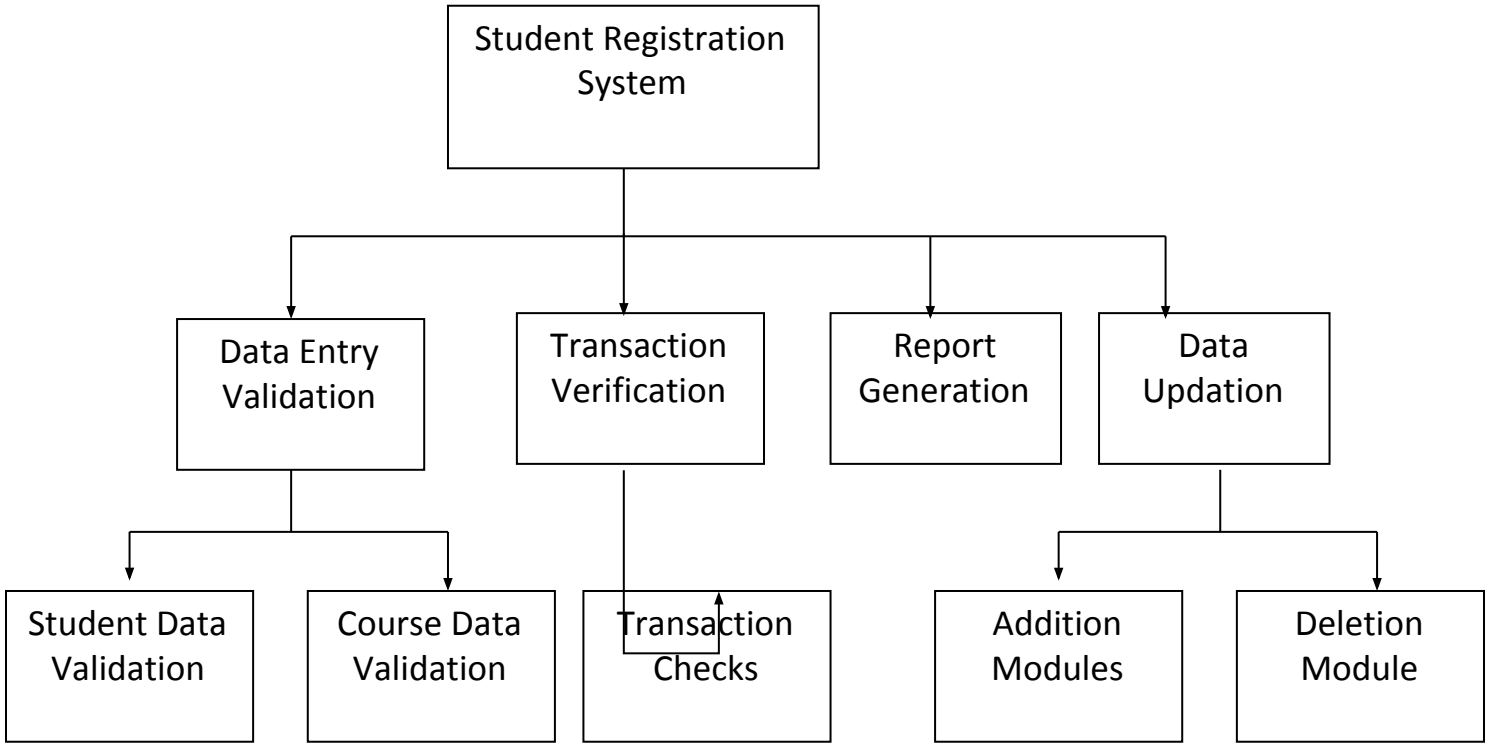
- Suppose that you need to design software that can be used by the patients to make an appointment with the doctor.



Functional Decomposition Example



Functional Decomposition Example



Design Methods

- **Feature-oriented design:**
  - Assigns features to modules.
  - High-level design describes the system in terms of a service and a collection of features.
  - Lower-level designs describe how each feature augments the service and identifies interactions among features.
- **Data-oriented decomposition:**
  - Focuses on how data will be partitioned into modules.
  - High-level design describes conceptual data structures
  - Lower-level designs provide detail as to how data are distributed among modules and how the distributed data realize the conceptual models.

Design Methods

- **Process-oriented decomposition:**
  - Partitions the system into concurrent processes.
  - High-level design

  (1) Identifies the system's main tasks

  (2) Assigns tasks to runtime processes

  (3) Explains how the tasks coordinate with each other.
  - Lower-level designs describe the processes in more detail.
- **Event-oriented decomposition:**
  - Focuses on the events that the system must handle and assigns responsibility for events to different modules.
  - High-level design catalogues the system's expected input events.
  - Lower-level designs decompose the system into states and describe how events trigger state transformations.

Design Methods

- **Object-oriented design:**
  - Assigns objects to modules.
  - High-level design identifies the system's object types and explains how objects are related to one another.
  - Lower-level designs detail the objects' attributes and operations.

Decomposition and Views

- 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
- ***Modularity*** is the most common manifestation of separation of concerns. Software is divided into separately named and addressable components, sometimes called *modules,* that are integrated to satisfy problem requirements.

Architectural Styles

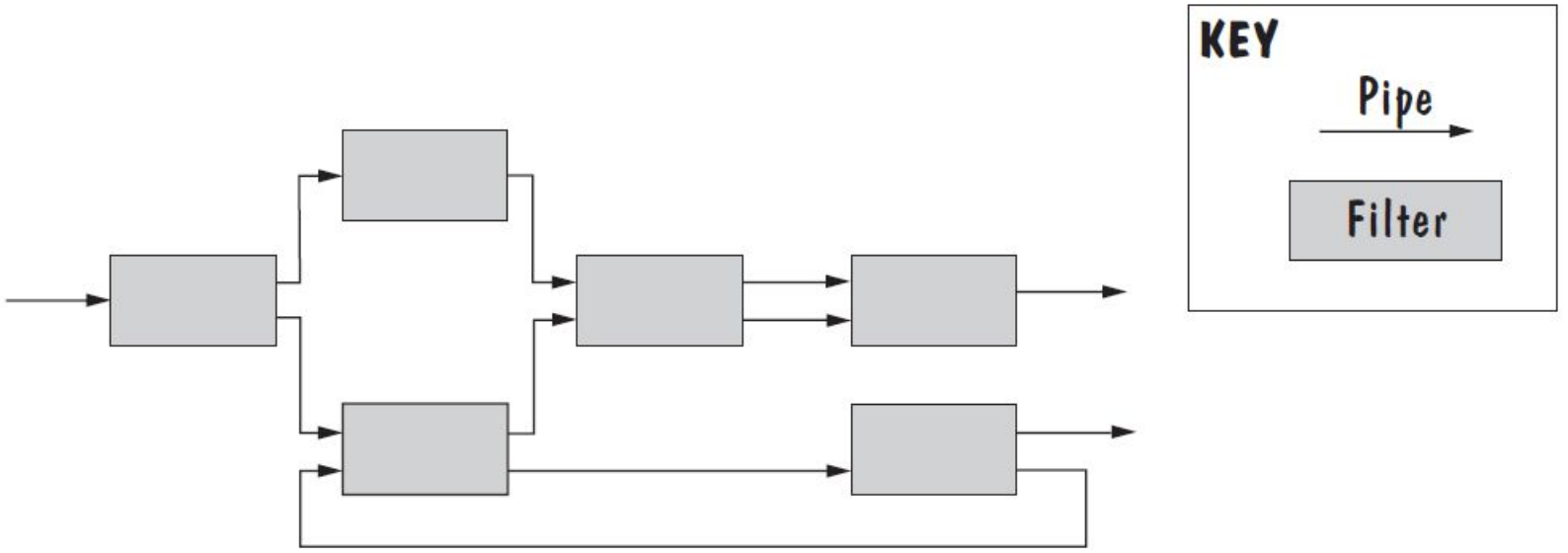
- Software **architectural styles** are established, large-scale patterns of system structure.
- They have **defining rules, elements, and techniques** that result in designs with recognizable structures and well understood properties.
- Not complete detailed solutions. Rather, they are **loose templates** that offer distinct solutions for coordinating a system's components.
- Focus on the different ways that components might **communicate, synchronize, or share data** with one another.

Architectural Styles

- Pipe-Filter
- Client-Server
- Peer-to-Peer
- Publish-Subscribe
- Repositories
- Layering
- Call-Return

Pipe and Filter

- System functionality is achieved by passing input data through a sequence of data-transforming components, called **filters**, to produce output data.
- **Pipes** are connectors that simply transmit data from one filter to the next without modifying the data.



Pipe and Filter

- **Several important properties**
  - The designer can understand the entire system's effect on input and output as the composition of the filters
  - The filters can be reused easily on other systems
  - System evolution is simple
  - Allow concurrent execution of filters
- **Drawbacks**
  - To support a fixed data format during data transmission, each filter must parse input data before performing its computation and then convert its results back to the fixed data format for output. This repeated parsing and unparsing of data can hamper system performance.
  - Not good for handling interactive application

Client Server

- Two types of components:
  - **Server** components offer services
  - **Clients** access them using a **request/reply protocol**.
- The components execute **concurrently** and are usually **distributed** across several computers.
- There may be one centralized server, several replicated servers distributed over several machines, or several distinct servers each offering a different set of services.
- The relationship between clients and servers is **asymmetric**: Clients know the identities of the servers from which they request information, but servers know nothing about which, or even how many, clients they serve.

Client Server

- Clients initiate communications by issuing a ***request***, as a message or a remote-procedure call
- Servers respond by fulfilling the request and ***replying*** with a result.
- Normally, servers are **passive** components that simply react to clients' requests, but in some cases, a server may initiate actions on behalf of its clients.
- For example, a client may send the server an executable function, called a **callback**, which the server subsequently calls under specific circumstances.

Client Server

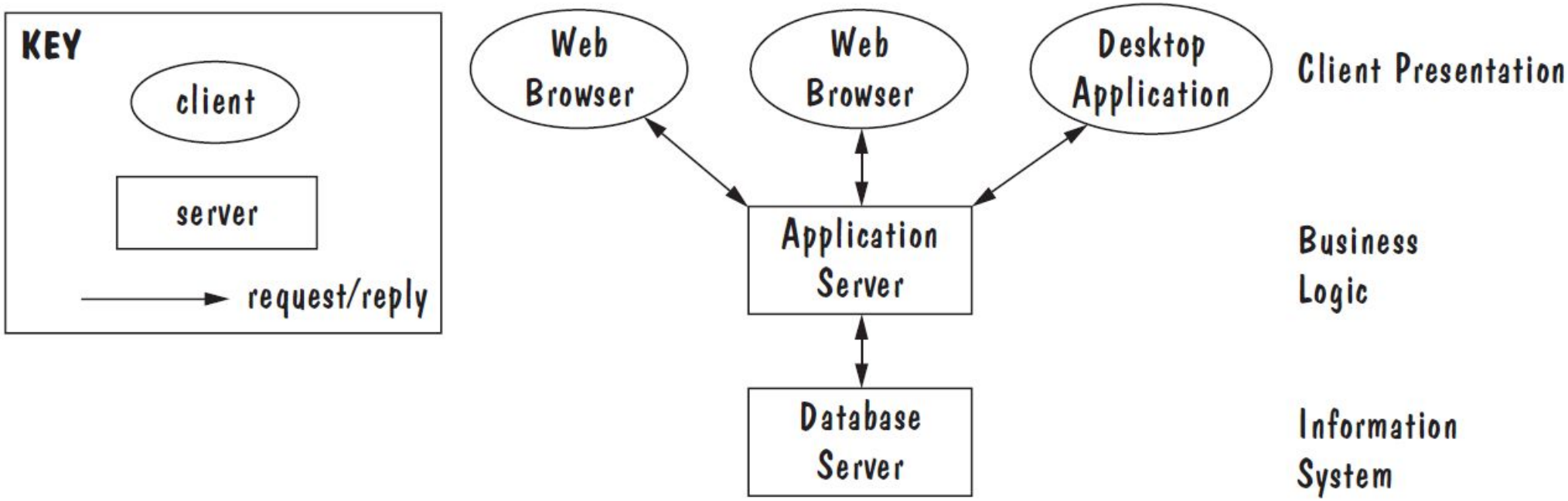


FIGURE 5.7 Three-tiered client-server architecture.

Peer-to-Peer

- A **peer-to-peer** (P2P) architecture is one in which each component executes as its own process and acts as both a client of and a server to other peer components.
- Each component has an **interface** that specifies not only the services it provides, but also the services that it requests from other peer components.
- Peers communicate by **requesting** services from each other.
- Any component can **initiate** a request to any other peer component.
- What differs among components are the data each component stores locally.
- System's data are distributed among the components; whenever a component needs information not stored locally, it retrieves it from a peer component.

Peer-to-Peer

- The best-known P2P architectures are file-sharing networks, such as **Napster** and **Freenet**, in which the components provide similar services to each other.
- **P2P Networks Advantages:**
  - They scale up well.
  - Each added component increases the system's capabilities, in the form of new or replicated data and as additional server capacity.
  - Highly tolerant of network failures, because data are replicated and distributed over multiple peers.
- **P2P Networks Disadvantage:**
  - Each added component increases demands on the system in the form of additional requests.

Peer-to-Peer

- When NOT to use:
  - When file contents change frequently (e.g. prices)
  - When sharing speed has importance (e.g. large files are needed quickly)
  - File quality is critical
  - When trust between peers is required (e.g. the content is protected)

Publish-Subscribe

- In a publish-subscribe architecture, components interact by **broadcasting** and **reacting** to events.
  - Component expresses interest in an event by **subscribing** to it.
  - When another component announces (**publishes**) that the event has taken place, the subscribing components are notified.
  - **Implicit invocation** is a common form of publish-subscribe architecture, in which a subscribing component associates one of its procedures with each event of interest (called **registering** the procedure).
  - When the event occurs, the publish-subscribe infrastructure invokes all the event's registered procedures.
- In contrast to client-server and P2P components, publish-subscribe components know nothing about each other.

Publish-Subscribe

- Publishing component simply announces events and then waits for interested components to react; each subscribing component simply reacts to event announcements, regardless of how they are published.
- **For Example**, tools such as editors register for events that might occur during a debugger's functioning.
  - Consider that the debugger processes code, one line at a time.
  - When it recognizes that it has reached a set breakpoint, it announces the event **"reached breakpoint"**
  - Then, the system forwards the event to all registered tools, including the editor
  - Editor reacts to the event by automatically scrolling to the source-code line that corresponds to the breakpoint.

Publish-Subscribe

- **Strengths**:
  - Provide strong support for system evolution and customization.
  - Any component can be added to the system and can register itself without affecting other components.
  - Can easily reuse publish-subscribe components in other event-driven systems.
- **Weaknesses**:
  - Components can pass data at the time they announce events. But if components need to share persistent data, the system must include a shared repository to support that interaction. This sharing can diminish the system's extensibility and reusability.
  - Difficult to test, because the behavior of a publishing component will depend on which subscribing components are monitoring its events. Thus, we cannot test the component in isolation and infer its correctness in an integrated system.

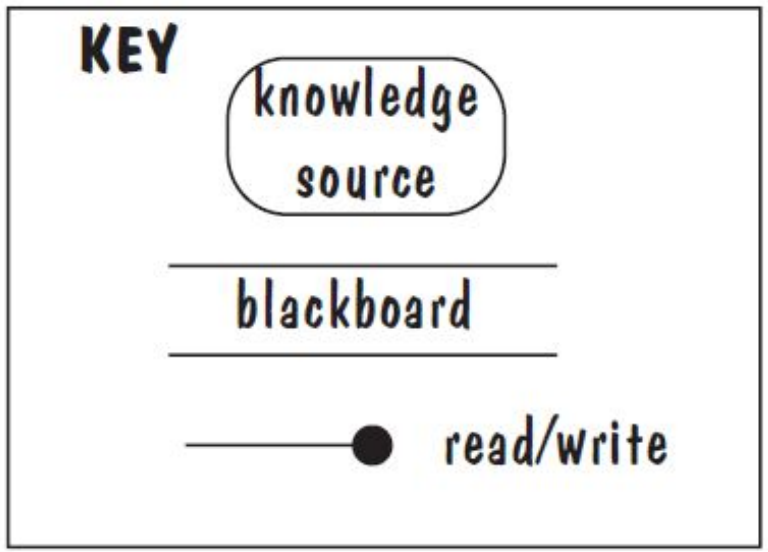
Repositories

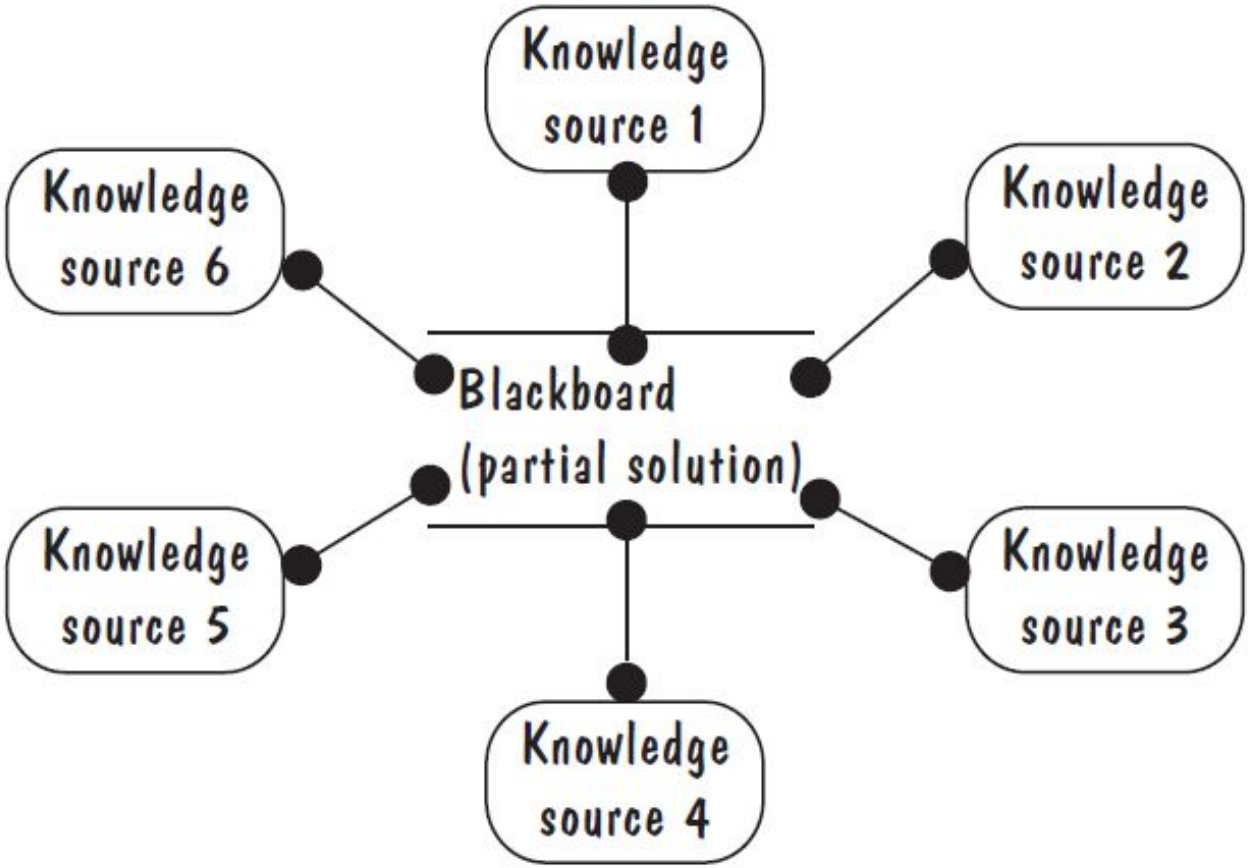
- A **repository** style of architecture consists of two types of components:
  - Central data store
  - Associated data-accessing components.
- Shared data are stockpiled in the **data store**.
- **Data accessors** are computational units that store, retrieve, and update the information.
- In the **blackboard** type of repository, the data accessing components are ***reactive***: they execute in reaction to the current contents of the data store.
- Blackboard contains information about the current state of the system's execution that triggers the execution of individual data accessors, called **knowledge sources**.
- For example, the blackboard may store computation tasks, and an idle knowledge source checks a task out of the blackboard, performs the computation locally, and checks the result back into the blackboard.

Repositories

- **For example**, in a rule-based system, the current state of the solution is stored in the blackboard, and knowledge sources iteratively revise and improve the solution by applying rewriting rules.
- An important property of this style of architecture is the **centralized management** of the system's key data.
- In the data store, we can
  - Localize responsibility for storing persistent data
  - Managing concurrent access to the data
  - Enforcing security and privacy policies, and protecting the data against faults (e.g., via backups).

Blackboard


KEY
knowledge source
blackboard
read/write


Knowledge source 1
Knowledge source 6
Knowledge source 2
Blackboard
(partial solution)
Knowledge source 5
Knowledge source 3
Knowledge source 4

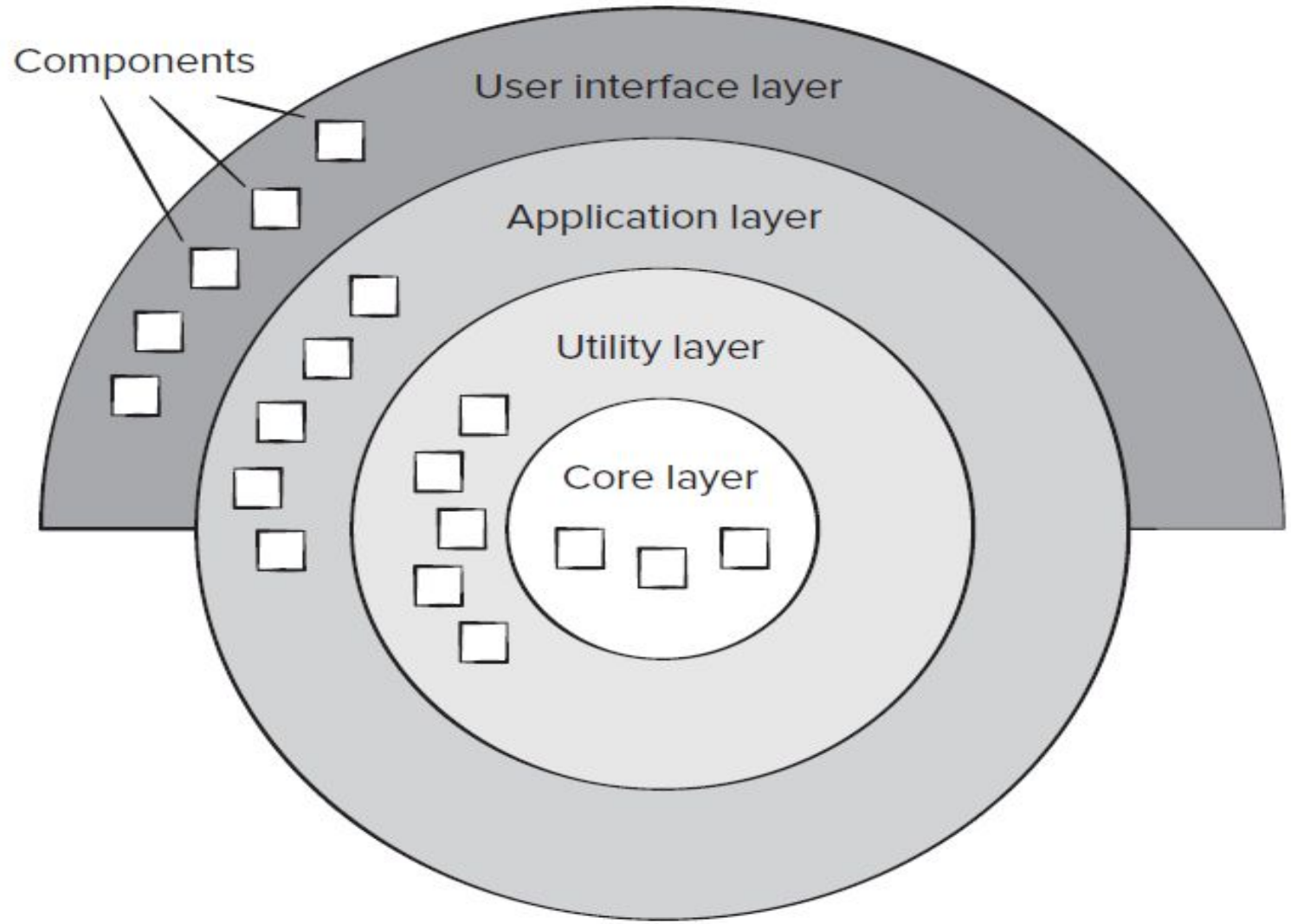
Layering

- Layers are **hierarchical**
  - Each layer provides **service** to the one outside it and acts as a client to the layer inside it
  - **Layer bridging**: allowing a layer to access the services of layers below its lower neighbor
- The design includes **protocols**
  - Explain how each pair of layers will interact
- Under no circumstances does a layer access the services offered by a higher-level layer; the resulting architecture would no longer be called layered.

Layering

- **Strengths**:
  - The layering style is useful whenever we can decompose our system's functionality into steps, each of which builds on previous steps.
  - Layer modification is relatively easy; such changes should affect at most only the two adjacent layers.
- **Weaknesses**:
  - It is not always easy to structure a system into distinct layers of increasingly powerful services, especially when software units are seemingly interdependent.
  - It may appear that layers introduce a performance cost from the set of calls and data transfers between system layers.

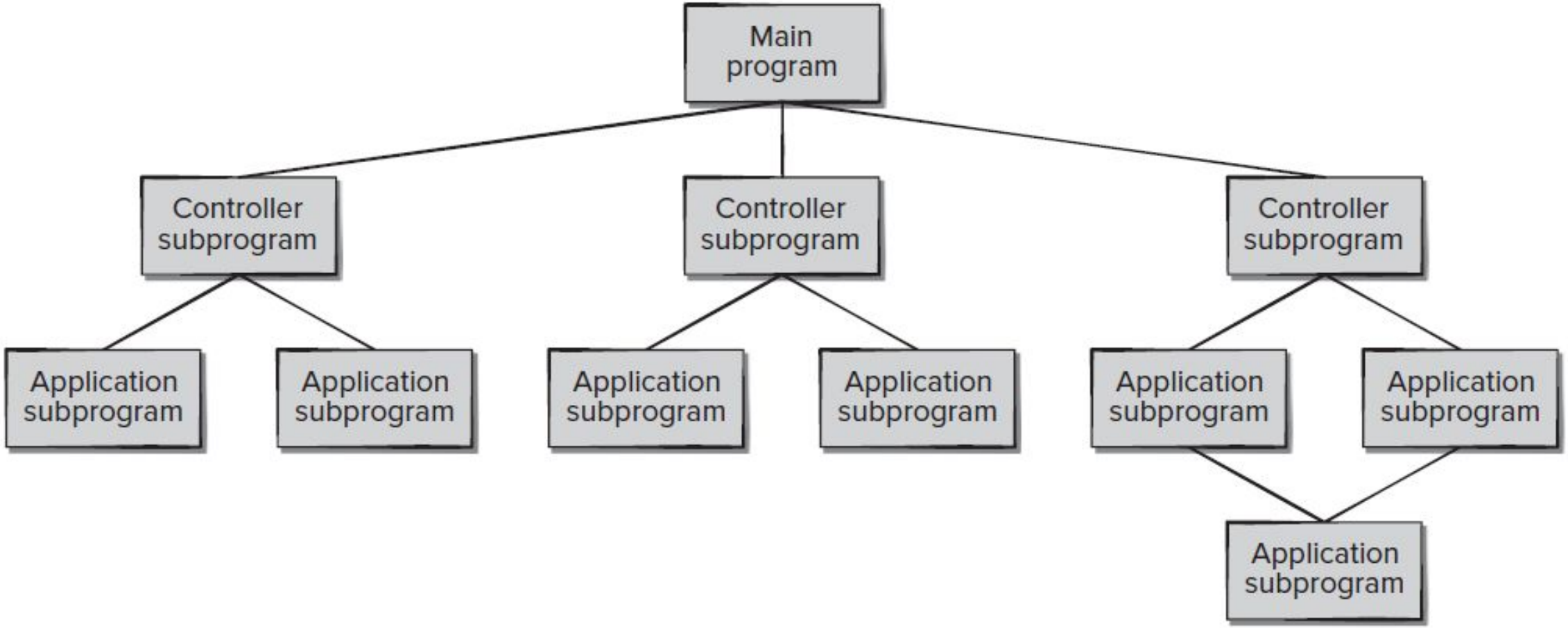
Layering


Components
User interface layer
Application layer
Utility layer
Core layer

Call-and-Return

- Enables you to achieve a program structure that is relatively easy to modify and scale.
- Two substyles that exist within this category:
  - ***Main program/subprogram architectures***. This classic program structure decomposes function into a control hierarchy where a “main” program invokes several program components, which in turn may invoke still other components.
  - ***Remote procedure call architectures.*** The components of a main program/subprogram architecture are distributed across multiple computers on a network.

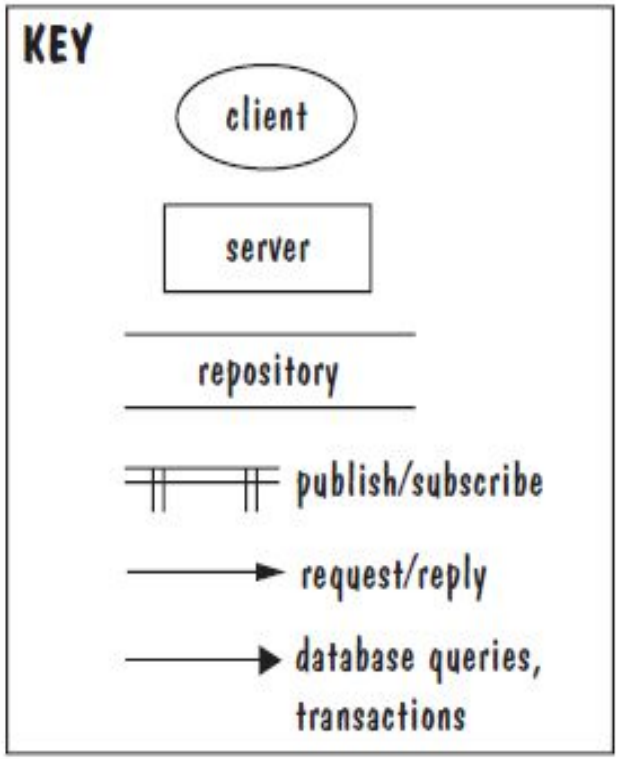
Call-and-Return

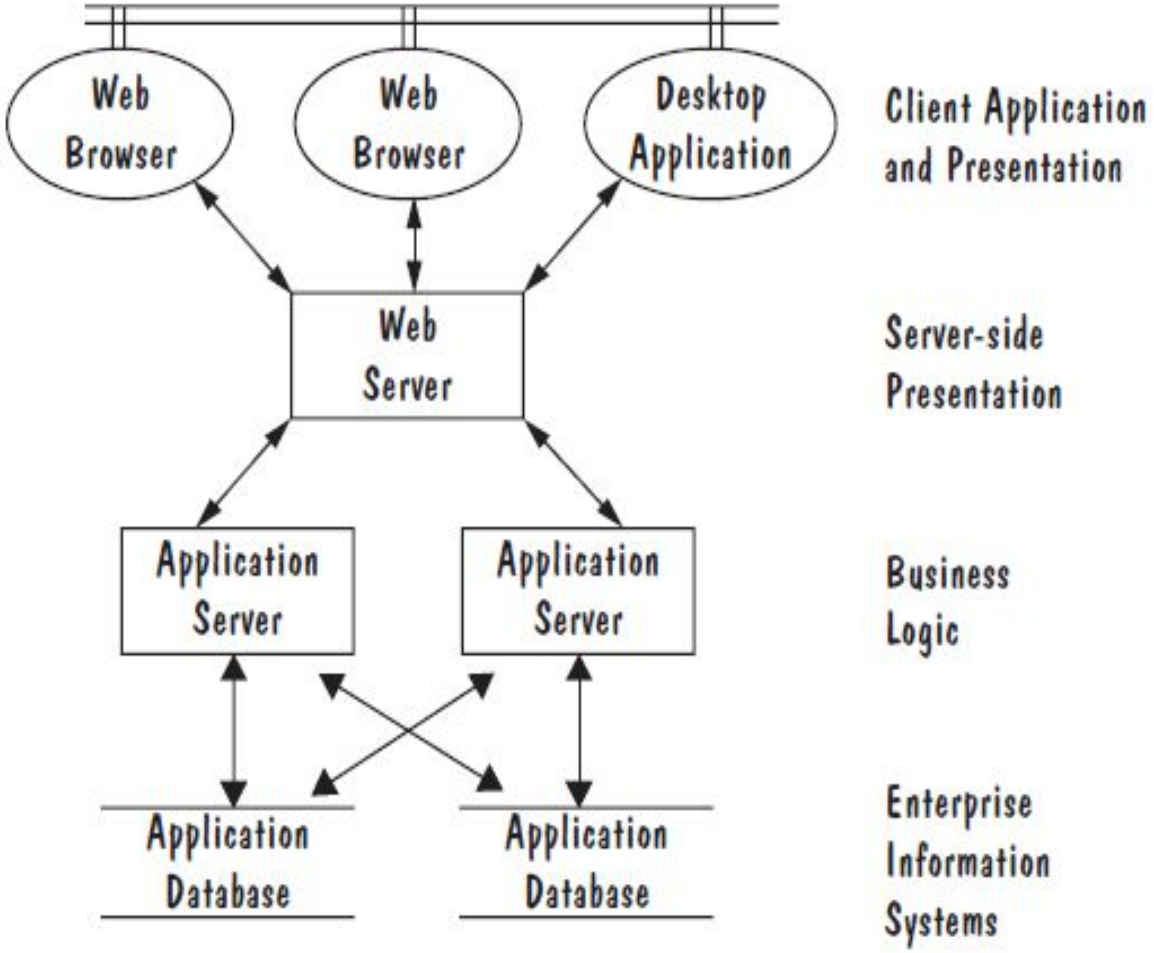


Combining Architectural Style

- Actual software architectures rarely based on purely one style
- 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
- If architecture is expressed as collection of models, documentation must be created to show relation between models
- Integration of different architectural styles is easier if the styles are compatible

Combining Architectural Style





Combination of Publish-Subscribe, Client-Server and Repository Architectural Style