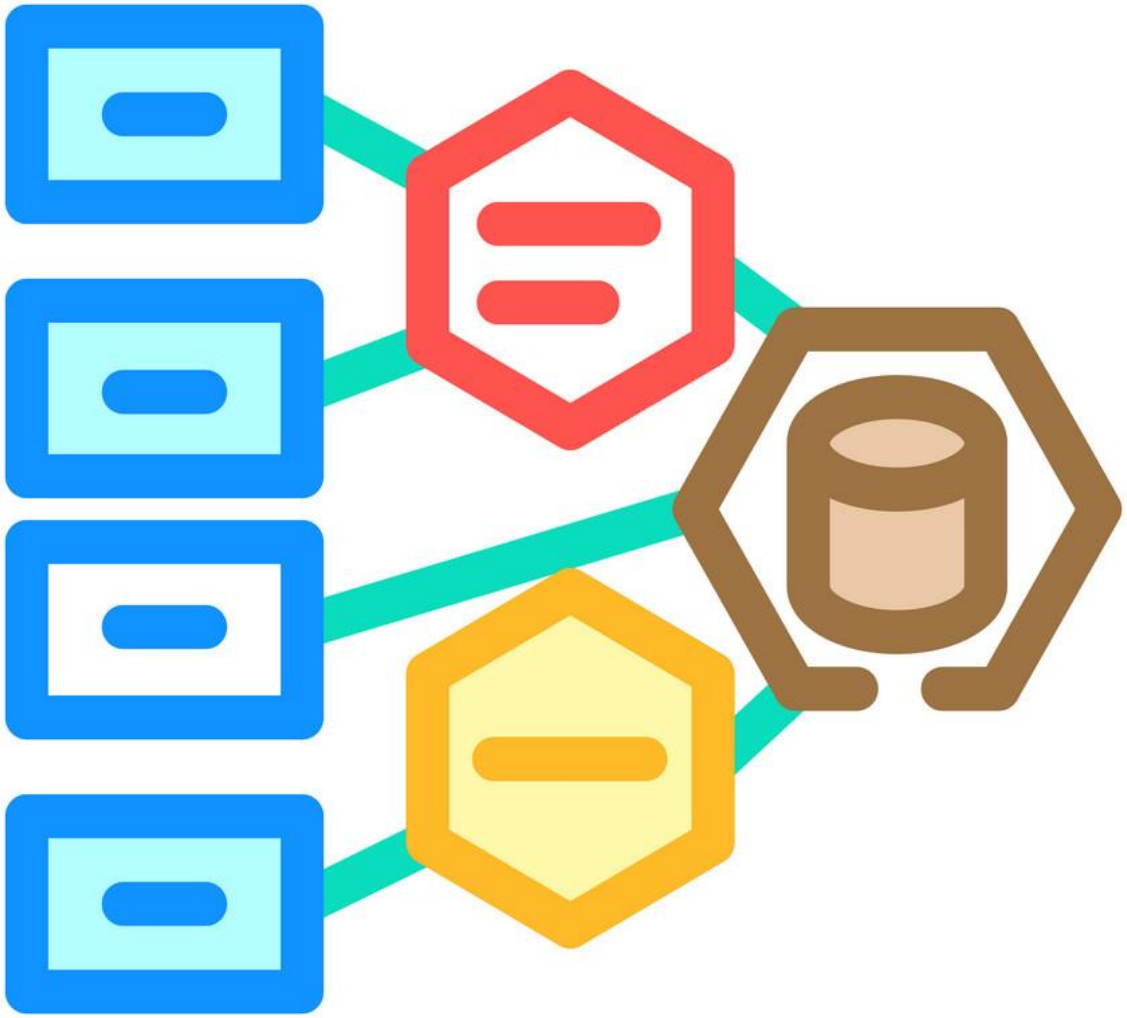
Software Design and architecture

Fall 2024

Dr. Natalia Chaudhry



Design vs architecture

Software Architecture:

•Focuses on the **high-level structure** of the software system.
•Defines the overall **organization** of the system, its **components**, and how they interact.
•Deals with **non-functional requirements** such as scalability, performance, security, and maintainability.
•Specifies **architectural styles** or patterns like microservices, layered architecture, etc.
•Concerned with **global decisions** that affect the system as a whole.

Software Design:

•Focuses on the **detailed structure** of individual components or modules within the architecture.
•Concerned with **functional requirements**, ensuring the system meets specific user needs.
•Includes defining **algorithms**, **data structures**, and **interfaces**.
•Covers **class diagrams**, **object-oriented design principles**, and **design patterns** (e.g., Factory, Singleton).

Relationship:

- **Software architecture comes before software design**. Architecture sets the stage by deciding the high-level organization, while design fills in the details within that structure.
- **Design can exist within architecture**; however, design tends to deal with a lower level of abstraction.
- In practice, **software design is sometimes considered a subset** of software architecture because it works within the constraints and guidelines set by the architecture.

Course learning outcomes

- Critique an existing architecture or design.
- Differentiate how various architectural styles and design patterns enhance and degrade a system's functional-and non-functional properties.
- Generate and justify architecture and/or design given a collection of requirements.
- Produce and present concise and unambiguous architecture and design descriptions.
- Create and implement an architecture and design, refining it into a complete system.

The course is divided into **two major modules**

The **first module** covers the introduction to software design and provides details of software architecture and architectural styles.

The **second module** covers the detailed design phase with introduction to Object Oriented Design and various design patterns.

	Topics
1	Introduction Software design and architecture
2	Principles of good software design. Correctness and Robustness Flexibility, reusability and efficiency
3	Introduction to software architecture
4	Software architectural design Software architectural attributes Attribute types Trade-off of attributes and choices
5	Software Architectural Styles Data flow architctures Layered Event-based Data-centered MVC Multi-tier distributed Service Oriented

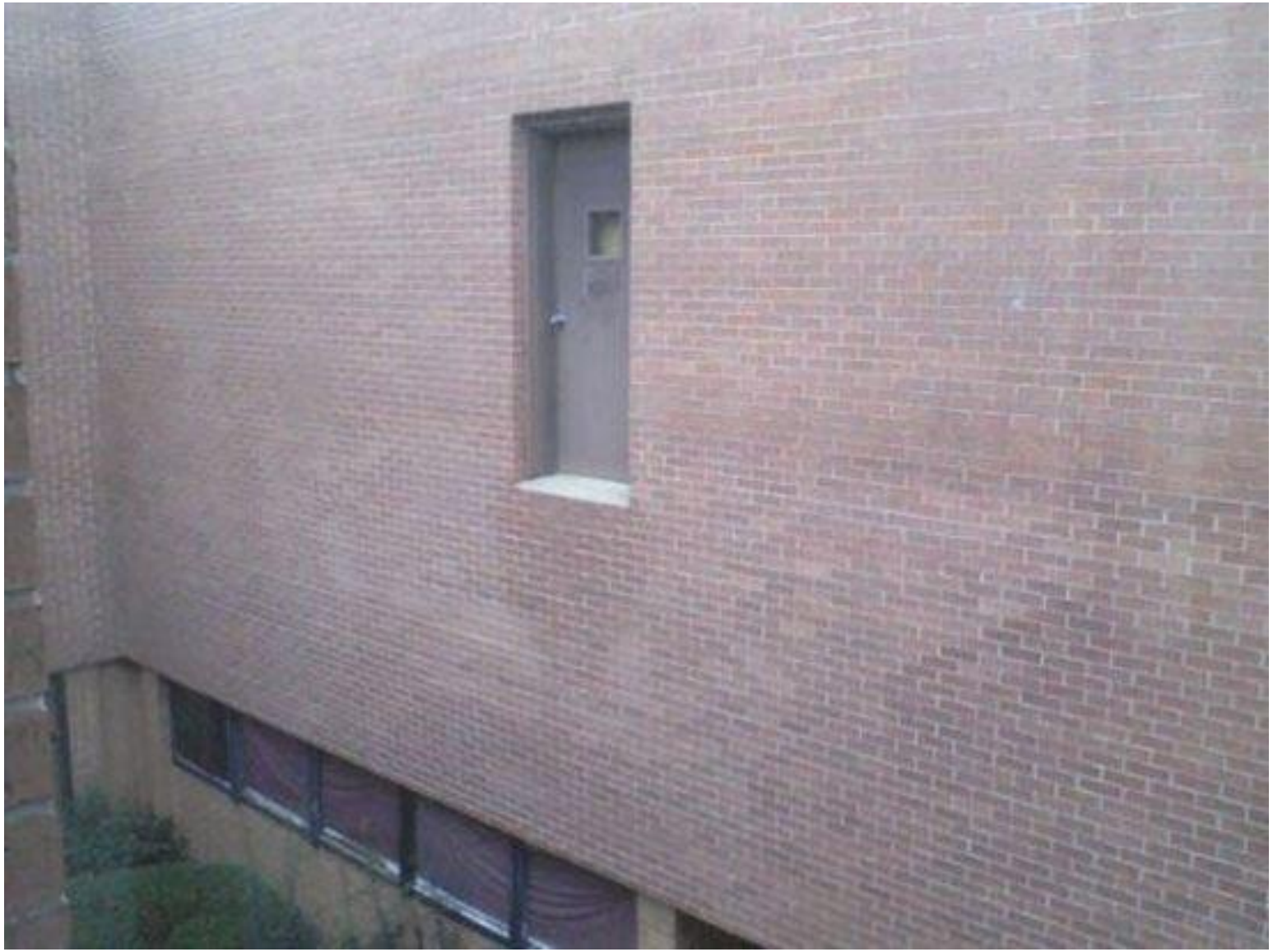
	Topics
6	ANALYSIS PHASE (OBJECT ORIENTED DESIGN) Functional Modeling Activity Diagrams Use case diagrams
7	ANALYSIS PHASE (OBJECT ORIENTED DESIGN) Structural Modeling Classes, Attributes, and Operations, Relationships CRC Cards Class Diagrams
8	ANALYSIS PHASE (OBJECT ORIENTED DESIGN) Behavioural Modeling Objects, operations, and messages Sequence Diagrams Behavioural State Machines – States, Events, Transitions, Actions, and Activities
9	DESIGN PHASE (OBJECT ORIENTED DESIGN) Evolving the analysis models into design models
6	ANALYSIS PHASE (OBJECT ORIENTED DESIGN) Functional Modeling Activity Diagrams Use case diagrams

	Topics
10	Introduction to Components and Component Oriented Design
11	Pattern oriented Design Creational patterns Structural patterns Behavioral pattern

Motivation

A staircase that leads right into a wall!!!

A door that would drop you 10 feet down!



An impossible to use ATM machine

What do you think is wrong in these real life scenarios?

- The requirements are correct!
  - A staircase next to the outer wall
  - A door on the first floor
  - An ATM outside the bank branch
  - The bridge
- The design is flawed!
  - The execution based on the design results in disaster.

- Software are no different.

- For a useful software, it has to be ‘engineered’; which involves giving specific attention to every phase of software development.

What is design?

Design is the first step in the development phase for any engineered product or system.

Design is about HOW the system will perform its functions.

Software design

- Software design is like drawing up blueprints for a building before it's constructed. It's the process of planning and creating a detailed plan for how a piece of software will work, what it will do, and how different parts will interact with each other.
- Just like architects consider the layout, materials, and functionality of a building, software designers think about the structure, features, and flow of a program. It's all about organizing ideas and instructions so that programmers can build the software effectively.
- The goal of software design is to build a model that meets all customer requirements and leads to successful implementation.

Requirements specification was about the WHAT the system will do

Design is about the HOW the system will perform its functions
provides the overall decomposition of the system
allows to split the work among a team of developers
also lays down the groundwork for achieving non-functional requirements (performance, maintainability, reusability, etc.)
takes target technology into account (e.g., kind of middleware, database design, etc.)

Without proper design, software can become chaotic, difficult to maintain, and prone to errors.

- Imagine you're developing a mobile banking app. Before writing any code, you need to carefully design the software. You'll need to consider various aspects such as user interface design, security measures, data storage methods, communication with banking servers, and error handling.
- Without a solid design, you might end up with a confusing user interface that frustrates customers, security vulnerabilities that expose sensitive financial information, or inefficient data storage that slows down the app.

Software Development Activities

- **Requirements Elicitation**
- **Requirements Analysis** (e.g., Structured Analysis, OO Analysis)

  – analyzing requirements and working towards a conceptual model without taking the target implementation technology into account

  – useful if the conceptual gap between requirements and implementation is large
  - part of requirements engineering (but may produce more than what is going to be part of the requirement spec)
- **Design**
  - coming up with solution models taking the target implementation technology into account
- **Implementation**
- **Testing**

Software Design in SDLC

- In SDLC (Software Development Life Cycle), Design phase is one of the most important phases.
- In the software engineering context, design focuses on four major areas of concern: data, architecture, interfaces and components.

Design Process Activities

- **Architectural design**
  - **Modules, inter-relationships etc**
- Abstract specification
  - Services of each sub-system, constraints etc
- Interface design
  - Interface to other sub-system or outside enviornment
- Component design
  - Services allocated to components and their interfaces designed
- Data structure design
- Algorithm design

Patient Management Module

•**Functions**: Manages patient records, demographics, medical history, and contact information.
•**Inter-relationships**: Interacts with the Appointment Scheduling Module to update and retrieve patient data during the booking process. It also interfaces with the Billing Module for generating invoices based on patient services and with the Prescription Management Module for maintaining patient medication history.

Appointment Scheduling Module

•**Functions**: Manages the scheduling of patient appointments, including booking, rescheduling, and cancellations.
•**Inter-relationships**: Integrates with the Patient Management Module to pull patient data when appointments are made. Communicates with the Doctor Availability Module to ensure appointments are scheduled according to the availability of medical professionals.

Billing Module

- **Functions**: Handles the financial aspects, including generating bills, processing payments, and managing insurance claims.
- **Inter-relationships**: Works closely with the Patient Management Module to retrieve patient service records and the Insurance Claims Module to submit and track claims. It also interfaces with the Prescription Management Module to bill for prescribed medications.

Prescription Management Module

•**Functions**: Manages prescriptions, including writing, renewing, and tracking medication orders.

•**Inter-relationships**: Connects with the Patient Management Module to access patient medical history and with the Billing Module to ensure that prescribed medications are correctly billed. It also integrates with third-party pharmacies for e-prescriptions.

Lab Results Module

•**Functions**: Manages laboratory results, allowing doctors to view and analyze patient lab tests.

•**Inter-relationships**: Integrates with the Patient Management Module to store and retrieve lab results. It also interacts with the Doctor Availability Module to notify doctors of new lab results that need review.

Doctor Availability Module

•**Functions**: Tracks the availability of medical professionals and manages their schedules.
•**Inter-relationships**: Ties into the Appointment Scheduling Module to ensure appointments are only made when doctors are available. Also communicates with the Lab Results Module to ensure doctors are notified of new test results.

Insurance Claims Module

•**Functions**: Manages the submission and tracking of insurance claims.
•**Inter-relationships**: Interfaces with the Billing Module to submit claims based on services rendered and with the Patient Management Module to ensure all necessary patient information is included in the claims.

Design Process Activities

- Architectural design
  - Modules, inter-relationships etc
- **Abstract specification**
  - **Services of each sub-system, constraints etc**
- Interface design
  - Interface to other sub-system or outside enviornment
- Component design
  - Services allocated to components and their interfaces designed
- Data structure design
- Algorithm design

Smart Home Automation System

Sub-Systems and Their Abstract Specifications

A. Lighting Control Sub-System

Abstract Specification:

- **Services**:
  - Turn lights on/off based on user input or schedule.
  - Adjust brightness and color temperature.
  - Automate lighting based on room occupancy.
  - Integrate with security systems for alert-based lighting.
- **Constraints**:
  - Must respond to commands within 1 second.
  - Integration with third-party smart bulbs must comply with specific API standards.
  - Must operate over both Wi-Fi protocols.

B. Security and Surveillance Sub-System

Abstract Specification:

- **Services**:
  - Monitor and record video feeds from security cameras.
  - Detect motion and trigger alerts.
  - Manage door locks and grant access remotely.
  - Integrate with emergency services (e.g., automatic alarm triggering).
- **Constraints**:
  - Video feed latency must not exceed 2 seconds.
  - Must store video recordings securely, adhering to encryption standards.
  - Must comply with local data protection laws.

C. Climate Control Sub-System

Abstract Specification:

- **Services**:
  - Monitor and adjust the home's temperature and humidity.
  - Provide energy usage analytics and suggestions for optimization.
  - Enable geofencing to adjust climate settings based on occupants' proximity.
  - Integrate with weather forecasting services for predictive adjustments.
- **Constraints**:
  - Must maintain temperature within ±1°C of the setpoint.
  - Must operate efficiently with a 24-hour battery backup in case of power loss.

Home Automation Hub (Central Control)

Abstract Specification:

- **Services**:
  - Provide a unified interface for controlling all sub-systems.
  - Enable remote access and control via mobile app or web interface.
  - Manage automation rules across different sub-systems (e.g., “If the security system detects motion, turn on the lights”).
  - Offer voice control integration with popular virtual assistants (e.g., Amazon Alexa, Google Assistant).
- **Constraints**:
  - Must operate with high availability (99.9% uptime).
  - Latency for cross-sub-system commands must be less than 500ms.
  - Must ensure data privacy and security, particularly for remote access.

Design Process Activities

- Architectural design
  - Modules, inter-relationships etc
- Abstract specification
  - Services of each sub-system, constraints etc
- **Interface design**
  - **Interface to other sub-system or outside enviornment**
- Component design
  - Services allocated to components and their interfaces designed
- Data structure design
- Algorithm design

User Interface (UI) Design

The User Interface (UI) design is crucial for ensuring that homeowners can easily control and monitor their smart home devices through intuitive and responsive interfaces.

A. Mobile App Interface

Dashboard:

- A centralized screen where users can quickly see the status of all connected devices and systems (e.g., lights on/off, current temperature, security status).
- Visual indicators for critical alerts (e.g., security breach, high energy usage).
- Quick access buttons for common actions (e.g., turning all lights off, activating security mode).

- **Control Panels**:
  - **Lighting Control**: A panel with sliders for brightness and color temperature, along with buttons for on/off and preset lighting scenes (e.g., "Relax," "Movie Time").
  - **Climate Control**: A temperature slider, humidity control, and energy usage graph. Options for setting schedules or geofencing rules.
  - **Security**: A live video feed with playback controls, door lock/unlock buttons, and an event log for recent security alerts.
  - **Entertainment**: Controls for volume, track selection, and synchronized playback across multiple rooms.
- **Settings and Automation**:
  - A section where users can configure system settings, set up automation rules (e.g., "Turn off lights when the security system is armed"), and integrate third-party devices.

B. Web Interface

- **Responsive Design**:
  - The web interface should mirror the functionality of the mobile app but optimized for larger screens. It should provide a more detailed overview and deeper configuration options.
- **Advanced Controls**:
  - Access to advanced system configurations, detailed logs, and historical data (e.g., energy usage trends over months).
- **User Management**:
  - Interface for managing user permissions, creating guest access profiles, and setting up multi-factor authentication.

Design Process Activities

- Architectural design
  - Modules, inter-relationships etc
- Abstract specification
  - Services of each sub-system, constraints etc
- Interface design
  - Interface to other sub-system or outside enviornment
- **Component design**
  - **Services allocated to components and their interfaces designed**
- Data structure design
- Algorithm design

Component Design

1. Lighting Control Sub-System

- **Components**:
  - **Light Controller**:
    - **Services Allocated**:
      - Turning lights on/off.
      - Adjusting brightness.
      - Changing color temperature.
    - **Interfaces**:
      - **API Interface**:
        - POST /light/{id}/on: Turns on the light with a specific ID.
        - POST /light/{id}/off: Turns off the light with a specific ID.
      - **Hardware Interface**:
        - Communication with smart bulbs over Wi-Fi using predefined protocols.

2. Security and Surveillance Sub-System

- **Components**:
  - **Camera Controller**:
    - **Services Allocated**:
      - Managing live video feeds.
      - Recording and storing video.
      - Motion detection.
    - **Interfaces**:
      - **API Interface**:
        - GET /camera/{id}/feed: Retrieves live video feed.
        - POST /camera/{id}/record: Starts recording on a specific camera.
        - GET /camera/{id}/recordings: Retrieves stored recordings.
      - **Hardware Interface**:
        - Interface with IP cameras, handling video encoding, and transmission protocols.

Design Process Activities

- Architectural design
  - Modules, inter-relationships etc
- Abstract specification
  - Services of each sub-system, constraints etc
- Interface design
  - Interface to other sub-system or outside enviornment
- Component design
  - Services allocated to components and their interfaces designed
- **Data structure design**
- Algorithm design

1. Lighting Control Sub-System

A. Data Structures

•Light Structure

```
{
  "light_id": "string",          // Unique identifier for the light
  "name": "string",              // Name or label for the light (e.g., "Living Room Light")
  "room_id": "string",           // Identifier for the room where the light is located
  "status": "boolean",           // Current status (on/off)
  "brightness": "int",           // Brightness level (0-100)
  "color_temperature": "int",    // Color temperature in Kelvin
  "last_updated": "timestamp"    // Last time the light's status was updated
}
```

Occupancy Sensor Structure

```
{
  "sensor_id": "string",          // Unique identifier for the occupancy sensor
  "room_id": "string",            // Identifier for the room being monitored
  "status": "boolean",            // Occupied (true) or vacant (false)
  "last_detected": "timestamp"    // Last time occupancy was detected
}
```

B. Relationships

- **Room → Lights**: One-to-Many
  - A room can have multiple lights associated with it.
- **Room → Occupancy Sensor**: One-to-One
  - Each room can have one occupancy sensor that monitors presence.

2. Climate Control Sub-System

A. Data Structures

- **Thermostat Structure**

```
{
  "thermostat_id": "string",       // Unique identifier for the thermostat
  "room_id": "string",             // Identifier for the room the thermostat controls
  "current_temperature": "float",  // Current temperature reading
  "target_temperature": "float",   // Target temperature setting
  "humidity": "float",             // Current humidity level
  "mode": "string",                // Operating mode (e.g., "cooling", "heating", "off")
  "last_updated": "timestamp"      // Last time the thermostat data was updated
}
```

Energy Usage Structure

```
{
  "usage_id": "string",           // Unique identifier for the energy usage record
  "time_period": "string",        // Time period for the usage data (e.g., "hourly",
"daily")
  "energy_consumed": "float",     // Amount of energy consumed in kWh
  "cost": "float",                // Cost associated with the energy consumed
  "suggestions": "string"         // Energy-saving suggestions based on usage patterns
}
```

Design Process Activities

- Architectural design
  - Modules, inter-relationships etc
- Abstract specification
  - Services of each sub-system, constraints etc
- Interface design
  - Interface to other sub-system or outside enviornment
- Component design
  - Services allocated to components and their interfaces designed
- Data structure design
- **Algorithm design**

1. Lighting Control Sub-System

A. Occupancy-Based Lighting Algorithm

Objective: Automatically control the lights based on room occupancy.

Algorithm:

1.Input:

- occupancy_status from the Occupancy Sensor.
- Current time and brightness level.

2.Steps:

- If occupancy_status is true (room occupied):
  - Check the current time and set the brightness level accordingly (e.g., dimmer at night).
  - If the lights are off, turn them on with the appropriate brightness.
- If occupancy_status is false (room unoccupied):
  - If the lights are on, turn them off after a specified delay (e.g., 5 minutes of no occupancy).

3.Output:

- Updated light_status (on/off) and brightness level.

2. Security and Surveillance Sub-System

A. Motion-Triggered Recording Algorithm

Objective: Start recording video when motion is detected by a camera.

Algorithm:

1.Input:

- •motion_detected signal from the camera.
- •Camera camera_id.

2.Steps:

- •If motion_detected is true:
  - •Fetch the current timestamp and create a new recording entry in the database.
  - •Start recording and store the video feed with metadata (camera_id, start_time).
- •If the motion persists, continue recording until no motion is detected for a certain period (e.g., 30 seconds).
- •Once the motion ends, stop recording and update the end_time in the recording entry.

3.Output:

- •Recorded video file saved to the server.
- •Updated database entry with recording details.

Levels of Software Design

- Architectural design (high-level design)
  - architecture - the overall structure, main modules and their connections
  - addresses the main non-functional requirements (e.g., reliability, performance)
  - hard to change
- Detailed design (low-level design)
  - the inner structure of the main modules
  - detailed enough to be implemented in the programming language

Architectural Design Example

Objective: Provide a high-level structure of the Smart Home Automation System, showing the major subsystems and their interactions.

High-Level Architecture

1.Subsystems:

1. Lighting Control Sub-System
2. Security and Surveillance Sub-System
3. Climate Control Sub-System
4. Entertainment Sub-System
5. Home Automation Hub (Central Control)

2. Interaction Overview:

1. **Home Automation Hub** acts as the central coordinator, communicating with each subsystem.
2. **Lighting Control** interacts with **Occupancy Sensors** to automate lighting based on presence.
3. **Security and Surveillance** communicates with **Cameras** and **Door Locks** to monitor and secure the home.
4. **Climate Control** manages **Thermostats** and **Humidity Sensors** to maintain comfortable living conditions.
5. **Entertainment** controls media devices and interacts with user playlists for synchronized media playback.

3. Communication Patterns:

1. **REST APIs** for control commands and data queries between subsystems and the Home Automation Hub.
2. **Real-Time Events** for triggering actions, such as turning on lights when motion is detected.

Detailed Design of the Lighting Control Sub-System

1.Components:

- **Light Controller**
  - Manages the status of lights (on/off, brightness, color temperature).
- **Occupancy Sensor**
  - Detects presence and sends signals to the Light Controller.
- **Light Management API**
  - Provides endpoints for controlling lights via external commands or automation rules.

2.Data Structures:

- **Light Object**:

{ "light_id": "string", "name": "string", "room_id": "string", "status": "boolean", "brightness": "int", "color_temperature": "int", "last_updated": "timestamp" }

- **Occupancy Event**:

{ "sensor_id": "string", "room_id": "string", "status": "boolean", "timestamp": "timestamp" }

3.Detailed Algorithms:

•**Occupancy-Based Lighting Control Algorithm**:

Input: occupancy_event

Output: Updated light status

1. Fetch light objects associated with the room_id from occupancy_event.
2. If occupancy_event.status == true:
   - For each light in the room, set light.status = true.
   - Adjust brightness based on the time of day (e.g., lower brightness at night).
3. If occupancy_event.status == false:
   - Set a delay timer (e.g., 5 minutes).
   - After the delay, check occupancy status again.
   - If still unoccupied, set light.status = false.
4. Update light objects in the database and send control signals to the lights.

- **Light Management API**:
  - POST /lights/{light_id}/on: Turns on a specific light.
  - POST /lights/{light_id}/off: Turns off a specific light.
  - PUT /lights/{light_id}/brightness: Adjusts the brightness of a specific light.
  - PUT /lights/{light_id}/color: Changes the color temperature of a specific light.

Design vs. Architecture

Software Architecture refers to the high-level structure of a software system. It involves the overall organization of the system, including its major components, their interactions, and the principles guiding its design and evolution. The architecture defines the blueprint for the system, focusing on how the system's components fit together to fulfill the requirements.

Key aspects of Software Architecture:

- **Components:** Major parts of the system (e.g., modules, services).
- **Connectors:** How these components interact (e.g., APIs, messaging systems).
- **Patterns:** Reusable solutions to common problems (e.g., MVC, Microservices).
- **Principles:** Guidelines and rules that govern the architecture (e.g., SOLID principles).
- **Non-functional Requirements:** Performance, scalability, security, etc.

Software Design refers to the process of defining the detailed structure of individual components within the software system. It deals with the lower-level implementation details and how specific parts of the system will be built. Design focuses on the algorithms, data structures, and the flow of control within the system's components.

Key aspects of Software Design:

- **Class Design:** The internal structure of classes and their relationships.
- **Data Structures:** Choice of data structures for efficient data handling.
- **Algorithms:** The logic to perform specific tasks.
- **Design Patterns:** Standard solutions for common design problems (e.g., Singleton, Factory)

The difference between **Software Architecture** and **Software Design** in terms of patterns lies in the scope and level of abstraction at which these patterns operate.

Architecture patterns are high-level, reusable solutions to common problems in software architecture. They describe the overall structure and interaction of major components in a software system.

Common Architecture Patterns:

- **Layered (N-tier) Architecture:** Organizes the system into layers, such as presentation, business logic, and data access layers. Each layer has a specific responsibility and interacts with adjacent layers.
- **Microservices Architecture:** Breaks down the system into small, independently deployable services that communicate over a network. Each service focuses on a specific business capability.
- **Event-Driven Architecture:** Uses events to trigger communication between loosely coupled components, allowing for asynchronous processing and scalability.
- **Client-Server Architecture:** Divides the system into client and server components, where the client requests services, and the server provides them.
- **Service-Oriented Architecture (SOA):** Structures the system as a collection of services that communicate over a network, focusing on reusability and interoperability.

Design patterns operate at a lower level of abstraction and are concerned with solving common problems in the design and implementation of individual components or classes. These patterns provide templates for handling specific design challenges, such as object creation, object structure, and object behavior.

Common Design Patterns:

•Creational Patterns:

- **Singleton:** Ensures a class has only one instance and provides a global point of access to it.
- **Factory Method:** Defines an interface for creating an object, but lets subclasses alter the type of objects that will be created.

- **Structural Patterns:**
  - **Adapter:** Allows incompatible interfaces to work together by converting the interface of a class into another interface clients expect.
  - **Decorator:** Attaches additional responsibilities to an object dynamically, providing a flexible alternative to subclassing.
- **Behavioral Patterns:**
  - **Observer:** Defines a one-to-many dependency between objects, so that when one object changes state, all its dependents are notified.
  - **Strategy:** Defines a family of algorithms, encapsulates each one, and makes them interchangeable.

Example:
Within a component of a microservices architecture (say, the payment processing service), you might use a **Factory Method** pattern to create different payment processors based on the type of payment (credit card, PayPal, etc.). The design pattern provides a solution for managing object creation, making the system more flexible and easier to maintain.

Architecture Patterns *provide solutions for structuring an entire system, while* ***Design Patterns*** *provide solutions for designing and implementing the components within that structure.*

The architecture design representation is derived from the system requirement specification and the analysis model.

Who is responsible for developing the architecture design?
Software architects and designers are involved in this process. They translate (map) the software system requirements into architecture design.

During the translation process, they apply various design strategies to divide and conquer the complexities of an application domain and resolve the software architecture.

Why is software architecture design so important?

There are several reasons.

- A poor design may result in a deficient product that does not meet system requirements,
- is not adaptive to future requirement changes,
- is not reusable,
- exhibits unpredictable behavior or performs badly.
- Without proper planning in the architecture design stage, software production may be very inefficient in terms of time and cost.

Just like sketch of a building helps constructor to correctly construct the building software architecture helps software developer to develop the software properly.

Software design is a process of problem solving and planning for a software solution.

Architecture is:

All about communication.

- What „parts“ are there?
- How do the „parts“ fit together?

Software Architecture Design Guidelines

- identify the right architecture styles to decompose a complex system into its constituent elements.
- Functional and nonfunctional requirements should be identified, verified, and validated before architecture and detailed design work is done.
- Think of abstract design before thinking of concrete design. Always start with an abstract design that specifies interfaces of components and abstract data types.
- Think of nonfunctional requirements early in the design process. When you map functional requirements to an architecture design, you should consider nonfunctional requirements as well.
- Communicate with stakeholders and document their preferences for quality attributes
- Think of software reusability and extensibility as much as possible to increase the reliability and cost-effectiveness of new systems.
- Try to promote high cohesion within each element and loose coupling between elements.

Functional Requirements:

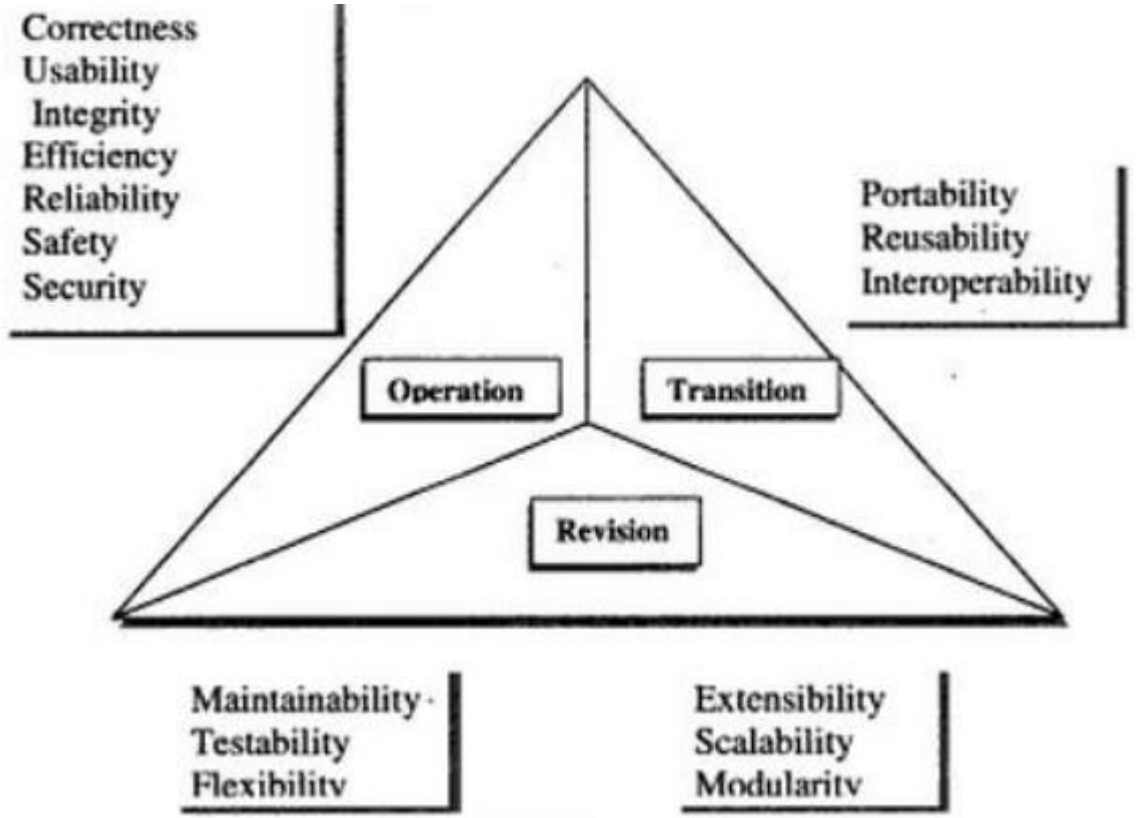
1. **Definition:** Functional requirements describe the specific functionalities or features that a system must provide. They outline the actions the system must perform in response to various inputs.
2. **Focus:** Functional requirements focus on what the system should do to meet the user's needs and achieve the intended goals.
3. **Examples:**
    1. The system must allow users to log in with a valid username and password.
    2. The system must be able to process online payments for customer orders.
    3. The system must generate monthly financial reports for management.

Non-functional Requirements:

1. **Definition:** Non-functional requirements are attributes that characterize the system's overall behavior, performance, and qualities. These requirements are not directly related to specific functionalities but rather describe how the system should behave or perform under certain conditions.
2. **Focus:** Non-functional requirements focus on qualities such as performance, reliability, usability, security, and maintainability.
3. **Examples:**
    1. The system must respond to user inputs within 2 seconds to ensure a responsive user interface.
    2. The system must be available 99.9% of the time to ensure high reliability.
    3. The system must be able to handle 1000 concurrent users to ensure scalability.
    4. The system must encrypt sensitive user data to ensure data security.

Software quality triangle

- In general, the characteristics of a software product can be divided into
  - Operational characteristics
  - Transition characteristics
  - Revision characteristics



What Operational Characteristics should a software have ? *exterior quality*

a) Correctness: The software which we are making should meet all the specifications stated by the customer.

b) Usability/Learnability: The amount of efforts or time required to learn how to use the software should be less. This makes the software user-friendly even for IT illiterate people.

c) Integrity : Just like medicines have side-effects, in the same way a software may have a side-effect i.e. it may affect the working of another application. But a quality software should not have side effects.

d) Reliability : The software product should not have any defects. Not only this, it shouldn't fail while execution.

What Operational Characteristics should a software have ? *exterior quality*

e) Efficiency : This characteristic relates to the way software uses the available resources. The software should make effective use of the storage space and execute command as per desired timing requirements.

f) Security : With the increase in security threats nowadays, this factor is gaining importance. The software shouldn't have ill effects on data / hardware. Proper measures should be taken to keep data secure from external threats.

g) Safety : The software should not be hazardous to the environment/life.

What are the Revision Characteristics of software ? *interior quality*

a) **Maintainability :** Maintenance of the software should be easy for any kind of user.

b) **Flexibility :** Changes in the software should be easy to make.

c) **Extensibility :** It should be easy to increase the functions performed by it.

d) **Scalability :** It should be very easy to upgrade it for more work(or for more number of users).

e) **Testability :** Testing the software should be easy.

f) **Modularity :** Any software is said to made of units and modules which are independent of each other. These modules are then integrated to make the final software. If the software is divided into separate independent parts that can be modified, tested separately, it has high modularity.

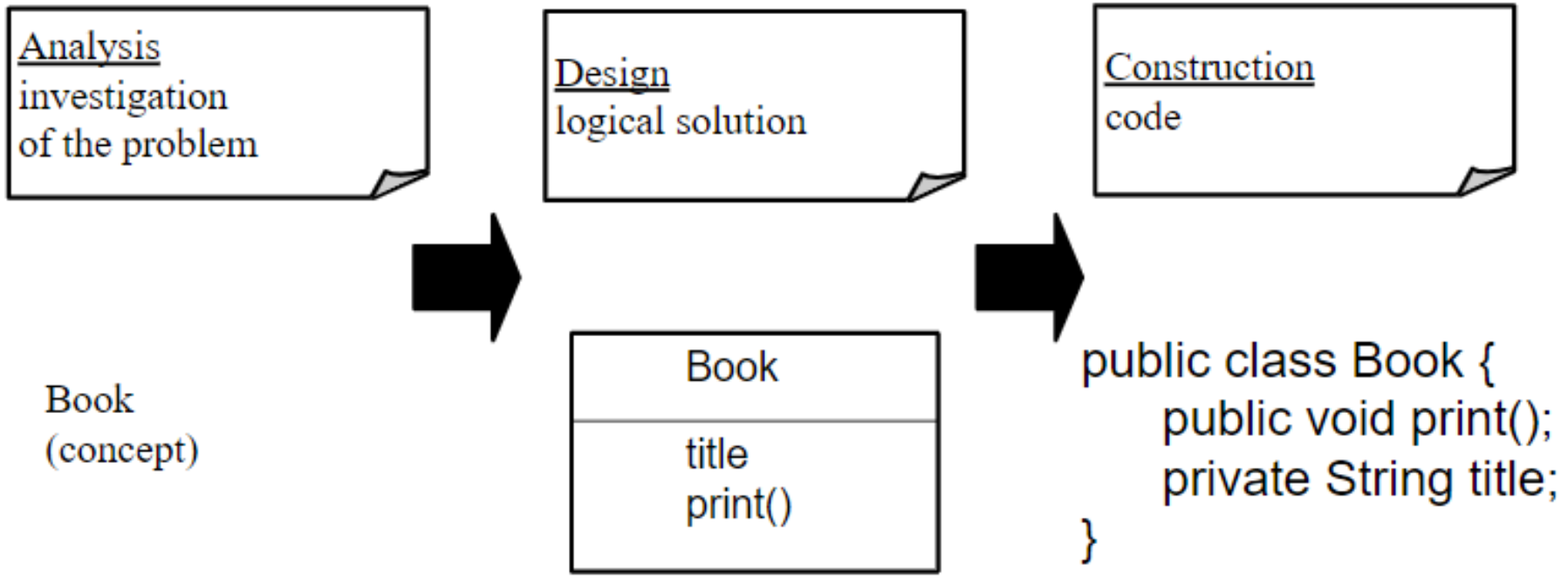
Transition Characteristics of the software :

a) Interoperability : Interoperability is the ability of software to exchange information with other applications and make use of information transparently.

b) Reusability : If we are able to use the software code with some modifications for different purpose then we call software to be reusable.

c) Portability : The ability of software to perform same functions across all environments and platforms, demonstrate its portability.

From Design of Implementation



Reading(s)

Chapter 3, Basic concepts, Software Architecture: Foundations, Theory, and Practice by *Eric Dashofy, Nenad Medvidović, and Richard N Taylor.*

That’s it