CMPG224 – Software Engineering

SU7– Software Design

04/09/2024

Outline

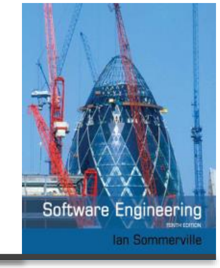
- Introduction to software design
- Principles of software design
- Software design process
- Architectural design
- Detailed design
- Software design tools

Software design

- Software design
  - the process of defining the architecture, components, interfaces, data structures, algorithms, and other characteristics of a system or component
  - serves as the detailed plan or blueprint for software system, laying out how the software will function and how it will be implemented
    - impacting maintainability, scalability, performance, and user experience.

- Importance
  - provides a clear plan, reducing complexity and guiding developers
  - ensures the software meets requirements and is maintainable.
  - helps identify potential issues early, saving time and resources

Good Software Design

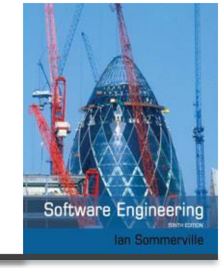


- Correctness - ensure the software meets all specified requirements.
- Efficiency - optimize resource usage, including time and memory.
- Maintainability - well-designed software facilitate easy updates and modifications as it is organized and modular, allowing changes to be made without affecting other parts of the system.
- Reusability - a well-designed system promotes code reuse, saving time and effort in future projects.
- Scalability - good design accommodates future growth and changes in requirements, making it easier to scale and adapt the software
- Performance - good design improves the performance of the software and make it easier to debug and test.
- Customer satisfaction - a well-designed software system is more likely to meet user expectations, leading to higher client satisfaction.

High-level vs Detailed design

- High-level design (HLD) - also known as system or architectural design focuses on the overall system architecture and structure.
  - It defines major components, modules, and their interactions to meet functional and non-functional requirements.
  - Key aspects include system architecture, major components, data flow, module relationships, external interfaces.
  - Example - designing the architecture of a web application with client-server interactions, databases, and APIs.
- Detailed or Low-level design (LLD) - focuses on the internal structure of individual components and modules, specifying the logic, algorithms, and data structures to be used.
  - Key aspects include class diagrams, function definitions, database schemas, algorithms, and data structure specifics.
  - Example - specifying the logic of a user authentication module, including database queries and encryption algorithms.

Software design concepts



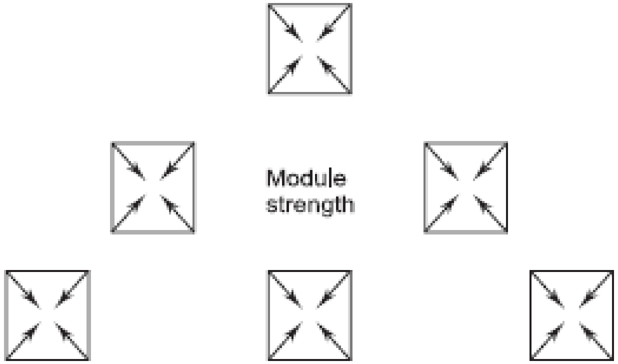
- Abstraction
  - Simplifies complex systems by focusing on essential properties (functionality) appropriate to the problem and hiding unnecessary (implementation) details
  - Example - In a banking system, abstracting the concept of an "Account" with properties like balance and methods like deposit and withdraw

.

- Modularity
  - Breaking down a software system into smaller, manageable modules or components.
    - each module should have a well-defined interface and perform a specific function.
    - increase flexibility, reusability, and maintainability
  - Example:
    - a banking system can consist of different modules for account management, transaction processing, security, user interface, etc.
    - In a web application, separating the user interface, business logic, and data access layers into distinct modules

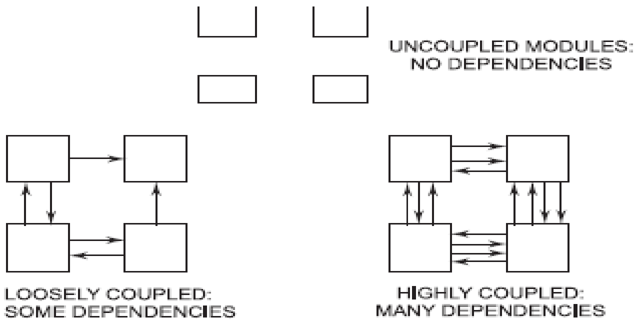
Software design concepts

- Encapsulation
  - Bundling the data and methods that operate on the data within one unit, and restricting access to some of the object's components
  - Example - in an OOP, a class encapsulates data (attributes) and methods (functions) that manipulate the data, hiding the internal state from outside interference.
- Cohesion
  - degree to which elements of module or component belong together in terms of functionality.
  - High cohesion indicates that the elements within a module serve a single task - improves maintainability, readability, and reusability



Software design concepts

- Coupling
  - Degree of interdependence between software modules or components
  - Low coupling minimizes the impact of changes in one module on others - implies that modules are loosely connected and interact minimally with each other
    - reduce complexity, increase modularity, and improve maintainability of software design
  - High coupling makes a design difficult to understand and maintain, increases development effort and making it difficult to implement and debug.



An important design objective is to maximize the module cohesion and minimize the module coupling

  - Example: In a music player app, modules could include Playback Control, Playlist Management, and User Interface.
    - Each module has high cohesion (handling a single responsibility) and low coupling (interacting with each other through defined interfaces).

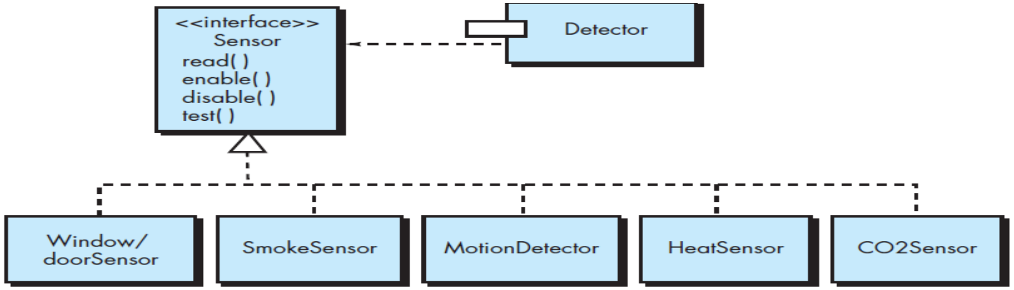
Software design principles

- Software design principles help in creating software that is maintainable, scalable, and robust
  - create designs that are more open to change and to reduce the propagation of side effects when changes do occur.
  - guide developers in writing clean, efficient, and modular code.
- These principles include:
  - SOLID Principles
  - KISS (Keep It Simple, Stupid) principles
  - DRY (Don't Repeat Yourself) principles
  - YAGNI (You Aren’t Gonna Need It) principles

Software design principles –SOLID Principles

- Single Responsibility Principle (SRP) - a class should have only one reason to change - it should have only one job or responsibility (high cohesion)
  - Example: A UserAuthentication class should only handles user login, while a separate UserProfile class manages user profile information.
  - A class that handles database operations should not also be responsible for rendering the UI.
  - It ensures that changes in user data management do not affect user operations.

- Open/Closed Principle (OCP) - software entities (classes, modules, functions) should be open for extension but closed for modification.
  - Example: a Shape class with a method draw() can be extended by creating subclasses like Circle and Square that implement the draw() method, without modifying the original Shape class.
  - Better create a new class that extends or implements the existing one to add new functionality than modifying it

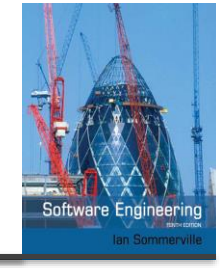


Software design principles

- Liskov Substitution Principle (LSP) - objects of a superclass should be replaceable with objects of a subclass without affecting the correctness of the program.
  - Example: If Bird is a superclass and Penguin is a subclass, the Penguin class should be able to replace the Bird class without causing errors, even if Penguin cannot fly.

- Interface Segregation Principle (ISP) - clients should not be forced to depend on interfaces they do not use.
  - related methods should be collected into separate interfaces rather than mixing unrelated methods
  - It promotes smaller, focused interfaces.
  - Example: Instead of having a large Animal interface with methods like fly(), swim(), and walk(), create smaller, specific interfaces like Flyable, Swimmable, and Walkable that can be implemented by relevant classes.

Software design principles



- Dependency Inversion Principle (DIP) - high-level modules should not depend on low-level modules.
  - Both should depend on abstractions
  - Abstractions should not depend on details
  - Details should depend on abstractions.

  - Example: Instead of a Database class directly depending on a MySQLDatabase class, both should depend on an interface IDatabase.
    - This allows for easy swapping of database implementations.

Design pattern

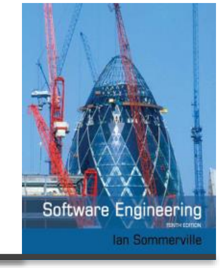
- Design patterns – proven or reusable solutions to common problems that occur in software design.
  - provide templates for how to solve problems that can be used in many different situations.
    - Specify the pattern name, the problem, solution and consequences
  - capture best practices and provide a way to communicate and document design decisions

- Design patterns are important because they lead to more reusable, modular, maintainable, scalable, flexible, and communicative code.
  - Use software engineers, enabling them to build robust and efficient software systems.

Common Design Pattern – OO design

- Creational patterns
  - focus on object creation mechanisms, providing ways to create objects in a flexible, decoupled, and reusable manner
  - Examples: Singleton, Factory Method, Abstract Factory, Builder, Prototype.
- Structural patterns
  - deal with the composition of classes and objects, providing ways to form larger structures while keeping them flexible and efficient.
  - Examples: Adapter pattern, Composite pattern, Decorator pattern, and Facade pattern.
- Behavioural patterns
  - focus on communication between objects and the assignment of responsibilities, providing ways to define how objects interact and behave.
  - Examples: Observer pattern, Strategy pattern, Template Method pattern, and Command pattern.

Common Design Pattern – Examples



- Singleton pattern - ensures a class has only one instance and provides a global point of access to it.
  - Example - a Logger class that manages logging across an application can be a Singleton to avoid multiple instances and ensure consistent logging.

```
public class Singleton {
    private static Singleton instance;

    private Singleton() {}

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

- Factory pattern - provides an interface for creating objects but lets subclasses alter the type of objects that will be created.
  - Example - in a game, a CharacterFactory class might create various types of characters (Warrior, Mage, etc) based on input without specifying the exact class.

Common Design Pattern – Factory pattern


```
// Product interface
interface Shape {
    void draw();
}
 // Concrete Products
class Circle implements Shape {
    public void draw() {
        System.out.println("Drawing a Circle");
    }}
 class Rectangle implements Shape {
    public void draw() {
        System.out.println("Drawing a
Rectangle");
    }}
```

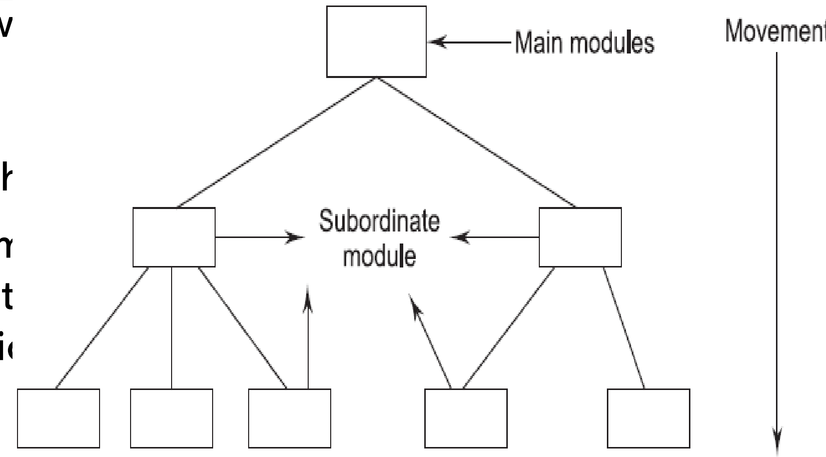
```
// Factory class
class ShapeFactory {
    public Shape getShape(String shapeType)
{
        if (shapeType == null) {
            return null;
        }
        if
(shapeType.equalsIgnoreCase("CIRCLE")) {
            return new Circle();
        } else if
(shapeType.equalsIgnoreCase("RECTANGLE"))
{
            return new Rectangle();
        }
        return null;
    }
}
```

Common Design Pattern – examples

- Observer pattern - defines a one-to-many dependency between objects so that when one object changes state, all its dependents are notified automatically
  - Example - in a stock trading application, when the price of a stock changes (subject), all registered users (observers) are notified.
- Decorator pattern - adds behaviour to objects dynamically
  - Example - in a coffee shop ordering system, a Coffee class can be decorated with additional options like Milk, Sugar, or Whipped Cream

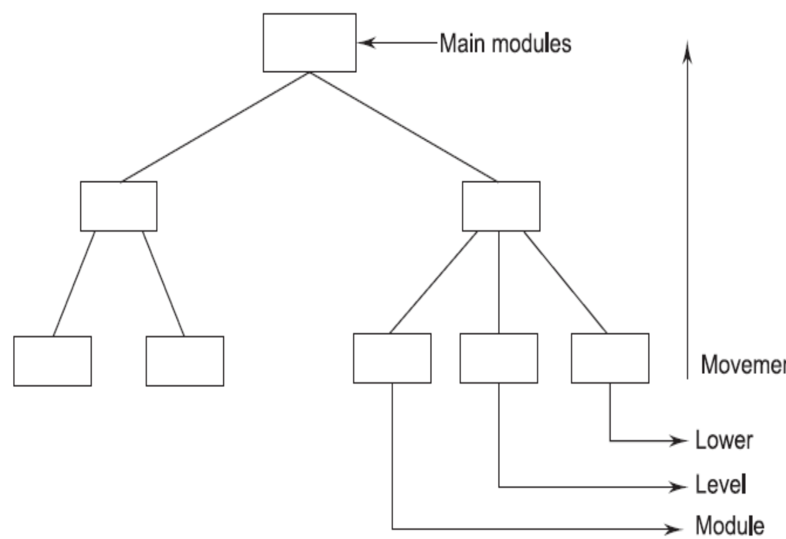
Software Design Methodologies

- Top-down design - starts with a high-level overview and breaks down into more detailed components (data format, algorithms, etc)
  - suitable only if the system develop from scratch
- Example - designing a hospital management system from the main modules like Patient Management, Staff Management, and Billing down to specific functionalities.



(Movement in top-down approach)

- Bottom-up design - begins with designing low-level components and integrates them into higher-level modules.
  - suitable for a system to be built from an existing system
- Example - developing encryption algorithms first and then integrating them into a broader security system.



(Movement in bottom-up approach)

- Best practice is to use hybrid design (top-down and bottom-up)

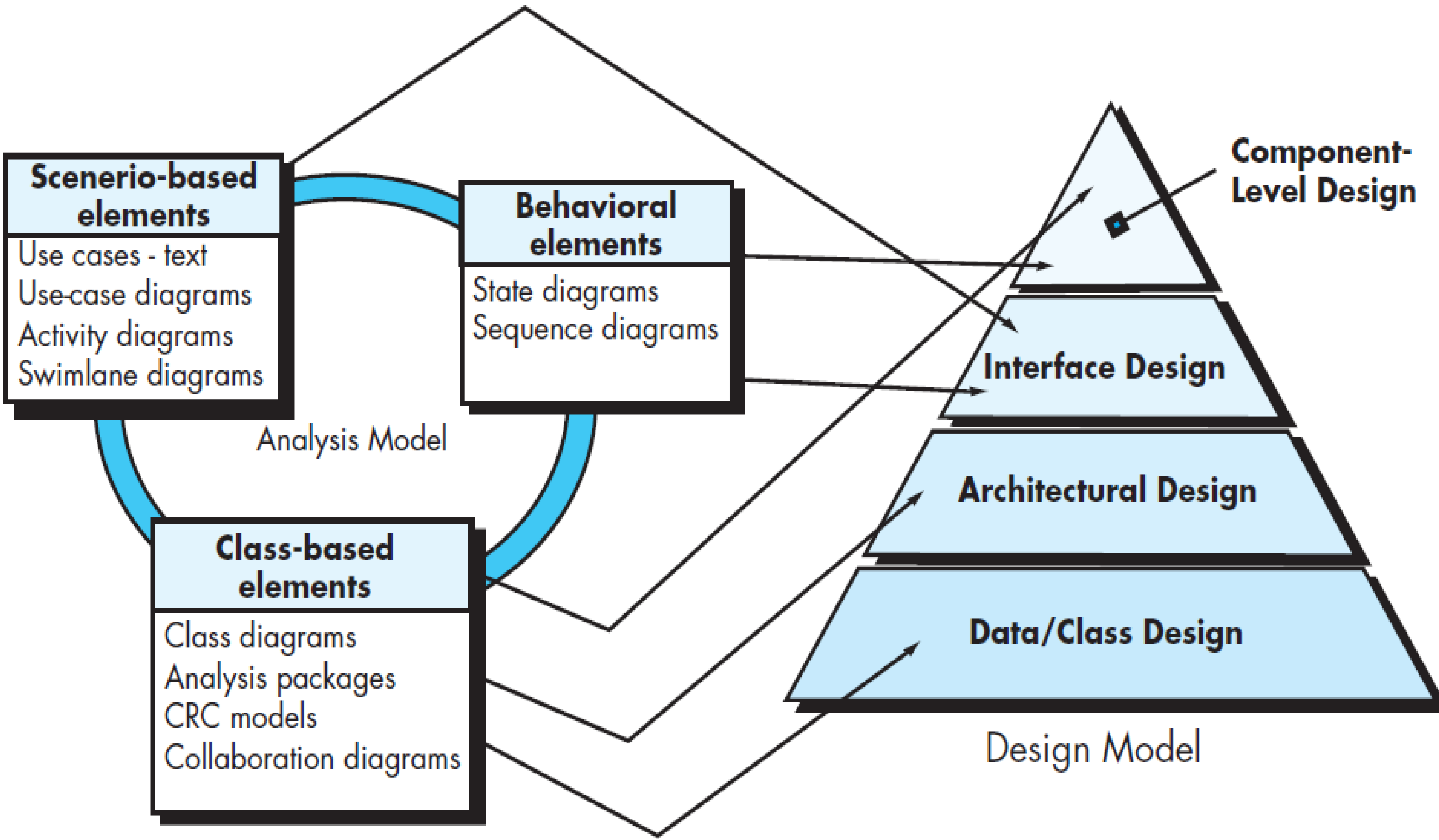
Software Design Methodologies

- Object-Oriented Design (OOD) - focuses on designing software using objects that represent real-world entities, encapsulating data and behaviour.

- key concepts
  - Classes and objects
  - Encapsulation
  - Inheritance
  - Polymorphism - ability to present the same interface for different underlying forms (data types).

- Example - in a shopping cart system, objects could be Customer, Cart, Item, each with their own properties and methods (e.g., addItem`, calculateTotal, checkout, makePayment, etc).

Software Design Models vs UML Diagrams



Software Design Models

- Architectural design - defines the high-level structure of a system, specifying major components and their interactions to meet system-wide requirements.

- Component-level design - focuses on the internal structure and functionality of individual software components, ensuring they perform their specific tasks effectively.

- Data design - involves designing the system's data structures, databases, and the flow of data between system components to ensure data integrity and accessibility.

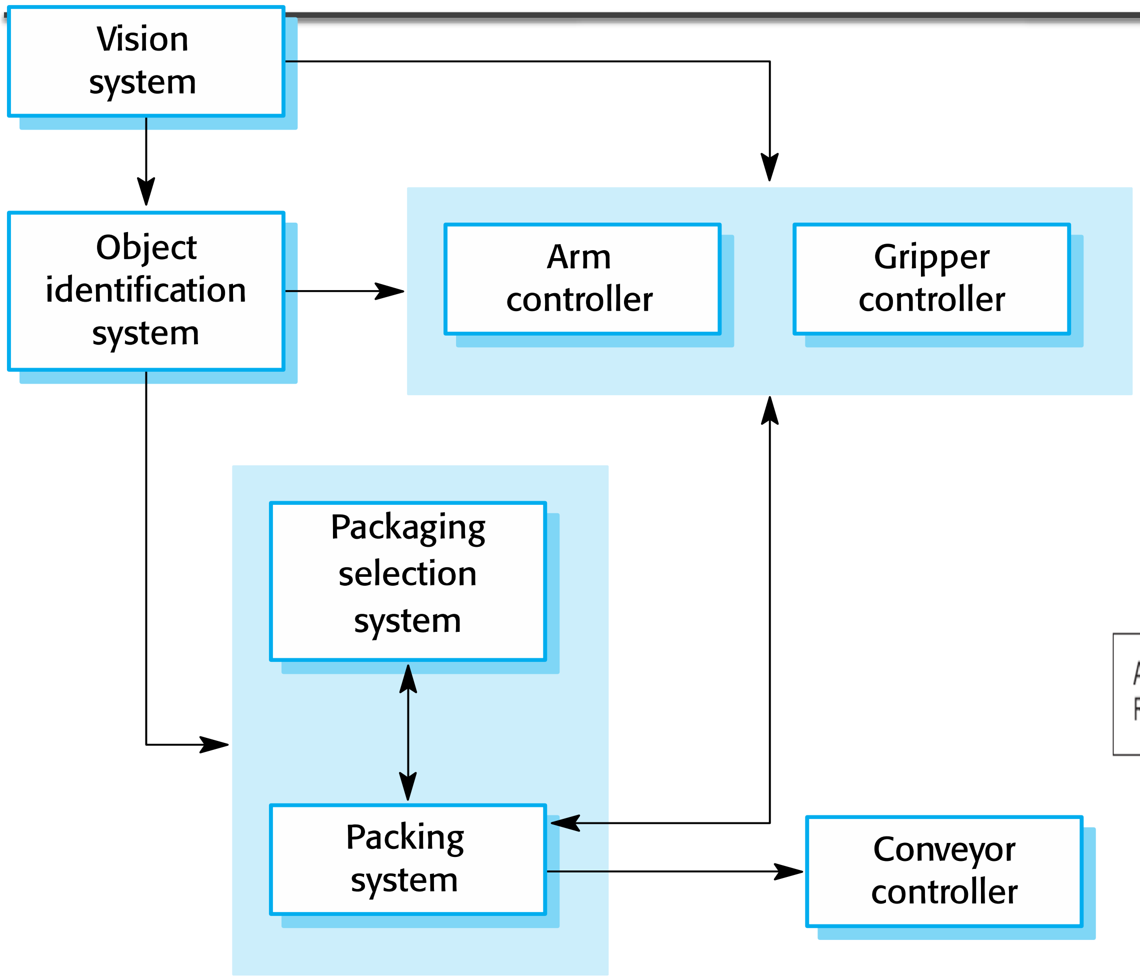
- Interface design - defines how components and systems interact with each other and with users, focusing on communication protocols, user experience, and input/output flows.

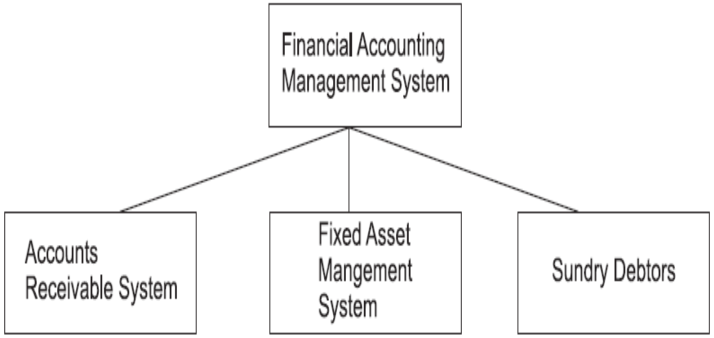
Phases of the design process – Architectural design

- Architectural design - defines the overall high-level structure of the software, including the organization of its components, their relationships, and how they interact.
  - a blueprint for the system, guiding the development process and ensuring that the system meets its requirements.
    - Involve selecting architectural styles and patterns, components, and defining their relationships to meet the system's requirements, and constraints that affect the way in which architecture can be implemented

- Architectural design process output is an architectural model which is derived from sources such as:
  - information about the application domain,
  - specific requirements model elements such as use cases or analysis classes, their relationships and collaborations for the problem at hand; and
  - availability of architectural patterns and styles

Examples of software architectures





packing robot control system

Importance of Software Architecture

- Provides a clear framework for developers, reducing complexity
- Ensures the system is scalable, maintainable, and meets performance requirements.
- Facilitates communication among stakeholders by providing a common understanding of the system.
- Helps identify and mitigate risks early in the development process.

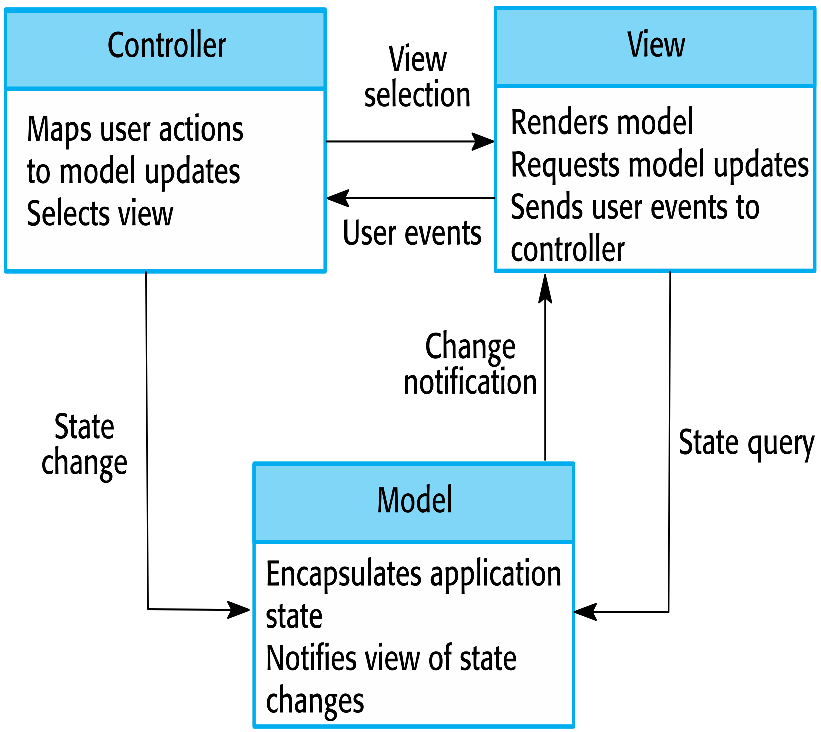
Architectural patterns and styles

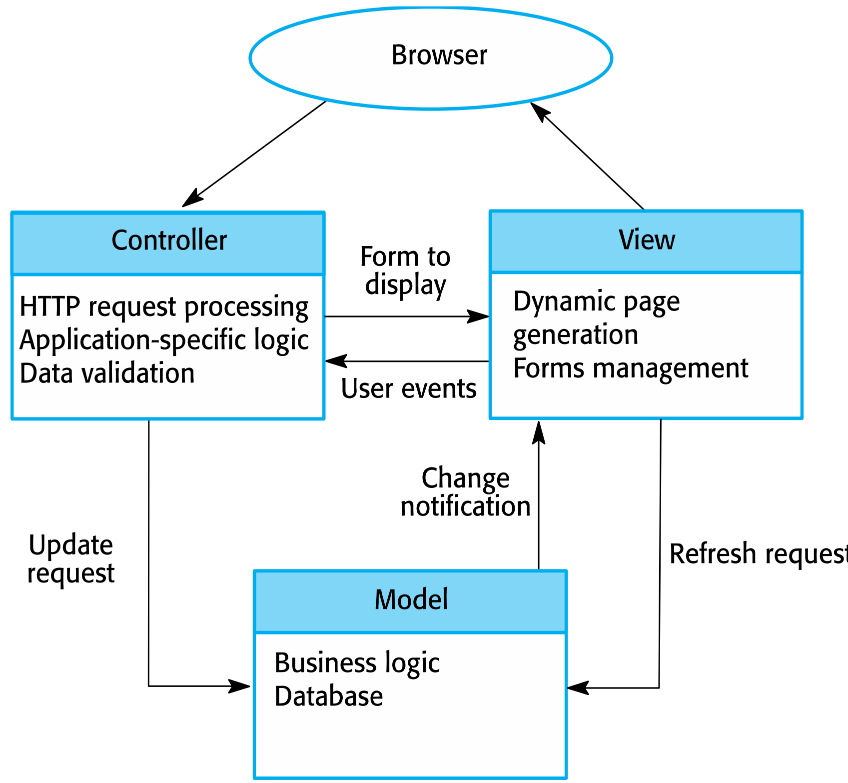
- Architectural patterns
  - a general, reusable solution to a commonly occurring problem in software architecture within a given context
    - provides a template for how to solve a specific problem but does not specify concrete implementation details.
  - Is more about best practices and provide guidelines for structuring and organizing software components.
  - Examples: Model-View-Controller(MVC), Repository, Event-driven application, Microservices architecture, etc
- Architectural styles
  - provides a set of principles for organizing software elements, their interactions, and constraints
  - Is more about the overall structure and organization rather than solving a specific problem
  - Examples - Layered architecture, Client-server architecture, Pipe-and-Filter architecture, etc

Common Architectural Patterns

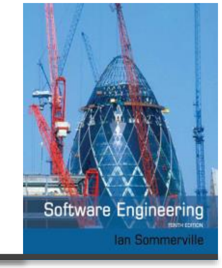
- Model-View-Controller (MVC)
  - Separates an application into three interconnected components (Model, View, and Controller) to separate internal representations of information from how it is presented and accepted by the user.
    - Model - manages data and business logic
    - View - manages the display of data (user interface)
    - Controller - handles user input, manages interactions and updates the Model or View accordingly.
  - It promote modularity, maintainability, and scalability
  - suitable for web, mobile, desktop and enterprise applications, game, e-commerce system, real-time application, etc,
- Example -In a web application, the:
  - Model could represent the database
  - View could be the web page that displays data, and
  - Controller could handle user requests and update the Model and View accordingly

Common architectural pattern - MVC

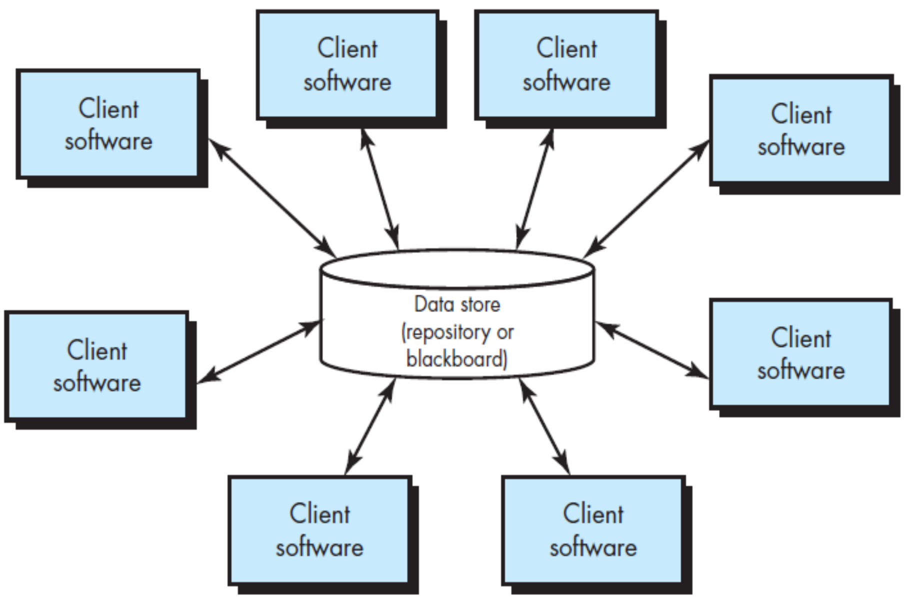


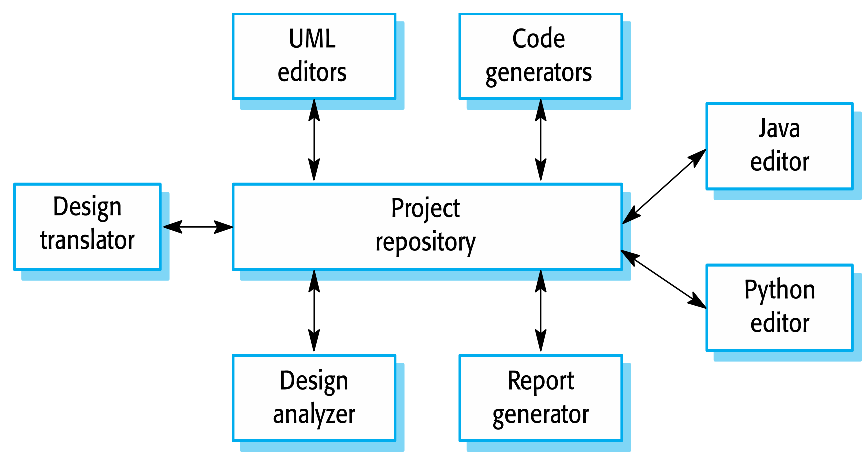


Common architectural pattern - Repository



- Repository pattern
  - Centralizes data access logic, providing a clean separation between the data access and business/application logic layers.
  - benefits include data consistency, maintainability, and security, making it a valuable choice for systems where data access and persistence are critical.
  - suitable for systems such as content management systems, e-commerce platforms, IDE, etc.





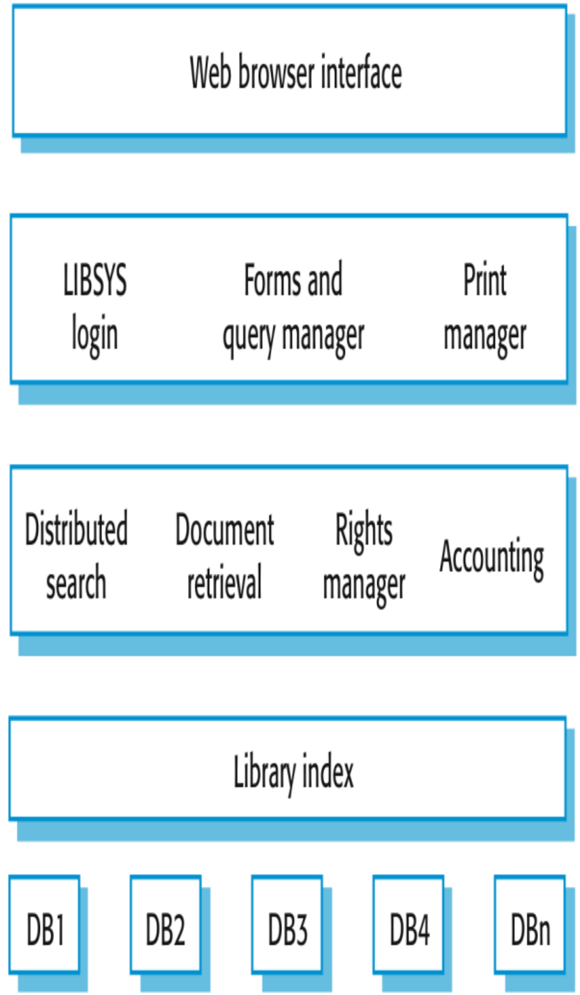
an IDE

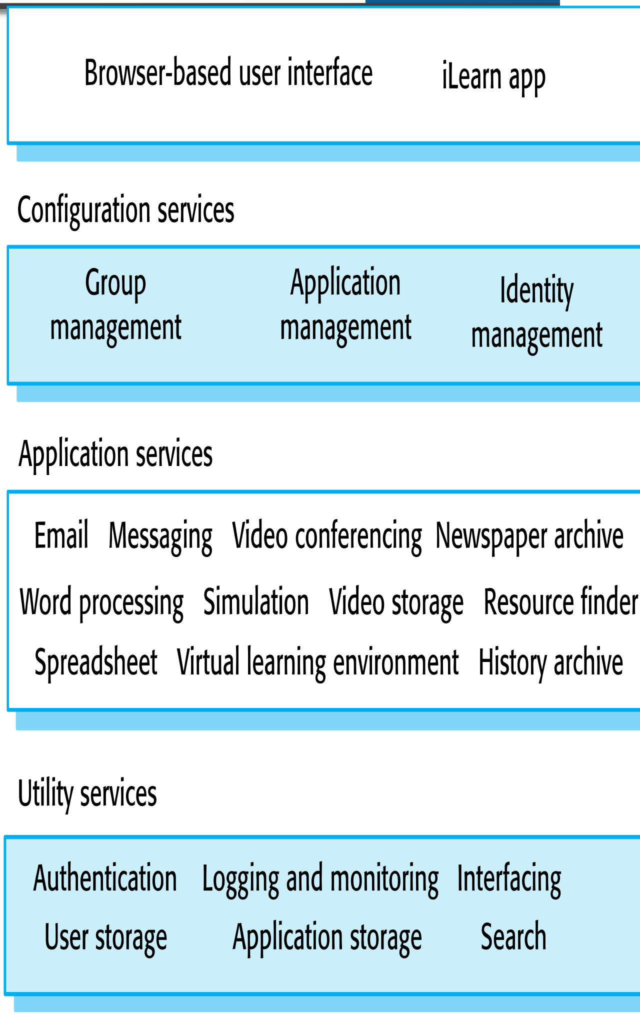
Common architectural styles

- Layered architecture
  - Organizes a system into layers, each with a specific role, such as presentation, business logic, and data access.
    - A layer provides services to the layer above it so the lowest-level layers represent core services that are likely to be used throughout the system
  - It is suitable for information systems including web and enterprise applications, health, financial, e-commerce and educational systems, content management, etc.
- Example
  - A banking application might have a presentation layer for the user interface, a business logic layer for transaction processing, and a data access layer for database interactions.
  - Web applications often have layers like User Interface (UI), Application Logic, and Database.
  - Enterprise Applications may use more layers, such as Presentation, Application, Business, and Data layers.

A generic layered architecture style

e.g. layered architecture





e.g. library and iLearn system architecture

Common architectural styles

- Client-Server architecture
  - Divides the system into two main components: the client (user interface) and the server (business logic and data).
    - Clients request services from the server and the
    - Server processes the requests and returns results, while the
    - Network which allows clients to access servers

- Suitable for systems such as web applications, enterprise systems, file sharing and storage, email server, printer server, game servers, distributed system, etc,
- Examples
  - A web-based email service where the email client (browser) communicates with the email server to send, receive, and manage messages.
  - Web applications where Browsers (clients) send requests to web servers (server) for resources.

client–server architecture for a film library

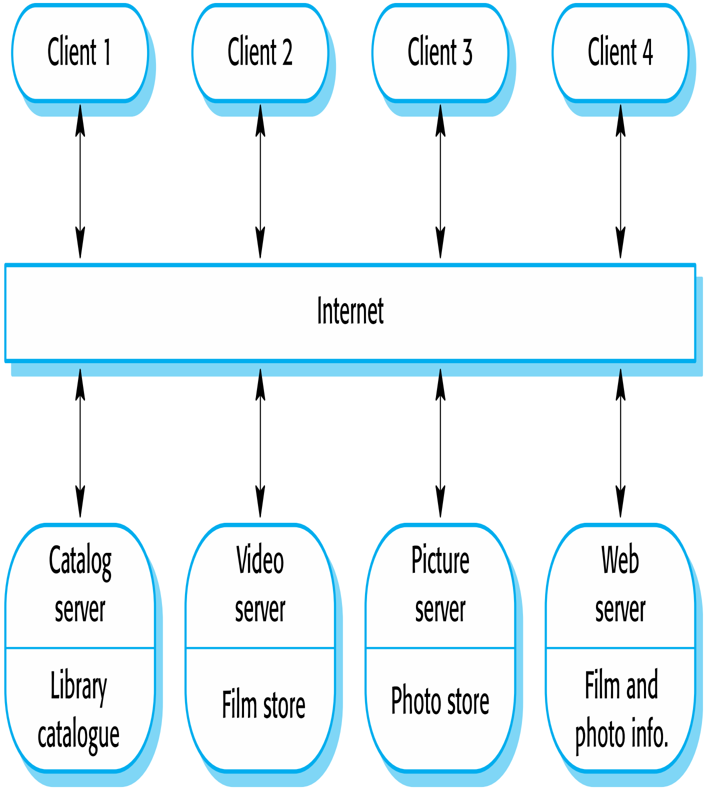


Table 16.2: Architectural Styles, Nature of Computations, and Quality Attributes

Architecture	Nature of computations	Quality attributes
Data Flow	Well-defined input and output. Sequential transformation of input.	Integratability, Reusability
Batch-sequential	Single output operation on a single collection of input. Sequential processing of input.	Reusability Modifiability
Pipe-and-filter	Transformation of continuous streams of data. Simultaneous transformation of available data elements.	Scalability Response to input element before the whole stream of data becomes available.
Call-and-Return	Fixed order of computation	Modifiability, Integratability, Reusability
Object-oriented	Computations restricted to fixed number of operations for each element of a set of entities.	Reusability, Modifiability
Layered	Division of computational tasks between application-specific and platform-specific layers.	Portability, Reusability
Independent-Process	Independent computations on a network of computer systems.	Modifiability Performance
Communicating	Message passing as an interaction mechanism.	Modifiability, Performance
Event-base implicit invocation	Computations triggered by a collection of events.	Flexibility, Scalability, Modifiability
Agent	Computations performed by interacting information processing systems.	Reusability, Performance, Modularity
Virtual Machine		Portability
Interpreter	Computation on data controlled by internal state.	Portability
Intelligent system	Both cognitive and reactive forms of computation.	Portability
CHAM	Computations mimic laws of chemistry.	Portability
Repository	Computation on highly structured data. Order of computation governed by query requests.	Scalability Modifiability
Reuse library	Computation on passive data acquisition, storage, change of forms, and retrieval.	Scalability Modifiability
Blackboard	Computation on active data control.	Scalability, Modifiability

Questions?