Software Design: Introduction and Key Concepts

Software design is the process of conceptualizing, planning, and refining the structure and behavior of a software system based on end-user requirements before actual implementation begins. This phase transforms requirements-often documented in a Software Requirements Specification (SRS)-into a blueprint for developers, ensuring that the system will meet user needs, be maintainable, and support future changes.

Key Aspects of Software Design:

- The design process results in both documented and conceptual models of how the software will function.
- It involves problem-solving, planning, and modeling various aspects of the system before coding starts.
- Software design must be robust, adaptable to change requests, and maintainable throughout the development lifecycle.

Approaches to Software Design

Two primary approaches are used to break down and organize software design:

- **Top-Down Approach:** The design process starts with the overall system architecture, which is then decomposed into smaller, more detailed components. This approach emphasizes planning and a comprehensive understanding of the system before coding begins. Testing of individual components is delayed until significant design is complete.
- **Bottom-Up Approach:** The process begins with designing and implementing the most basic or foundational components first. These are then integrated to form larger subsystems and, eventually, the complete system. This approach allows for early coding and testing but may risk integration challenges if high-level design is not considered from the start. Code reusability is a major benefit.

Modern software design often combines both approaches to leverage their respective strengths-planning from the top down while reusing and integrating pre-existing modules from the bottom up.

Software Design Techniques

Several techniques are employed to create effective software designs:

- **Prototyping:** Building early versions of the system to clarify requirements and design choices.
- **Structural Models:** Defining the system's structure, often using data flow diagrams or module decomposition.
- **Object-Oriented Design (OOD):** Organizing the system as a collection of interacting objects, each encapsulating data and behavior. OOD uses principles like abstraction, encapsulation, modularization, and hierarchy, and is often visualized using UML diagrams.
- **Entity-Relationship Models:** Used primarily for database design, illustrating how data entities relate to each other.

- **Refactoring:** Iteratively improving the design to accommodate new features or resolve design issues that arise as the system evolves.

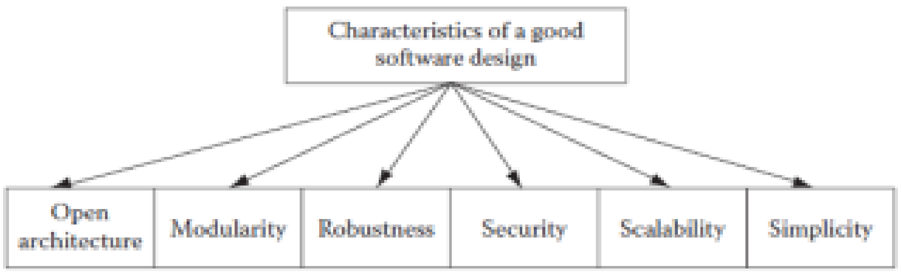
Design Principles and Concepts

Effective software design is guided by several core principles:

- **Abstraction:** Focusing on essential features while omitting unnecessary details.
- **Refinement:** Breaking down high-level functions into more detailed steps.
- **Modularity:** Dividing the system into distinct modules or components.
- **Software Architecture:** Defining the overall structure and ensuring conceptual integrity.
- **Control Hierarchy:** Organizing program components in a hierarchical structure.
- **Information Hiding:** Restricting access to the internal details of modules.
- **Data Structures and Procedures:** Designing the logical relationships among data and the processes for manipulating them.

Key Fundamentals of Good Software Design:

- **Strong Foundation and Structure:** The initial design must be sound and scalable to accommodate future growth, similar to how a building's foundation supports additional floors or features. If the software's structure is weak, adding new features can destabilize the system.
- **Incremental Development:** Modern software is often developed incrementally, starting with a minimal feature set and expanding over time. This approach requires a flexible and extensible design from the outset to avoid instability during growth.
- **Refactoring:** In agile and incremental development, refactoring is a technique used to reorganize and improve the design without altering its external behavior. This helps maintain stability and adaptability as new features are added.

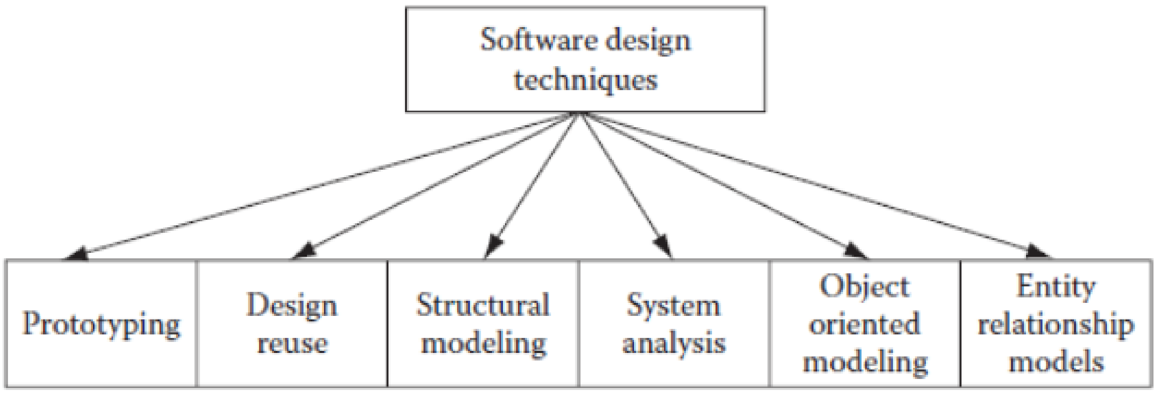


- **Open architecture:** Allows easy integration and future enhancements.
- **Modularity:** Divides the system into independent, manageable components.
- **Robustness:** Ensures the system can handle errors and unexpected situations.
- **Security:** Protects the system and data from unauthorized access.
- **Scalability:** Supports growth in users, features, or data without major changes.
- **Simplicity:** Keeps the design easy to understand, maintain, and extend.

Software Design Techniques:

High-Level vs. Low-Level Design

Aspect	High-Level Design (HLD)	Low-Level Design (LLD)
Abstraction	System architecture and structure	Detailed implementation of components
Focus	Components, interactions, data flow	Functions, algorithms, data structures
Deliverables	Architecture diagrams, flowcharts	Class diagrams, pseudo-code, database schemas
Purpose	Blueprint for development	Guide for actual coding and integration
Example Techniques	Prototyping, structural models, OOD, SOA	Database modeling, detailed application logic



1.Prototypes

Prototyping is a software design approach where early, simplified versions (prototypes) of an application are built to visualize and test ideas before full-scale development.

Advantages of Prototyping

- **Fast and Cost-Effective:** Prototyping is quick and inexpensive, allowing teams to test concepts without heavy investment.
- **Early Customer Buy-In:** Demonstrating a prototype helps secure customer approval and clarifies requirements early in the project.
- **Reduces Miscommunication:** Prototypes provide a visual reference, helping to resolve misunderstandings between customers and developers at an early stage.
- **Quick Feedback and Iteration:** Customers can interact with the prototype and provide feedback, enabling rapid improvements and reducing the risk of missing key features.

- **Error and Cost Reduction:** Early detection of issues through prototyping saves time and money by preventing costly changes later in development.
- **Higher Satisfaction and Engagement:** Clients are more satisfied and engaged when they can see and influence the product early.

Limitations of Prototyping

- **Customer Expectations:** Clients may mistakenly believe the prototype is a near-complete product, leading to unrealistic expectations about development timelines.
- **Limited Scope:** Prototypes mainly showcase user interfaces and basic flows; they usually do not represent complex internal logic or backend processes.
- **Not a Substitute for Full Development:** While useful for visualizing and refining requirements, prototypes cannot replace the need for thorough development and testing.

2.Design Reuse

Design reuse in software development refers to leveraging existing design components, modules, or services in new projects or within different parts of the same project. This practice reduces redundancy, increases efficiency, and enhances software quality.

Types of Design Reuse

- **Internal Design Reuse:** Within large software products, common design elements are standardized and reused across multiple modules. This reduces design effort and ensures consistency throughout the system.
- **Open Source and External Reuse:** With the rise of open source, developers can directly use or adapt existing designs from external sources, saving significant time and resources.
- **Service-Oriented Architecture (SOA):** SOA enables true reuse by allowing applications to consume services provided by others without copying or modifying the underlying design. Developers simply integrate with published service interfaces, using the service as-is.

Benefits of Design Reuse

- **Efficiency and Speed:** Reusing proven components accelerates development and reduces the need to design from scratch.
- **Reliability:** Reused components are often well-tested, leading to more dependable software.
- **Consistency:** Standardized components ensure consistent design and functionality across the product.
- **Cost Savings:** Less time and effort are spent on design, reducing overall project costs.
- **Simplified Maintenance:** Updates or bug fixes in a reused component benefit all instances where it is used, making maintenance easier.

SOA and the Future of Design Reuse

SOA represents a major shift in design reuse. Instead of copying or modifying code, developers integrate existing services by consuming them through published interfaces. This model offers:

- No need to purchase or own the component-just use the service.
- Full documentation and integration details are provided by the service owner.
- The service is used as-is, promoting loose coupling and flexibility.

SOA and similar paradigms are transforming software development by making reuse more practical, scalable, and efficient, fundamentally changing how applications are built and maintained.

3.Structural Models

Structural models in software engineering visually represent how a system is organized in terms of its components and the relationships between them. These models help break down complex applications into manageable parts, making development, maintenance, and team collaboration more efficient.

- **Hierarchy of Components:** At the lowest level are functions and procedures, which are grouped into classes or packages. Multiple classes form a component, components build modules, and modules together make up the complete application.
- **Packages and Blocks:** Packages act as containers (like folders) that group related elements, including other packages, classes, or components. The fundamental unit of structure is often called a "Block," which can represent anything from a subsystem to a physical or logical part of the system.
- **Structural Analysis:** Breaking an application into parts is achieved through structural analysis. Starting from requirement specifications, features are identified and then decomposed into smaller sets that fit into different modules. This decomposition is captured in structural models, such as class diagrams, package diagrams, and block diagrams.
- **Visualization and Documentation:** Structural models serve as blueprints, helping stakeholders understand system architecture, allocate development tasks, and ensure the system’s structure aligns with requirements. They also provide valuable documentation for ongoing maintenance and future enhancements.

Benefits

- **Clarity:** Makes complex systems understandable by showing how parts fit together.
- **Maintainability:** Facilitates updates and debugging by isolating changes to specific components or modules.
- **Team Collaboration:** Enables division of work among teams or developers, each responsible for different modules or components.

Structural models are fundamental for designing robust, scalable, and maintainable software systems by providing a clear, organized view of the system’s architecture and its constituent parts.

4. Object-Oriented Design (OOD)

Object-oriented design (OOD) is a way to plan and build software by modeling real-world entities as objects. Each object represents a business entity (like a Customer or Order) and has properties and behaviors similar to the real thing.

- **Objects and Classes:** Objects are created from classes. A class is like a blueprint (e.g., Car), and objects are actual items (e.g., a specific car). Classes can have child classes that inherit properties from parent classes, and can also add their own features.
- **Inheritance:** Child classes reuse and extend the behavior of parent classes, making it easy to represent groups of similar things (e.g., Vehicle → Car, Bike).
- **Real-World Alignment:** This approach matches real-world scenarios, making it easier to model business processes in software.
- **Input Sources:** OOD uses information from use cases, activity diagrams, and user interfaces to design the system.

5.Systems Analysis

Systems analysis is the process of studying business needs and breaking them down into detailed requirements to determine if and how they can be addressed by a software system. The main goal is to understand the problem, identify objectives, and specify what the system should do so it can support users in their routine business tasks.

Key Steps in Systems Analysis

- **Understanding Business Needs:** Analyze the business scenario to see if it can be effectively supported by a software application. For example, enabling customers to access bank accounts online requires analyzing all steps, objects, and interactions involved.
- **Decomposition:** Break down the overall business process into smaller components or activities, such as user actions, website functions, and backend system interactions.
- **Requirement Gathering:** Collect facts and requirements from stakeholders using interviews, questionnaires, and observations to ensure all user needs and system constraints are understood.
- **Modeling and Documentation:** Create models (like use cases or process diagrams) that describe user activities, system objects, and their interactions. This helps visualize how the system will work and ensures all requirements are covered.
- **Feasibility and Analysis:** Assess whether the proposed system is practical and cost-effective, and analyze potential solutions to select the best approach.

6. Entity Relationship Models

Entity Relationship Models (ER Models) are a fundamental technique in software and database design used to visually represent business entities and the relationships between them.

- **Entities:** Entities are objects or concepts (like Customer, Order, Product) about which data needs to be stored. In ER diagrams, entities are usually depicted as rectangles.

- **Attributes:** Each entity has attributes (properties or characteristics), such as a Customer having a Name or ID. Attributes are often shown as ovals connected to their entity.
- **Relationships:** Relationships illustrate how entities are associated with one another (e.g., a Customer places an Order). These are represented by lines connecting entities, sometimes with diamonds to label the type of relationship.
- **Cardinality:** ER diagrams also specify cardinality (one-to-one, one-to-many, many-to-many) to define how many instances of one entity relate to another.

Uses in Database Design

- **Database Structure:** ER models help determine what tables are needed, how they are linked, and what data each table will hold.
- **Normalization:** By mapping entities and relationships, ER diagrams help avoid data redundancy and ensure a clean, organized database structure.
- **Implementation:** Each entity typically becomes a table in a relational database, with attributes as columns and relationships implemented through foreign keys.

Visual and Practical Benefits

- **Clarity:** ER diagrams provide a clear visual representation, making it easier for both technical and non-technical stakeholders to understand the data model.
- **Foundation for Development:** They serve as blueprints for building databases and are essential in the early stages of system design.
- **Object Correlation:** In modern development, objects in object-oriented models can be directly mapped to entities in ER diagrams, aligning software objects with database tables for seamless design and implementation.

Architectural Design

- Architectural Design is about deciding **how the software system is structured**.
- Think of it like planning a building before construction.
- It ensures the parts of the software (called components) work well together.
- It focuses on both **what the system does** and **how well it performs** (speed, security, flexibility).

Why Software Architecture is Important

- Software architecture is like a **blueprint**.
- It helps:
  1. Check if the design meets goals.
  2. Try different designs early.
  3. Lower risks before coding starts.
- It also improves **communication** between team members (designers, developers, managers).

IEEE Software Architecture Standard (IEEE-1471)

- This standard helps describe software architecture clearly.
- It suggests showing **multiple views** of architecture for different people:

- Developer's view
- User's view
- Manager's view

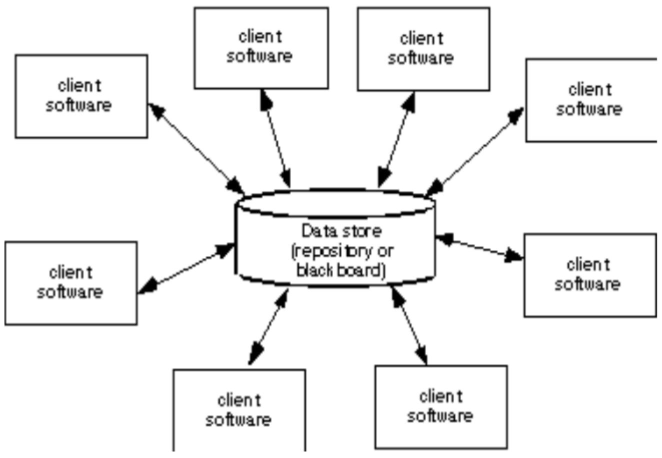
Architectural Genres and Styles

- **Genre**: Type of software (e.g., banking software, embedded software).
- **Style**: Common structure used in a system.
  - Has components, connectors, rules, and models.
  - Helps understand how the system works as a whole.

Types of Architectural Styles

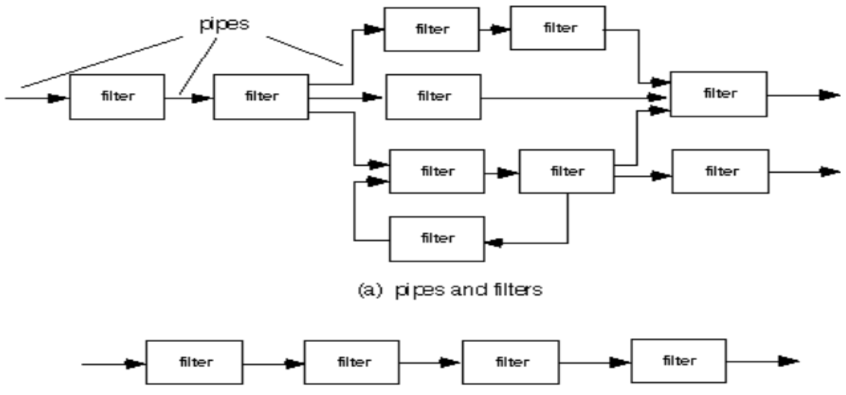
1. Data-Centered Architecture:

- Central database or file.
- All parts of the system read/write data to/from this center.
- Ex: Banking system with a central customer database.



2. Data Flow Architecture:

- Data moves in **stages**, like an assembly line.
- Each stage changes the data a bit.
- Ex: Video file gets compressed -> encoded -> uploaded.

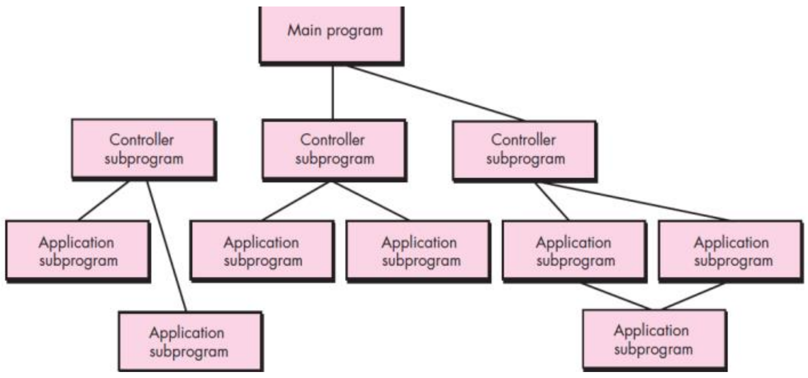


(a) pipes and filters

(b) batch sequential

3. Call and Return Architecture:

- Traditional programming style.
- Main program calls sub-programs.
- Easy to understand and modify.

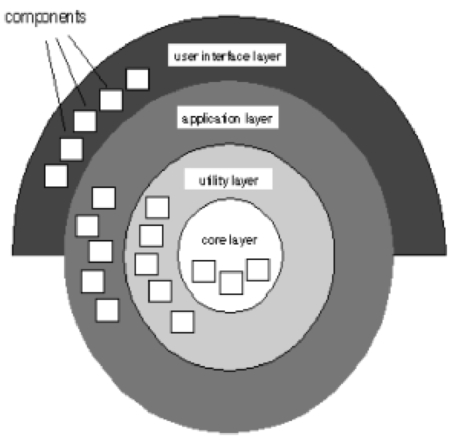


4. Object-Oriented Architecture:

- Based on objects that represent real-world things.
- Each object has data + functions.
- Communicates using messages.

5. Layered Architecture:

- Software is split into layers.
- Each layer does a specific job.
- Ex:
  - Top layer – UI
  - Middle layer – Logic
  - Bottom layer – Database



Architectural Patterns

- Common **patterns** solve repeating problems.

Some Patterns:

1. **Concurrency**: Handles many tasks at once.
2. **Persistence**: Saves data even after app closes.
3. **Distribution**: Software parts work over a network (like client-server apps).
4. **Broker Pattern**: Middle-man between parts (used in distributed systems).

Representing Architecture with Diagrams

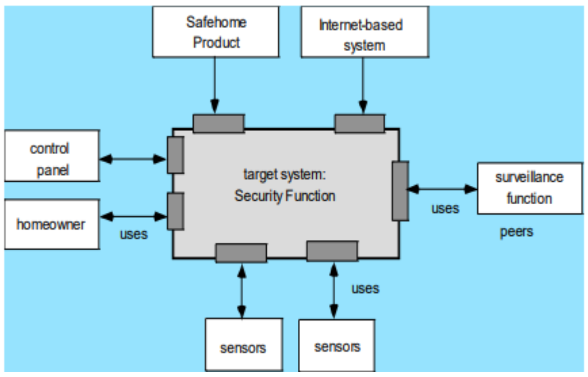
a) Architectural Context Diagram (ACD):

Shows how your software interacts with outside systems, people, and devices.

At the architectural design level, a software architect uses an architectural context diagram (ACD) to model the manner in which software interacts with entities external to its boundaries.

1. Superordinate systems—those systems that use the target system as part of some higher-level processing scheme.
2. Subordinate systems—those systems that are used by the target system and provide data or processing that are necessary to complete target system functionality.
3. Peer-level systems—those systems that interact on a peer-to-peer basis (i.e., information is either produced or consumed by the peers and the target system.
4. Actors—entities (people, devices) that interact with the target system by producing or consuming information that is necessary for requisite processing.

The below diagram illustrates Architectural context diagram for the SafeHome security function.



b) Archetypes:

- Basic building blocks or **abstract roles** in the system.
- In a **home security system**:
  - **Node** = input/output devices (e.g., sensors)
  - **Detector** = devices that sense motion/fire
  - **Indicator** = alarm lights/sounds
  - **Controller** = manages turning alarm on/off

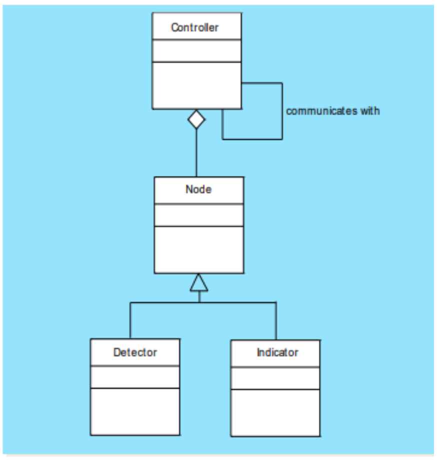


Figure 10.7 UML relationships for SafeHome security function archetypes

Refining into Components

- The system is broken into **components**.
- Example: SafeHome Security System
  - **Detector Management**: handles sensor data
  - **Alarm Processing**: handles alarms
  - **Control Panel**: handles user input
- Some parts are **application components** (business logic).
- Others are **infrastructure components** (supporting tasks like data storage).

Analyzing Architectural Design

Steps:

1. List real-world **scenarios**.
2. Collect **requirements** and system environment info.
3. Choose the best **architecture styles** (layered, client-server, etc.).
4. Test for quality like speed, security.
5. See how quality is affected by your design choices.
6. Compare different design options.

Complexity and ADL

- More connections = more complexity.
- Use **ADL (Architectural Description Language)** to:
  - Break big design into parts.
  - Combine parts together.
  - Describe how parts talk to each other.

Partitioning the Architecture

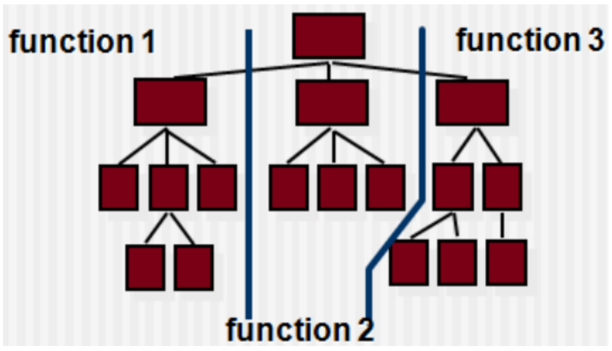
Why partition the system?

- Makes software **easier to test, maintain, and extend**.
- Reduces bugs and unwanted effects from code changes.

Types of Partitioning:

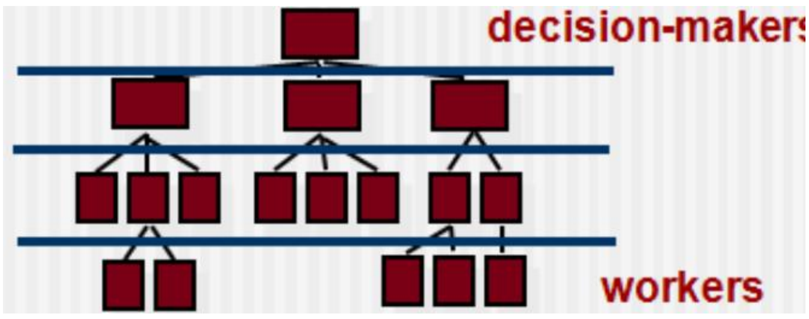
1. **Horizontal Partitioning**:

- Divide system by functions.
- Example: One branch handles UI, another handles logic.
- Use control modules to manage these branches.



2. **Vertical Partitioning (Factoring)**:

- Divide system by **decision-making** levels.
- Top-level modules make decisions.
- Bottom-level modules do the work.



Software Design Methods: Top Down and Bottom Up:

1. Top-Down Design Approach

- Start designing from the **top/main structure** of the software.
- Break it down step by step into smaller components and features.

How it works:

1. Design the **overall system structure** first.
2. Then, design the **main components** of the system.
3. Finally, design the **details and features** inside each component.

Example: Imagine designing a car**:**

- First, design the whole car (overall layout).

- Then design the engine, seats, wheels (main components).
- Finally, design the small parts like spark plugs, seat cushions (details).

Benefits:

- Non-functional things like **security, performance, usability** are handled early.
- Ensures a **consistent and unified design**.
- Components are reusable, so it's **easier to maintain and extend**.
- Works well with **Waterfall Model** projects.

Drawbacks:

- **Risky**, because everything must be planned from the start.
- Hard to adjust later if something was missed early.
- Not suitable for projects that need flexibility.

2. Bottom-Up Design Approach

- Start designing from the **smallest parts or functions** first.
- Then build up to design **middle components**, and finally the **overall system**.

How it works:

1. Design **small functions or modules** first.
2. Combine them to make **bigger components**.
3. Finally, connect everything to form the **complete system**.

Example: Think of building a **house**: First build bricks, then walls, then combine them to make the full house.

Benefits:

- Allows **incremental development** — you can keep adding new parts.
- **Easier to make changes** as the design grows.
- Works well with **Agile and Iterative** development methods.
- Better if not all information is available at the beginning.

Why popular today:

- Agile projects don’t plan the entire design at once.
- Each sprint or iteration creates a **small working part**.
- **Refactoring** is used later to clean and improve the growing design.

Refactoring

- Refactoring is the process of improving the internal structure of software code **without changing its external behavior**. It makes the code cleaner, more maintainable, and easier to scale.
- Think of it as reorganizing the internal layout of a system while keeping its output and functionality the same.

- Even with careful planning, software often becomes difficult to extend or maintain as new features are added. This is especially true in long-term or Agile projects where the software evolves over time.
- Refactoring helps to: Improve the code structure, Reduce complexity, Make the system easier to understand and maintain

When to Refactor?

1. **Duplicate code** – Same logic appears in multiple places.
2. **Long methods** – Functions or methods that are too large and hard to follow.
3. **Large number of call parameters** – Too many inputs make functions complex and error-prone.
4. **Message chaining** – Calling one method that calls another, forming a chain that is hard to trace.
5. **Classes with many concepts** – A single class doing too many unrelated things.
6. **Large class size** – Classes becoming too big and handling too much, which makes them difficult to manage.

How Refactoring Helps

- Reduces **coupling** (dependency) between classes
- Increases **cohesion** (focused responsibility within a class)
- Encourages code **reuse** by creating modular and independent classes
- Makes future changes or scaling up easier

In Agile, complete software design is not done upfront. Instead, software is built in iterations. As a result, the design must adapt over time. Refactoring allows the design to be adjusted during each iteration to maintain code quality.

Module Coupling

Module coupling refers to the level of **dependency between different modules or classes** in a software system. When one module depends heavily on others, they are said to be "tightly coupled."

Why is High Coupling a Problem?

- As the software product **grows** and more code is added, the **connections between modules** increase.
- When **one module is changed**, it can affect many other modules, leading to **more bugs and errors**.
- High coupling makes the system **hard to maintain, update, and test**.

How to Reduce Coupling?

- **Reduce the number of direct calls** between modules or classes.
- Keep modules more **independent and self-contained**.
- Apply **Service-Oriented Architecture (SOA)**:

    - In SOA, each module or service works **independently** and communicates through **well-defined interfaces**.
    - This is called **loose coupling** and it improves flexibility and reliability.

User Interface Design (UI Design)

UI Design is about **how the user interacts with the software** — what the software looks like, how easy it is to use, and how users complete tasks.

Golden Rules of UI Design:

1. Place the User in Control:

- Don't force actions.
- Allow flexible, undoable actions.
- Hide technical details from the user.
- Make actions feel natural and direct.

2. Reduce User's Memory Load:

- Use meaningful defaults.
- Provide shortcuts.
- Use real-world visual design (e.g., icons).
- Don't show too much info at once.

3. Make the Interface Consistent:

- Similar tasks should look and behave the same.
- Maintain design across the app.
- Follow user expectations from past experience.

User Interface Design Models:

1. **User Model** – Who are the users?
2. **Design Model** – How the system should appear and behave.
3. **Mental Model** – What users think the system does.
4. **Implementation Model** – The actual design of screens and behaviors.

UI Design Process:

1. Interface Analysis

- Who are the users? (age, job, skills)
- What tasks will they perform?
- What content will be shown?
- What environment will the software be used in?

2. User Analysis Questions:

- Are users experienced or new?

- Are they trained?
- What is their language?
- How important is correctness?
- Do they work full-time or part-time?

3.**Task Analysis and Modeling:**

- What work will the user do?
- What are the tasks and subtasks?
- What objects are involved?
- What is the task sequence?
- Use **use cases** and **workflow diagrams** (like swimlane diagrams) to describe this.

4.Display Content Analysis:

- Is the content placed consistently?
- Can users customize it?
- Are large reports broken into easy parts?
- Is the content readable and understandable?
- Are color and warnings used properly?

Interface Design Steps:

1. Define interface objects and actions.
2. Identify events (user actions) that change the interface.
3. Show each interface state visually.
4. Design how users will understand the system state.

Design issues include: Response time, Help and error messages, Menu labelling, Accessibility, Internationalization

Other UI Design Principles:

- **Anticipation** – Predict what users might want to do next.
- **Communication** – Show status of user actions.
- **Consistency** – Use same design across the app.
- **Controlled autonomy** – Let users move freely but follow structure.
- **Efficiency** – Focus on making user tasks faster.
- **Focus** – Keep screens relevant to the current task.
- **Fitts's Law** – Bigger buttons = faster clicking.
- **Learnability** – Easy to learn and remember.
- **Latency Reduction** – Let users continue while the app processes.
- **Readability** – Easy to read for all users.
- **Track state** – Save user progress.
- **Visible Navigation** – Show where users are and where they can go.

UI Design Workflow:

1. Review analysis results.
2. Sketch the layout.

3. Map user goals to actions.
4. Create storyboards.
5. Refine layout using design feedback.
6. Identify UI elements (buttons, forms).
7. Describe user behavior and flow.
8. Design screens for each interface state.

Aesthetic Design Tips:

- Use white space wisely.
- Focus on content.
- Place key info top-left to bottom-right.
- Avoid too much scrolling.
- Make sure layout fits different screen sizes.

Design Evaluation Cycle

1. **Design**
    - Create an initial version of the user interface.
    - This includes layout, controls, workflows, and visual design.
2. **Prototype**
    - Build a basic, working version of the interface (even if it's just for demonstration).
    - This helps users and designers understand how the system will look and behave.
3. **Evaluate**
    - Test the design with real or representative users.
    - Collect feedback through usability testing, observation, or questionnaires.
4. **Analyze**
    - Review the feedback from users.
    - Identify any **problems, confusion, or areas for improvement**.
5. **Refine**
    - Make necessary changes to the interface based on the feedback.
    - Improve layout, labels, navigation, or interactions.
6. **Repeat**- Go through the cycle again until the UI is easy, clear, and user-friendly.

Pattern Oriented Design

Pattern Oriented Design uses **design patterns**—reusable solutions to common design problems. These patterns help in creating software that is **efficient, scalable, and easier to maintain**.

Why Use Patterns?

- Patterns **capture expert knowledge**.
- They help developers **solve problems faster** by reusing proven solutions.
- Patterns can be used across **different projects and domains**.

Types of Patterns in Requirement Modeling:

1. Analysis Patterns

- Describe **reusable solutions** for common requirements.
- Help convert **real-world business needs** into software models.
- Discovered during requirement gathering by analyzing **use cases**.

2. Semantic Analysis Patterns (SAPs)

- A type of analysis pattern that comes from a set of related use cases.
- Help in creating models that can be reused across different systems.

Use Case Example:

- A car has a **rear-view camera** and a **proximity sensor**.
- When the car is in reverse:
  - Camera shows the rear view on screen.
  - Sensor checks for objects behind.
  - If an object is too close, the **brake is applied automatically**.

This system has:

- **Sensors** – to detect distance
- **Actuators** – to apply the brake

Actuator-Sensor Pattern

Intent: Define how different types of sensors and actuators **interact with a control unit.**

Motivation:

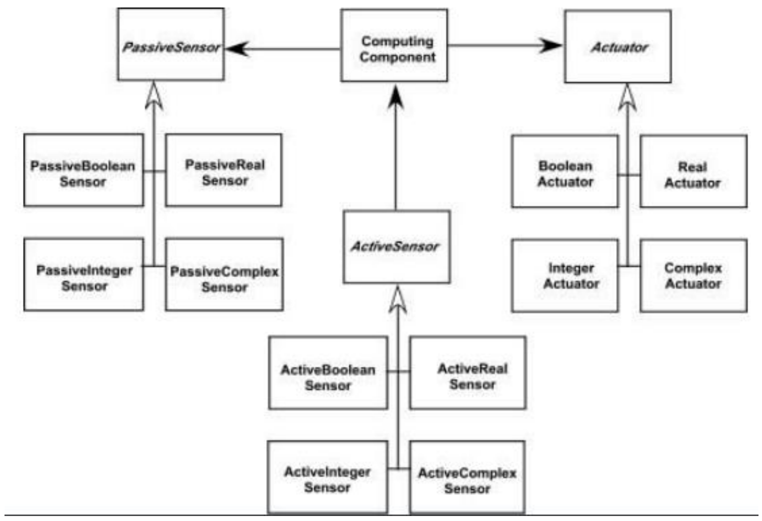
- In embedded systems (like cars, home automation), there are many sensors (input) and actuators (output).
- This pattern **organizes** and **standardizes** how to use them.

Structure (UML Class Diagram):

- Abstract classes:
  - **Sensor**
  - **Actuator**
- Types:
  - Boolean, Integer, Real (basic sensors/actuators)
  - Complex (like radar, advanced types)

Each has:

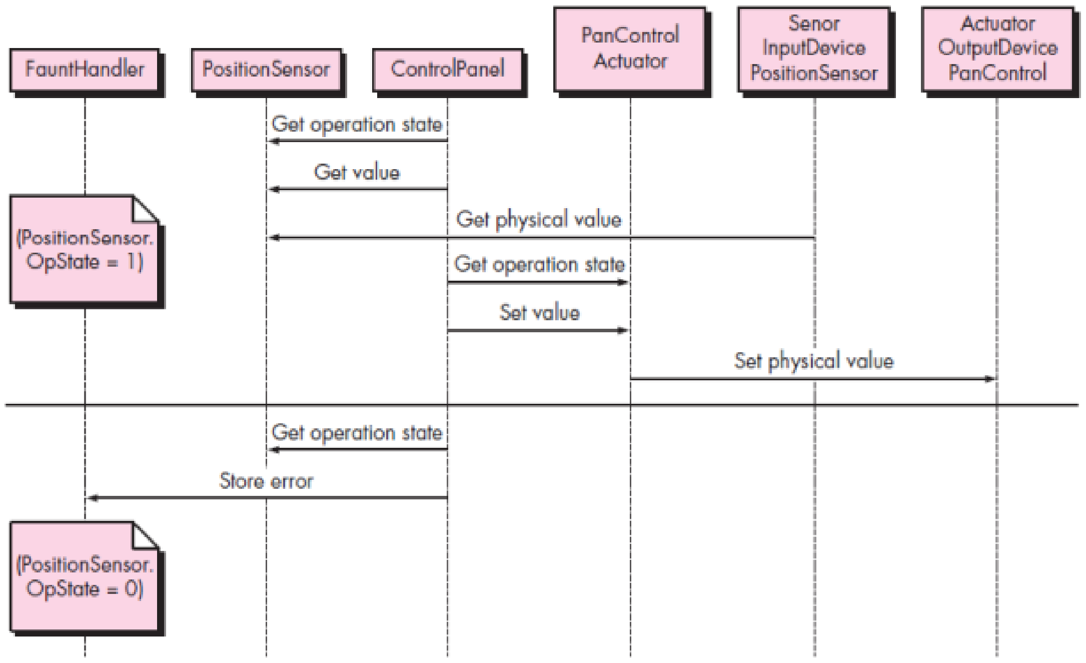
- Methods to read or write values
- Functions to check operational state
- Status reporting features



Behavior (Sequence Diagram):

- The **control panel** checks sensors and actuators.
- If a **fault** is found (e.g., sensor not working), it is reported to a **Fault Handler**.
- The **Computing Component** processes sensor data and sends commands to actuators.

The figure below presents a UML sequence diagram for an example of the Actuator-Sensor Pattern as it might be applied for the SafeHome function that controls the positioning (e.g., pan, zoom) of a security camera.



Applicability: Useful in systems with multiple sensors and actuators**:**

- Home security systems
- Cars
- Smart devices

Web Application (WebApp) Design

Designing a WebApp involves **planning how it looks, works, and behaves** for users interacting with it through a web browser.

Main Parts of WebApp Design:

1. Content Analysis

- Identify **all types of content**: text, images, videos, audio, etc.
- Use **data modeling** to define each data object clearly.

2. Interaction Analysis

- Describe **how users interact** with the WebApp.
- Use **use cases** to show detailed user actions and system responses.

3. Functional Analysis

- Define what **functions and operations** the WebApp must perform.
- These are based on **use cases** and include both content-related and system-level processing.

4. Configuration Analysis

- Describe the **technical environment**:
    - What kind of server and OS will be used
    - How the server and browser interact
    - What communication protocols are needed

When is Analysis Needed?

Explicit analysis is important when:

- The WebApp is **large or complex**
- There are **many users or developers**
- The WebApp is **critical** to the business

Content Model

- Extract **content objects** from use cases.
- Define **attributes** of each object.
- Show relationships using:
    - **ER diagrams**
    - **Data trees** (hierarchy of content)

Example: Data Tree

- Shows the **structure of information** in a WebApp.
- Shaded boxes = content objects (e.g., user profile)

- Unshaded boxes = data values (e.g., user name, email)

Interaction Model

Includes:

1. **Use Cases** – user actions and system responses
2. **Sequence Diagrams** – how objects interact in order
3. **State Diagrams** – how the WebApp changes states
4. **UI Prototype** – a sample screen showing layout and controls

Functional Model

- Shows two things:
  - What users **see and use**
  - What internal operations are needed
- Often represented using **Activity Diagrams**

Configuration Model

- **Server-Side:** Define hardware, OS, and interoperability. Specify communication protocols and interfaces.
- **Client-Side:** Define browser setup, screen sizes, and testing needs.

Navigation Modeling

Ask design questions like:

- Which pages or elements are most important?
- Should some content be easier to reach?
- How are navigation **errors** handled?
- Should users be able to **track their path** through the site?
- Should **search or menus** be used for navigation?

Navigation Design Tips:

- Show a full menu/map for easy navigation.
- Adjust navigation based on user actions.
- Consider storing the user's navigation history.
- Design navigation based on:
  - **User behavior**
  - **Element importance**
- Handle **external links** smartly (new tab, same window, etc.)

Concurrent Engineering in Software Design

Concurrent Engineering **means using** information from earlier stages **(like design) to start some work for later stages (like development)** at the same time**, instead of waiting.**

Normally, you cannot start development **until** design is complete**.**

But some decisions, like:

- Which **programming language** to use,
- How to **divide the software** for teams,
- Planning for **maintenance and support**
  can be made during the design stage itself.

Benefits:

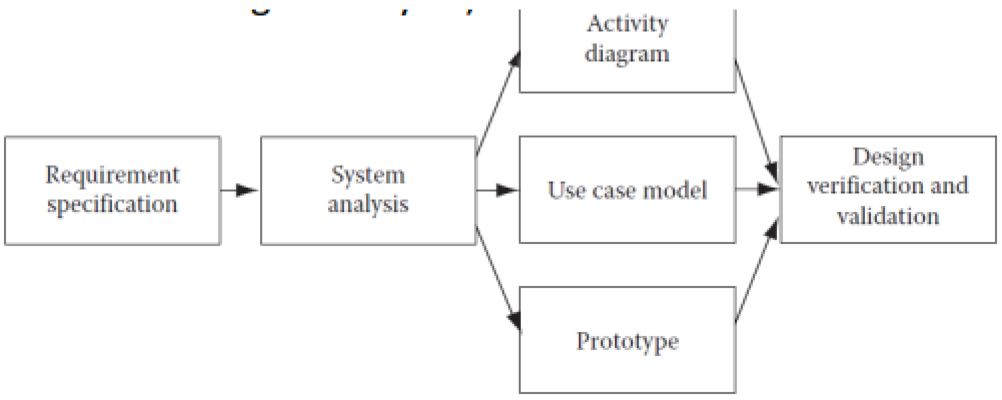
- Saves time by allowing **parallel work**.
- Helps teams **prepare in advance**.
- Reduces risks and **delays** in later stages.
- Ensures smoother progress through the project.

Design Life-Cycle Management

This is the **step-by-step process** of converting software requirements into a proper **software design**.

Steps involved:

1. **System Analysis**:
   - Study the requirements.
   - Check if converting them into a software design is **possible** (feasibility).
2. **Design Creation**:
   - Based on the analysis, create the **actual design**.
   - Use tools like **activity diagrams**, **use cases**, **prototypes**, etc.
3. **Verification and Validation**:
   - Once the design is ready, it is **reviewed** for correctness and completeness.
   - If the design passes the review, it is **approved** for the development phase.


Requirement specification
System analysis
Activity diagram
Use case model
Prototype
Design verification and validation