Slide 3

background of component-level design:

1. **When It Happens**:
   - Component-level design takes place after the first round of architectural design.
2. **Purpose**:
   - It aims to create a detailed design model based on earlier analysis and architectural models.
3. **Potential for Errors**:
   - The transition from design to implementation can introduce subtle errors that are hard to detect and fix later.
4. **Prevention Philosophy**:
   - Effective programmers should focus on not introducing bugs in the first place, rather than spending time debugging later. This principle is attributed to Edsgar Dijkstra.
5. **Representation**:
   - Component-level designs can be shown using intermediate representations like diagrams, tables, or text. These representations help in translating the design into actual source code.
6. **Guidelines**:
   - The design of data structures, interfaces, and algorithms should follow well-established guidelines to minimize errors.

==========

Slide 5

What is a Software Component?

- **Modular Building Block**: A software component is a self-contained part of a larger software system.
- **Deployable and Replaceable**: It can be independently deployed, replaced, or updated without affecting the entire system.
- **Encapsulates Implementation**: It hides its internal workings and only exposes certain interfaces for interaction.

Communication and Collaboration

- **With Other Components**: Components interact with each other within the system.
- **With External Entities**: Components can also communicate with elements outside the system (like databases or external services).

Three Views of a Component

1. **Object-Oriented View**:
   - Sees components as objects or classes with properties and methods.
2. **Conventional View**:
   - Sees components as more traditional, structured pieces of software (like functions or procedures).
3. **Process-Related View**:
   - Focuses on how components interact and manage processes within the system.

==========

Slide 6

Object-Oriented View of a Component

- **Collaborating Classes**: A component is seen as a group of classes working together.
- **Elaboration of Classes**:
   - **Problem Domain Classes**: Classes that represent real-world entities relevant to the problem being solved.

    - **Infrastructure Classes**: Classes that support the system's functionality but aren't part of the problem domain.

Steps in the Process:

1. **Identify Attributes and Operations**:
    - Determine all properties (attributes) and functions (operations) each class should have.
2. **Define Interfaces**:
    - Specify how classes will communicate and work together.

Further Steps After Initial Design:

1. **Elaborate on Details**:
    - Add more detail to each attribute, operation, and interface.
2. **Specify Data Structures**:
    - Choose the right data structures for each attribute.
3. **Design Algorithms**:
    - Create detailed algorithms for the operations.
4. **Design Interfaces**:
    - Implement the methods and messaging for objects to interact.

Summary

- **Definition**: Component as a group of collaborating classes.
- **Elaboration**: Identify attributes, operations, and interfaces.
- **Further Detailing**: Specify data structures, design algorithms, and implement interfaces for communication.

==========

Slide 7

Conventional View of a Component

A component is seen as a module in a program that includes:

1. **Processing Logic**: The code that performs the component's tasks.
2. **Internal Data Structures**: The data storage needed to support the processing logic.
3. **Interface**: The method through which the component can be called and data can be passed to it.

Roles of a Component

A component can have one of these roles:

1. **Control Component**:
    - Manages and coordinates the activation of other components.
2. **Problem Domain Component**:
    - Performs a specific function or part of a function required by the user.
3. **Infrastructure Component**:
    - Provides support functions that are essential for the problem domain components to work.

Summary

- **Definition**: Component as a functional module with logic, data, and interface.
- **Roles**: Can be a control component, problem domain component, or infrastructure component.

==========

Slide 8

Conventional View (continued)

In the conventional view, software components are created based on Data Flow Diagrams (DFDs) from the analysis model. Here's how this process works:

1. **Mapping DFDs to Modules**:
    - Each "transform bubble" (module) from the DFD is converted into a hierarchy of modules.
    - **Control components** are placed at the top.
    - **Problem domain components** and **infrastructure components** are placed toward the bottom.
2. **Goal of Functional Independence**:
    - Aim to ensure that each module operates independently as much as possible.
3. **Steps for Each Transform**:
    - **Define the Interface**:
        - Specify the order, number, and types of parameters for the transform.
    - **Define Internal Data Structures**:
        - Identify the data structures used within the transform.
    - **Design the Algorithm**:
        - Develop the algorithm using a stepwise refinement approach (breaking down the process into smaller, more manageable steps).

Summary

- **Derived from DFDs**: Components are based on the lowest-level transform bubbles in DFDs.
- **Module Hierarchy**: Control components at the top, problem domain and infrastructure components at the bottom.
- **Functional Independence**: Aim for independent operation of modules.
- **Development Steps**:
    - Define the interface for each transform.
    - Define internal data structures.
    - Design the algorithm using stepwise refinement.

==========

Slide 9

Process-related View

Key Points:

- **Reuse of Existing Components**: Instead of creating each component from scratch, systems are built using pre-existing components from a library.
- **Component Library**: A collection of reusable components designed for easy integration.

Steps:

1. **Formulate Software Architecture**: Plan the structure of the software system.
2. **Select Components**: Choose suitable components from the library to fill the architecture.

Components in the Library:

- **Interface Description**: Detailed information on how to interact with the component.
- **Functions Performed**: A clear list of tasks the component can execute.
- **Communication and Collaboration**: Information on how the component interacts with other components.

Summary:

In the process-related view, the focus is on using ready-made components to build software. These components come with complete documentation on their interfaces, functions, and interaction requirements, making the system development more efficient and standardized.

==========

Slide 11

Component-level Design Principles

1. Open-Closed Principle

- **Definition**: A module or component should be open for extension but closed for modification.
- **Explanation**: Design components so they can be extended with new functionality without altering existing code.
- **Simplified**: You can add new features to a component without changing its existing code.

2. Liskov Substitution Principle

- **Definition**: Subclasses should be substitutable for their base classes.
- **Explanation**: If a component uses a base class, it should work correctly even if a subclass of that base class is used instead.
- **Simplified**: A child class should be able to replace its parent class without breaking the system.

3. Dependency Inversion Principle

- **Definition**: Depend on abstractions (interfaces); do not depend on concretions (concrete implementations).
- **Explanation**: Components should rely on interfaces or abstract classes instead of specific implementations to make the system more flexible and easier to extend.
- **Simplified**: Use interfaces instead of concrete classes to make components more adaptable and easier to change.

4. Interface Segregation Principle

- **Definition**: Many client-specific interfaces are better than one general-purpose interface.
- **Explanation**: Instead of having one large interface with many methods, create multiple smaller, client-specific interfaces. Each interface should only include methods relevant to the particular client category.
- **Simplified**: Create smaller, focused interfaces for different clients instead of one big interface.

==========

Slide 12

Component Packaging Principles

1. Release Reuse Equivalency Principle

- **Definition**: The granularity of reuse is the granularity of release.
- **Explanation**: Reusable classes should be grouped into packages that can be managed, upgraded, and controlled together as new versions are created.
- **Simplified**: Package reusable classes together so they can be easily managed and updated.

2. Common Closure Principle

- **Definition**: Classes that change together belong together.
- **Explanation**: Classes that are likely to change for the same reasons should be packaged together. This ensures that when one class changes, the related classes are also considered.
- **Simplified**: Group classes that are likely to change for the same reason into the same package.

3. Common Reuse Principle

- **Definition**: Classes that aren't reused together should not be grouped together.
- **Explanation**: Avoid packaging classes together if they are not likely to be reused together. This prevents unnecessary integration and testing for classes that haven't changed but are in the same package as those that have.
- **Simplified**: Only package classes together if they are likely to be reused together, to avoid unnecessary updates and testing.

==========

Slide 13

Component-Level Design Guidelines

Components

1. **Naming Conventions**:
    - **Definition**: Establish consistent and clear naming conventions.
    - **Explanation**: Components should have names that reflect their function and are meaningful to all stakeholders.
    - **Simplified**: Name components clearly and meaningfully.
2. **Architectural Component Names**:
    - **Definition**: Names should be derived from the problem domain.
    - **Explanation**: Names like "Calculator" should be easily understood by everyone involved.
    - **Simplified**: Use names from the problem domain that everyone understands.
3. **Infrastructure Component Names**:
    - **Definition**: Names should reflect their implementation-specific role.
    - **Explanation**: Names like "Stack" should indicate the technical function of the component.
    - **Simplified**: Use technical names that describe what the component does.

Dependencies and Inheritance in UML

1. **Modeling Dependencies**:
    - **Definition**: Show dependencies from left to right.
    - **Explanation**: This makes it clear how components depend on each other.
    - **Simplified**: Draw dependencies from left to right in diagrams.
2. **Modeling Inheritance**:
    - **Definition**: Show inheritance from top (base class) to bottom (derived classes).
    - **Explanation**: This clarifies the hierarchy and relationships.
    - **Simplified**: Draw inheritance from top to bottom in diagrams.
3. **Using Interfaces**:
    - **Definition**: Model dependencies as interfaces.
    - **Explanation**: Using interfaces rather than direct dependencies makes the system more flexible.
    - **Simplified**: Use interfaces to show dependencies for better flexibility.

==========

Slide 14-15

Certainly! Cohesion refers to how closely the elements within a module or component are related to each other. Here's a simplified breakdown of the types of cohesion mentioned:

1. **Functional Cohesion**:
    - A module performs a single task or computation and returns a result. This is the highest level of cohesion.
2. **Layer Cohesion**:
    - Higher-layer components access services provided by lower-layer components. This focuses on the interaction between layers of abstraction.
3. **Communicational Cohesion**:
    - Operations that access the same data or share data closely related to each other are grouped within a class. This ensures operations are logically related.
4. **Sequential Cohesion**:
    - Components or operations are grouped in a way that allows one to provide input to the next in a sequence, implementing a sequence of operations.
5. **Procedural Cohesion**:
    - Operations are grouped in a manner where each operation is invoked immediately after the preceding one, even if no data is passed between them.
6. **Temporal Cohesion**:
    - Operations are grouped to perform a specific behavior or establish a particular state, such as during program startup or error handling.
7. **Utility Cohesion**:
    - Components, classes, or operations are grouped based on similar general functions but are otherwise unrelated to each other in terms of their data or logic.

The goal of maximizing cohesion is to ensure that components or classes are focused, maintainable, and easier to understand and modify. Higher levels of cohesion generally lead to better software design.

==========

Slide 16-18

Coupling refers to the degree of dependency between different modules, classes, or components in a system. Lower coupling is generally better because it makes the system easier to implement, test, and maintain. Here's a simplified breakdown of the types of coupling:

1. **Data Coupling**:
    - Operation A passes one or more simple data values to operation B. Fewer data values mean lower coupling.
2. **Stamp Coupling**:
    - A whole data structure or object is passed as a parameter to an operation, increasing the level of dependency.
3. **Control Coupling**:
    - Operation A passes a control flag to operation B to direct its logic. Changes in B might require changes in A to keep the logic correct.
4. **Common Coupling**:
    - Multiple components use a shared global variable, which can cause unintended interactions and errors.
5. **Content Coupling**:
    - One component directly modifies the internal data of another component, creating a strong dependency.

Other types of coupling (unranked):

1. **Subroutine Call Coupling**:
    - One operation calls another within itself.
2. **Type Use Coupling**:
    - Component A uses a data type defined in component B. Changes in the type definition require updates in all dependent components.
3. **Inclusion or Import Coupling**:
    - Component A includes or imports the contents of component B.
4. **External Coupling**:

- A component interacts with external systems or services, like operating system functions or database services.

In summary, coupling measures how much different parts of a system depend on each other. The goal is to minimize coupling to keep the system simple and maintainable.

==========

Slide 19-20

steps involved in conducting component-level design:

1. **Identify Design Classes**:
   - Find all classes that match the problem domain (from the analysis and architectural models).
   - Find all classes related to infrastructure (like GUI components, OS components, data management, networking). These might not be in the initial models.
2. **Elaborate Design Classes**:
   - Detail the design for classes that aren't reusable components.
   - Specify the details of messages when classes or components interact.
   - Identify interfaces for each component, such as abstract classes.
   - Define attributes, data types, and data structures needed for implementation.
   - Describe processing flow within each operation using pseudocode or UML activity diagrams.
3. **Persistent Data Management**:
   - Describe databases and files.
   - Identify the classes needed to manage these data sources.
4. **Behavioral Representations**:
   - Develop detailed behavioral models for each class or component using UML state diagrams and relevant use cases.
5. **Deployment Diagrams**:
   - Create deployment diagrams to show the location of key components in the system.
   - Use class instances and specify hardware and operating system environments.
6. **Consider Alternatives**:
   - Evaluate multiple design solutions before finalizing the design.
   - Use established design principles and guidelines to make the final decision.

In summary, component-level design involves identifying and detailing all necessary classes and components, managing data, developing behavioral models, and creating deployment diagrams. It also involves considering alternative designs to find the best solution.

==========

Slide 21-22

designing conventional components:

1. **Conventional Design Constructs**:
   - Focus on creating maintainable functional or procedural programs.
   - Use basic constructs: sequence, condition, and repetition.
     - **Sequence**: Steps are executed one after another.
     - **Condition**: Decisions are made using if-then-else structures.
     - **Repetition**: Loops are used to repeat actions.
2. **Logical Structure**:
   - Each construct follows a predictable path: control starts at the top and exits at the bottom.
   - This makes it easier to understand and maintain the program.
3. **Design Notations**:
   - Different ways to represent the use of these constructs.
4. **Graphical Design Notation**:
   - Visual representations of sequence, if-then-else, selection, and repetition.
5. **Tabular Design Notation**:

    - Tables are used to represent logic (details usually provided in accompanying slides or materials).
6. **Program Design Language (PDL)**:
    - Similar to programming languages but uses descriptive text within the program statements to explain the logic.

In summary, conventional component design uses clear and predictable structures (sequence, condition, repetition) to ensure maintainability. Various notations, such as graphical, tabular, and program design languages, help represent and explain these constructs.

==========

Slide 25-26

tabular design notation:

1. **List Actions**:
    - Write down all the actions that a specific procedure or module can perform.
2. **List Conditions**:
    - Write down all the conditions or decisions that are made during the execution of the procedure.
3. **Associate Conditions with Actions**:
    - Match specific sets of conditions with the corresponding actions.
    - Eliminate combinations of conditions that are impossible.
    - Alternatively, list every possible combination of conditions.
4. **Define Rules**:
    - Create rules that indicate what actions occur for each set of conditions.

In summary, tabular design notation involves creating a table to match conditions with actions, ensuring clear rules for what actions should occur under specific conditions. This helps in organizing and visualizing the logic of a procedure or module.

rules and conditions using a tabular design notation example:

Rules Table

Conditions	1	2	3	4
Condition A	T	T		F
Condition B		F	T	
Condition C	T			T
Actions				
Action X	P		P	
Action Y				P
Action Z	P	P		P

Explanation:

1. **List Conditions**:
    - **Condition A**: Can be True (T) or False (F).
    - **Condition B**: Can be True (T) or False (F).
    - **Condition C**: Can be True (T) or False (F).
2. **List Actions**:
    - **Action X**: Performed (P) or not performed.
    - **Action Y**: Performed (P) or not performed.
    - **Action Z**: Performed (P) or not performed.
3. **Associating Conditions with Actions**:
    - In Rule 1:
        - Condition A is True.
        - Condition C is True.
        - Action X and Action Z are performed.

    - In Rule 2:
        - Condition A is True.
        - Condition B is False.
        - Action Z is performed.
    - In Rule 3:
        - Condition B is True.
        - Action X is performed.
    - In Rule 4:
        - Condition A is False.
        - Condition C is True.
        - Action Y and Action Z are performed.
4. **Define Rules**:
    - The table shows what actions (X, Y, Z) occur under specific sets of conditions (A, B, C).

Summary

The table helps visualize the relationship between conditions and actions in a structured way. Each rule (1 to 4) corresponds to a unique combination of conditions, and the table clearly defines what actions should be taken for each combination. This organized approach simplifies understanding and implementing procedural logic.