Lecture Title: Chapter 10: Component-Level Design (Part 2 - WebApps, Notation, and CBD)

Subject: Software Engineering

Program: BTech Computer Science and Engineering

Duration: 1 Hour

I. LECTURE INTRODUCTION & OBJECTIVES

Opening Hook: "We've learned how to design cohesive, low-coupled class-based components. But how do we design a 'component' for a WebApp, which is fundamentally about content and interaction? And when we're not using OO, how do we specify the detailed logic of a traditional functional module? Finally, what if we could build software by assembling pre-existing parts like Lego blocks? Today, we complete the component-level design picture with WebApp components, design notations, and Component-Based Development (CBD)."

Objectives: By the end of this lecture, you will be able to:

1. Apply component-level design concepts to WebApps for both content and functionality.
2. Use graphical, tabular, and Program Design Language (PDL) notations to specify traditional component logic.
3. Describe the key activities in Component-Based Development (CBD): domain engineering, qualification, adaptation, and composition.

II. PART 1: COMPONENT DESIGN FOR WEBAPPS

10.4 Component-Level Design for WebApps

- In WebApps, a "component" is broader than a class. It's any cohesive element that contributes to a Web page or its behavior.
- We design at two levels: Content and Functionality.

10.4.1 Content Design at the Component Level

- Focus: The structural elements that define content objects and their presentation.
- Key Artifact: The Content Object. A chunk of text, graphic, video, or audio that can be treated as a distinct entity (e.g., a `Product Description`, a `News Article`, a `User Profile`).
- Design Tasks:
    1. Define Content Objects: Refine them from the analysis model.
    2. Define Content Relationships: How are objects associated? (e.g., a `Product` *has* an `ImageGallery` and `Reviews`).
    3. Define Content Hierarchy: Organize objects from general to specific.
    4. Design Content Templates: Create a consistent visual structure (e.g., an HTML/CSS template) for displaying each type of content object.
- Example: For an `Article` content object: Design a template with slots for `headline`, `byline`, `publication date`, `body text`, `featured image`, and `related links`.

10.4.2 Functional Design at the Component Level

- Focus: The processing elements that manipulate content and perform computations.
- Key Artifacts:
    1. Functional Components: These are the server-side (and sometimes client-side) software modules that deliver functionality (e.g., a `ShoppingCartController`, a `SearchEngine`, a `PaymentGatewayInterface`).
    2. Client-Side Scripts: JavaScript functions that provide interactivity within the browser (e.g., form validation, dynamic content updates).
- Design Tasks:
    1. Allocate Functions: Decide which functions execute on the server vs. the client.

2. Design Server-Side Components: Apply traditional OO or procedural component design principles (cohesion, coupling) to server-side classes/modules.
3. Design Client-Side Scripts: Specify the behavior of interactive scripts, often using UML sequence diagrams or statecharts for complex interactions (e.g., a multi-step checkout wizard).

- Integration: Content templates invoke functional components to populate dynamic data (e.g., a template calls a `ProductRecommender` component to display "customers also bought...").

Quick Discussion (2 min): For a social media "Create Post" feature, what would be a key content object and a key functional component?

III. PART 2: DESIGNING TRADITIONAL COMPONENTS - SPECIFYING THE LOGIC

10.5 Designing Traditional Components

- For procedural, non-OO systems (or for the detailed logic *inside* a method of an OO system), we need notations to specify algorithms and procedural logic.

10.5.1 Graphical Design Notation

- Tool of Choice: The Flowchart.
- Elements: Ovals (start/end), rectangles (process), diamonds (decision), parallelograms (I/O), arrows (flow).
- Strengths: Excellent for visualizing control flow, especially for complex conditional logic and loops.
- Weaknesses: Can become "spaghetti" code in diagram form. Poor at representing data structure.
- Modern Use: Largely superseded by UML Activity Diagrams (which are more structured flowcharts with swimlanes, forks/joins).

10.5.2 Tabular Design Notation

- Tool of Choice: Decision Tables.
- Purpose: To specify complex business rules where multiple conditions lead to different actions.
- Structure:
    - Condition Stub: Lists all relevant conditions.
    - Action Stub: Lists all possible actions.
    - Rules Columns: Each column defines a unique combination of condition values (Y/N) and the resulting actions (X).
- Strengths: Excellent for ensuring completeness and consistency in logic. Removes ambiguity.
- Example: Rules for approving a loan based on income, credit score, and employment status.

In-Class Exercise (5 min): Create a simple decision table for a login system with two conditions: `Username Valid?` (Y/N) and `Password Correct?` (Y/N). Define the actions: `Grant Access`, `Show Invalid User Error`, `Show Invalid Password Error`.

10.5.3 Program Design Language (PDL)

- Also called Structured English or Pseudocode.
- The most important and widely used notation.
- Definition: A text-based, language-independent notation that describes procedural logic using the vocabulary of a natural language (e.g., English) and the syntax of a programming language.
- Characteristics:
    - Uses fixed syntax keywords for `IF-THEN-ELSE`, `DO WHILE`, `REPEAT UNTIL`, `CASE`.
    - Uses natural language for processing steps.
    - Can declare data structures.
    - NO SPECIFIC LANGUAGE SYNTAX (no `{ }`, `;`, pointer arithmetic).
- Example PDL:
- `text`

```
PROCEDURE calculateDiscount (orderTotal, customerStatus)
IF customerStatus = "gold" THEN
    discountRate = 0.15
ELSE IF customerStatus = "silver" AND orderTotal > 100 THEN
    discountRate = 0.10
```

```
ELSE
    discountRate = 0.05
ENDIF
discount = orderTotal * discountRate
RETURN discount
```

- `END PROCEDURE`
- Strengths: Easy to write and read, reviewable, maps directly to code, enforces structured programming concepts.

IV. PART 3: COMPONENT-BASED DEVELOPMENT - THE REUSE PARADIGM

10.6 Component-Based Development (CBD)

- Philosophy: "Buy, don't build." Assemble software from pre-existing, commercially available or in-house reusable components.

10.6.1 Domain Engineering

- The foundational activity for CBD.
- Goal: To identify, model, construct, catalog, and disseminate artifacts (especially components) for a specific application domain (e.g., healthcare, banking).
- Process: Analyze domain -> Build domain model -> Define architecture -> Identify reusable components -> Build component library.

10.6.2 Component Qualification, Adaptation, and Composition

- The core CBD process for building a specific application:
    1. Qualification: Assessing if a candidate Commercial Off-The-Shelf (COTS) or in-house component fits the architectural and functional needs. Does it match interfaces? Is it reliable?
    2. Adaptation: Modifying the component to work in the new context. Techniques include:
        - White-Box Wrapping: Modify source code (rare for COTS).

        - Gray-Box Wrapping: Use the component's extension API.
        - Black-Box Wrapping: Build an adapter component that translates between your system's interface and the component's interface (most common for COTS).
    3. Composition: Integrating the qualified and adapted components into the architectural framework. Requires a run-time infrastructure (e.g., middleware like CORBA, .NET, EJB) to handle communication.

10.6.3 Analysis and Design for Reuse

- When building new components, design them with future reuse in mind.
- Principles:
    - Keep it generic: Avoid hard-coding context-specific details.
    - Keep it cohesive: A reusable component should do one thing exceptionally well.
    - Create a clear, simple, and complete interface.
    - Document thoroughly: Provide clear specifications and usage examples.

10.6.4 Classifying and Retrieving Components

- Problem: How do you find a reusable component in a large library?
- Solution: A component classification scheme.
- Common Schemes:
    - Keyword/Enumerated Classification: Assign keywords from a restricted vocabulary.
    - Faceted Classification: Describe a component using multiple, independent "facets" (e.g., `Function`, `Object`, `Medium`, `Language`). More powerful and flexible.
    - Attribute-Value Classification: Similar to facets.
- Coupled with a component repository (database) and search tools.

V. CONCLUSION & KEY TAKEAWAYS

1. WebApp Component Design has two facets: Content Design (objects, templates) and Functional Design (server-side modules, client-side scripts).

2. Traditional Component Logic is specified using Flowcharts (graphical), Decision Tables (tabular for complex rules), and PDL/Pseudocode (the most versatile and common text-based notation).
3. Component-Based Development (CBD) is a reuse-centric process involving Domain Engineering, followed by Qualification, Adaptation (wrapping), and Composition of pre-built components.
4. Design for Reuse requires extra effort to create generic, well-documented components, which are then classified in a repository for future retrieval.

Final Thought: "Component-level design is the last bastion of pure design thought. Whether you're sketching a PDL algorithm, wrapping a COTS component, or designing a content template, you are making decisions that will either amplify or hinder the productivity of the construction team. Do it with care."

Suggested Reading:

- *The Mythical Man-Month* by Frederick Brooks – Essays on conceptual integrity, relevant to CBD.
- *Component Software: Beyond Object-Oriented Programming* by Clemens Szyperski – The classic CBD text.