Lecture Title: Chapter 10: Component-Level Design
Subject: Software Engineering
Program: BTech Computer Science and Engineering
Duration: 1 Hour

I. LECTURE INTRODUCTION & OBJECTIVES

Opening Hook: "Architecture gives us the big boxes and how they connect. Now we zoom in. What's *inside* those boxes? How do we design the individual, concrete building blocks that developers will actually code? This is Component-Level Design—the phase where we specify the internal details of each software component with enough precision that coding becomes almost a mechanical translation."

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

1. Define a software component from object-oriented, traditional, and process-related perspectives.
2. Apply basic design principles and guidelines to create well-designed class-based components.
3. Evaluate component quality using the metrics of cohesion and coupling.
4. Describe the activities involved in conducting component-level design.

II. PART 1: DEFINING THE BUILDING BLOCK - WHAT IS A COMPONENT?

10.1 What Is a Component?

- A component is a modular, deployable, and replaceable part of a system that encapsulates implementation and exposes a set of interfaces.
- It is the physical realization of an abstraction from the architectural design.

10.1.1 An Object-Oriented View

- A component is a set of collaborating classes.
- Each class has a well-defined set of responsibilities (what it does) and collaborates with other classes to fulfill them.
- The component is represented by one or more design classes and includes all auxiliary classes (e.g., user interfaces, data access classes) needed to implement the abstraction.
- Focus: Classes, objects, inheritance, collaboration.

10.1.2 The Traditional View

- A component is a functional element of a program—a module.
- It encompasses processing logic, internal data structures, and an interface that enables it to be invoked and data to be passed to it.
- Derived from structured programming: a component is a procedural abstraction (a function or subroutine) or a data abstraction (a package of related functions, like a `Math` library).
- Focus: Functions, data structures, functional decomposition.

10.1.3 A Process-Related View

- From the perspective of the software process, a component is a work product that results from component-level design.
- It is a design artifact (e.g., a UML class diagram, a detailed pseudocode description) that will be passed to the construction phase.
- Key Idea: In modern processes (especially Agile), a component might correspond to a story or feature that is designed and implemented within a single iteration.

Synthesis: Regardless of view, a good component is:

- Coherent (it makes sense as a unit).
- Encapsulated (details are hidden).
- Has a clear interface.

III. PART 2: DESIGNING CLASS-BASED COMPONENTS - THE PRINCIPLES

- Assuming an OO approach (most common), we now focus on designing the classes that make up a component.

10.2.1 Basic Design Principles

- The Open-Closed Principle (OCP): A module/class should be open for extension but closed for modification. New functionality should be added by adding new code, not changing old code.
- The Liskov Substitution Principle (LSP): Subtypes must be substitutable for their base types. If `S` is a subtype of `T`, then objects of type `T` can be replaced with objects of type `S` without altering the correctness of the program.
- Dependency Inversion Principle (DIP): Depend on abstractions, not on concretions. High-level modules should not depend on low-level modules; both should depend on abstractions (e.g., interfaces).
- Interface Segregation Principle (ISP): Clients should not be forced to depend on interfaces they do not use. Many client-specific interfaces are better than one general-purpose interface.
- The Release Reuse Equivalency Principle (REP): The granule of reuse is the granule of release. What you reuse, you must be able to release and track.

10.2.2 Component-Level Design Guidelines

- Practical rules derived from principles:
    1. Components: Should be named for their overall function (e.g., `AccountManager`, `SensorController`).
    2. Interfaces: Should provide a clear, consistent abstraction. Use nouns for data-only interfaces, verbs for operation-only interfaces.
    3. Dependencies: Should be explicitly modeled and minimized. In UML, show dependencies with dashed arrows.
    4. Functionality: Should be entirely defined by its interface contract (preconditions, postconditions).
    5. Error Handling: Should be part of the component's design, not an afterthought.

10.2.3 Cohesion (Revisited in Detail)

- Definition: The degree to which the responsibilities of a single component/class are related.
- The Cohesion Spectrum (from WORST to BEST):
    - Coincidental Cohesion (Worst): Parts are unrelated. (A "utility" class with random functions `print()` and `calculateInterest()`).
    - Logical Cohesion: Parts are logically related (all input handlers, all output routines), but functionally different.
    - Temporal Cohesion: Parts are related by timing (e.g., an `initialize()` routine that sets up everything).
    - Procedural Cohesion: Parts execute in a sequence (steps in an algorithm).
    - Communicational Cohesion: Parts operate on the same input data or produce the same output data.
    - Sequential Cohesion: Output from one part is input to the next (like an assembly line). Good.
    - Functional Cohesion (BEST): Every part is essential for performing one single, well-defined task. (e.g., a class that *only* calculates tax).
- Goal: MAXIMIZE FUNCTIONAL COHESION.

10.2.4 Coupling (Revisited in Detail)

- Definition: The degree of interdependence between components/classes.
- The Coupling Spectrum (from WORST to BEST):
    - Content Coupling (Worst): One component modifies the internal data or logic of another (e.g., using `goto` into another module, modifying another's private data).
    - Common/Global Coupling: Components share global data. (Very bad: causes side effects, hard to debug).
    - Control Coupling: One component controls the logic of another by passing control flags or commands.
    - Stamp/Data-Structure Coupling: A component passes a whole data structure to another, which only uses part of it.
    - Data Coupling (BEST): Components communicate via simple data items (parameters). This is the ideal.
- Goal: MINIMIZE COUPLING. AIM FOR DATA COUPLING.

The Golden Rule: STRIVE FOR HIGH COHESION AND LOW COUPLING.

In-Class Exercise: Evaluate this component: A class called `StudentProcessor` with methods: `registerForCourse()`, `calculateGPA()`, `printTranscript()`, `sendEmailToParents()`. What is its likely cohesion level? How could you improve it?

IV. PART 3: CONDUCTING COMPONENT-LEVEL DESIGN

10.3 Conducting Component-Level Design

- This is the process of creating the detailed design.
- Inputs: Architectural design, analysis classes, use cases.
- Outputs: Detailed specifications for each component/class.

Key Activities:

1. Identify all components/classes from the architectural model. These are the "big boxes" to be elaborated.
2. Elaborate design classes from analysis classes:
    - Add detailed attributes (data types, visibility).
    - Define detailed operations (full method signatures: name, parameters, return types).
    - Specify interfaces precisely.
3. Specify persistence and data structures: How will objects be stored? Define DB schemas or file formats.
4. Specify algorithms and logic: For complex operations, define the algorithm using pseudocode, activity diagrams, or decision tables.
5. Apply design principles and patterns: Use patterns (e.g., Strategy, Factory) to solve recurring design problems elegantly.
6. Refine the design: Apply cohesion/coupling analysis. Refactor if needed.
7. Review the design: Conduct a technical review to uncover errors before coding.

Tools & Notation:

- UML Class Diagrams: For static structure (classes, attributes, methods, relationships).
- UML Sequence/Communication Diagrams: For interaction details.
- UML State Diagrams: For components with complex state behavior.

- Pseudocode (PDL - Program Design Language): The primary tool for specifying algorithm logic in a language-agnostic way.

V. CONCLUSION & KEY TAKEAWAYS

1. A Component is a modular, replaceable building block, viewable as collaborating classes (OO), functional modules (Traditional), or a process work product.
2. Class-Based Component Design is guided by principles (OCP, LSP, DIP) and the ultimate goal of High Functional Cohesion and Low Data Coupling.
3. Cohesion measures internal relatedness; Coupling measures external interdependence.
4. Conducting component-level design is a systematic elaboration process, using UML and pseudocode to specify classes, data, algorithms, and interfaces in detail.

Final Thought: "Component-level design is where the rubber meets the road. It's the last checkpoint where we can think deeply about structure and clarity before diving into the syntax of a programming language. A minute spent perfecting cohesion here saves an hour of debugging later."

Suggested Reading:

- *Clean Code* by Robert C. Martin – Excellent for practical class/component design.
- *The Pragmatic Programmer* – Has great sections on coupling, cohesion, and design.