SNS COLLEGE OF ENGINEERING

Kurumbapalayam (po), Coimbatore – 641 107

Accredited by NAAC-UGC with ‘A’ Grade

Approved by AICTE & Affiliated to Anna University, Chennai

DEPARTMENT OF INFORMATION TECHNOLOGY

Unit 3

Coupling and Cohesion

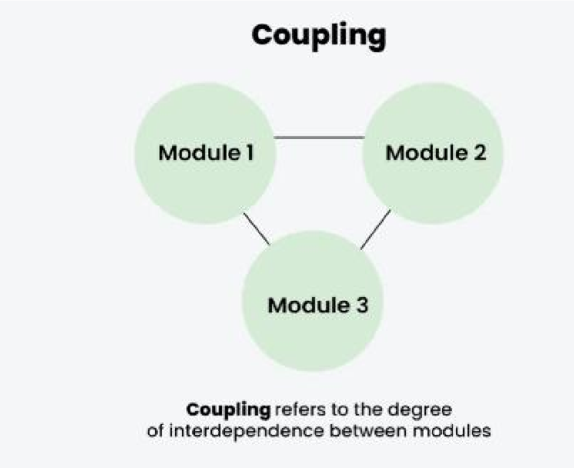
Coupling

What it means: Coupling refers to how much one module or part of the system depends on other modules. Low coupling means the modules are independent, while high coupling means the modules are tightly connected and depend a lot on each other.

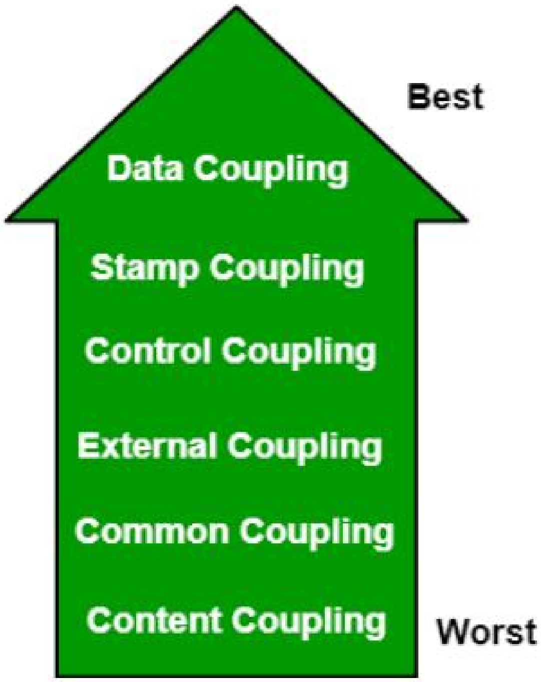
Why it’s important: Low coupling is generally better because it allows modules to work independently of each other. This makes your software more flexible and easier to modify because you can change one part without affecting others.

Example of Low Coupling: Imagine you have a "User Profile" module and a "Settings" module. If these two modules don’t need to know much about each other, they are loosely coupled. Each one can work on its own without disturbing the other.

Example of High Coupling: Now, imagine that the "User Profile" module needs constant updates from the "Settings" module for everything to work. If you want to change the "Settings" module, you have to change the "User Profile" module too. That’s high coupling.


Coupling
Module 1
Module 2
Module 3
Coupling refers to the degree
of interdependence between modules

Types of Coupling:



Types of Coupling

1. Data Coupling

- **What it is**: This is the best kind of coupling. It happens when modules communicate **only by passing data**. They don't rely on each other's internal details.
- **Example**: Think of a **Customer Billing System**. One part calculates the bill, and the other part generates the invoice. They just pass the total amount between them, not much else.
- **Why it's good**: The modules are **independent**, so if one changes, the other won't be affected much.

2. Stamp Coupling

- **What it is**: Here, one module passes a **whole data structure** (like an object or a list) to another module, even though the second module only needs part of it.
- **Example**: Imagine a **Student Record System**. One module passes the entire student record (name, ID, grades) to another module, even though it only needs the grades.

- **Why it's okay**: It might be necessary for efficiency, but it’s **not ideal** because the second module is getting more data than it actually needs.

3. Control Coupling

- **What it is**: This happens when modules communicate by passing **control information** (like flags or parameters that tell what to do next).
- **Example**: Think of a **sort function** that takes a **comparison function** as a parameter. The comparison function decides how the sorting will work.
- **Why it can be good**: The function is reusable and flexible because it can sort in different ways depending on the comparison function passed.

4. External Coupling

- **What it is**: This happens when a module depends on **external** systems or hardware, like a **file format** or **protocol**.
- **Example**: A module that **reads from an external file**. The file format must remain the same, or the module won’t work.
- **Why it's problematic**: Dependencies on external systems make the software harder to **maintain** and **update**.

5. Common Coupling

- **What it is**: Modules share **global data**. If one module changes the global data, it affects all other modules using it.
- **Example**: Imagine two modules that use a **global variable** like totalBalance. If one module changes this balance, the other module’s behavior might change unexpectedly.
- **Why it's bad**: Global data can make the system harder to understand and modify, and changes in one place can break other parts of the system.

6. Content Coupling

- **What it is**: This is the **worst** kind of coupling. One module can **modify the internal data** of another module, or the control flow is passed directly from one module to another.

- **Example**: One module directly changes the **internal data** of another module, which breaks the module's internal workings.
- **Why it’s bad**: This makes the system very **tightly coupled**, and it’s **hard to change** or maintain without breaking things.

7. Temporal Coupling

- **What it is**: Modules depend on the **timing** or **order of events**. One module must execute before the other.
- **Example**: Imagine a process where **Module A** must finish saving data before **Module B** starts sending an email. If the order is wrong, it could break the system.
- **Why it’s problematic**: If the timing is not handled properly, things may not work as expected, and it becomes hard to **test** and **maintain**.

8. Sequential Coupling

- **What it is**: This happens when the **output** of one module is used as the **input** of another module.
- **Example**: Module A processes data and sends the result to Module B, which then processes it further. They depend on each other’s output and input.
- **Why it’s difficult**: If Module A changes its output, Module B may break. It can make the system hard to modify.

9. Communicational Coupling

- **What it is**: This occurs when modules share a common communication method, like a **shared message queue** or **database**.
- **Example**: Two modules share a **database** to send and receive information.
- **Why it’s problematic**: Sharing a communication system can create **performance issues** and **difficulties in debugging**.

10. Functional Coupling

- **What it is**: This happens when two modules depend on each other’s **functionality**.
- **Example**: Module A calls a function in Module B to perform a task. If Module B’s function changes, Module A might break.
- **Why it’s bad**: Tight dependencies like this make the system **difficult to maintain** and **update**.

11. Data-Structured Couplng

- **What it is**: This occurs when two modules share the same **data structure**, like a **database table**.
- **Example**: Two modules share a **database table** to read and update data. If the table structure changes, both modules can break.
- **Why it's problematic**: Changes to the shared data structure can cause problems in multiple places, making maintenance difficult.

12. Interaction Coupling

- **What it is**: This happens when methods from one class **invoke** methods of another class.
- **Example**: Module A directly calls methods in Module B to perform its tasks. This is interaction between the two modules.
- **Why it's bad**: If methods in Module B change, Module A might break, making the system hard to modify.

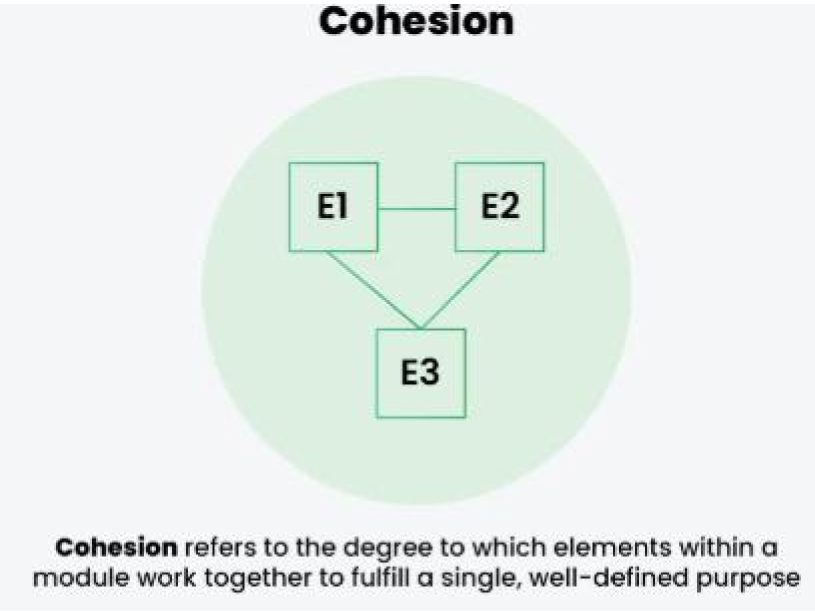
13. Component Coupling

- **What it is**: This refers to a class having **variables** or **methods** of another class.
- **Example**: Class A has an **instance variable** of Class B. If Class B changes, Class A might need to change as well.
- **Why it's problematic**: Changes in one class may force changes in other classes, leading to tight coupling and hard maintenance.

Cohesion

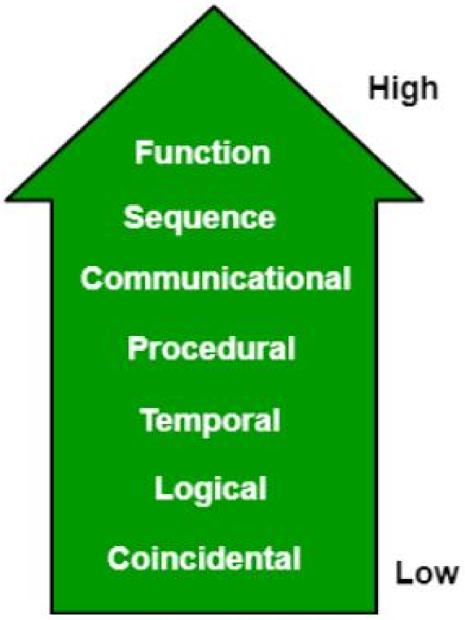
1. Cohesion

- **What it means**: Cohesion refers to how closely the responsibilities of a module (or part of a program) are related to each other. A module with **high cohesion** means that the things it does are very closely related, whereas a module with **low cohesion** means it tries to do many different things that aren't really connected.
- **Why it's important**: High cohesion is usually better because it makes the module easier to understand, test, and maintain. It also leads to code that's more organized and focused.
- **Example of High Cohesion**: Imagine a "Weather Forecast" module that only focuses on getting and showing weather information. Everything inside this module is related to weather, so it has high cohesion.
- **Example of Low Cohesion**: Now imagine a "Weather Forecast" module that also handles user logins, payment processing, and sends emails. This module has low cohesion because it's trying to do too many different things that aren't related to each other.



Cohesion refers to the degree to which elements within a module work together to fulfill a single, well-defined purpose

Types of Cohesion:



Types of Cohesion

1. Functional Cohesion

- **What it is**: This is the best type of cohesion. All the elements (or tasks) in a module are focused on performing a **single well-defined task**. They work together to achieve one function.
- **Example**: A module that **calculates the area of a rectangle**. It only does this task and nothing else. All the tasks in this module are related and focused on calculating the area, making the module very **cohesive**.
- **Why it's good**: It leads to easy-to-understand, reusable, and maintainable code.

2. Sequential Cohesion

- **What it is**: This occurs when the output of one part of the module becomes the input for another part. There's a **flow of data** between the elements.
- **Example**: A module that **reads data**, then **processes the data**, and finally **prints the results.** The tasks are performed one after another in a sequence, and each task depends on the output of the previous one.

- **Why it’s useful**: It’s common in functional programming, where the output of one function naturally leads to the next function.

3. Communicational Cohesion

- **What it is**: This occurs when the elements in a module work on the **same data** or contribute to the **same output**.
- **Example**: A module that **updates a record in a database** and **sends the updated record to a printer**. Both actions involve the same data (the record).
- **Why it’s good**: The tasks are related because they use the same data and contribute to a common purpose.

4. Procedural Cohesion

- **What it is**: This type of cohesion happens when elements are grouped together based on the **order of execution**, even though the tasks may not be closely related in function.
- **Example**: A module that **calculates a student's GPA**, **prints the student record**, **calculates cumulative GPA**, and **prints the cumulative GPA**. These tasks are related in terms of the sequence of actions, but they are not directly related to each other in purpose.
- **Why it’s less ideal**: It can be harder to maintain because tasks aren't really related in what they do, just in when they happen.

5. Temporal Cohesion

- **What it is**: This occurs when tasks are grouped together because they need to be executed at the **same time** or **in the same time span**. These tasks may not be functionally related, but they happen together.
- **Example**: A module that **initializes the system** by setting up database connections, loading configuration files, and starting background tasks. These tasks don’t really relate to each other but need to be done together when the system starts.
- **Why it’s useful**: It’s often used in real-time or embedded systems where tasks need to be performed together within a specific timeframe.

6. Logical Cohesion

- **What it is**: This occurs when tasks are logically related, but not functionally. The tasks perform different operations, but they are grouped because they are related by **type**.
- **Example**: A module that handles **input from multiple sources**, like a **disk**, a **network**, and a **tape**. All these inputs are logically related because they're types of input, but they perform different functions.
- **Why it's not ideal**: The tasks are related in type but not in functionality, so it can make the module hard to maintain.

7. Coincidental Cohesion

- **What it is**: This is the **worst form of cohesion**. Tasks in a module are **not related at all** and have no logical or functional connection. The only thing they have in common is that they are in the same module.
- **Example**: A module that **prints the next line** and **reverses a string**. These tasks don't share any purpose or function but happen to be in the same module.
- **Why it's bad**: It makes the code hard to understand, maintain, and modify because the tasks are completely unrelated.

8. Informational Cohesion

- **What it is**: This happens when the tasks in a module are grouped because they operate on the **same data structure or object**.
- **Example**: A module that handles operations for a **Student object** — like **adding grades**, **calculating average grade**, and **printing the student's details**. All these tasks work on the same data, which is the **Student object**.
- **Why it's good**: It's useful in **object-oriented programming**, where tasks related to a specific data structure are grouped together.

9. Layer Cohesion

- **What it is**: This occurs when tasks are grouped together based on their **level of abstraction or responsibility**. A module handles only specific **layers** of the system.
- **Example**: A **database access layer** module might handle all operations related to **reading and writing to a database**, while a **business logic layer** module handles operations related to calculations or processing business rules.

- **Why it’s useful**: It organizes code by its purpose, making it easier to maintain and update.