🔥 Exam-Focused Summary: Coupling

Definition

Coupling → The degree of **dependency** between two modules in a software system.
Goal → Keep **low coupling** for **better maintainability** and **reusability**.

🔢 Types of Coupling (From Best to Worst)

Type	Definition	Example	Good/Bad
1️⃣ **Data Coupling**	Modules communicate via **simple data** (e.g., `int`, `float`).	`computeSalary(int hours, float rate)`	✅ **Best (Low Coupling)**
2️⃣ **Stamp Coupling**	Modules share **structured data** (e.g., `struct` in C).	Passing a `struct Student` instead of `student_id`.	⚠️ **Acceptable**
3️⃣ **Control Coupling**	One module **controls** another via **flags/decision variables**.	`performOperation(1); // 1 = Read, 2 = Write`	❌ **Bad (High Dependency)**
4️⃣ **Common Coupling**	**Global variables** are shared across modules.	`extern int globalCounter;`	🚨 **Very Bad (Unpredictable Changes)**
5️⃣ **Content Coupling**	One module **modifies/accesses** another's internal code.	`extern float accountBalance;`	🚨🚨 **Worst (Breaks Encapsulation)**

📌 Shortcut to Remember

"Dogs Sleep Calmly, Chasing Cats"
1️⃣ Dogs → **Data Coupling** ✅ (Best)
2️⃣ Sleep → **Stamp Coupling**
3️⃣ Calmly → **Control Coupling** ❌
4️⃣ Chasing → **Common Coupling** 🚨
5️⃣ Cats → **Content Coupling** 🚨🚨 (Worst)

💡 Key Points for Exams

✅ **Always prefer Data Coupling** – Use **function parameters** instead of global variables.
❌ **Avoid Content Coupling** – One module **should not** access another module's internals.
⚠️ **Common Coupling is risky** – Global variables cause **unintended side effects**.
🔥 **Low Coupling = Better Software** – More **maintainable, reusable, and testable** code.

This should be **enough for exam prep!** Let me know if you need quick MCQs or diagrams! 😊📖

-Hierarchy of modules

🔹 Shortcut to Remember

📌 **"H.L.A.F" → A Good Hierarchy has High-Level Abstraction & Low Fan-out**
1️⃣ **H** → **Hierarchy** should be **clear & structured** (tree-like).
2️⃣ **L** → **Low Fan-out** → Each module should **control fewer submodules**.
3️⃣ **A** → **Abstraction** → Hide unnecessary details, **expose only necessary functions**.
4️⃣ **F** → **Flexibility** → A **good hierarchy** makes the system **easy to extend & modify**.

🔥 Key Exam Takeaways

✅ **Control hierarchy = Defines module relationships & invocation**
✅ **Structure chart = Tree-like representation of control flow**
✅ **Low Fan-out = Each module should control limited submodules**
✅ **Abstraction = Hide internal details, expose only what's necessary**
✅ **Good hierarchy = Easier maintenance, debugging & reusability**

Characteristics of module structure

🔹 Summary Table (For Exam Revision)

Concept	Definition	Good/Bad?	Shortcut to Remember
Layering	Modules are arranged in **layers** where higher layers depend on lower layers	✅ Good	**Top modules control lower ones, not vice versa!**

Control Abstraction	Modules should only **invoke** functions in the layer immediately below	✅ Good	**Follow strict top-down design!**
Fan-Out	Number of **submodules** controlled by a module	❌ Bad (If too high)	**Too many dependencies = Spaghetti code!**
Fan-In	Number of **modules that call** a given module	✅ Good (If high)	**More reuse = Better maintainability!**

🔥 Key Takeaways for Exams

✅ **Use layering** to keep modules organized.
✅ **Follow control abstraction** → Higher modules should control lower ones.
✅ **Keep fan-out low** → Avoid controlling too many submodules.
✅ **Encourage high fan-in** → More modules should reuse common utilities.

🎯 Key Differences: FOD vs. OOD

Feature	Function-Oriented Design (FOD)	Object-Oriented Design (OOD)
Approach	Focuses on **functions (procedures)**	Focuses on **objects (real-world entities)**
Data Management	Data is **shared globally**	Data is **encapsulated within objects**
State	**Centralized** (Global Data)	**Decentralized** (Each object has its own state)
Communication	Functions call each other	Objects communicate via **message passing**
Reusability	**Low** (Harder to reuse individual functions)	**High** (Encapsulation and inheritance improve reusability)
Scalability	**Difficult** (New functions require modifying existing code)	**Easier** (New classes can be added without affecting others)

DFD (Data Flow Diagram) – Exam Shortcuts

DFD Helps Create

📌 **Function Model** → Breaks system into smaller functions.
📌 **Data Model** → Defines what data is exchanged and refined.

Key DFD Rules

✔ **External entities** only appear in the **Context Diagram**, not in detailed DFDs.
✔ **DFD does NOT show control information**, conditions, or order of execution.
✔ Only **data flow** is represented, not decision-making logic.

Shortcomings of DFD

❌ **Imprecise** (May not handle missing/wrong input).
❌ **No control flow** (Doesn't show when/how functions run).
❌ **No synchronization** (Doesn't define parallel operations).
❌ **Subjective decomposition** (Different ways to break down the system, no clear best approach).

Quick Memory Trick 🧠

"DFD = What flows, not how it flows!" 🚀

Shortcuts to Remember SRS Characteristics & Types of Requirements

✅ Shortcut for Characteristics of a Good SRS:

👉 **C.C.C.U.V.M.T.F.** (💡Think: "**Cool Cats Can Use Very Modern Tech Features**")

- **C** - Correctness
- **C** - Completeness
- **C** - Consistency
- **U** - Unambiguity
- **V** - Verifiability
- **M** - Modifiability
- **T** - Traceability
- **F** - Feasibility

✅ Shortcut for Types of Requirements:

👉 **"FND" (Found)**

1️⃣ **F** - Functional
2️⃣ **N** - Non-functional
3️⃣ **D** - Domain-specific

💡 **Tip:** Just remember "**FND the requirements!**"

Extra Quick Tips for Exam

✔ **For Definitions:** Always break into **What? Why? Example.**
✔ **For Characteristics:** Look for words ending in **-ness** (Correctness, Completeness, etc.)
✔ **For Requirements:** Think **Function vs. Quality vs. Industry**

Let me know if you need more! 🚀😃

Shortcut to Remember: "H-CLIT"

✅ **H**igh Cohesion
✅ **Cl**ear Interfaces
✅ **L**ow Coupling
✅ **I**ndependent Testing
✅ **T**estability & Maintainability

👉 If a design meets these criteria, it's **modular**!