Memory Management LLD Project

**SOLID Principles**

**S – Single Responsibility**

* Each class has one clear responsibility:
  + MemoryManager → orchestrates memory handling
  + TLB → manages cache lookups
  + Logger → handles logging
  + PageTable → manages page-frame mapping
  + ProcessManager → manages process lifecycle

**O – Open/Closed**

* PageReplacementPolicy lets you add new strategies
* TLB can be extended with FIFO or Random eviction without altering core logic
* Logger can be replaced by FileLogger, MockLogger, etc.

**L – Liskov Substitution**

* Any PageReplacementPolicy implementation works interchangeably at runtime (LRU, FIFO, Random), no surprises.

**I – Interface Segregation**

* Interfaces are minimal and focused (e.g., PageReplacementPolicy has just two methods)

**D – Dependency Inversion**

* MemoryManager depends on abstractions (interfaces) rather than concrete classes
* You can inject different Logger, TLB, or page policies easily, keeping code loosely coupled

**Structural Principles**

**Composition over Inheritance**

* MemoryManager **has** a TLB, **has** a PageTable, **has** a Logger, **has** a PageReplacementPolicy, **has** a ProcessManager

**High Cohesion**

* Each class deals only with its own domain, without mixing unrelated functionality

**Loose Coupling**

* Classes interact via well-defined interfaces—a change in one module (e.g. replacing LRU with FIFO) doesn’t ripple across the system

**Architectural Patterns**

**Strategy Pattern**

* PageReplacementPolicy interface → allows switching replacement behavior without altering memory manager logic

**Dependency Injection**

* Modular approach supports testing and easy substitution

**Reusability & Extensibility**

* Adding new policies, TLB eviction tactics, or logging sinks requires **zero changes** to your core MemoryManager
* Doing so enriches functionality **while keeping design stable**

**Testability**

* You can **mock** or substitute individual components like the logger or page policy for targeted unit tests
* Benefit from clean boundaries and no hidden dependencies

**Performance Awareness**

* You added **metrics** (optional) such as page faults, TLB hits/misses, and eviction count—showing quantitative insight into system behavior

**Why It Matters in Interviews**

* Demonstrates deep understanding of **LLD principles**, not just coding ability
* Reflects best practices in software architecture design
* Provides concrete talking points (e.g., “I used strategy pattern for page policy, enabling open-closed extensibility…”)

**TL;DR:**

Your project showcases **SOLID**, **composition**, **strategy**, **dependency injection**, and **loose coupling**, all in a clean, modular system. It’s a strong, credible candidate demonstration of real production-quality Low-Level Design.

SUMMARY

**This project is a simulation of an OS memory management unit, focusing on virtual memory, TLB caching, and multiple page replacement policies. At its core, the MemoryManager class orchestrates the system, coordinating components like the page table, TLB, logger, and the replacement policy.**

**One of the most important design decisions I made was to follow SOLID principles: I separated responsibilities into focused classes—for example, the TLB only handles cache lookups, the Logger logs events, and each replacement policy like LRU or Random is implemented using the Strategy pattern via a common interface. This makes the system open for extension—like plugging in a new LFU policy—without modifying core logic.**

**All components are injected as dependencies, making the design loosely coupled and testable. For example, in testing, I can use a mock logger or a stub TLB. I also implemented multilevel paging and a metrics system to track performance—like TLB hit/miss rates and page faults—which gives insight into how the system behaves.**

**Overall, this project not only demonstrates the functionality of memory management but also reflects a production-style architecture that’s clean, modular, and extensible.**