SOLID and Clean Design

SOLID Principles

- The SOLID principles are a restatement of design guidelines
  - These come out of the OO world
  - Many of them are restatements of principles that we have already seen
  - Some are very OO specific principles
  - They are not rules, just design guidelines
  - You will notice that the same themes about design are repeated in various places in the course
  - This demonstrates how foundational these concepts are to engineering and software development
- Clean Design
  - A term used to describe macro program components that follow the engineering design principles and SOLID concepts discussed here

SOLID Principles

- SOLID were introduced by Robert C. Martin (Uncle Bob)
  - Goal was to make engineering design principles more accessible for programmers
- SOLID
  - Single responsibility principle
  - Open–closed principle
  - Liskov substitution principle
  - Interface segregation principle
  - Dependency inversion principle

Single Responsibility

- This is a restatement of the idea of high cohesion
- A class should have only one reason to change
  - Each class or module should do one thing, and do it well
  - Makes the code easier to understand
  - Less risk of breaking unrelated functionality when making changes.
  - Makes the code easier to test
- Restating the principle
  - If a module has only one responsibility, then the only thing that should cause it to change is a change in that responsibility
  - If a module has multiple responsibilities, then changes to any of the responsibilities will force the module to change

Open-Closed Principle

- Software entities should be open for extension, but closed for modification
  - Add new behavior without rewriting existing code
  - Prevents introducing bugs into stable code
- Restating the principle
  - Once a module is in production, changes to the module may cause clients to crash
  - If we need to add functionality to a module, there should be a way to do it
  - And we should not have to change what is currently there
  - Rather than remove or change functionality, we deprecate it
    - That means we still support the deprecated functionality, but we warn users that there is something else should be used
    - This avoids crashing systems because of enhancements or updates

Liskov Substitution Principle

- Subtypes must be substitutable for their base types
  - If S is a subtype of T, objects of type T should work when replaced with objects of type S
  - Inheritance should preserve expected behavior
  - Violations lead to surprises and runtime errors
- Restatement of the principle
  - A subclass should avoid overriding a concrete method of the superclass
  - Refers to functionality that is the responsibility of the superclass, not the subclass
  - “S is a type of T, so everything T is expected to do should be done by S”
  - Does not apply to superclass methods that are intended to be overridden by subclasses

Interface Segregation Principle

- No client should be forced to depend on methods it does not use
  - Favor small, specific interfaces over large, “fat” ones
  - Prevents bloated, fragile contracts
  - Keeps implementations lean and focused
- Restating the principle
  - An interface should be crafted for the needs of a specific group of stakeholders
  - It should only offer functionality that is relevant to that group
  - Exposing new functionality via a new interface should not require changing existing interfaces

Dependency Inversion Principle

- We have already seen this one
- Depend on abstractions, not on concretions
  - High-level modules shouldn't depend on low-level details
  - Both should depend on interfaces/abstractions
  - Reduces coupling
  - Improves testability (e.g. easy to mock dependencies)
  - Makes module implementation easier to change

Clean Design

- This refers to applying the engineering design principles and SOLID to designing an application architecture
- Readability First
  - Code is read far more often than it's written
  - Favor clarity over cleverness
  - Use meaningful names: for example, getCustomerOrders() vs. gco()
  - Keep functions short: do one thing, do it well
- Modularity
  - Break systems into small, well-defined components
  - Each module/class has a clear responsibility (ties to SRP)
  - Benefits: easier debugging, parallel development, and reuse

Clean Design

- Low coupling, high cohesion
  - Cohesion = how focused a module is on a single task. High cohesion is good
  - Coupling = how dependent modules are on each other. Low coupling is good
  - A clean design maximizes cohesion and minimizes coupling
- Encapsulation and information hiding
  - Hide internal details, expose only what’s necessary
  - Reduces accidental misuse and allows internal changes without breaking clients
- Consistent error handling
  - Handle errors gracefully and consistently
  - Use exceptions rather than silent failures or cryptic codes
  - Provide useful error messages without leaking sensitive information (ties to secure coding)

Clean Design

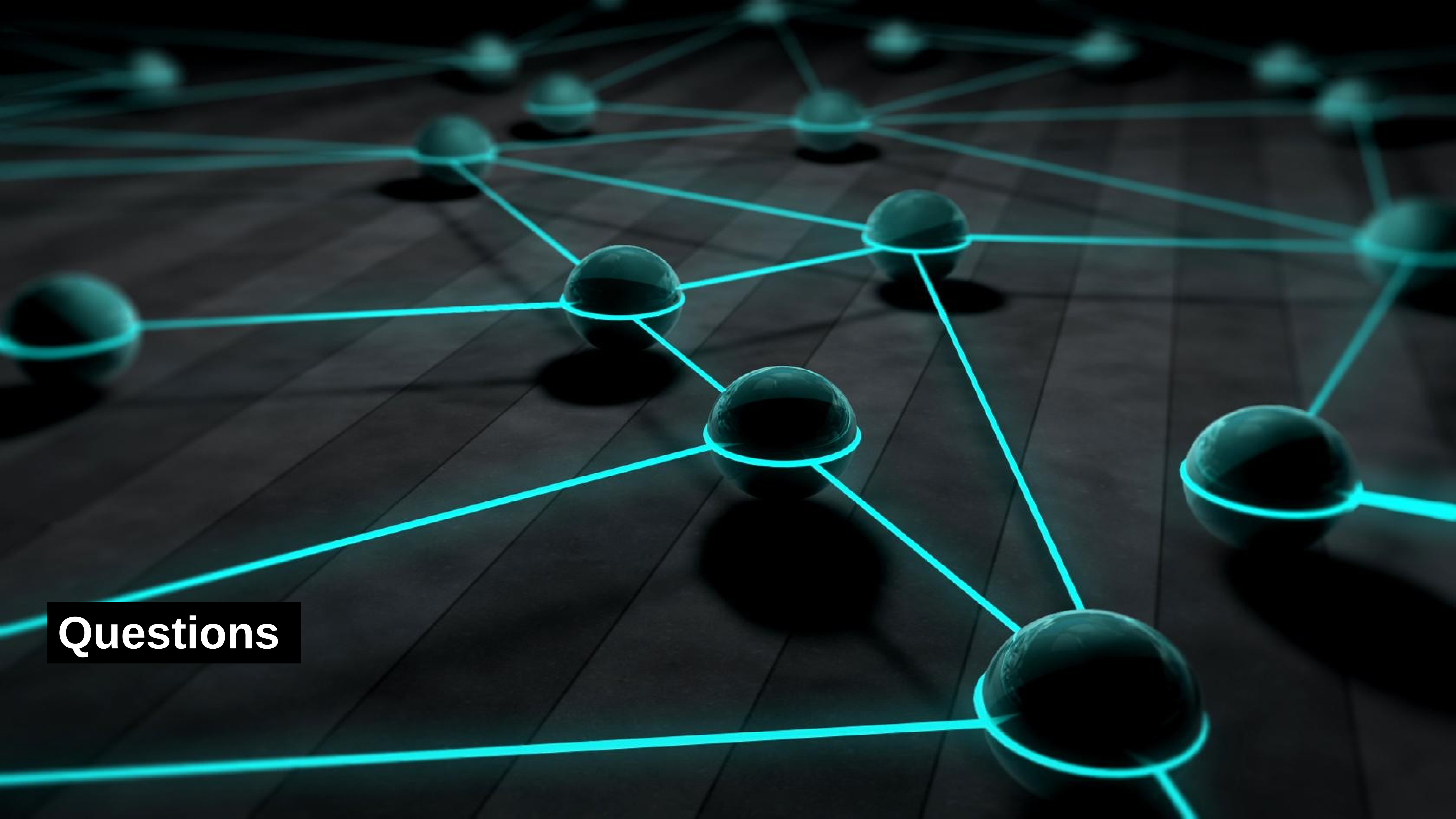
- Defensive Programming
  - Anticipate misuse and unexpected inputs
  - Example: Validate parameters before processing
  - Fail early with clear error messages
- Avoiding Code Smells
  - Common indicators of poor design (“bad smells”):
    - God Class: a class knows too much or does too much
    - Duplicated Code: logic repeated across modules
    - Long Method: hard to test and understand
    - Primitive Obsession: using raw strings/ints instead of proper types

Clean Design

- Refactoring as Discipline
  - Continuously improve design without changing behavior
  - Examples of refactorings:
    - Extract method (split large function)
    - Introduce parameter object (group related params)
    - Replace magic numbers with named constants
- Design for Testability
  - Code should be easy to test in isolation
  - Use interfaces and dependency injection to allow mocking
  - Keep functions pure where possible (no hidden side effects)

Clean Design

- Consistency and Style
  - Follow consistent naming, formatting, and structure
  - Adopt a shared coding style (e.g., PEP 8 for Python, Google/Oracle style for Java)
  - Makes codebases easier for teams to navigate



Questions