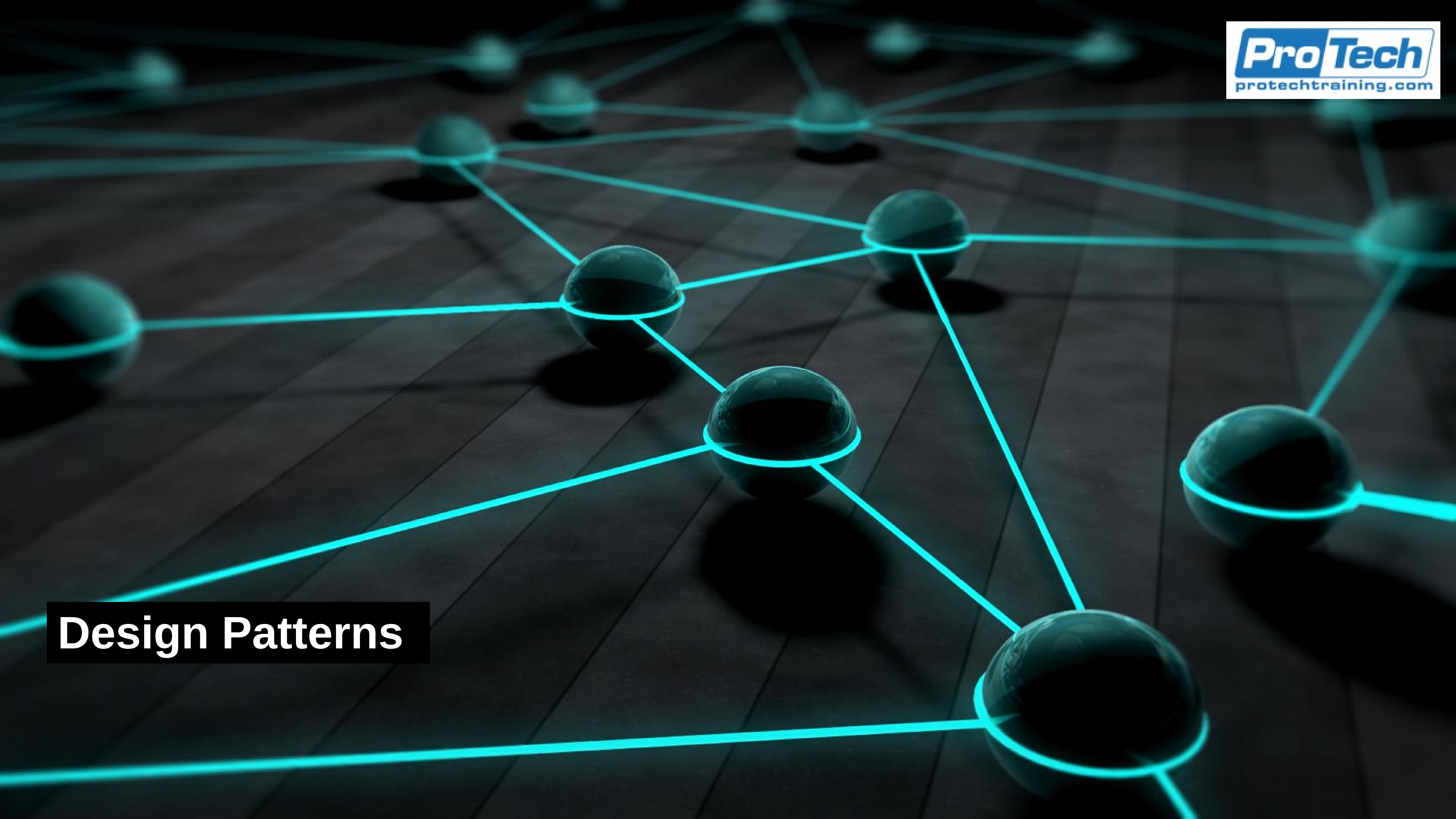
ProTech
protechtraining.com



Design Patterns

Patterns as Best Practices

- Design patterns are reusable solutions to common software design problems.
  - First proposed by Kent Beck and Ward Cunningham
  - Noticed that expert programmers tended to solve the same problem in similar ways
  - Cataloged in the book “Design Patterns: Elements of Reusable Object-Oriented Software”
- Reasons for use
  - Proven solution templates to build on instead of reinventing the wheel
  - Improve communication between developers
    - Referring to “Singleton” or “Observer” is faster and more precise than describing the entire design
  - Makes code more maintainable, scalable, and flexible
- Analogy:
  - Like architectural blueprints for buildings, they are not code but guidelines for structure

Patterns as Best Practices

- Patterns are not functional solutions
  - Algorithms are functional solutions, they provide guidance on producing specific results at the code level
  - Patterns focus on structural issues that affect performance
  - Patterns are about how to organize code
  - Primary purpose is often to improve performance
- For example
  - A resource may be expensive to create when requested and continuous requests for the resource are slowing the system down
  - The Flyweight pattern shows how to pre-allocate a number of instances of the resource into a pool or reusable instances

Types of Patterns

- Patterns were originally limited to OO type programming
- However the concept has been extended to other areas
- Software Architecture Patterns
  - Patterns at the level of macro system design
  - Helps teams structure large systems consistently.
  - Examples:
    - Layered Architecture (UI - Business Logic - Data).
    - Microservices (independent services with APIs).
    - Event-Driven Architecture (systems built around events).

Types of Patterns

- Enterprise Integration Patterns
  - Patterns for connecting different systems
  - From Hohpe & Woolf’s book Enterprise Integration Patterns
  - Examples:
    - Message Bus
    - Publish/Subscribe
    - Message Translator (Adapter at integration scale)
  - Widely used in messaging systems like Kafka, RabbitMQ

Design Pattern Structure

- To prevent anything from just being called a pattern
  - The original design pattern book defined a template to document patterns
  - The intent was to make patterns a more usable tool
- The template has the following sections
  - Pattern Name and Classification
    - The name gives a shorthand way to refer to the solution.
    - Classification (Creational, Structural, Behavioral).
  - Intent
    - What the pattern does, its purpose, and rationale.
    - Answers “What problem does this pattern solve?”
  - Also Known As
    - Any alternative names.

Design Pattern Structure

- Motivation
  - Example scenario illustrating the problem and how the pattern provides a solution
- Applicability
  - Situations where the pattern is useful
  - Recognizable symptoms of the problem
- Structure
  - Diagrams (class diagrams, interaction diagrams) showing the pattern components
- Participants
  - The classes and objects involved in the pattern
- Collaborations
  - How participants interact with each other
- Consequences
  - Results of applying the pattern (benefits, trade-offs, costs)

Design Pattern Structure

- Implementation
  - Tips, pitfalls, and language-specific notes for implementing the pattern
- Sample Code
  - Concrete examples (in C++ and Smalltalk in the original book, but now often Java/Python).
- Known Uses
  - Examples of real systems where the pattern has been applied successfully
- Related Patterns
  - Connections to other patterns (complementary or alternative approaches)

Antipatterns

- Documented in the book “Antipatterns” by Malveau et al
  - Informally, describing the systematic patterns of how people break things
- “Solutions” that look attractive but create more problems than they solve
  - Opposite of design patterns
- Common Anti-Patterns:
  - God Object: one class does too much (violates single responsibility)
  - Spaghetti Code: tangled, hard-to-maintain code without structure
  - Singleton Abuse: making everything a singleton, thereby creating hidden dependencies
  - Golden Hammer: applying the same pattern everywhere, even when not appropriate

Antipatterns

- Like design patterns, antipatterns have a specific structure
- This is intended to provide a way to correct the antipattern
- Some parts of the structure are:
  - Problem
    - The context and recurring problem that leads to the AntiPattern
    - Symptoms or indicators that it exists
  - Symptoms
    - Observable signs in code, architecture, or process
    - Often framed as “smells” (e.g., excessive complexity, lack of modularity)
  - Consequences
    - Negative outcomes of the AntiPattern (technical debt, performance bottlenecks, maintainability issues)

Antipatterns

- Root Cause
  - The underlying reason this AntiPattern tends to emerge (e.g., lack of experience, deadline pressure, poor planning)
- Refactored Solution (a.k.a. Refactored AntiPattern)
  - A proven method to resolve or avoid the AntiPattern
  - Often framed in terms of corresponding design patterns or best practices
- Examples
  - Real-world cases or anecdotes where the AntiPattern has been observed

Design Patterns

- Design patterns do not originate in software
- They are solutions to real-world problems
  - Adapted to solve similar problems in software
  - Since they work in the real-world, it’s not surprising they work in software
- Design patterns are divided into three categories
  - Creational: Deal with object creation in a flexible, reusable way.
  - Structural: Focus on composition of classes and objects.
  - Behavioral: Focus on object interactions and communication.

Creational Patterns

- Singleton:
  - Ensure that there is only one instance of a class.
    - Example: a logger in an application
    - Real world: President of the US
- Factory Method
  - Lets subclasses decide what concrete type of object to create
  - The superclass only specifies an abstract type
  - Used when you need to create objects without knowing the exact class at compile time.
    - Example: A GUI library where you call a factory to create Button objects (Windows vs Mac look)
    - Real world: A coffee kiosk with buttons for “espresso,” “latte,” “americano”: the machine decides which internal process to run based on the requested type derived from an abstract type

Creational Patterns

- Abstract Factory
  - Used when you need families of related objects.
    - Example: Cross-platform UI frameworks (create a whole Windows widget set vs Mac widget set)
    - Real world; Choosing a furniture “style” (e.g., Scandinavian set) from a showroom
    - You get a matching sofa, chair, and table produced as a coordinated family
- Builder
  - Used when constructing complex objects with many optional parts
  - The builder object has the logic to execute the requested construction
    - Example: Creating a Meal object in a fast-food ordering system (burger + fries + drink)
    - Real world: Ordering a customized item of any type

Creational Patterns

- Prototype
  - Used when object creation is expensive and cloning is faster
    - Example: Copying a pre-configured document or template file and modifying it
    - Real world: Cutting a copy of a key from the original

Structural Patterns

- Adapter
  - Used when two incompatible systems need to work together
    - Example: Making a new payment gateway API fit into your old billing system
    - Real world: A travel plug adapter lets your device’s plug fit foreign wall sockets without changing either side
- Decorator
  - Used when you want to add features dynamically without changing the base class.
    - Example: Wrapping a basic Printer class with decorators like RemotePrinter or LaserPrinter
    - Real world: You take a present and add wrapping to enhance it without altering the gift itself
- Facade
  - Used to hide complexity behind a simple interface.
    - Example: A ComputerFacade class that provides a single start() method instead of many subsystem calls
    - Real world: A hotel concierge: one desk provides a simple interface to many complex services

Structural Patterns

- Proxy
  - Used when you want to control access, defer loading, or add security
    - Example: A Virtual Proxy that loads an image only when it’s actually displayed
    - Real world: Proxy voting in companies or politics
- Composite
  - Used when individual objects and groups of objects should be treated the same
    - Example: A file system where files and folders are both "components" and can be traversed uniformly
    - Real world: Departments on a company organization chart
- Bridge
  - Used to decouple an interface from its implementation so both can vary independently
    - Example: A “FileStorage” API that can switch implementations without changing callers.
    - Real world: A universal remote (interface) that controls different devices (implementation)

Structural Patterns

- Flyweight
  - Used to manage large reusable fine-grained objects that are expensive to create or use
    - *Example: Tokenization: Reusing interned strings/tokens/AST that repeat heavily*
    - *Real world: A motor pool for a company that supplies cars and drivers*

Behavioral Patterns

- Observer
  - Used when changes in one object should automatically notify others
    - *Example: A stock market app where stock price updates notify all subscribed dashboards*
    - *Real world: SMS weather alerts: subscribers automatically get notified when the weather service publishes an update*
- Strategy
  - Used when you want to switch between different algorithms at runtime
    - *Example: Payment system choosing between credit card, PayPal, or bank transfer strategies*
    - *Real world: Choosing travel routes, fastest vs. shortest vs. scenic to the same destination*
- Template Method
  - Used when an algorithm has a fixed skeleton but certain steps vary
    - *Example: Data parser (read/ process/save), where only the "process" step changes depending on data format.*
    - *Real world: A cake recipe: fixed overall steps (mix, bake, cool), but specific parts (flavor, frosting) are filled in by the baker.*

Behavioral Patterns

- Command
  - Used when you want to encapsulate actions as objects
    - *Example: Undo/redo functionality in text editors (each action = a command object)*
    - *Real world: Writing down an order in a restaurant instead of telling the cook what to do directly*
- Iterator
  - Used when you want to traverse a collection without exposing its internal structure
  - An iterator object encapsulates the “what’s next” logic
    - *Example: Iterating through a playlist of songs, regardless of whether it’s stored in an array, list, or database*
    - *Real world: A triage nurse deciding who gets to see the doctor next in an emergency room*

Behavioral Patterns

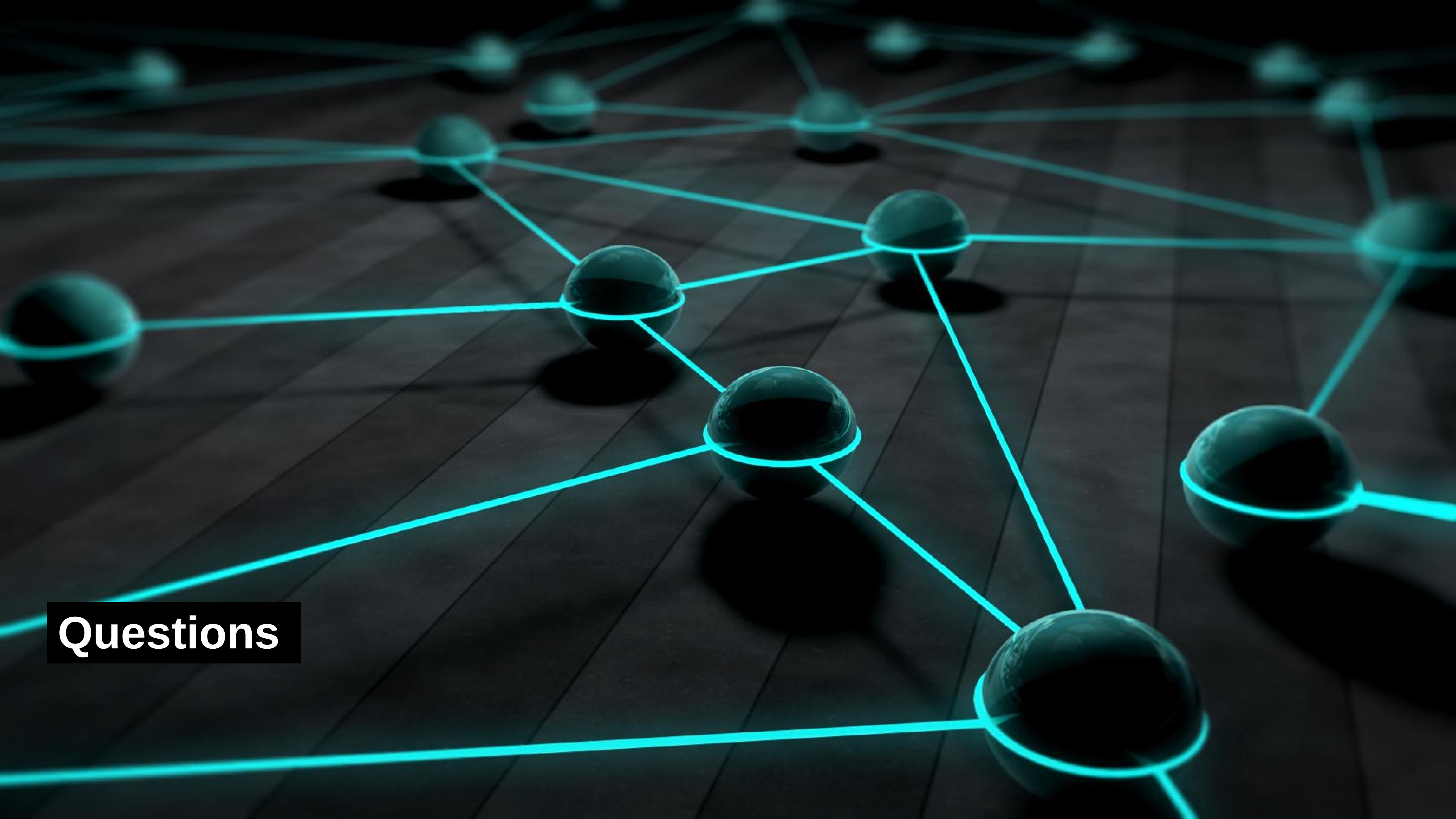
- Interpreter
  - Used when you need to evaluate expressions in a simple domain-specific language (DSL) with a well-defined grammar
    - *Example: Parsing and executing filter rules like status = "open" AND priority > 2 in a query feature*
    - *Real world: Musicians read musical notation and plays accordingly following shared grammar/rules*
- Mediator
  - Used to reduce tight coupling by centralizing how multiple objects/components communicate with each other
    - *Example: A comments system that allows users to communicate through a central location*
    - *Real world: An air-traffic controller coordinating pilots so planes don’t talk directly to each other*

Behavioral Patterns

- Memento
  - Used to capture and restore an object’s internal state without exposing its internals
    - *Example: A text editor saving snapshots of the document so users can undo/redo edits safely.*
    - *Real world: A saved photo of a table setting that lets you restore it to its original state*
- Visitor
  - Used to add new operations to a complex object structure without modifying the classes of the elements being operated on
    - *Example: Running different analyses such as pretty-printing, linting or type-checking on source code*
    - *Real world: A CPA who does tax reporting for different clients*

Behavioral Patterns

- Chain of Responsibility
  - Used to decouple request senders from receivers by passing a request through a sequence of handlers so the first capable one processes it (or it’s dropped)
    - *Example: An HTTP request moves through middleware (logging - authentication - authorization - rate limiting - handler), with each step handling or forwarding the request*
    - *Real world: Customer support escalation: frontline rep - specialist - manager; each in turn tries to handle your request until someone can*
- State
  - Used when an object’s behavior must change based on its internal state, replacing complex if/else or switch logic with state-specific objects and clear transitions
    - *Example: A media player where the same buttons (Play/Pause/Stop) act differently in Stopped, Playing, or Paused states, and each state object controls the next transition*
    - *Real world: A turnstile: “locked” requires a coin, “unlocked” lets you pass; same device, different behavior depending on current state*



Questions