1. Singleton

Intent: Ensure a class has only one instance and provide a global point of access to it.

Key Focus: Managing global state while preventing multiple instantiations (e.g., lazy initialization, thread safety).

2. Observer (or Publish-Subscribe)

Intent: Define a one-to-many dependency between objects so that when one object changes state, all its dependents are notified automatically.

Key Focus: Decoupling subjects (observables) from observers (listeners) to enable dynamic event handling.

3. Strategy

Intent: Define a family of algorithms, encapsulate each one, and make them interchangeable, allowing the algorithm to vary independently from clients that use it.

Key Focus: Enabling runtime algorithm selection via composition rather than inheritance.

4. Factory Method

Intent: Define an interface for creating an object but let subclasses decide which class to instantiate, promoting loose coupling.

Key Focus: Delegating object creation to subclasses to support extensibility.

5. Abstract Factory

Intent: Provide an interface for creating families of related or dependent objects without specifying their concrete classes.

Key Focus: Ensuring consistency among related object types (e.g., UI themes for different platforms).

6. Adapter

Intent: Allow incompatible interfaces to work together by wrapping one interface to match another.

Key Focus: Bridging legacy or third-party code with modern systems without modification.

7. Decorator

Intent: Attach additional responsibilities to an object dynamically without altering its structure.

Key Focus: Enhancing functionality at runtime via composition (e.g., Java I/O streams).

8. Command

Intent: Encapsulate a request as an object, enabling parameterization, queuing, logging, and undo/redo operations.

Key Focus: Decoupling invocation from execution (e.g., GUI button actions or transaction systems).

9. State

Intent: Allow an object to alter its behavior when its internal state changes, appearing as if the object changed its class.

Key Focus: Managing state transitions explicitly to avoid conditional logic sprawl.

10. Template Method

Intent: Define the skeleton of an algorithm in a method, deferring some steps to subclasses to enable customization.

Key Focus: Enforcing algorithm structure while allowing subclass-specific variations.

11. Builder

Intent: Separate the construction of a complex object from its representation, enabling the same construction process to create different representations.

Key Focus: Simplifying object creation with step-by-step assembly (e.g., `StringBuilder` or `AlertDialog.Builder`).

12. Prototype

Intent: Specify the kinds of objects to create using a prototypical instance, then clone new objects by copying this prototype.

Key Focus: Avoiding expensive initialization by reusing existing objects (e.g., deep copying).

13. Proxy

Intent: Provide a surrogate or placeholder for another object to control access to it (e.g., lazy loading, access control).

Key Focus: Adding pre- or post-processing logic without modifying the original object.

14. Chain of Responsibility

Intent: Pass a request along a chain of handlers, each deciding whether to process it or pass it to the next handler.

Key Focus: Decoupling senders from receivers to support dynamic handler chains (e.g., event bubbling).

15. Mediator

Intent: Reduce direct communication between objects by introducing a central mediator that handles interactions.

Key Focus: Centralizing complex communication logic to avoid tight coupling (e.g., chat room participants).

16. Memento

Intent: Capture and externalize an object's state so it can be restored later without exposing its internal structure.

Key Focus: Enabling undo/redo functionality while maintaining encapsulation.

17. Iterator

Intent: Provide a way to access elements of an aggregate object sequentially without exposing its underlying representation.

Key Focus: Unifying traversal logic across different collection types (e.g., `List` vs. `Set`).

18. Flyweight

Intent: Use sharing to support large numbers of fine-grained objects efficiently by reusing existing instances.

Key Focus: Minimizing memory usage via shared immutable state (e.g., font glyphs in text editors).

---

1. Single Responsibility Principle (SRP)

Core Intent: A class should have only one reason to change (i.e., it should handle only one function).

Key Practices:

- Avoid "god classes" (classes with excessive responsibilities) by splitting functionality to reduce coupling.
- Define responsibilities based on business needs, not technical implementation (e.g., separate logging or validation logic).
- **Benefits**: Improves maintainability (changes in one function don’t affect others) and testability (single responsibilities are easier to unit test).
- **Example**: A `UserManager` class should not handle both database operations and email sending; the latter should be delegated to an `EmailService`.

---

2. Open-Closed Principle (OCP)

Core Intent: Software entities (classes, modules, etc.) should be open for extension but closed for modification.

Key Practices:

- Use abstraction (interfaces/abstract classes) to define stable behaviors and enable extension via inheritance or polymorphism.
- Avoid modifying existing code directly; instead, adapt to changes by adding new classes or configurations (e.g., Strategy Pattern).
- **Benefits**: Reduces the risk of introducing new bugs and lowers maintenance costs.
- **Example**: A payment system supporting multiple methods (credit card, Alipay) should allow adding new methods by implementing a `PaymentGateway` interface without altering existing code.

---

3. Liskov Substitution Principle (LSP)

Core Intent: Subclasses must be substitutable for their parent classes without affecting program correctness.

Key Practices:

- Subclasses should not override parent methods in ways that violate expected behavior (e.g., a parent `sort()` ascending but a subclass sorts descending).
- Follow contract design: Subclasses must honor parent preconditions (input constraints) and postconditions (output guarantees).

- **Benefits**: Prevents hidden errors caused by inheritance and enhances code robustness.
- **Example**: If `Square` inherits from `Rectangle`, forcing `setWidth()` to also modify height violates LSP; composition should be used instead of inheritance.

---

4. Interface Segregation Principle (ISP)

Core Intent: Clients should not be forced to depend on interfaces they don't need; interfaces should be "small and focused."

Key Practices:

- Split bloated interfaces into multiple smaller ones, each containing only methods required by clients.
- Avoid "fat interfaces" that force implementations to include irrelevant methods (e.g., a printer interface with scanning functionality).
- **Benefits**: Reduces unnecessary coupling and improves flexibility.
- **Example**: Split a `Worker` interface into `TaskWorker` (task handling) and `ReportWorker` (report generation), allowing roles to implement only relevant interfaces.

---

5. Dependency Inversion Principle (DIP)

Core Intent: High-level modules should not depend on low-level modules; both should depend on abstractions. Abstractions should not depend on details; details should depend on abstractions.

Key Practices:

- Decouple modules via dependency injection (DI) or interface abstraction (e.g., Spring's `@Autowired`).
- Avoid hardcoding concrete implementations (e.g., directly instantiating `DatabaseConnection`); instead, depend on interfaces or abstract classes.
- **Benefits**: Lowers module coupling and improves replaceability (e.g., switching databases by modifying configurations).
- **Example**: The data persistence layer should depend on a `Repository` interface, not concrete implementations like `MySQLRepository` or `MongoDBRepository`.

---

6. Law of Demeter (LoD)

Core Intent: An object should interact with as few other objects as possible ("only talk to immediate friends").

Key Practices:

- Avoid accessing deep properties through intermediate objects (e.g., `a.getB().getC().doSomething()`); instead, use direct calls or expose methods.
- Limit a class's "friends" to itself, parameters, member variables, or objects it creates.
- **Benefits**: Reduces inter-class dependencies and minimizes the risk of ripple effects from changes.
- **Example**: An `Order` class should not directly manipulate a customer's address (`Customer.getAddress().getCity()`); instead, use a method like `Customer.getShippingCity()`.

Summary

- **SOLID** focuses on robust and scalable class design, while **LoD** emphasizes minimizing inter-class coupling.
- These principles are often used together (e.g., DIP + SRP for decoupling, ISP + LoD for interface optimization).
- Following them improves code quality but requires balancing against over-engineering (e.g., excessive class splitting increasing complexity).

Please note that the key focuses of this exam are the Decorator Pattern, Observer Pattern, Adapter Pattern, Abstract Factory Pattern, and State Pattern. Be sure to memorize the class diagrams and code examples for these patterns.

(Good luck to you all!)