Design Patterns

Modular Architecture, Behavioral Decoupling, and Enterprise-Ready Abstractions

There are **Creational Patterns**, **Structural Patterns** and **Behavioral Patterns**.

🧱 1. Creational Patterns

Creational patterns abstract **how objects are created**, providing flexibility and decoupling in object instantiation.

1.1 🔒 Singleton

📌 Definition:

Ensures a class has **exactly one instance** and provides a **global point of access** to it.

🧩 Spring Boot Application:

Spring's `ApplicationContext` manages beans as **singletons by default**, eliminating the need for manual singleton logic.

```
@Service
public class AppConfigService {
    // Singleton by default
}
```

🧠 Considerations:

- Avoid `synchronized` blocks — let Spring manage lifecycle.
- Be careful with `static` state — it is not tracked by the container.
- Use for shared resources: configuration, logging, application metrics.

1.2 🏭 Factory Method

📌 Definition:

Defines an interface for creating objects, but lets **subclasses or configuration** decide which class to instantiate.

🧩 Spring Boot Application:

```
@Configuration
public class AlertFactory {

    @Bean
    public AlertService alertService() {
        return isProd() ? new SlackAlertService() : new LogAlertService();
    }
}
```

🧠 Considerations:

- Integrates well with `@Profile`, `@ConditionalOnProperty`, and custom conditions.
- Enables **strategy injection** or **plugin-based architecture**.
- Factory methods make unit testing easier via bean substitution.

1.3 🧰 Abstract Factory

📌 Definition:

Provides an interface to create families of related objects **without specifying their concrete classes**.

🧩 Spring Boot Application:

```
public interface NotificationFactory {
    NotificationSender createSender();
    NotificationFormatter createFormatter();
}

@Component
@Profile("sms")
public class SmsNotificationFactory implements NotificationFactory {
    ...
}
```

🧠 Considerations:

- Use when objects vary **in groups** by environment, feature, or context.
- Often paired with `@Profile`, custom `Condition`, or factory beans.
- Enables complete switching of infrastructure with zero client changes.

1.4 🏗️ Builder

📌 Definition:

Separates object **construction from representation**, especially useful for **complex objects** or **immutable structures**.

🧩 Spring Boot Application:

```
@Builder
public class UserRegistrationRequest {
    private String name;
    private String email;
    private LocalDate birthDate;
}
```

🧠 Considerations:

- Use `@Builder` (Lombok) for DTOs, responses, and test fixtures.
- Avoid telescoping constructors in favor of fluent builders.
- Improves test readability and immutability.

1.5 🧬 Prototype

📌 Definition:

Creates new object instances by **cloning an existing instance**.

🧩 Spring Boot Application:

```
@Bean
@Scope(ConfigurableBeanFactory.SCOPE_PROTOTYPE)
public ReportBuilder reportBuilder() {
    return new ReportBuilder();
}
```

🧠 Considerations:

- Useful when object instantiation is **expensive** and **non-shared**.
- Applies to `ThreadLocal` scope or per-request bean instances.
- Beware: prototype beans are **not automatically garbage collected** if leaked from singleton scope.

Here's the enhanced and complete documentation including all missing structural patterns from your provided image:

🧩 2. Structural Patterns

Structural patterns define how classes and objects are **composed** to form larger, more flexible structures.

2.1 🔌 Adapter

📌 Definition:

Allows objects with **incompatible interfaces** to collaborate by converting the interface of one class into another expected by clients.

🧩 Spring Boot Application:

```
@Component
public class StripeAdapter implements PaymentProcessor {
    private final StripeClient stripe;

    public PaymentResponse process(PaymentRequest req) {
```

```
        return stripe.charge(req.toStripePayload());
    }
}
```

🧠 Considerations:

- Integrates external or legacy systems.
- Promotes **hexagonal architecture**.
- Simplifies testing by abstracting external dependencies.

2.2 🌉 Bridge

📌 Definition:

Splits a **large class or closely related classes** into two separate hierarchies—**abstraction and implementation**—which evolve independently.

🧩 Spring Boot Application:

```
public interface MessageSender {
    void send(String content);
}

@Service
public class NotificationService {
    private final MessageSender sender;

    public void notifyUser(String msg) {
        sender.send(msg);
    }
}
```

🧠 Considerations:

- Separates business logic from platform-specific implementation.
- Useful when **implementations change more frequently** than abstractions.

2.3 🌳 Composite

📌 Definition:

Allows you to compose objects into **tree structures** and work with these structures as if they were individual objects.

🧩 Spring Boot Application:

```
public interface MenuComponent {
    void render();
}

public class MenuItem implements MenuComponent {
    public void render() { /* render single item */ }
}

public class Menu implements MenuComponent {
    private List<MenuComponent> items;

    public void render() {
        items.forEach(MenuComponent::render);
    }
}
```

🧠 Considerations:

- Enables uniform treatment of individual and composite objects.
- Useful for nested structures like menus, directory hierarchies, or UI widgets.

2.4 🎨 Decorator

📌 Definition:

Lets you dynamically attach new **behaviors** to objects by placing them into special wrapper objects.

🧩 Spring Boot Application:

```
@Component
@Primary
public class LoggingInvoiceService implements InvoiceService {
    private final InvoiceService delegate;

    public void generate(Invoice i) {
        log.debug("Generating invoice");
        delegate.generate(i);
    }
}
```

🧠 Considerations:

- Adds cross-cutting concerns like logging, caching, or metrics.

- Preferable over inheritance for flexible runtime composition.

2.5 🏛️ Facade *(Newly added from image)*

📌 Definition:

Provides a simplified interface to a **complex subsystem**, library, or framework, reducing complexity for clients.

🧩 Spring Boot Application:

```
@Service
public class OrderFacade {
    private final PaymentService paymentService;
    private final InventoryService inventoryService;

    public void placeOrder(Order order) {
        paymentService.process(order.getPaymentInfo());
        inventoryService.reserve(order.getItems());
    }
}
```

🧠 Considerations:

- Simplifies client interactions with complex subsystems.
- Reduces coupling between subsystems and clients.

2.6 ⚖️ Flyweight *(Newly added from image)*

📌 Definition:

Optimizes memory use by sharing **common state** among multiple objects rather than storing it repeatedly.

🧩 Spring Boot Application:

```
@Component
public class IconFactory {
    private final Map<String, Icon> cache = new HashMap<>();

    public Icon getIcon(String type) {
        return cache.computeIfAbsent(type, Icon::new);
    }
}
```

🧠 Considerations:

- Useful for memory-intensive applications.
- Effective when objects share considerable amounts of data.

2.7 🪞 Proxy

📌 Definition:

Provides a **placeholder** for another object to control access to it, for purposes such as lazy loading, security checks, logging, or transaction handling.

🧩 Spring Boot Application:

```
@Transactional
public void processOrder() {
    // Transaction proxy automatically manages transaction boundaries
}
```

🧠 Considerations:

- Integral to Spring AOP mechanisms (`@Transactional` , `@Cacheable` , etc.).
- Controls and manages resource access transparently.

🔁 3. Behavioral Patterns

Behavioral patterns focus on **communication between objects**, defining how responsibilities are distributed, how algorithms are encapsulated, and how interactions occur.

3.1 📣 Observer

📌 Definition:

Defines a **one-to-many dependency** so that when one object changes state, all its dependents are notified automatically.

🧩 Spring Boot Application:

```
@EventListener
public void handle(UserRegisteredEvent event) {
    sendWelcomeEmail(event.getUser());
}
```

🧠 Considerations:

- Decouples event emitters from handlers.
- Integrates with `ApplicationEventPublisher`.
- Enables async behavior with `@Async` listeners.

3.2 🧠 Strategy

📌 Definition:

Defines a **family of interchangeable algorithms**, encapsulates each, and allows them to be selected at runtime.

🧩 Spring Boot Application:

```
@Service
public class CheckoutService {
    private final Map<String, DiscountStrategy> strategies;

    public BigDecimal calculate(String type, BigDecimal amount) {
        return strategies.get(type).applyDiscount(amount);
    }
}
```

🧠 Considerations:

- Replaces cumbersome `if-else` structures.
- Supports open/closed principle (easily extendable).
- Commonly used in payment gateways, discount mechanisms, algorithms selection.

3.3 📜 Template Method

📌 Definition:

Defines the **skeleton of an algorithm**, deferring certain implementation steps to subclasses.

🧩 Spring Boot Application:

```
public abstract class DataImporter {
    public final void importData(String path) {
        validate(path);
        parse(path);
        persist();
    }

    protected abstract void validate(String path);
```

```
    protected abstract void parse(String path);
    protected abstract void persist();
}
```

🧠 Considerations:

- Enforces algorithm structure consistently.
- Facilitates subclass-specific variations (e.g., different data formats).
- Commonly applied in frameworks and ETL processes.

3.4 📦 Command

📌 Definition:

Turns a **request into a stand-alone object**, encapsulating all relevant information to execute the request later, support undo operations, or schedule its execution.

🧩 Spring Boot Application:

```
public interface Command {
    void execute();
}

@Component
public class EmailCommand implements Command {
    public void execute() {
        emailService.sendEmail();
    }
}
```

🧠 Considerations:

- Ideal for job scheduling, undo functionality, transactional systems.
- Easily serialized for deferred processing or audit trails.

3.5 🔗 Chain of Responsibility

📌 Definition:

Allows you to pass requests **along a chain of handlers**. Each handler decides to either process the request or forward it along the chain.

🧩 Spring Boot Application:

```
@Component
public class AuthenticationFilter implements Filter {
    private final Filter nextFilter;

    public void handle(Request req) {
        if (authenticated(req)) nextFilter.handle(req);
        else reject(req);
    }
}
```

🧠 Considerations:

- Common in middleware, request validation, and error handling.
- Promotes loose coupling between sender and receiver.

3.6 🔄 Iterator

📌 Definition:

Allows sequential access to elements in a collection without exposing its underlying representation.

🧩 Spring Boot Application:

```
public interface Iterator<T> {
    boolean hasNext();
    T next();
}
```

🧠 Considerations:

- Abstracts traversal logic, supports different underlying structures.
- Built into Java (`Iterator`, enhanced for-loops).

3.7 💾 Memento

📌 Definition:

Enables capturing and restoring an object's internal state without exposing internal details.

🧩 Spring Boot Application:

```
public class Editor {
    private String text;
```

```
    public EditorMemento save() {
        return new EditorMemento(text);
    }

    public void restore(EditorMemento memento) {
        text = memento.getSavedText();
    }
}
```

🧠 Considerations:

- Supports undo mechanisms, snapshots.
- Useful in editor applications, transactional rollback scenarios.

3.8 ⚖️ Mediator

📌 Definition:

Reduces direct dependencies between objects by forcing communication via a central **mediator object**.

🧩 Spring Boot Application:

```
@Component
public class ChatRoomMediator {
    public void sendMessage(User user, String msg) {
        // Distribute messages to other participants
    }
}
```

🧠 Considerations:

- Simplifies complex object interactions.
- Centralizes communication logic.
- Often applied in GUI frameworks, chat applications, or workflows.

3.9 🌡️ State

📌 Definition:

Allows an object to **change its behavior dynamically** based on its internal state. It appears as if the object has changed its class.

🧩 Spring Boot Application:

```
public class Order {
    private OrderState state;

    public void proceed() {
        state.handle(this);
    }

    public void setState(OrderState state) {
        this.state = state;
    }
}
```

🧠 Considerations:

- Avoids large conditional state management code.
- Useful in order processing, workflow management, and state machines.

3.10 🧳 Visitor

📌 Definition:

Separates algorithms from the objects on which they operate by moving the algorithm into a separate class.

🧩 Spring Boot Application:

```
public interface ReportElement {
    void accept(Visitor visitor);
}

public class SalesReport implements ReportElement {
    public void accept(Visitor visitor) {
        visitor.visit(this);
    }
}
```

🧠 Considerations:

- Useful for operations performed on composite object structures.
- Commonly applied in parsing, AST processing, reporting, and validation scenarios.

✅ Summary: Design Pattern Matrix

Pattern Type	Pattern	Core Benefit	Spring Example
Creational	Singleton	One instance, lifecycle managed	`@Component`, `@Service`
	Factory Method	Centralize object creation	`@Bean` methods, `@Configuration`
	Abstract Factory	Families of related objects	`@Profile`, strategy factories
	Builder	Complex object construction	Lombok `@Builder`, HTTP clients
	Prototype	Cloning with isolation	`@Scope("prototype")` beans
Structural	Adapter	Interface bridging	CRM/ERP API integration
	Decorator	Dynamic behavior injection	Logging wrappers
	Proxy	Access control, lifecycle	`@Transactional`, AOP
	Composite	Hierarchical uniform APIs	Menu trees, permission hierarchies
	Bridge	Abstraction/implementation separation	Messaging, external service abstraction
Behavioral	Observer	Event-based decoupling	`@EventListener`, `ApplicationEvent`
	Strategy	Pluggable, dynamic behavior	`Map<String, Strategy>` injection
	Template Method	Workflow skeleton	Data import, test bases
	Command	Encapsulated request logic	Async workers, message handlers

📚 References & Further Reading

1. **Design Patterns: Elements of Reusable Object-Oriented Software** — Erich Gamma, Richard Helm, Ralph Johnson, John Vlissides (Gang of Four)
   The definitive source for creational, structural, and behavioral patterns, with foundational concepts applied across languages and frameworks.
2. **Effective Java (3rd Edition)** — Joshua Bloch
   A must-read for Java developers covering idiomatic use of patterns like Builder, Singleton, and Factory in a modern Java context.
3. **Spring Framework Reference Documentation**
   https://docs.spring.io/spring-framework/docs/current/reference/html/

The official guide that explains Spring's use of proxy, factory, AOP, and dependency injection mechanisms.

4. **Refactoring Guru – Design Patterns Explained Simply**
   https://refactoring.guru/design-patterns
   A visual and beginner-friendly resource for understanding the intent, structure, and use-cases of each pattern.
5. **Spring Boot Design Patterns** — Dinesh Rajput (Book)
   Practical applications of common design patterns using Spring Boot, including Singleton, Factory, Template Method, Strategy, and more.
6. **Martin Fowler – Patterns of Enterprise Application Architecture**
   A broader look at enterprise architecture patterns like Repository, Service Layer, and Transaction Script, frequently reflected in Spring Boot design.
7. **Head First Design Patterns (Updated for Java 8)** — Eric Freeman & Elisabeth Robson
   An engaging introduction to object-oriented design and the patterns behind reusable, testable components.
8. **Spring AOP and Proxy Pattern Use Cases**
   https://www.baeldung.com/spring-aop
   Learn how Spring uses the Proxy and Decorator patterns internally for cross-cutting concerns like transactions and logging.
9. **Clean Architecture: A Craftsman's Guide to Software Structure and Design** — Robert C. Martin
   A deep dive into abstraction, modular boundaries, and the role of interfaces and dependency inversion in scalable systems.
10. **Java Design Patterns (TutorialsPoint)**
    https://www.tutorialspoint.com/design_pattern/index.htm
    Lightweight summaries of each pattern with example code in Java.