PARTE 1: Accoppiamento e Astrazione - Il Cuore del Design

Perché Esistono Questi Concetti

Il software evolve. Il codice che scrivi oggi verrà modificato domani. **L'accoppiamento determina quanto costa ogni modifica**.

Accoppiamento: Definizione Operativa

```
// ALTO ACCOPPIAMENTO - fragile
class GestoreOrdini {
    public void processaOrdine(Ordine o) {
        MySQLDatabase db = new MySQLDatabase();
        db.connect("localhost", "user", "pass");
        db.save(o);

        EmailSMTP email = new EmailSMTP();
        email.configura("smtp.gmail.com", 587);
        email.invia(o.getCliente(), "Ordine confermato");
    }
}
```

Problema: cambiare database da MySQL a PostgreSQL richiede modificare `GestoreOrdini`. Passare da email a SMS? Altra modifica. Testare senza DB reale? Impossibile.

Costo: ogni dipendenza concreta è un punto di rottura.

```
// BASSO ACCOPPIAMENTO - flessibile
class GestoreOrdini {
    private RepositoryOrdini repository;
    private NotificatoreClienti notificatore;

    public GestoreOrdini(RepositoryOrdini repo, NotificatoreClienti notif) {
        this.repository = repo;
        this.notificatore = notif;
    }

    public void processaOrdine(Ordine o) {
        repository.salva(o);
        notificatore.notifica(o.getCliente(), "Ordine confermato");
    }
}
```

```
interface RepositoryOrdini {
    void salva(Ordine o);
}

interface NotificatoreClienti {
    void notifica(Cliente c, String messaggio);
}
```

Beneficio: `GestoreOrdini` non sa se usa MySQL, MongoDB, o mock in memoria. Non sa se notifica via email, SMS, o push. **Dipende da astrazioni, non da implementazioni**.

Astrazione: Il Contratto Stabile

L'astrazione è **l'invariante** che resiste al cambiamento.

```
// CATTIVA astrazione - troppo specifica
interface InviatoreMail {
    void inviaTramiteSMTP(String destinatario, String oggetto, String
corpo);
}
// Se domani serve REST API o message queue? Interfaccia inadeguata.

// BUONA astrazione - invariante del dominio
interface Notificatore {
    void notifica(Destinatario dest, Messaggio msg);
}
// L'invariante è: "inviare un messaggio a un destinatario"
// Il "come" (SMTP, HTTP, SMS) è dettaglio implementativo
```

Legge Fondamentale

> **Dipendi dall'astrazione più stabile possibile.**
>
> Stabilità = quanto raramente cambia il contratto, non l'implementazione.

PARTE 2: Cosa Sono Davvero i Pattern

Pattern: Soluzione Ricorrente a Problema Ricorrente in Contesto

Non sono:

- ❌ Ricette da copiare
- ❌ Garanzia di codice "giusto"
- ❌ Obiettivo del design

Sono:

- ✅ **Vocabolario condiviso**: "usa un Observer" dice più di 50 righe di spiegazione
- ✅ **Soluzione dimostrata**: qualcuno ha già pagato il costo dell'errore
- ✅ **Trade-off documentati**: sappiamo quando funzionano e quando no

Esempio Concreto: Strategy Pattern

Problema ricorrente: algoritmo che varia indipendentemente dal contesto d'uso.

```
// SENZA pattern - logica condizionale sparsa
class Calcolatrice {
    public double calcola(Ordine ordine, String tipoSconto) {
        double totale = ordine.getTotale();

        if (tipoSconto.equals("BLACK_FRIDAY")) {
            return totale * 0.7;
        } else if (tipoSconto.equals("CLIENTE_FEDELE")) {
            return totale - 10;
        } else if (tipoSconto.equals("PRIMO_ORDINE")) {
            return totale * 0.9;
        }
        return totale;
    }
}
// Aggiungere uno sconto = modificare Calcolatrice (Open/Closed violato)
```

```
// CON Strategy - variazione isolata
interface StrategiaScontistica {
    double applica(double totale);
}

class ScontoBlackFriday implements StrategiaScontistica {
    public double applica(double totale) { return totale * 0.7; }
}

class ScontoClienteFedele implements StrategiaScontistica {
    public double applica(double totale) { return totale - 10; }
}

class Calcolatrice {
    private StrategiaScontistica strategia;

    public void setStrategia(StrategiaScontistica s) {
        this.strategia = s;
    }

    public double calcola(Ordine ordine) {
```

```
        return strategia.applica(ordine.getTotale());
    }
}
```

Risultato:

- Nuovo sconto = nuova classe, zero modifiche esistenti
- Testabile: mock della strategia
- Combinabile: catena di sconti = Decorator su Strategy

PARTE 3: Tassonomia Pattern - Perché Cardin Distingue

Pattern a Livelli Diversi

```
ARCHITETTURALI (struttura sistema)
    ├─ Layered Architecture
    ├─ Microservices
    ├─ Event-Driven
    └─ Hexagonal (Ports & Adapters)
         ↓ definiscono macro-organizzazione

DESIGN (struttura componenti)
    ├─ Creazionali (Factory, Builder, Singleton)
    ├─ Strutturali (Adapter, Decorator, Proxy)
    └─ Comportamentali (Strategy, Observer, Template)
         ↓ definiscono relazioni tra classi

IDIOMATICI (linguaggio-specifici)
    ├─ RAII (C++)
    ├─ Context Manager (Python)
    └─ Optional chaining (linguaggi moderni)
```

Dependency Injection: Pattern Architetturale

Perché architetturale e non design?

Perché risolve un problema di **organizzazione dell'intero sistema**, non di singole classi.

Il Problema che Risolve

```
// SENZA DI - accoppiamento transitivo
class Controller {
    private Service service = new Service();
}
```

```
class Service {
    private Repository repo = new Repository();
}

class Repository {
    private Database db = new MySQLDatabase();
}
```

Problema:

- `Controller` dipende transitivamente da `MySQLDatabase`
- Testare `Controller` richiede DB reale
- Configurazione sparsa in ogni classe
- Impossible sostituire implementazioni a runtime

```
// CON DI - inversione controllo
class Controller {
    private Service service;

    @Inject
    public Controller(Service service) {
        this.service = service;
    }
}

class Service {
    private Repository repo;

    @Inject
    public Service(Repository repo) {
        this.repo = repo;
    }
}

// Container DI (Spring, Guice, etc.)
Container container = new Container();
container.bind(Repository.class).to(MySQLRepository.class);
container.bind(Service.class).to(ServiceImpl.class);
container.bind(Controller.class).to(ControllerImpl.class);

// Risoluzione automatica grafo dipendenze
Controller ctrl = container.get(Controller.class);
```

Effetto architetturale:

- Configurazione centralizzata
- Dipendenze iniettate dall'esterno (Hollywood Principle: "don't call us, we'll call you")

- Sostituibilità globale (test: inietta mock; prod: inietta implementazioni reali)
- Gestione lifecycle (singleton, prototype, request-scoped)

Altri Pattern Architetturali

CQRS (Command Query Responsibility Segregation)

```
// Separa scrittura da lettura
interface CommandService {
    void creaOrdine(CreaOrdineCommand cmd);
    void annullaOrdine(AnnullaOrdineCommand cmd);
}

interface QueryService {
    OrdineDTO trovaOrdine(String id);
    List<OrdineDTO> ordiniCliente(String clienteId);
}

// Implementazioni possono usare DB diversi
// Comandi → DB transazionale normalizzato
// Query → DB denormalizzato ottimizzato lettura
```

Architetturale perché: cambia organizzazione persistenza, non singola classe.

Repository Pattern

```
interface RepositoryOrdini {
    Ordine trova(String id);
    void salva(Ordine ordine);
    List<Ordine> trovaTutti();
}

class RepositoryOrdiniJPA implements RepositoryOrdini {
    // implementazione con JPA
}
```

Architetturale perché:

- Isola logica dominio da persistenza
- Permette cambio strategia persistenza (SQL → NoSQL → cache)
- Definisce boundary tra layer applicativo e infrastrutturale

---

PARTE 4: Perché Non Richiesti in Esame

Pattern Architetturali vs Design: Differenza di Scope

Aspetto	Design Pattern	Architetturali
Scala	Classe/modulo	Sistema/subsistema
Decisione	Sviluppatore	Architetto/team
Reversibilità	Alta	Bassa
Impatto	Locale	Globale
Apprendimento	Codice	Esperienza progetti reali

Ragionamento di Cardin (ipotesi fondata)

Pattern Design (GoF):

- Insegnabili con esempi isolati
- Applicabili immediatamente in esercizi
- Verificabili in classe diagram
- Comprensibili senza aver visto grandi sistemi

Pattern Architetturali:

- Richiedono contesto di sistema complesso
- Trade-off comprensibili solo con esperienza (es: "quando CQRS vale la complessità?")
- Non verificabili in 2h d'esame
- Rischiano apprendimento mnemonico senza comprensione

In sintesi: pattern architetturali sono **preparazione professionale**, non materiale esaminabile in corso base.

PARTE 5: Sintesi Operativa

Accoppiamento - Checklist Pratica

Quando scrivi codice, chiedi:

1. **Se cambio X, quante classi devo modificare?** (X = DB, framework, libreria esterna)
2. **Posso testare questa classe senza dipendenze reali?**
3. **Le dipendenze sono interfacce o classi concrete?**

Astrazione - Euristica

Buona astrazione se:

- ✅ Rappresenta concetto di dominio, non tecnologia
- ✅ Stabile nel tempo (contratto non cambia spesso)

- ✅ Nasconde dettagli implementativi
- ❌ Non è "interfaccia perché sì" (no marker interface inutili)

Pattern - Quando Applicarli

```
1. Ho un problema ricorrente?
   NO → non serve pattern
   SÌ → vai a 2

2. Pattern noto risolve questo problema?
   NO → design ad-hoc
   SÌ → vai a 3

3. Complessità pattern < complessità problema?
   NO → soluzione più semplice
   SÌ → applica pattern
```

Anti-pattern: pattern hunting (cercare dove ficcare pattern, non problemi da risolvere).
