



NoiStudiamo

DISPENSA DI LABORATORIO



Ingegneria del Software

Laboratorio: codice Java commentato, test con JUnit, design pattern implementati ed esercizi svolti sugli stream

A CURA DI

Damiano La Pusata

Ottobre 2026

Indice dei capitoli

Ogni esempio è stato compilato ed eseguito; gli output sono quelli reali.

1. Dal metodo unico alla classe	4
2. Test manuali e test con JUnit	6
3. Refactoring passo per passo	10
4. Ereditarietà, polimorfismo e Template Method	13
5. Dal codice ai diagrammi UML	17
6. Singleton e Multiton	19
7. Factory Method, Object Pool, Dependency Injection e Abstract Factory	22
8. Prototype	27
9. Builder	29
10. Adapter	31
11. Facade	33
12. Stream: esercizi svolti	35
13. Composite	40
14. Decorator	42
15. Bridge	44
16. Observer e Publisher Subscriber	46
17. State	49

18. Mediator	52
19. Chain of Responsibility	55
20. Command	57
21. Test a partire dai requisiti	59
22. Riconoscere i design pattern dal codice	62

1. Dal metodo unico alla classe

La classe che legge le letture di un contatore, scritta con un metodo per ogni responsabilità: lettura da file, inserimento senza duplicati, somma, massimo.

Che cosa deve fare il programma

Un file di testo contiene una lettura per riga. Il programma deve leggere i valori, scartare i duplicati, calcolarne la somma e il massimo. Come spiegato nella dispensa di teoria, invece di un unico metodo che fa tutto si scrive una classe in cui ogni metodo ha un solo compito: è la differenza fra spaghetti code e codice orientato agli oggetti.

CONSUMI.JAVA

```
import java.io.File;
import java.io.FileReader;
import java.io.IOException;
import java.io.LineNumberReader;
import java.util.ArrayList;
import java.util.List;

// Tiene le letture di un contatore e calcola somma e picco
public class Consumi {
    private List<Float> letture = new ArrayList<>(); // valori letti, senza duplicati

    // legge un file con un valore per riga e inserisce ogni valore
    public void leggiFile(String cartella, String nome) throws IOException {
        LineNumberReader f = new LineNumberReader(new FileReader(new File(cartella, nome)));
        while (true) {
            String riga = f.readLine(); // null quando il file e' finito
            if (riga == null) break;
            inserisci(Float.parseFloat(riga)); // da testo a numero
        }
        f.close();
    }

    // inserisce un valore solo se non e' gia' presente
    public void inserisci(float x) {
        if (!letture.contains(x)) letture.add(x);
    }

    // query: restituisce la somma senza modificare lo stato
    public float calcolaSomma() {
        float somma = 0;
        for (float v : letture) somma += v; // enhanced for
        return somma;
    }
}
```

```
// query: restituisce il valore massimo (0 se non ci sono letture)
public float calcolaPicco() {
    if (letture.isEmpty()) return 0;
    float picco = letture.get(0);          // si parte dal primo valore
    for (float v : letture)
        if (v > picco) picco = v;
    return picco;
}
```

```
// command: svuota la lista
public void svuota() {
    letture = new ArrayList<>();
}
}
```

Alcuni dettagli da notare. Gli `import` in cima dicono dove trovare le classi di libreria. La lista è di `Float` e non di `stringhe`, così la conversione avviene una volta sola e i duplicati si riconoscono per valore. `calcolaSomma()` e `calcolaPicco()` sono **query**: restituiscono un valore e non modificano lo stato. Il massimo parte dal primo valore della lista e non da zero, così funziona anche con valori negativi; con la lista vuota restituisce 0.

MAIN.JAVA

```
import java.io.IOException;

public class Main {
    public static void main(String[] args) {
        Consumi c = new Consumi();
        try {
            c.leggiFile("dati", "gennaio.txt"); // il file contiene: 12, 7, 30, 7
        } catch (IOException e) {
            System.out.println("File non trovato");
        }
        System.out.println("somma: " + c.calcolaSomma());
        System.out.println("picco: " + c.calcolaPicco());
    }
}
```

Il file `dati/gennaio.txt` contiene i valori 12, 7, 30 e 7. Il secondo 7 è un duplicato e viene scartato, quindi la somma è 12 più 7 più 30, cioè 49, e il massimo è 30.

OUTPUT DELL'ESECUZIONE

```
somma: 49.0
picco: 30.0
```

Per compilare ed eseguire, dalla cartella che contiene i file: `javac *.java` e poi `java Main`.

DA RICORDARE

Un metodo per ogni responsabilità; metodi **query** che restituiscono valori senza modificare lo stato, metodi **command** che modificano lo stato senza restituire nulla. Il **for avanzato** scorre le collezioni senza indici.

2. Test manuali e test con JUnit

La stessa classe verificata due volte: prima con una classe di test scritta a mano, poi con JUnit, compresi i test parametrici e la configurazione Maven.

Test scritti a mano

Ogni metodo di test prepara dati noti, chiama l'operazione e confronta il risultato con quello atteso. Il metodo `preparaDati()`, chiamato all'inizio di ogni test, rende i test indipendenti. Le somme di float si confrontano con una tolleranza, mai con l'uguaglianza esatta.

TESTCONSUMI.JAVA

```
// Test scritti a mano, senza librerie: ogni test prepara i dati e controlla il risultato
public class TestConsumi {
    private final Consumi c = new Consumi();
```

```
    private void preparaDati() {           // dati noti, sempre uguali
        c.svuota();
        c.inserisci(12f);
        c.inserisci(7f);
        c.inserisci(30f);
    }
```

```
    public void testSomma() {
        preparaDati();
        if (Math.abs(c.calcolaSomma() - 49f) < 0.001) System.out.println("OK somma");
        else System.out.println("FAILED somma");
    }
```

```
    public void testPicco() {
        preparaDati();
        if (c.calcolaPicco() == 30f) System.out.println("OK picco");
        else System.out.println("FAILED picco");
    }
```

```
    public void testDuplicati() {
        preparaDati();
        c.inserisci(7f);           // duplicato: non deve cambiare la somma
        if (Math.abs(c.calcolaSomma() - 49f) < 0.001) System.out.println("OK duplicati");
        else System.out.println("FAILED duplicati");
    }
```

```
    public void testVuota() {           // caso limite: lista vuota
        c.svuota();
        if (c.calcolaSomma() == 0 && c.calcolaPicco() == 0) System.out.println("OK vuota");
        else System.out.println("FAILED vuota");
    }
```

```

    public static void main(String[] args) {
        TestConsumi t = new TestConsumi();
        t.testSomma();
        t.testPicco();
        t.testDuplicati();
        t.testVuota();
    }
}

```

OUTPUT DELL'ESECUZIONE

```

OK somma
OK picco
OK duplicati
OK vuota

```

Gli stessi test con JUnit

Con JUnit spariscono il main e le stampe: il motore trova i metodi annotati con `@Test`, esegue prima di ciascuno il metodo annotato con `@BeforeEach` e segnala i test falliti. Le asserzioni vengono da `Assertions`, importate con `import static`.

CONSUMITEST.JAVA

```

import static org.junit.jupiter.api.Assertions.assertEquals;
import org.junit.jupiter.api.BeforeEach;
import org.junit.jupiter.api.Test;

```

```

// La stessa verifica con JUnit: niente main, niente stampe
public class ConsumiTest {
    private final Consumi c = new Consumi();

```

```

    @BeforeEach                                // eseguito prima di ogni test
    public void prepara() {
        c.svuota();
        c.inserisci(12f);
        c.inserisci(7f);
        c.inserisci(30f);
    }

```

```

    @Test
    public void somma() {
        assertEquals(49f, c.calcolaSomma(), 0.001f); // atteso, effettivo, tolleranza
    }

    @Test
    public void picco() {
        assertEquals(30f, c.calcolaPicco(), 0.001f);
    }

```

```

@Test
public void duplicatoIgnorato() {
    c.inserisci(7f);
    assertEquals(49f, c.calcolaSomma(), 0.001f);
}

@Test
public void listaVuota() {
    c.svuota();
    assertEquals(0f, c.calcolaSomma(), 0.001f);
    assertEquals(0f, c.calcolaPicco(), 0.001f);
}
}

```

Test parametrici

Un test parametrico viene eseguito una volta per ogni riga di `@CsvSource`; i valori di ogni riga vengono assegnati ai parametri del metodo, nell'ordine.

CALCOLATRICE.JAVA

```

public class Calcolatrice {
    public int somma(int a, int b) {
        return a + b;
    }
}

```

CALCOLATRICETEST.JAVA

```

import static org.junit.jupiter.api.Assertions.assertEquals;
import org.junit.jupiter.params.ParameterizedTest;
import org.junit.jupiter.params.provider.CsvSource;

public class CalcolatriceTest {
    private final Calcolatrice calc = new Calcolatrice();

    @ParameterizedTest // eseguito una volta per riga
    @CsvSource({ "2, 3, 5", "10, 5, 15", "0, 0, 0", "-4, 4, 0" })
    void somma(int a, int b, int atteso) {
        assertEquals(atteso, calc.somma(a, b));
    }
}

```

Il motore esegue quattro volte il metodo: con 2 e 3 si attende 5, con 10 e 5 si attende 15, con 0 e 0 si attende 0, con meno 4 e 4 si attende 0. Insieme ai quattro test di `ConsumiTest`, gli esiti sono otto test superati.

Il progetto Maven

In un progetto Maven le classi stanno in `src/main/java` e i test in `src/test/java`. Nel `pom.xml` si dichiarano le dipendenze di JUnit con ambito test. Il comando `mvn test` compila ed esegue tutti i test.

ESTRATTO DI POM.XML

```
<dependencies>
  <dependency>
    <groupId>org.junit.jupiter</groupId>
    <artifactId>junit-jupiter</artifactId>    <!-- API, motore e test parametrici -->
    <version>5.11.3</version>
    <scope>test</scope>                      <!-- usata solo per i test -->
  </dependency>
</dependencies>
```

L'artefatto `junit-jupiter` raccoglie API, motore e supporto ai test parametrici. Per JUnit 6, la versione più recente, si indica il numero di versione corrispondente; il codice dei test resta identico.

DA RICORDARE

Test **indipendenti** (dati ricreati prima di ogni test) e **autovalutanti**. In JUnit: `@Test`, `@BeforeEach`, `assertEquals(atteso, effettivo, tolleranza)`, `@ParameterizedTest` con `@CsvSource`.

3. Refactoring passo per passo

Un metodo lungo trasformato con Estrai Metodo, Sostituisci Temp con Query e Dividi Variabile Temp, verificando che l'output resti identico.

Il punto di partenza

Il metodo `stampa()` fa tutto: intestazione, calcolo del totale, sconto, stampa. I commenti segnalano i frammenti da estrarre, e la variabile `temp` viene usata con due significati diversi: prima il totale, poi l'importo scontato.

RICEVUTAPRIMA.JAVA

```
import java.util.List;

// Prima del refactoring: un metodo lungo con commenti e variabili temporanee
public class RicevutaPrima {
    private final String cliente;
    private final List<Double> prezzi;

    public RicevutaPrima(String cliente, List<Double> prezzi) {
        this.cliente = cliente;
        this.prezzi = prezzi;
    }

    public void stampa() {
        // intestazione
        System.out.println("*** RICEVUTA ***");
        // calcolo del totale
        double temp = 0;
        for (double p : prezzi) temp += p;
        // sconto
        double sconto;
        if (temp > 100) sconto = 0.90;
        else sconto = 1.0;
        System.out.println("cliente: " + cliente);
        System.out.println("totale: " + temp);
        temp = temp * sconto; // temp riusata con un altro significato
        System.out.println("da pagare: " + temp);
    }
}
```

I passi del refactoring

1. **Dividi Variabile Temp**: il secondo uso di `temp` diventa una variabile con un nome proprio, l'importo da pagare.
2. **Sostituisci Temp con Query**: il totale e lo sconto diventano metodi, `totale()` e `sconto()`, che qualunque metodo può chiamare; anche l'importo da pagare diventa la query `daPagare()`.
3. **Estrai Metodo**: senza più variabili locali da passare, i frammenti commentati diventano `stampaIntestazione()` e `stampaDettagli()`.
4. Dopo ogni passo si ricompila e si controlla che l'output non sia cambiato.

RICEVUTA.JAVA

```
import java.util.List;

// Dopo il refactoring: Estrai Metodo, Sostituisci Temp con Query, Dividi Variabile Temp
public class Ricevuta {
    private final String cliente;
    private final List<Double> prezzi;

    public Ricevuta(String cliente, List<Double> prezzi) {
        this.cliente = cliente;
        this.prezzi = prezzi;
    }
}
```

```
public void stampa() {                                // si legge come una sequenza di passi
    stampaIntestazione();
    stampaDettagli();
}

private void stampaIntestazione() {
    System.out.println("*** RICEVUTA ***");
}
```

```
private void stampaDettagli() {
    System.out.println("cliente: " + cliente);
    System.out.println("totale: " + totale());
    System.out.println("da pagare: " + daPagare());
}
```

```
private double totale() {                            // query al posto della variabile temp
    double totale = 0;
    for (double p : prezzi) totale += p;
    return totale;
}
```

```
private double sconto() {                            // query al posto della variabile sconto
    if (totale() > 100) return 0.90;
    return 1.0;
}

// nuovo nome per il secondo significato di temp
private double daPagare() {
    return totale() * sconto();
}
}
```

MAIN.JAVA

```
import java.util.List;

public class Main {
    public static void main(String[] args) {
        List<Double> prezzi = List.of(40.0, 35.0, 45.0);
        new RicevutaPrima("Rossi", prezzi).stampa();
        // stesso output: il comportamento non cambia
        new Ricevuta("Rossi", prezzi).stampa();
    }
}
```

Con prezzi 40, 35 e 45 il totale è 120; poiché supera 100 si applica lo sconto del 10 per cento: 120 per 0,90 fa 108. Le due versioni stampano esattamente le stesse righe: il comportamento esterno non è cambiato, come richiede il refactoring.

OUTPUT DELL'ESECUZIONE

```
*** RICEVUTA ***
cliente: Rossi
totale: 120.0
da pagare: 108.0
*** RICEVUTA ***
cliente: Rossi
totale: 120.0
da pagare: 108.0
```

DA RICORDARE

Prima i test (o l'output di riferimento), poi piccoli passi. Spesso conviene applicare **Sostituisci Temp con Query** prima di **Estrai Metodo**, perché elimina le variabili locali che complicherebbero l'estrazione.

4. Ereditarietà, polimorfismo e Template Method

Sottoclassi e override, la differenza fra tipo statico e tipo dinamico, il dispatch che riparte dal tipo dinamico e l'implementazione di Template Method.

Superclasse e sottoclasse

DIPENDENTE.JAVA

```
public class Dipendente {
    private String nome;
    protected float stipendio;           // visibile anche alle sottoclassi

    public Dipendente(String nome, float stipendio) {
        this.nome = nome;
        this.stipendio = stipendio;
    }

    public String getNome() {
        return nome;
    }

    public void stampa() {
        System.out.println(nome + " guadagna " + stipendio);
    }
}
```

RESPONSABILE.JAVA

```
import java.util.ArrayList;
import java.util.List;

// Sottoclasse: eredita tutto da Dipendente, aggiunge un team e ridefinisce stampa()
public class Responsabile extends Dipendente {
    private final List<Dipendente> team = new ArrayList<>();

    public Responsabile(String nome, float stipendio) {
        super(nome, stipendio);           // chiama il costruttore della superclasse
    }
}
```

```

    public void aggiungi(Dipendente d) {        // metodo nuovo
        team.add(d);
    }

    @Override
    public void stampa() {                      // override
        super.stampa();                        // riusa la versione della superclasse
        System.out.println("  coordina " + team.size() + " persone");
    }
}

```

Responsabile eredita nome, stipendio e metodi; nel costruttore chiama `super(...)`, che deve essere la prima istruzione. Ridefinisce `stampa()` e al suo interno riusa la versione della superclasse con `super.stampa()`. L'annotazione `@Override` chiede al compilatore di controllare che il metodo ridefinisca davvero un metodo ereditato.

Il dispatch

CONTO.JAVA

```

public class Conto {
    protected float saldo = 1500;

    public void preleva(float importo) {
        System.out.println("preleva() di Conto");
        // quale verifica()? lo decide il tipo dinamico
        if (verifica(importo)) saldo -= importo;
        System.out.println("saldo: " + saldo);
    }

    public boolean verifica(float importo) {
        System.out.println("verifica() di Conto");
        return saldo - importo >= 0;
    }
}

```

CONTODEPOSITO.JAVA

```

public class ContoDeposito extends Conto {
    @Override
    public boolean verifica(float importo) {        // regola piu' severa
        System.out.println("verifica() di ContoDeposito");
        return saldo - importo >= 1000;
    }
}

```

Seguiamo `c.preleva(800)` con `c` dichiarato `Conto` ma creato come `ContoDeposito`.

1. Il tipo dinamico è `ContoDeposito`: `preleva()` lì non c'è, si risale a `Conto` e si esegue quella.
2. Dentro `preleva()` la chiamata a `verifica()` riparte dal tipo dinamico: in `ContoDeposito` il metodo c'è, e si esegue la versione severa.
3. Il saldo è 1500; 1500 meno 800 fa 700, che è minore di 1000, quindi la verifica fallisce e il saldo resta 1500.

Template Method

REPORT.JAVA

```
// Template Method: genera() fissa la sequenza, le sottoclassi definiscono i passi
public abstract class Report {
    public final void genera() {                // final: la sequenza non si puo' cambiare
        intestazione();
        corpo();
        chiusura();
    }

    protected void intestazione() {            // passo con implementazione predefinita
        System.out.println("=== REPORT ===");
    }

    protected abstract void corpo();           // passo obbligatorio per le sottoclassi

    protected void chiusura() {
        System.out.println("=== FINE ===");
    }
}
```

REPORTVENDITE.JAVA

```
public class ReportVendite extends Report {
    @Override
    protected void corpo() {
        System.out.println("vendite del mese: 120");
    }
}
```

MAIN.JAVA

```
public class Main {
    public static void main(String[] args) {
        Responsabile r = new Responsabile("Anna", 3000);
        r.aggiungi(new Dipendente("Luca", 1800));
        // tipo statico Dipendente, tipo dinamico Responsabile
        Dipendente d = r;
        // late binding: esegue stampa() di Responsabile
        d.stampa();

        // dispatch: preleva() di Conto, verifica() di ContoDeposito
        Conto c = new ContoDeposito();
        c.preleva(800);

        Report rep = new ReportVendite();
        rep.genera();
    }
}
```

OUTPUT DELL'ESECUZIONE

```
Anna guadagna 3000.0
  coordina 1 persone
preleva() di Conto
verifica() di ContoDeposito
saldo: 1500.0
=== REPORT ===
vendite del mese: 120
=== FINE ===
```

La prima riga dell'output viene da `super.stampa()`, la seconda dalla parte aggiunta in `Responsabile`: anche se la variabile è di tipo `Dipendente`, si esegue il metodo della sottoclasse. Le righe del report mostrano `Template Method: genera()` di `Report` chiama intestazione e chiusura predefinite e il corpo implementato dalla sottoclasse.

DA RICORDARE

extends per ereditare, **super(...)** nel costruttore, **super.metodo()** per riusare la versione della superclasse, **@Override** per farsi controllare dal compilatore. Il dispatch parte sempre dal **tipo dinamico**. `Template Method`: metodo **final** con la sequenza, passi **abstract** o con implementazione predefinita.

5. Dal codice ai diagrammi UML

Come ricavare il diagramma delle classi e il diagramma di sequenza da un codice Java, con le regole di traduzione e un esempio completo.

Regole di traduzione

NEL CODICE JAVA	NEL DIAGRAMMA DELLE CLASSI
<code>class A</code>	rettangolo con nome A; corsivo se <code>abstract</code>
<code>interface I</code>	rettangolo con lo stereotipo «interface»
<code>private int x;</code>	attributo – <code>x: int</code>
<code>public void f(int a)</code>	metodo + <code>f(a: int);</code> protected si scrive #
<code>static</code>	membro sottolineato oppure annotato {static}
<code>class B extends A</code>	freccia continua con triangolo vuoto da B verso A
<code>class C implements I</code>	freccia tratteggiata con triangolo vuoto da C verso I
attributo di tipo B	associazione da A verso B; se le parti vivono con il tutto, composizione (rombo pieno)
<code>List</code> come attributo	associazione con molteplicità * (oppure 0..*)
B usato solo come parametro o variabile locale	dipendenza: freccia tratteggiata aperta

Esempio: il diagramma delle classi del capitolo 4

Applichiamo le regole al codice del capitolo precedente: `Responsabile` `extends` `Dipendente` diventa una generalizzazione; l'attributo `List<Dipendente> team` diventa un'aggregazione con molteplicità 0..*, perché i dipendenti esistono anche senza il responsabile; `Report` è astratta, quindi il suo nome va in corsivo.

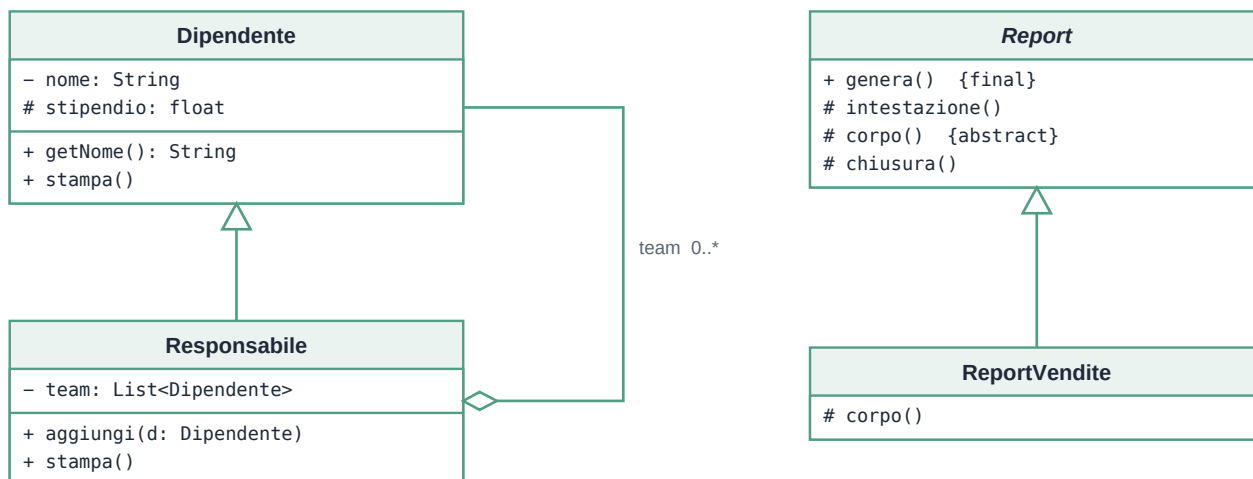


Figura 5.1: il diagramma delle classi ricavato dal codice del capitolo 4.

Esempio: il diagramma di sequenza del main

Per il diagramma di sequenza si segue il codice riga per riga: ogni `new` è un messaggio `create` che fa nascere un oggetto, ogni chiamata di metodo è una freccia piena dal chiamante al chiamato, con una barra di attivazione sul chiamato. Una chiamata che un oggetto fa a se stesso, come `super.stampa()`, è una freccia che torna sulla stessa linea della vita.

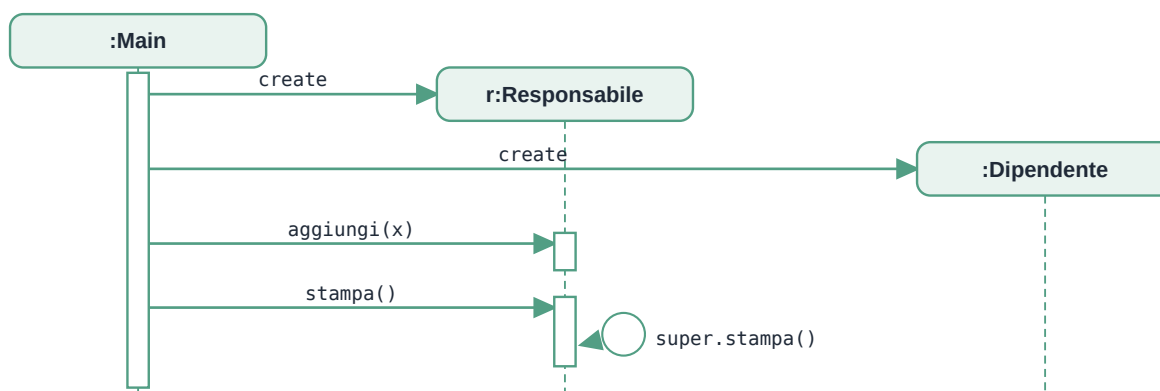


Figura 5.2: il diagramma di sequenza delle prime righe del main del capitolo 4.

DA RICORDARE

Ogni elemento del codice ha un corrispondente UML: **extends** diventa triangolo vuoto con linea continua, **implements** triangolo vuoto con linea tratteggiata, un **attributo** diventa associazione, una **lista** molteplicità *. Nel sequenza ogni **new** è un `create` e ogni chiamata una freccia piena.

6. Singleton e Multiton

Tre implementazioni: Singleton con creazione pigra, Singleton con creazione anticipata e Multiton con un'istanza per chiave.

Come si implementa

1. Costruttore **private**, così nessun'altra classe può usare `new`.
2. Attributo **private static** che tiene l'unica istanza.
3. Metodo **public static getInstance()** che crea l'istanza se serve e la restituisce.

Creazione pigra

REGISTRO.JAVA

```
import java.util.ArrayList;
import java.util.List;

// Singleton con creazione pigra (lazy)
public class Registro {
    private static Registro obj;           // l'unica istanza
    private final List<String> messaggi = new ArrayList<>();

    // privato: nessuno può fare new Registro()
    private Registro() { }

    public static Registro getInstance() {    // punto di accesso globale
        if (obj == null) obj = new Registro(); // creata alla prima richiesta
        return obj;
    }

    public void scrivi(String m) {
        messaggi.add(m);
    }

    public String ultimo() {
        return messaggi.get(messaggi.size() - 1);
    }

    public int quanti() {
        return messaggi.size();
    }
}
```

Creazione anticipata

L'istanza viene creata quando la classe è caricata, quindi `getInstance()` non ha bisogno di controlli e la classe è sicura anche con più thread.

IMPOSTAZIONI.JAVA

```
// Singleton con creazione anticipata (eager): sicuro anche con piu' thread
public class Impostazioni {
    // creata al caricamento della classe
    private static final Impostazioni OBJ = new Impostazioni();
    private String lingua = "it";

    private Impostazioni() { }

    public static Impostazioni getInstance() {
        return OBJ;
    }

    public String getLingua() { return lingua; }
    public void setLingua(String l) { lingua = l; }
}
```

Multiton

Una mappa associa a ogni chiave la sua istanza; getInstance(chiave) crea l'istanza solo se per quella chiave non esiste ancora.

ARCHIVIO.JAVA

```
import java.util.HashMap;
import java.util.Map;

// Multiton: al massimo un'istanza per ogni cartella
public class Archivio {
    private static final Map<String, Archivio> istanze = new HashMap<>();
    private final String cartella;

    private Archivio(String cartella) {
        this.cartella = cartella;
    }

    public static Archivio getInstance(String cartella) {
        if (!istanze.containsKey(cartella)) // crea solo se manca
            istanze.put(cartella, new Archivio(cartella));
        return istanze.get(cartella);
    }

    public String getCartella() { return cartella; }
}
```

MAIN.JAVA

```
public class Main {  
    public static void main(String[] args) {  
        Registro r1 = Registro.getInstance();  
        Registro r2 = Registro.getInstance();  
        r1.scrivi("avvio");  
        r2.scrivi("connessione");  
        System.out.println(r1 == r2);           // true: stesso oggetto  
        System.out.println(r1.ultimo() + " " + r1.quanti());  
        // new Registro() darebbe un errore di compilazione: costruttore privato  
  
        Impostazioni.getInstance().setLingua("en");  
        System.out.println(Impostazioni.getInstance().getLingua());  
  
        Archivio a = Archivio.getInstance("foto");  
        Archivio b = Archivio.getInstance("foto");  
        Archivio c = Archivio.getInstance("musica");  
        System.out.println((a == b) + " " + (a == c)); // true false  
    }  
}
```

OUTPUT DELL'ESECUZIONE

```
true  
connessione 2  
en  
true false
```

`r1 == r2` confronta i riferimenti e vale true: le due variabili puntano allo stesso oggetto, quindi i due messaggi finiscono nello stesso registro, che ne contiene 2. Due richieste per “foto” restituiscono la stessa istanza, la richiesta per “musica” un’istanza diversa.

COME RICONOSCERLO NEL CODICE

Costruttore privato, attributo statico del tipo della classe stessa, metodo statico che lo restituisce: è un **Singleton**. Una mappa statica di istanze indicizzata da una chiave: è un **Multiton**.

7. Factory Method, Object Pool, Dependency Injection e Abstract Factory

I pattern che separano la creazione degli oggetti dal loro uso, ciascuno con un esempio completo.

Factory Method

L'interfaccia Biglietto è il Product, le due classi concrete sono i ConcreteProduct. La classe Cassa svolge insieme i ruoli Creator e ConcreteCreator, con un metodo factory statico e parametrico: decide in base all'età quale classe istanziare.

BIGLIETTO.JAVA

```
// Product
public interface Biglietto {
    double prezzo();
    String descrizione();
}
```

BIGLIETTOINTERO.JAVA

```
// ConcreteProduct
public class BigliettoIntero implements Biglietto {
    public double prezzo() { return 10; }
    public String descrizione() { return "intero"; }
}
```

BIGLIETTORIDOTTO.JAVA

```
// ConcreteProduct
public class BigliettoRidotto implements Biglietto {
    public double prezzo() { return 6; }
    public String descrizione() { return "ridotto"; }
}
```

CASSA.JAVA

```
// Creator e ConcreteCreator nella stessa classe; factory method statico con parametro
public class Cassa {
    public static Biglietto creaBiglietto(int eta) {
        if (eta < 18 || eta >= 65) return new BigliettoRidotto();
        return new BigliettoIntero();
    }
}
```

VARIANTE CON RIFLESSIONE

La classe da istanziare arriva come stringa, per esempio letta da un file di configurazione; nel codice non compare nessun new di una classe concreta.

CASSARIFLESSIVA.JAVA

```
// Variante con riflessione: la classe da istanziare e' un nome in una stringa
public class CassaRiflessiva {
    public static Biglietto creaBiglietto(String nomeClasse) {
        try {
            Class<?> cls = Class.forName(nomeClasse);           // classe dal nome
            return (Biglietto) cls.getConstructor().newInstance(); // nuova istanza
        } catch (ReflectiveOperationException e) {
            return new BigliettoIntero();                       // valore di riserva
        }
    }
}
```

VARIANTE CON SUPPLIER

Una mappa associa a ogni nome il riferimento al costruttore; aggiungere un tipo significa aggiungere una voce, senza toccare uno switch.

CASSASUPPLIER.JAVA

```
import java.util.Map;
import java.util.function.Supplier;

// Variante con Supplier: una mappa dal nome al costruttore, niente switch
public class CassaSupplier {
    private static final Map<String, Supplier<Biglietto>> MAPPA =
        Map.of("intero", BigliettoIntero::new, "ridotto", BigliettoRidotto::new);

    public static Biglietto creaBiglietto(String tipo) {
        return MAPPA.getOrDefault(tipo, BigliettoIntero::new).get();
    }
}
```

Dependency Injection

NOTIFICATORE.JAVA

```
public interface Notificatore {
    void invia(String testo);
}
```

NOTIFICATOREEMAIL.JAVA

```
public class NotificatoreEmail implements Notificatore {
    public void invia(String testo) { System.out.println("email: " + testo); }
}
```

ORDINE.JAVA

```
// Dependency Injection: la dipendenza arriva dal costruttore
public class Ordine {
    private final Notificatore notificatore;

    public Ordine(Notificatore n) {          // l'Ordine non crea il notificatore
        notificatore = n;
    }

    public void conferma() {
        notificatore.invia("ordine confermato");
    }
}
```

Ordine conosce solo l'interfaccia Notificatore; quale implementazione usare lo decide chi crea l'ordine, nel main.

Object Pool

CONNESSIONE.JAVA

```
public class Connessione {
    private static int create = 0;          // conta quante connessioni sono state create
    private final int id;

    public Connessione() { id = ++create; }

    public int getId() { return id; }
}
```

POOLCONNESSIONI.JAVA

```
import java.util.ArrayList;
import java.util.List;

// Object Pool: riusa le istanze restituite invece di crearne di nuove
public class PoolConnessioni {
    private final List<Connessione> libere = new ArrayList<>();

    public Connessione prendi() {           // metodo factory
        if (!libere.isEmpty()) return libere.remove(0);
        return new Connessione();
    }

    public void restituisci(Connessione c) { // il client la rende riusabile
        libere.add(c);
    }
}
```

Abstract Factory

Due famiglie di prodotti, chiara e scura, ciascuna con un bottone e una finestra. Ogni factory concreta crea solo prodotti della propria famiglia, quindi non si possono mescolare per errore.

BOTTONE.JAVA

```
public interface Bottone { String disegna(); }
```

FINESTRA.JAVA

```
public interface Finestra { String apri(); }
```

BOTTONECHIARO.JAVA

```
public class BottoneChiaro implements Bottone {  
    public String disegna() { return "[ ok ]"; }  
}
```

FINESTRASCURA.JAVA

```
public class FinestraScura implements Finestra {  
    public String apri() { return "finestra scura"; }  
}
```

FACTORYTEMA.JAVA

```
// AbstractFactory: un metodo di creazione per ogni prodotto della famiglia  
public interface FactoryTema {  
    Bottone creaBottone();  
    Finestra creaFinestra();  
}
```

FACTORYCHIARA.JAVA

```
// ConcreteFactory: crea solo prodotti della famiglia chiara  
public class FactoryChiara implements FactoryTema {  
    public Bottone creaBottone() { return new BottoneChiaro(); }  
    public Finestra creaFinestra() { return new FinestraChiara(); }  
}
```

FACTORYSCURA.JAVA

```
public class FactoryScura implements FactoryTema {  
    public Bottone creaBottone() { return new BottoneScuro(); }  
    public Finestra creaFinestra() { return new FinestraScura(); }  
}
```

Le classi BottoneScuro e FinestraChiara sono analoghe e non sono ripetute.

MAIN.JAVA

```
public class Main {  
    public static void main(String[] args) {  
        Biglietto b1 = Cassa.creaBiglietto(30);           // il client conosce solo Biglietto  
        Biglietto b2 = Cassa.creaBiglietto(70);  
        System.out.println(b1.descrizione() + " " + b1.prezzo());  
        System.out.println(b2.descrizione() + " " + b2.prezzo());  
        System.out.println(CassaRiflessiva.creaBiglietto("BigliettoRidotto").descrizione());  
        System.out.println(CassaSupplier.creaBiglietto("ridotto").descrizione());  
    }  
}
```

```

    new Ordine(new NotificatoreEmail()).conferma();    // wiring in un solo punto

    PoolConnessioni pool = new PoolConnessioni();
    Connessione c1 = pool.prendi();
    pool.restituisce(c1);
    Connessione c2 = pool.prendi();                  // riusa c1
    System.out.println(c1.getId() + " " + c2.getId());

```

```

        FactoryTema tema = new FactoryScura();        // si cambia famiglia solo qui
        System.out.println(tema.creaFinestra().apri() + " " + tema.creaBottone().disegna());
    }
}

```

OUTPUT DELL'ESECUZIONE

```

intero 10.0
ridotto 6.0
ridotto
ridotto
email: ordine confermato
1 1
finestra scura [#ok#]

```

Un cliente di 30 anni riceve un biglietto intero, uno di 70 un ridotto. La riga "1 1" mostra il pool: la seconda richiesta riceve la stessa connessione restituita, con identificativo 1, invece di crearne una nuova. L'ultima riga usa solo prodotti della famiglia scura.

COME RICONOSCERLO NEL CODICE

Un metodo che restituisce un'interfaccia e al suo interno fa **new** di classi diverse a seconda di un parametro o di una sottoclasse: **Factory Method**. Un'interfaccia con **più metodi di creazione**, implementata da classi che creano prodotti coordinati: **Abstract Factory**. Una lista di oggetti liberi e i metodi prendi e restituisci: **Object Pool**. Dipendenze passate nel **costruttore** come interfacce: **Dependency Injection**.

8. Prototype

Clonazione con copia profonda, confrontata con una copia superficiale sbagliata.

Implementazione

AGENDA.JAVA

```
// Prototype
public interface Agenda {
    void aggiungi(String impegno);
    void stampa();
    Agenda clona();
}
```

CALENDARIO.JAVA

```
import java.util.ArrayList;
import java.util.List;

// ConcretePrototype con copia profonda
public class Calendario implements Agenda {
    private List<String> impegni = new ArrayList<>();

    public void aggiungi(String impegno) {
        impegni.add(impegno);
    }
```

```
    public void stampa() {
        System.out.println(impegni);
    }

    public Agenda clona() {
        Calendario copia = new Calendario();
        copia.impegni = new ArrayList<>(impegni);    // nuova lista con gli stessi elementi
        return copia;
    }
}
```

Il cuore è `clona()`: crea un nuovo oggetto e gli assegna una **nuova lista** con gli stessi elementi, `new ArrayList<>(impegni)`. Dopo la clonazione le due agende sono indipendenti.

L'errore della copia superficiale

CALENDARIOSUPERFICIALE.JAVA

```
import java.util.ArrayList;
import java.util.List;

// Versione sbagliata: copia superficiale, la lista e' condivisa
public class CalendarioSuperficiale implements Agenda {
    private List<String> impegni = new ArrayList<>();

    public void aggiungi(String impegno) { impegni.add(impegno); }
    public void stampa() { System.out.println(impegni); }

    public Agenda clona() {
        CalendarioSuperficiale copia = new CalendarioSuperficiale();
        copia.impegni = impegni;                // stesso riferimento: errore
        return copia;
    }
}
```

MAIN.JAVA

```
public class Main {
    public static void main(String[] args) {
        Agenda a = new Calendario();
        a.aggiungi("dentista");
        Agenda b = a.clona();                // b parte dagli impegni di a
        b.aggiungi("palestra");
        a.stampa();                          // [dentista]
        b.stampa();                          // [dentista, palestra]

        Agenda s = new CalendarioSuperficiale();
        s.aggiungi("dentista");
        Agenda t = s.clona();
        t.aggiungi("palestra");
        s.stampa();                          // anche l'originale e' cambiato
    }
}
```

OUTPUT DELL'ESECUZIONE

```
[dentista]
[dentista, palestra]
[dentista, palestra]
```

Le prime due righe mostrano la copia profonda: aggiungere "palestra" al clone non tocca l'originale. La terza mostra l'errore: con la copia superficiale originale e clone condividono la lista, e l'originale si ritrova un impegno aggiunto solo al clone.

COME RICONOSCERLO NEL CODICE

Un'interfaccia con un metodo che **restituisce una copia** dell'oggetto (clone, clona, copia) e un client che crea oggetti chiamandolo invece di usare new: **Prototype**. Controllare sempre che le collezioni interne vengano duplicate.

9. Builder

Un oggetto immutabile costruito con chiamate concatenate e verificato prima della creazione.

Implementazione

PRENOTAZIONE.JAVA

```
// Product: immutabile, si crea solo tramite il builder
public class Prenotazione {
    private final String cliente;           // obbligatorio
    private final int ospiti;               // facoltativi
    private final boolean colazione;

    Prenotazione(String cliente, int ospiti, boolean colazione) {
        this.cliente = cliente;
        this.ospiti = ospiti;
        this.colazione = colazione;
    }

    @Override
    public String toString() {
        return cliente + ", ospiti " + ospiti + ", colazione " + colazione;
    }
}
```

PRENOTAZIONEBUILDER.JAVA

```
// ConcreteBuilder: raccoglie i valori e crea l'oggetto in build()
public class PrenotazioneBuilder {
    private final String cliente;
    private int ospiti = 1;                // valori predefiniti
    private boolean colazione = false;

    public PrenotazioneBuilder(String cliente) { // parametri obbligatori nel costruttore
        this.cliente = cliente;
    }

    public PrenotazioneBuilder ospiti(int n) {
        ospiti = n;
        return this;                       // restituisce se stesso: chiamate concatenate
    }

    public PrenotazioneBuilder colazione(boolean c) {
        colazione = c;
        return this;
    }
}
```

```

    public Prenotazione build() {                // controlla la coerenza prima di creare
        if (cliente == null || ospiti < 1) return null;
        return new Prenotazione(cliente, ospiti, colazione);
    }
}

```

Il costruttore di Prenotazione non è pubblico: ha visibilità di package, quindi dall'esterno si crea una prenotazione solo passando dal builder. Ogni metodo del builder restituisce `this`, e `build()` controlla la coerenza: con zero ospiti restituisce `null`.

MAIN.JAVA

```

public class Main {
    public static void main(String[] args) {
        Prenotazione p = new PrenotazioneBuilder("Bianchi")
            .ospiti(2)
            .colazione(true)
            .build();
        System.out.println(p);
        // valori predefiniti
        System.out.println(new PrenotazioneBuilder("Verdi").build());
        // incoerente: null
        System.out.println(new PrenotazioneBuilder("Neri").ospiti(0).build());
    }
}

```

OUTPUT DELL'ESECUZIONE

```

Bianchi, ospiti 2, colazione true
Verdi, ospiti 1, colazione false
null

```

COME RICONOSCERLO NEL CODICE

Una classe con metodi che impostano un valore e fanno **return this**, e un metodo finale **build()** che crea un oggetto con attributi **final**: **Builder**.

10. Adapter

La stessa classe di libreria adattata in due modi: Object Adapter e Class Adapter.

Target e Adaptee

ETICHETTATRICE.JAVA

```
// Target: l'interfaccia che l'applicazione si aspetta
public interface Etichettatrice {
    String prossimaEtichetta();
}
```

GENERATORECODICI.JAVA

```
// Adaptee: classe di libreria, che non possiamo modificare
public class GeneratoreCodici {
    private final String prefisso;
    private int numero = 1;

    public GeneratoreCodici(String prefisso) {
        this.prefisso = prefisso;
    }

    public String emettiCodice() {                // nome diverso da quello atteso
        return prefisso + numero++;
    }
}
```

Object Adapter

ADATTATOREOGGETTO.JAVA

```
// Object Adapter: implementa Target e contiene l'Adaptee
public class AdattatoreOggetto implements Etichettatrice {
    private GeneratoreCodici generatore;    // riferimento all'adaptee
    private final String prefisso;

    public AdattatoreOggetto(String prefisso) {
        this.prefisso = prefisso;
    }

    public String prossimaEtichetta() {
        if (generatore == null)                // lazy initialization
            generatore = new GeneratoreCodici(prefisso);
        return generatore.emettiCodice();      // delega
    }
}
```

Class Adapter

ADATTATORECLASSE.JAVA

```
// Class Adapter: estende l'Adaptee e implementa Target (adapter a due vie)
public class AdattatoreClasse extends GeneratoreCodici implements Etichettatrice {
    public AdattatoreClasse(String prefisso) {
        super(prefisso);
    }

    public String prossimaEtichetta() {
        return emettiCodice(); // metodo ereditato
    }
}
```

MAIN.JAVA

```
public class Main {
    public static void main(String[] args) {
        // il client usa solo Etichettatrice
        Etichettatrice e = new AdattatoreOggetto("MAG");
        System.out.println(e.prossimaEtichetta() + " " + e.prossimaEtichetta());

        AdattatoreClasse a = new AdattatoreClasse("SCA");
        System.out.println(a.prossimaEtichetta()); // come Etichettatrice
        System.out.println(a.emettiCodice()); // come GeneratoreCodici
    }
}
```

OUTPUT DELL'ESECUZIONE

```
MAG1 MAG2
SCA1
SCA2
```

L'object adapter crea il generatore alla prima chiamata e poi lo riusa: le etichette proseguono, MAG1 e MAG2. Il class adapter risponde sia come Etichettatrice sia come GeneratoreCodici, perché è entrambe le cose: è l'adapter a due vie.

COME RICONOSCERLO NEL CODICE

Una classe che **implementa un'interfaccia** e nei suoi metodi **chiama un metodo con un altro nome** su un oggetto contenuto (Object Adapter) o ereditato (Class Adapter): **Adapter**.

11. Facade

Un'unica classe che offre un metodo semplice al posto di una sequenza di chiamate al sottosistema.

Implementazione

TRADUTTORE.JAVA

```
import java.util.List;

// Facade: un'interfaccia semplice davanti ai due dizionari
public class Traduttore {
    private final Dizionario inglese = new Dizionario(List.of("Hello", "Yes", "Thanks"));
    private final Dizionario italiano = new Dizionario(List.of("Ciao", "Si'", "Grazie"));

    public void aggiungi(String parola) {          // nasconde i quattro passi
        if (inglese.contiene(parola)) {            // 1. verifica
            inglese.aggiungiAlTesto(parola);        // 2. testo inglese
            int i = inglese.posizione(parola);      // 3. posizione
            italiano.aggiungiAlTesto(italiano.parola(i)); // 4. traduzione
        }
    }
}

public void stampaTutto() {
    System.out.println("EN:" + inglese.getTesto());
    System.out.println("IT:" + italiano.getTesto());
}

// classe del sottosistema annidata: il client non puo' usarla direttamente
private static class Dizionario {
    private final List<String> parole;
    private String testo = "";

    Dizionario(List<String> parole) { this.parole = parole; }
    boolean contiene(String p) { return parole.contains(p); }
    int posizione(String p) { return parole.indexOf(p); }
    String parola(int i) { return parole.get(i); }
    void aggiungiAlTesto(String p) { testo = testo + " " + p; }
    String getTesto() { return testo; }
}
}
```

La classe Dizionario è annidata e privata: il client non può nemmeno nominarla. Il metodo aggiungi() esegue i quattro passi nell'ordine giusto, che il client non deve conoscere.

MAIN.JAVA

```
public class Main {  
    public static void main(String[] args) {  
        Traduttore t = new Traduttore();    // il client vede solo il Facade  
        t.aggiungi("Hello");  
        t.aggiungi("Thanks");  
        t.aggiungi("Pizza");                // non nel dizionario: ignorata  
        t.stampaTutto();  
    }  
}
```

OUTPUT DELL'ESECUZIONE

EN: Hello Thanks
IT: Ciao Grazie

COME RICONOSCERLO NEL CODICE

Una classe che **contiene più oggetti di classi diverse** e offre pochi metodi, ciascuno dei quali chiama in sequenza metodi di quegli oggetti: **Facade**.

12. Stream: esercizi svolti

Nove esercizi risolti con gli stream, ciascuno con la pipeline e, dove serve, la traccia passo per passo dei valori.

Tutti gli esercizi sono metodi statici della classe `Esercizi`; il record `Prodotto` rappresenta un prodotto con nome, prezzo e categoria. Un record è una classe immutabile per cui Java genera costruttore e metodi di accesso, che si chiamano come i campi: `p.prezzo()`.

PRODOTTO.JAVA

```
// record: classe immutabile con costruttore e metodi di accesso generati
public record Prodotto(String nome, double prezzo, String categoria) { }
```

ESERCIZI.JAVA

```
import java.util.Comparator;
import java.util.List;
import java.util.Optional;
import java.util.stream.Collectors;
import java.util.stream.IntStream;
```

```
public class Esercizi {
```

```
    // 1. parole che iniziano con un prefisso
    public static List<String> conPrefisso(List<String> parole, String prefisso) {
        return parole.stream()
            .filter(p -> p.startsWith(prefisso))    // tiene solo quelle col prefisso
            .collect(Collectors.toList());
    }
```

```
    // 2. stringa con le iniziali di ogni parola
    public static String iniziali(List<String> parole) {
        return parole.stream()
            // ogni parola diventa la sua iniziale
            .map(p -> p.substring(0, 1))
            .reduce("", (acc, c) -> acc + c);        // concatena partendo da ""
    }
```

```
    // 3. perimetri delle terne che formano un triangolo
    public static List<Integer> perimetri(List<int[]> terne) {
        return terne.stream()
            // ogni lato minore della somma degli altri
            .filter(t -> t[0] < t[1] + t[2])
            .filter(t -> t[1] < t[0] + t[2])
            .filter(t -> t[2] < t[0] + t[1])
            .map(t -> t[0] + t[1] + t[2])            // terna -> perimetro
            .collect(Collectors.toList());
    }
```

```
// 4. terne di elementi contigui che formano un triangolo
public static List<int[]> terneContigue(List<Integer> lista) {
    return IntStream.rangeClosed(0, lista.size() - 3) // indici di partenza
        .mapToObj(i -> new int[] {lista.get(i), lista.get(i + 1), lista.get(i + 2)})
        .filter(t -> t[0] < t[1] + t[2] && t[1] < t[0] + t[2] && t[2] < t[0] + t[1])
        .collect(Collectors.toList());
}
```

```
// 5. la lista e' ordinata? cerca una coppia fuori ordine
public static boolean ordinata(List<Integer> lista) {
    return IntStream.rangeClosed(0, lista.size() - 2)
        .filter(i -> lista.get(i) > lista.get(i + 1)) // funzione pura
        .findAny() // si ferma alla prima
        // nessuna coppia: ordinata
        .isEmpty();
}
```

```
// 6. totale dei prezzi
public static double totale(List<Prodotto> prodotti) {
    return prodotti.stream()
        .mapToDouble(Prodotto::prezzo)
        .sum();
}

// 7. il prodotto piu' caro sotto una soglia
public static Optional<Prodotto> piuCaroSotto(List<Prodotto> prodotti, double soglia) {
    return prodotti.stream()
        .filter(p -> p.prezzo() < soglia)
        .max(Comparator.comparing(Prodotto::prezzo));
}
```

```
// 8. categorie presenti, senza duplicati e in ordine alfabetico
public static List<String> categorie(List<Prodotto> prodotti) {
    return prodotti.stream()
        .map(Prodotto::categoria)
        .distinct()
        .sorted()
        .collect(Collectors.toList());
}
```

```
// 9. quanti prodotti ha una categoria
public static long conta(List<Prodotto> prodotti, String categoria) {
    return prodotti.stream()
        .filter(p -> p.categoria().equals(categoria))
        .count();
}
}
```

Esercizio 1: parole con un prefisso

filter tiene le parole per cui startsWith(prefisso) è vero. In alternativa si può confrontare p.substring(0, prefisso.length()) con il prefisso, ma bisogna prima controllare che la parola non sia più corta del prefisso, altrimenti substring lancia un'eccezione.

Con le parole mare, marmo, monte, mano, lago e il prefisso "mar":

PAROLA	STARTSWITH("MAR")	ESITO
mare	vero	tenuta
marmo	vero	tenuta
monte	falso	scartata
mano	falso ("man")	scartata
lago	falso	scartata

Esercizio 2: le iniziali

map trasforma ogni parola nella sua prima lettera, reduce le concatena partendo dalla stringa vuota. Con ogni, regola, ha, eccezioni:

PASSO	ACC	INIZIALE	ACC + INIZIALE
inizio	""		
1	""	o	"o"
2	"o"	r	"or"
3	"or"	h	"orh"
4	"orh"	e	"orhe"

Esercizio 3: perimetri dei triangoli

Tre lati formano un triangolo se ciascuno è minore della somma degli altri due. Tre filter controllano le tre condizioni, poi map trasforma la terna nel perimetro.

TERNA	CONDIZIONI	ESITO
3, 4, 5	$3 < 9$, $4 < 8$, $5 < 7$: tutte vere	perimetro 12
1, 2, 8	$8 < 3$ falsa	scartata
2, 2, 3	$2 < 5$, $2 < 5$, $3 < 4$: tutte vere	perimetro 7

Esercizio 4: terne di elementi contigui

Si generano gli indici di partenza da 0 a n meno 3, e mapToObj costruisce per ogni indice i la terna con gli elementi i, i più 1, i più 2. Con la lista 2, 3, 5, 7, 12 gli indici vanno da 0 a 2:

INDICE	TERNA	TRIANGOLO?
0	2, 3, 5	no: $5 < 5$ è falso
1	3, 5, 7	sì
2	5, 7, 12	no: $12 < 12$ è falso

Esercizio 5: la lista è ordinata?

Si generano gli indici da 0 a n meno 2 e si filtrano quelli in cui un elemento è maggiore del successivo. `findAny()` si ferma alla prima coppia trovata; `isEmpty()` sull'Optional vale true se non ne ha trovata nessuna. Con 1, 4, 2: all'indice 0 si confronta 1 con 4 (in ordine), all'indice 1 si confronta 4 con 2 (fuori ordine): la ricerca si ferma e il risultato è false. La lambda è una funzione pura, quindi lo stream si potrebbe anche eseguire in parallelo.

Esercizi 6, 7, 8, 9: operazioni sui prodotti

Con i prodotti mela 1,5 (frutta), pane 2 (forno), kiwi 3 (frutta), torta 12 (forno):

- **totale:** `mapToDouble` trasforma in uno stream di double, `sum` somma: 1,5 più 2 più 3 più 12 fa 18,5;
- **più caro sotto 5:** `filter` tiene mela, pane e kiwi, `max` con il comparatore sul prezzo restituisce l'Optional con il kiwi;
- **categorie:** `map` dà frutta, forno, frutta, forno; `distinct` lascia frutta, forno; `sorted` le ordina: forno, frutta;
- **conta:** `filter` sulla categoria frutta e `count` dà 2.

MAIN.JAVA

```
import java.util.Arrays;
import java.util.List;
import java.util.stream.Collectors;
import java.util.stream.Stream;

public class Main {
    public static void main(String[] args) {
        List<String> parole = List.of("mare", "marmo", "monte", "mano", "lago");
        System.out.println(Esercizi.conPrefisso(parole, "mar"));
        System.out.println(Esercizi.iniziali(List.of("ogni", "regola", "ha", "eccezioni")));
```

```
List<int[]> terne = List.of(new int[] {3, 4, 5}, new int[] {1, 2, 8},
    new int[] {2, 2, 3});
System.out.println(Esercizi.perimetri(terne));

List<Integer> numeri = List.of(2, 3, 5, 7, 12);
Esercizi.terneContigue(numeri).forEach(t -> System.out.print(Arrays.toString(t)));
System.out.println();
System.out.println(Esercizi.ordinata(numeri)); // true
System.out.println(Esercizi.ordinata(List.of(1, 4, 2))); // false
```

```
List<Prodotto> prodotti = List.of(new Prodotto("mela", 1.5, "frutta"),
    new Prodotto("pane", 2.0, "forno"), new Prodotto("kiwi", 3.0, "frutta"),
    new Prodotto("torta", 12.0, "forno"));
System.out.println(Esercizi.totale(prodotti));
Esercizi.piuCaroSotto(prodotti, 5).ifPresent(p -> System.out.println(p.nome()));
System.out.println(Esercizi.categorie(prodotti));
System.out.println(Esercizi.conta(prodotti, "frutta"));
```

```
// peek mostra il percorso di ogni elemento nella pipeline
List<Integer> r = Stream.of(2, 3, 4, 5)
    .peek(x -> System.out.print(" da " + x))
    .map(x -> x + 17)
    .filter(x -> x % 2 == 0)
    .peek(x -> System.out.print(" tiene " + x))
    .collect(Collectors.toList());
System.out.println();
System.out.println(r);
```

```
Stream.iterate(3, n -> n * 2).limit(5).forEach(x -> System.out.print(x + " "));
System.out.println();
}
}
```

OUTPUT DELL'ESECUZIONE

```
[mare, marmo]
orhe
[12, 7]
[3, 5, 7]
true
false
18.5
kiwi
[forno, frutta]
2
  da 2 da 3 tiene 20 da 4 da 5 tiene 22
[20, 22]
3 6 12 24 48
```

La riga con “da” e “tiene” è prodotta dai due peek: mostra che ogni elemento attraversa tutta la pipeline prima che arrivi il successivo; il 2 diventa 19 e viene scartato, il 3 diventa 20 e passa, e così via.

DA RICORDARE

Schema mentale: **sorgente** (stream, Stream.of, IntStream.rangeClosed per gli indici), **trasformazioni** (filter per scartare, map per trasformare, distinct, sorted), **terminale** (collect, count, sum, reduce, max, findAny). Per confrontare elementi vicini si lavora sugli **indici**.

13. Composite

Un ordine fatto di prodotti e confezioni, che a loro volta contengono prodotti: il prezzo si calcola ricorsivamente.

Implementazione

ARTICOLO.JAVA

```
// Component: interfaccia comune a foglie e contenitori
public interface Articolo {
    double prezzo();
    void stampa(String rientro);
}
```

PRODOTTO.JAVA

```
// Leaf
public class Prodotto implements Articolo {
    private final String nome;
    private final double prezzo;

    public Prodotto(String nome, double prezzo) {
        this.nome = nome;
        this.prezzo = prezzo;
    }

    public double prezzo() { return prezzo; }

    public void stampa(String rientro) {
        System.out.println(rientro + nome + " " + prezzo);
    }
}
```

CONFEZIONE.JAVA

```
import java.util.ArrayList;
import java.util.List;

// Composite: contiene altri Articolo, foglie o confezioni
public class Confezione implements Articolo {
    private final String nome;
    private final List<Articolo> figli = new ArrayList<>(); // lista solo nel Composite
}
```

```

public Confezione(String nome) { this.nome = nome; }

public void add(Articolo a) { figli.add(a); }           // gestione dei figli
public void remove(Articolo a) { figli.remove(a); }

public double prezzo() {                               // inoltra la richiesta ai figli e somma
    double totale = 0;
    for (Articolo a : figli) totale += a.prezzo();
    return totale;
}

```

```

public void stampa(String rientro) {
    System.out.println(rientro + nome + " (totale " + prezzo() + ")");
    for (Articolo a : figli) a.stampa(rientro + " ");
}
}

```

La lista dei figli e i metodi add e remove stanno solo nel Composite: è la scelta della sicurezza. prezzo() della confezione somma i prezzi dei figli, e se un figlio è a sua volta una confezione la ricorsione scende di un livello.

MAIN.JAVA

```

public class Main {
    public static void main(String[] args) {
        Confezione regalo = new Confezione("pacco regalo");
        regalo.add(new Prodotto("profumo", 30));
        regalo.add(new Prodotto("biglietto", 2));

        Confezione ordine = new Confezione("ordine");
        ordine.add(regalo);                          // un composite dentro un composite
        ordine.add(new Prodotto("libro", 15));

        ordine.stampa("");
        // il client non distingue foglia e composito
        Articolo a = ordine;
        System.out.println("da pagare: " + a.prezzo());
    }
}

```

OUTPUT DELL'ESECUZIONE

```

ordine (totale 47.0)
  pacco regalo (totale 32.0)
    profumo 30.0
    biglietto 2.0
  libro 15.0
da pagare: 47.0

```

Il calcolo procede dal basso: il pacco regalo vale 30 più 2, cioè 32; l'ordine vale 32 più 15, cioè 47.

COME RICONOSCERLO NEL CODICE

Una classe che **implementa un'interfaccia** e **contiene una lista dello stesso tipo dell'interfaccia**, e i cui metodi scorrono la lista chiamando lo stesso metodo sui figli: **Composite**.

14. Decorator

Strati di comportamento aggiunti a un messaggio a runtime, combinabili in qualunque ordine.

Implementazione

TESTO.JAVA

```
// Component
public interface Testo {
    String testo();
}
```

MESSAGGIO.JAVA

```
// ConcreteComponent
public class Messaggio implements Testo {
    private final String contenuto;

    public Messaggio(String contenuto) { this.contenuto = contenuto; }

    public String testo() { return contenuto; }
}
```

DECORATORE.JAVA

```
// Decorator: e' un Testo e contiene un Testo a cui inoltra le richieste
public abstract class Decoratore implements Testo {
    private final Testo interno;

    public Decoratore(Testo interno) { this.interno = interno; }

    public String testo() { return interno.testo(); }
}
```

MAIUSCOLO.JAVA

```
// ConcreteDecorator: trasforma il risultato dell'oggetto interno
public class Maiuscolo extends Decoratore {
    public Maiuscolo(Testo t) { super(t); }

    @Override
    public String testo() { return super.testo().toUpperCase(); }
}
```

FIRMA.JAVA

```
// ConcreteDecorator con uno stato aggiunto
public class Firma extends Decoratore {
    private final String autore;

    public Firma(Testo t, String autore) {
        super(t);
        this.autore = autore;
    }

    @Override
    public String testo() { return super.testo() + " (" + autore + ")"; }
}
```

MAIN.JAVA

```
public class Main {
    public static void main(String[] args) {
        Testo base = new Messaggio("ciao a tutti");
        Testo t1 = new Firma(base, "Anna");
        Testo t2 = new Firma(new Maiuscolo(base), "Anna");           // ordine degli strati
        Testo t3 = new Maiuscolo(new Firma(base, "Anna"));
        // stesso decorator due volte
        Testo t4 = new Firma(new Firma(base, "Anna"), "Luca");
        System.out.println(t1.testo());
        System.out.println(t2.testo());
        System.out.println(t3.testo());
        System.out.println(t4.testo());
    }
}
```

OUTPUT DELL'ESECUZIONE

```
ciao a tutti (Anna)
CIAO A TUTTI (Anna)
CIAO A TUTTI (ANNA)
ciao a tutti (Anna) (Luca)
```

L'ordine degli strati conta. In t2 la firma è aggiunta dopo il maiuscolo, quindi resta "Anna"; in t3 il maiuscolo è lo strato esterno e trasforma anche la firma. In t4 lo stesso decorator è applicato due volte.

COME RICONOSCERLO NEL CODICE

Una classe che **implementa un'interfaccia** e **riceve nel costruttore un oggetto della stessa interfaccia**, a cui inoltra le chiamate aggiungendo qualcosa prima o dopo: **Decorator**. Si riconosce anche dagli annidamenti `new A(new B(new C()))`.

15. Bridge

Le forme separate dalla piattaforma che le disegna, con la piattaforma cambiata a runtime.

Implementazione

DISEGNO.JAVA

```
// Implementor: operazioni primitive della piattaforma
public interface Disegno {
    void tracciaLinea(int x1, int y1, int x2, int y2);
}
```

DISEGNOTESTO.JAVA

```
// ConcreteImplementor: piattaforma che scrive coordinate
public class DisegnoTesto implements Disegno {
    public void tracciaLinea(int x1, int y1, int x2, int y2) {
        System.out.println("linea (" + x1 + "," + y1 + ") (" + x2 + "," + y2 + ")");
    }
}
```

DISEGNOCOMPATTO.JAVA

```
// ConcreteImplementor: un'altra piattaforma
public class DisegnoCompatto implements Disegno {
    public void tracciaLinea(int x1, int y1, int x2, int y2) {
        System.out.print "[" + x1 + y1 + x2 + y2 + " ]";
    }
}
```

FORMA.JAVA

```
// Abstraction: tiene il riferimento all'Implementor (il ponte)
public abstract class Forma {
    private Disegno impl;

    public Forma(Disegno impl) { this.impl = impl; }

    public void setDisegno(Disegno impl) { this.impl = impl; } // cambio a runtime

    protected void linea(int x1, int y1, int x2, int y2) { // operazione di base
        impl.tracciaLinea(x1, y1, x2, y2);
    }

    // operazione di alto livello
    public abstract void disegna();
}
```

RETTANGOLO.JAVA

```
// RefinedAbstraction
public class Rettangolo extends Forma {
    private final int x1, y1, x2, y2;

    public Rettangolo(Disegno d, int x1, int y1, int x2, int y2) {
        super(d);
        this.x1 = x1; this.y1 = y1; this.x2 = x2; this.y2 = y2;
    }

    public void disegna() {                // quattro linee, qualunque sia la piattaforma
        linea(x1, y1, x2, y1);
        linea(x2, y1, x2, y2);
        linea(x2, y2, x1, y2);
        linea(x1, y2, x1, y1);
    }
}
```

MAIN.JAVA

```
public class Main {
    public static void main(String[] args) {
        Forma r = new Rettangolo(new DisegnoTesto(), 0, 0, 4, 2);
        r.disegna();
        r.setDisegno(new DisegnoCompatto());    // stessa forma, altra piattaforma
        r.disegna();
        System.out.println();
    }
}
```

OUTPUT DELL'ESECUZIONE

```
linea (0,0) (4,0)
linea (4,0) (4,2)
linea (4,2) (0,2)
linea (0,2) (0,0)
[0040][4042][4202][0200]
```

Lo stesso rettangolo viene disegnato da due piattaforme diverse: il codice di `Rettangolo` non cambia, cambia solo l'implementor ricevuto. Una nuova forma richiede solo una sottoclasse di `Forma`, una nuova piattaforma solo una classe che implementa `Disegno`.

COME RICONOSCERLO NEL CODICE

Una **gerarchia di astrazioni** (classe astratta e sottoclassi) che tiene un riferimento a un'**interfaccia di implementazione** con le sue classi concrete, e i metodi di alto livello costruiti chiamando quelli primitivi: **Bridge**.

16. Observer e Publisher Subscriber

Un Observer scritto da zero e la versione asincrona con le classi di Java 9.

Observer scritto da zero

OSSERVATORE.JAVA

```
// Observer
public interface Osservatore {
    void aggiorna(Soggetto s, Object dato); // riceve il mittente e il dato (modello push)
}
```

SOGGETTO.JAVA

```
import java.util.ArrayList;
import java.util.List;

// Subject: tiene la lista degli osservatori e li avvisa
public class Soggetto {
    private final List<Osservatore> osservatori = new ArrayList<>();
    private boolean cambiato = false;

    public void attach(Osservatore o) { osservatori.add(o); }
    public void detach(Osservatore o) { osservatori.remove(o); }

    protected void setCambiato() { cambiato = true; }

    protected void notifica(Object dato) {
        if (!cambiato) return; // niente notifiche inutili
        for (Osservatore o : osservatori) o.aggiorna(this, dato);
        cambiato = false;
    }
}
```

RUBRICA.JAVA

```
import java.util.ArrayList;
import java.util.List;

// ConcreteSubject
public class Rubrica extends Soggetto {
    private final List<String> contatti = new ArrayList<>();

    public void inserisci(String nome) {
        if (contatti.contains(nome)) return;
        contatti.add(nome);
        setCambiato();           // lo stato e' cambiato
        notifica(nome);          // avvisa tutti gli osservatori
    }

    public int quanti() { return contatti.size(); }
}
```

SALVATAGGIO.JAVA

```
// ConcreteObserver
public class Salvataggio implements Osservatore {
    public void aggiorna(Soggetto s, Object dato) {
        System.out.println("salvo su file: " + dato);
    }
}
```

SCHERMO.JAVA

```
// ConcreteObserver che interroga il subject (modello pull)
public class Schermo implements Osservatore {
    public void aggiorna(Soggetto s, Object dato) {
        Rubrica r = (Rubrica) s;
        System.out.println("a schermo: " + r.quanti() + " contatti");
    }
}
```

Salvataggio usa il dato passato con la notifica (modello push); Schermo chiede al subject il numero di contatti (modello pull).

La versione con Flow di Java 9

SCHERMOREATTIVO.JAVA

```
import java.util.concurrent.Flow.Subscriber;
import java.util.concurrent.Flow.Subscription;

// Subscriber di Java 9: riceve gli elementi in modo asincrono
public class SchermoReattivo implements Subscriber<String> {
    private Subscription sub;

    public void onSubscribe(Subscription s) {
        sub = s;
        sub.request(1);           // chiede il primo elemento
    }
}
```

```

    public void onNext(String nome) {
        System.out.println("ricevuto: " + nome);
        sub.request(1);           // pronto per il prossimo
    }

    public void onError(Throwable t) { System.out.println("errore"); }

    public void onComplete() { System.out.println("fine"); }
}

```

MAIN.JAVA

```

import java.util.concurrent.SubmissionPublisher;

public class Main {
    public static void main(String[] args) throws InterruptedException {
        Rubrica r = new Rubrica();
        r.attach(new Salvataggio());
        r.attach(new Schermo());
        r.inserisci("Anna");
        r.inserisci("Anna");           // duplicato: nessuna notifica
        r.inserisci("Luca");

        SubmissionPublisher<String> pub = new SubmissionPublisher<>();
        pub.subscribe(new SchermoReattivo());
        pub.submit("Marta");           // consegnato su un altro thread
        pub.close();                   // provoca onComplete()
        Thread.sleep(300);             // attende la consegna asincrona
    }
}

```

OUTPUT DELL'ESECUZIONE

```

salvo su file: Anna
a schermo: 1 contatti
salvo su file: Luca
a schermo: 2 contatti
ricevuto: Marta
fine

```

Il secondo inserimento di Anna non produce notifiche, perché lo stato non è cambiato. Con `SubmissionPublisher` la consegna avviene su un altro thread: per questo il main attende qualche istante prima di terminare.

COME RICONOSCERLO NEL CODICE

Una classe con una **lista di oggetti di un'interfaccia**, metodi per **aggiungerli e toglierli** e un metodo che chiama lo stesso metodo (update, aggiorna) su tutti: **Observer**. In Java moderno: **subscribe**, **submit**, **onNext**, **request(n)**.

17. State

Un ordine online prima con gli switch e poi con il pattern State.

Prima: gli switch

ORDINEPRIMA.JAVA

```
// Prima di State: uno switch sullo stato in ogni metodo
public class OrdinePrima {
    private enum Stato { NUOVO, SPEDITO, CONSEGNATO }
    private Stato stato = Stato.NUOVO;

    public void spedisci() {
        switch (stato) {
            case NUOVO -> { stato = Stato.SPEDITO; System.out.println("spedito"); }
            case SPEDITO -> System.out.println("gia' spedito");
            case CONSEGNATO -> System.out.println("gia' consegnato");
        }
    }

    public void consegna() {
        switch (stato) {
            case NUOVO -> System.out.println("prima va spedito");
            case SPEDITO -> { stato = Stato.CONSEGNATO; System.out.println("consegnato"); }
            case CONSEGNATO -> System.out.println("gia' consegnato");
        }
    }
}
```

Dopo: una classe per stato

STATOORDINE.JAVA

```
// State: ogni metodo restituisce lo stato successivo (interfaccia fluente)
public interface StatoOrdine {
    StatoOrdine spedisci();
    StatoOrdine consegna();
    StatoOrdine annulla();
}
```

NUOVO.JAVA

```
// ConcreteState
public class Nuovo implements StatoOrdine {
    public StatoOrdine spedisci() {
        System.out.println("spedito");
        return new Spedito();           // transizione
    }

    public StatoOrdine consegna() {
        System.out.println("prima va spedito");
        return this;                    // nessun cambio di stato
    }

    public StatoOrdine annulla() {
        System.out.println("annullato");
        return new Annullato();
    }
}
```

SPEDITO.JAVA

```
// ConcreteState
public class Spedito implements StatoOrdine {
    public StatoOrdine spedisci() {
        System.out.println("gia' spedito");
        return this;
    }

    public StatoOrdine consegna() {
        System.out.println("consegnato");
        return new Consegnato();
    }

    public StatoOrdine annulla() {
        System.out.println("non si puo' annullare: e' in viaggio");
        return this;
    }
}
```

CONSEGNATO.JAVA

```
// ConcreteState
public class Consegnato implements StatoOrdine {
    public StatoOrdine spedisci() { System.out.println("gia' consegnato"); return this; }
    public StatoOrdine consegna() { System.out.println("gia' consegnato"); return this; }
    public StatoOrdine annulla() { System.out.println("gia' consegnato"); return this; }
}
```

ANNULLATO.JAVA

```
// ConcreteState
public class Annullato implements StatoOrdine {
    public StatoOrdine spedisci() { System.out.println("ordine annullato"); return this; }
    public StatoOrdine consegna() { System.out.println("ordine annullato"); return this; }
    public StatoOrdine annulla() { System.out.println("gia' annullato"); return this; }
}
```

ORDINE.JAVA

```
// Context: delega allo stato corrente e lo sostituisce con quello restituito
public class Ordine {
    private StatoOrdine stato = new Nuovo();

    public void spedisci() { stato = stato.spedisci(); }
    public void consegna() { stato = stato.consegna(); }
    public void annulla() { stato = stato.annulla(); }

    public String getStato() { return stato.getClass().getSimpleName(); }
}
```

MAIN.JAVA

```
public class Main {
    public static void main(String[] args) {
        Ordine o = new Ordine();
        o.consegna();                // prima va spedito
        o.spedisci();
        o.annulla();                // non si puo' annullare
        o.consegna();
        System.out.println("stato finale: " + o.getStato());

        OrdinePrima p = new OrdinePrima(); // stesso comportamento, ma con gli switch
        p.spedisci();
        p.consegna();
    }
}
```

OUTPUT DELL'ESECUZIONE

```
prima va spedito
spedito
non si puo' annullare: e' in viaggio
consegnato
stato finale: Consegnato
spedito
consegnato
```

Ogni metodo dello stato restituisce lo stato successivo, e il Context si limita ad assegnarlo: `stato = stato.spedisci()`. Aggiungere uno stato, come `Annullato`, non ha richiesto di modificare gli switch esistenti, perché non ce ne sono.

COME RICONOSCERLO NEL CODICE

Un attributo di un tipo interfaccia che rappresenta lo **stato**, metodi del Context che **delegano** a quell'attributo e classi concrete che **restituiscono o impostano lo stato successivo**: **State**.

18. Mediator

Una finestra di ricerca in cui campo, bottone ed etichetta si coordinano solo attraverso il mediatore.

Implementazione

MEDIATORE.JAVA

```
// Mediator
public interface Mediatore {
    void cambiato(Componente c);           // un componente avvisa il mediatore
}
```

COMPONENTE.JAVA

```
// Colleague: conosce solo il mediatore
public abstract class Componente {
    protected final Mediatore mediatore;

    public Componente(Mediatore m) { mediatore = m; }
}
```

CAMPOTESTO.JAVA

```
public class CampoTesto extends Componente {
    private String testo = "";

    public CampoTesto(Mediatore m) { super(m); }

    public void scrivi(String t) {
        testo = t;
        mediatore.cambiato(this);           // non avvisa il bottone direttamente
    }

    public String getTesto() { return testo; }
}
```

BOTTONE.JAVA

```
public class Bottone extends Componente {
    private boolean attivo = false;

    public Bottone(Mediatore m) { super(m); }

    public void setAttivo(boolean a) {
        attivo = a;
        System.out.println("bottone " + (attivo ? "attivo" : "disattivo"));
    }

    public void premi() {
        if (attivo) mediatore.cambiato(this);
    }
}
```

ETICHETTA.JAVA

```
public class Etichetta extends Componente {
    public Etichetta(Mediatore m) { super(m); }

    public void mostra(String t) { System.out.println("etichetta: " + t); }
}
```

FINESTRARICERCA.JAVA

```
// ConcreteMediator: contiene tutta la logica di coordinamento
public class FinestraRicerca implements Mediatore {
    private final CampoTesto campo = new CampoTesto(this);
    private final Bottone cerca = new Bottone(this);
    private final Etichetta esito = new Etichetta(this);

    public void cambiato(Componente c) {
        // testo vuoto: bottone spento
        if (c == campo) cerca.setAttivo(!campo.getTesto().isEmpty());
        else if (c == cerca) esito.mostra("cerco " + campo.getTesto());
    }

    public CampoTesto getCampo() { return campo; }
    public Bottone getCerca() { return cerca; }
}
```

MAIN.JAVA

```
public class Main {
    public static void main(String[] args) {
        FinestraRicerca f = new FinestraRicerca();
        f.getCampo().scrivi("java");
        f.getCerca().premi();
        f.getCampo().scrivi("");
        f.getCerca().premi(); // disattivo: non succede nulla
    }
}
```

OUTPUT DELL'ESECUZIONE

```
bottone attivo  
etichetta: cerco java  
bottone disattivo
```

Il campo di testo non sa che esiste un bottone: avvisa il mediatore, che attiva il bottone se il testo non è vuoto. Con il testo vuoto il bottone si disattiva e premerlo non produce nulla.

COME RICONOSCERLO NEL CODICE

Classi che ricevono nel costruttore un riferimento a un **mediatore** e lo avvisano a ogni cambiamento, e un mediatore che conosce tutte e decide chi chiamare: **Mediator**.

19. Chain of Responsibility

L'approvazione di una spesa che sale lungo la gerarchia aziendale finché qualcuno ha l'autorità per approvarla.

Implementazione

GESTORE.JAVA

```
// Handler: ogni gestore conosce solo il successore
public abstract class Gestore {
    private Gestore successore;

    public Gestore setSuccessore(Gestore s) {
        successore = s;
        return s;
    }

    public void gestisci(double importo) {
        if (puoGestire(importo)) approva(importo);
        else if (successore != null) successore.gestisci(importo); // inoltra
        else System.out.println(importo + ": nessuno puo' approvare");
    }

    protected abstract boolean puoGestire(double importo);
    protected abstract void approva(double importo);
}
```

CAPOREPARTO.JAVA

```
// ConcreteHandler
public class CapoReparto extends Gestore {
    protected boolean puoGestire(double importo) { return importo <= 500; }
    protected void approva(double importo) {
        System.out.println(importo + ": approva il capo reparto");
    }
}
```

DIRETTORE.JAVA

```
// ConcreteHandler
public class Direttore extends Gestore {
    protected boolean puoGestire(double importo) { return importo <= 5000; }
    protected void approva(double importo) {
        System.out.println(importo + ": approva il direttore");
    }
}
```

CONSIGLIO.JAVA

```
// ConcreteHandler
public class Consiglio extends Gestore {
    protected boolean puoGestire(double importo) { return importo <= 50000; }
    protected void approva(double importo) {
        System.out.println(importo + ": approva il consiglio");
    }
}
```

Gestore contiene la logica comune: gestire se possibile, altrimenti inoltrare. È anche un piccolo Template Method, perché `gestisci()` fissa la sequenza e le sottoclassi definiscono `puoGestire()` e `approva()`.

MAIN.JAVA

```
public class Main {
    public static void main(String[] args) {
        Gestore catena = new CapoReparto();
        catena.setSuccessore(new Direttore()).setSuccessore(new Consiglio());
        catena.gestisci(200);           // il client conosce solo il primo anello
        catena.gestisci(3000);
        catena.gestisci(20000);
        catena.gestisci(90000);         // arriva in fondo senza essere gestita
    }
}
```

OUTPUT DELL'ESECUZIONE

```
200.0: approva il capo reparto
3000.0: approva il direttore
20000.0: approva il consiglio
90000.0: nessuno puo' approvare
```

COME RICONOSCERLO NEL CODICE

Una classe con un **riferimento a un oggetto dello stesso tipo** (il successore) e un metodo che **gestisce oppure inoltra** la richiesta al successore: **Chain of Responsibility**.

20. Command

Comandi su un calendario, eseguiti da un menù che conserva una pila per annullarli.

Implementazione

COMANDO.JAVA

```
// Command
public interface Comando {
    void execute();
    void unexecute();
}
```

CALENDARIO.JAVA

```
import java.util.ArrayList;
import java.util.List;

// Receiver: sa come svolgere le operazioni
public class Calendario {
    private final List<String> note = new ArrayList<>();

    public void aggiungi(String n) { note.add(n); }
    public void rimuovi(String n) { note.remove(n); }
    public void stampa() { System.out.println(note); }
}
```

AGGIUNGINOTA.JAVA

```
// ConcreteCommand: lega il receiver all'azione e ricorda cosa annullare
public class AggiungiNota implements Comando {
    private final Calendario calendario;
    private final String nota;

    public AggiungiNota(Calendario c, String nota) {
        calendario = c;
        this.nota = nota;
    }

    public void execute() { calendario.aggiungi(nota); }
    public void unexecute() { calendario.rimuovi(nota); } // operazione inversa
}
```

MENU.JAVA

```
import java.util.ArrayDeque;
import java.util.Deque;

// Invoker: esegue i comandi e tiene la pila per annullarli
public class Menu {
    private final Deque<Comando> eseguiti = new ArrayDeque<>();

    public void esegui(Comando c) {
        c.execute();
        eseguiti.push(c);           // in cima alla pila
    }

    public void annulla() {
        if (!eseguiti.isEmpty()) eseguiti.pop().unexecute(); // ultimo eseguito
    }
}
```

La pila (Deque con push e pop) permette più livelli di annullamento: si annulla sempre l'ultimo comando eseguito.

MAIN.JAVA

```
public class Main {
    public static void main(String[] args) {
        Calendario cal = new Calendario();
        Menu menu = new Menu();
        menu.esegui(new AggiungiNota(cal, "riunione")); // il client crea e configura
        menu.esegui(new AggiungiNota(cal, "palestra"));
        cal.stampa();
        menu.annulla(); // annulla "palestra"
        cal.stampa();
        menu.annulla();
        cal.stampa();
    }
}
```

OUTPUT DELL'ESECUZIONE

```
[riunione, palestra]
[riunione]
[]
```

COME RICONOSCERLO NEL CODICE

Un'interfaccia con **execute()** (ed eventualmente **unexecute()**), classi che **tengono un riferimento al receiver** e chiamano i suoi metodi, un invoker che **conserva i comandi**: **Command**.

21. Test a partire dai requisiti

Dai requisiti di un parcheggio ai casi di test e alla loro implementazione con JUnit.

Requisiti e progettazione

Un sistema registra quale auto occupa quale posto. Un'auto occupa un solo posto alla volta e un posto ospita una sola auto. Le classi sono Auto e Posto, record che contengono i dati, e Parcheggio, che tiene una mappa dalle auto ai posti.

AUTO.JAVA

```
public record Auto(String targa) { }
```

POSTO.JAVA

```
public record Posto(int numero) { }
```

PARCHEGGIO.JAVA

```
import java.util.HashMap;
import java.util.Map;

/** Registra quale auto occupa quale posto.
 * Un'auto occupa un solo posto, un posto ospita una sola auto. */
public class Parcheggio {
    private final Map<Auto, Posto> occupati = new HashMap<>();

    public boolean entra(Auto auto, Posto posto) {
        if (richiestaNonValida(auto, posto)) return false;
        occupati.put(auto, posto);
        return true;
    }

    private boolean richiestaNonValida(Auto auto, Posto posto) {
        return auto == null || posto == null
            || occupati.containsKey(auto)           // auto gia' parcheggiata
            || occupati.containsValue(posto);        // posto gia' occupato
    }

    public boolean esce(Auto auto, Posto posto) {
        return occupati.remove(auto, posto);        // rimuove solo se la coppia esiste
    }
}
```

`remove(chiave, valore)` di `Map` rimuove la coppia solo se la chiave è associata proprio a quel valore, e restituisce `true` se l'ha rimossa.

Casi di test

CASO	CLASSE DI EQUIVALENZA	ATTESO
un'auto entra in un posto libero	valida	true
la stessa auto in un secondo posto	auto già presente	false
una seconda auto nello stesso posto	posto occupato	false
auto null	non valida	false
posto null	non valida	false
uscita, poi nuova entrata nello stesso posto	valida	true, true

PARCHEGGIOTEST.JAVA

```
import static org.junit.jupiter.api.Assertions.assertFalse;
import static org.junit.jupiter.api.Assertions.assertTrue;
import org.junit.jupiter.api.BeforeEach;
import org.junit.jupiter.api.Test;
```

```
public class ParcheggioTest {
    private Parcheggio p;
    private Auto a1, a2;
    private Posto p1, p2;
```

```
@BeforeEach
void prepara() { // dati nuovi per ogni test
    p = new Parcheggio();
    a1 = new Auto("AB123CD");
    a2 = new Auto("EF456GH");
    p1 = new Posto(1);
    p2 = new Posto(2);
}
```

```
@Test
void entraUnaAuto() {
    assertTrue(p.entra(a1, p1));
}

@Test
void stessaAutoDuePosti() {
    p.entra(a1, p1);
    assertFalse(p.entra(a1, p2));
}
```

```

@Test
void stessoPostoDueAuto() {
    p.entra(a1, p1);
    assertFalse(p.entra(a2, p1));
}

@Test
void autoNull() { // classe di equivalenza non valida
    assertFalse(p.entra(null, p1));
}

```

```

@Test
void postoNull() {
    assertFalse(p.entra(a1, null));
}

@Test
void esceELiberaIlPosto() {
    p.entra(a1, p1);
    assertTrue(p.esce(a1, p1));
    assertTrue(p.entra(a2, p1)); // il posto ora e' libero
}
}

```

ESITO DEI TEST

```

OK   ParcheggioTest.entraUnaAuto
OK   ParcheggioTest.stessaAutoDuePosti
OK   ParcheggioTest.stessoPostoDueAuto
OK   ParcheggioTest.autoNull
OK   ParcheggioTest.postoNull
OK   ParcheggioTest.esceELiberaIlPosto
passati: 6, falliti: 0

```

DA RICORDARE

Ogni requisito e ogni vincolo diventano almeno un caso di test con il risultato atteso; si aggiungono le classi di equivalenza **non valide** (null) e i casi che verificano il ritorno allo stato iniziale. **@BeforeEach** ricrea i dati, così i test sono indipendenti.

22. Riconoscere i design pattern dal codice

Una tabella riassuntiva degli indizi che, nel codice o in un diagramma, rivelano quale pattern è stato usato.

Per riconoscere un pattern conviene guardare tre cose: quali **interfacce** ci sono, quali **riferimenti** tiene ciascuna classe (attributi, liste, parametri del costruttore) e come **si chiamano fra loro** i metodi. La tabella raccoglie gli indizi tipici.

PATTERN	INDIZI NEL CODICE
Singleton	costruttore privato, attributo static del tipo della classe, metodo static getInstance()
Factory Method	metodo che restituisce un'interfaccia e decide quale classe concreta istanziare
Abstract Factory	interfaccia con più metodi di creazione, implementazioni che creano prodotti della stessa famiglia
Prototype	metodo che restituisce una copia dell'oggetto; nuovi oggetti ottenuti clonando
Builder	metodi che impostano un valore e restituiscono this, metodo finale build()
Adapter	implementa un'interfaccia attesa e traduce le chiamate verso un oggetto con metodi di nome diverso
Facade	contiene oggetti di più classi e offre pochi metodi che li chiamano in sequenza
Composite	implementa un'interfaccia e contiene una lista di oggetti della stessa interfaccia
Decorator	implementa un'interfaccia e riceve nel costruttore un oggetto della stessa interfaccia a cui inoltra
Bridge	gerarchia di astrazioni con un riferimento a un'interfaccia di implementazione con più classi concrete
Observer	lista di osservatori, metodi attach e detach, notifica che chiama update su tutti
State	attributo di tipo interfaccia che rappresenta lo stato; metodi che delegano e cambiano stato
Mediator	componenti che conoscono solo un mediatore e lo avvertono; il mediatore conosce tutti
Chain of Responsibility	riferimento al successore dello stesso tipo; gestisce oppure inoltra
Command	interfaccia con execute(); oggetti comando con riferimento al receiver; invoker che li conserva
Template Method	metodo final in una classe astratta che chiama metodi abstract ridefiniti dalle sottoclassi

Alcuni pattern si somigliano. **Decorator** contiene **un** oggetto della propria interfaccia, **Composite** una **lista**. **State** e **Strategy** hanno lo stesso diagramma, ma lo stato cambia da solo, la strategia la sceglie il client. **Adapter** converte un'interfaccia esistente, **Facade** ne crea una più semplice.