NoiStudiamo

DISPENSA DI TEORIA

Ingegneria del Software

Qualità del software, progettazione a oggetti, design pattern, processi di sviluppo, stream Java, test e metriche

A CURA DI

Damiano La Pusata

Ottobre 2026

INDICE

Indice dei capitoli

I capitoli seguono l’ordine delle lezioni; ogni titolo è cliccabile.

1. Introduzione all’ingegneria del software 4

2. Qualità, test e principi di progettazione 10

3. Test con JUnit 13

4. Refactoring 16

5. Classi, ereditarietà e polimorfismo 20

6. Diagrammi UML 24

7. Design pattern e Singleton 29

8. Factory Method e Abstract Factory 32

9. Prototype 35

10. Builder 37

11. Adapter 39

12. Facade 41

13. Stream Java: lambda e operazioni di base 43

14. Stream Java: operazioni avanzate e parallelismo 48

15. Composite 51

16. Decorator 53

17. Bridge 55

18. Processi di sviluppo del software 57

19. Il processo Extreme Programming 61

20. Scrum 64

21. Observer e Model View Controller 66

22. State 70

23. Mediator 72

24. Chain of Responsibility 74

25. Command 76

26. Progettare con i design pattern 78

27. Evoluzione del software e metriche 79

28. Verifica, validazione e test 82

CAPITOLO 1

1. Introduzione all’ingegneria del software

Che cosa studia la disciplina, quali caratteristiche rendono il software diverso dagli altri prodotti, come si giudica la sua qualità e perché un codice che funziona può essere comunque scritto male.

Di che cosa si occupa l’ingegneria del software

Scrivere un programma che risolve un esercizio e sviluppare un sistema software destinato alla produzione sono attività molto diverse. Nel secondo caso il sistema è grande, viene scritto da più persone, deve rispettare tempi e costi concordati con un cliente e, soprattutto, deve continuare a vivere per anni, venendo corretto ed esteso molte volte. L’ingegneria del software nasce proprio per affrontare questo secondo scenario con metodi sistematici, come fanno le altre ingegnerie con ponti ed edifici.

DEFINIZIONE

L’**ingegneria del software** è la disciplina che studia come produrre sistemi software entro i costi e i tempi preventivati e con una qualità accettabile, organizzando lo sviluppo in fasi, con processi, tecniche di progettazione e strumenti di verifica.

Lo sviluppo di un sistema attraversa alcune fasi ricorrenti, che ritroveremo in tutto il corso. Nell’**analisi** si raccolgono i requisiti, cioè ciò che il cliente vuole, espressi in linguaggio naturale strutturato o con story card. Nella **progettazione** si decide come organizzare il sistema in classi e componenti, usando la programmazione orientata agli oggetti (OOP), i design pattern, i diagrammi UML e il refactoring. Nell’**implementazione** si scrive il codice, qui in linguaggio Java. Nel **test** (o convalida) si controlla che il codice faccia ciò che deve. Infine nella **manutenzione** il sistema viene corretto e fatto evolvere. Il modo in cui queste fasi si susseguono è detto **processo di sviluppo**: a cascata, evolutivo, agile come XP, e così via.

Ogni fase è un’**attività** che produce un **artefatto**, cioè un risultato concreto che passa alla fase successiva: un documento dei requisiti, un modello di progetto, il codice, l’esito dei test. La figura riassume attività, artefatti e le principali tecniche usate in ciascuna fase.

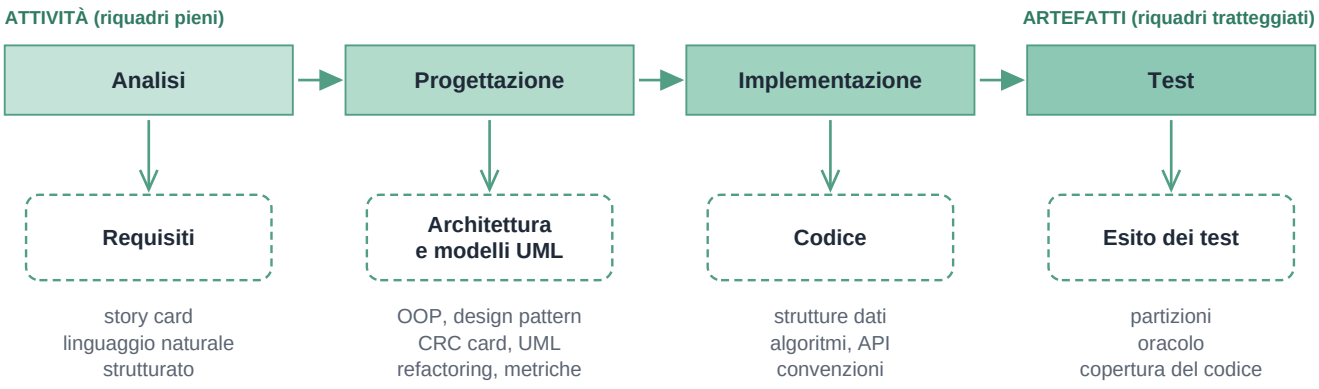


Figura 1.1: le fasi dello sviluppo; in alto le attività, al centro gli artefatti prodotti, in basso le tecniche usate.

Le caratteristiche del software

La caratteristica che distingue il software da qualunque altro prodotto ingegneristico è la **modificabilità**. Un sistema software non ha parti fisiche, non è fatto di atomi: per cambiarlo basta riscrivere del testo. Un ponte resta com'è; un programma viene modificato di continuo, ed è proprio questa facilità a renderlo difficile da governare.

C'è poi un paradosso: più un sistema ha successo, più dovrà cambiare. Le ragioni sono tre.

- La realtà in cui il sistema opera cambia, e con essa cambiano le esigenze degli utenti.
- Le richieste di estensione aumentano al crescere del successo: chi usa un sistema che funziona chiede subito nuove funzioni.
- Un sistema di successo sopravvive all'hardware per cui era stato sviluppato, e va quindi adattato a nuove piattaforme.

Per questo motivo, nel corso, scrivere codice **facile da modificare** sarà un obiettivo importante quanto scrivere codice corretto.

La qualità del software

Per dire se un sistema è "buono" servono criteri operativi, cioè proprietà che si possano valutare. Il primo e più importante è la correttezza.

DEFINIZIONE

Un sistema software è **corretto** quando aderisce al suo scopo ed è conforme alle specifiche. In pratica la correttezza si scompone in due domande: il sistema fa quello che il cliente vuole? Il sistema soddisfa le specifiche che erano state raccolte? Alla seconda domanda si risponde con il **testing**.

Accanto alla correttezza si considerano altri attributi di qualità, riassunti nella tabella.

ATTRIBUTO	SIGNIFICATO
Correttezza	Il sistema fa ciò che deve ed è conforme alle specifiche.
Efficienza	Il sistema non spreca risorse: tempo di calcolo, memoria, rete.
Manutenibilità	Il sistema può essere corretto ed esteso con poco sforzo e senza introdurre errori.
Dependability	Il sistema merita fiducia: è affidabile, disponibile e sicuro (vedi sotto).
Usabilità	Il sistema è facile da imparare e da usare per gli utenti a cui è destinato.

Il termine italiano "sicurezza" traduce due concetti inglesi diversi, entrambi compresi nella dependability. La **security** protegge sistema e dati riservati da accessi non autorizzati. La **safety** riguarda invece il fatto che il sistema, funzionando, non metta in pericolo la vita o la salute delle persone, come nel software di un pacemaker o di un impianto frenante.

DA RICORDARE

Il software si distingue per la **modificabilità**, e un sistema di successo cambia sempre. La qualità si misura con correttezza, efficienza, manutenibilità, dependability e usabilità. La correttezza si verifica con il **testing**. Security e safety non sono sinonimi: la prima protegge i dati, la seconda le persone.

Un primo sguardo a Java

Il corso usa Java, un linguaggio orientato agli oggetti. Un programma Java è fatto di classi; ogni classe pubblica sta in un file con lo stesso nome ed estensione `.java`, si compila con `javac Nome.java` e si esegue con `java Nome`. L'esecuzione parte dal metodo `main`. Le parole chiave che servono fin da subito sono raccolte nella tabella; chi conosce il C++ ne ritroverà molte.

PAROLA CHIAVE	USO
class	Definisce un tipo, da cui si creano le istanze (oggetti).
new	Crea un'istanza di una classe.
public, private	Indicano chi può accedere a classi, campi e metodi.
static	Il campo o il metodo appartiene alla classe, non alla singola istanza.
final	Campo o variabile assegnabile una sola volta; classe che non può essere ereditata; metodo che non può essere ridefinito.
void, return	void: il metodo non restituisce valori; return termina il metodo e restituisce l'eventuale valore.
import	Indica dove trovare una classe definita altrove (per esempio in una libreria).
throws	Dichiara le eccezioni che il metodo non gestisce e lascia passare al chiamante.

Un esempio: dal codice che funziona al codice ben scritto

Proviamo a sviluppare un componente piccolissimo, che potrebbe far parte di un sistema più grande, chiedendoci se il risultato sia riusabile, modificabile e corretto. I requisiti sono questi: abbiamo alcuni file di letture di un contatore elettrico, con un valore numerico per riga. Dobbiamo leggere i valori di ciascun file, tenere solo quelli non duplicati, calcolarne la somma e trovarne il massimo (il picco).

Una prima versione, scritta di getto e fatta crescere aggiungendo un requisito alla volta, potrebbe essere la seguente.

```
public class AnalisiConsumi {                 // versione monolitica
    private final List<String> letture = new ArrayList<>();
    private float somma, picco;

    public float analizza(String cartella, String nome) throws IOException {
        LineNumberReader f = new LineNumberReader(new FileReader(new File(cartella, nome)));
        somma = picco = 0;
        while (true) {
            String riga = f.readLine();          // legge una riga del file
            if (riga == null) break;             // fine del file: esce dal ciclo
            if (!letture.contains(riga)) {       // scarta i duplicati
                letture.add(riga);
                float x = Float.parseFloat(riga);  // da testo a numero
                somma += x;                        // aggiorna la somma
                if (picco < x) picco = x;          // aggiorna il massimo
            }
        }
        f.close();
        return somma;
    }
}
```

Il codice funziona, eppure ha un problema serio: il metodo `analizza` fa tutto in un unico flusso. Legge, filtra, somma e cerca il massimo mescolando le quattro operazioni nello stesso ciclo. Le conseguenze sono concrete.

1. **Non si può verificare dall'esterno.** Per controllare che tutti i valori del file siano stati letti bisognerebbe modificare il metodo; lo stesso vale per la somma e il massimo. Un buon codice deve poter essere verificato dall'esterno, chiamandolo.
2. **Non si può riusare.** Se un altro componente avesse bisogno solo del massimo, dovrebbe rieseguire tutto il resto.
3. **È difficile da modificare.** Se volessimo conservare anche i valori duplicati, toccando il ciclo rischieremmo di alterare la somma e il massimo, perché tutto è intrecciato.

Riscrivere il metodo separando i cicli, uno per leggere, uno per sommare e uno per il massimo, non risolve il problema: il codice resta un unico blocco che fa troppe cose. Questo difetto ha un nome.

DEFINIZIONE

Lo **spaghetti code** è un **antipattern**, cioè una soluzione ricorrente ma sbagliata. Il codice ha un flusso aggrovigliato come un piatto di spaghetti: metodi lunghi che fanno tutto, difficili da capire, verificare e modificare.

I sintomi tipici dello spaghetti code sono i seguenti.

- Metodi lunghi, senza parametri, che usano variabili globali.
- Il flusso di esecuzione è deciso dall'implementazione interna dell'oggetto e non da chi lo chiama.
- Interazioni minime fra oggetti; nomi di classi e metodi tipici della programmazione procedurale.
- Ereditarietà e polimorfismo non usati, quindi riuso impossibile.
- Gli oggetti non mantengono uno stato utile fra un'invocazione e l'altra.

Le cause sono quasi sempre l'inesperienza con la programmazione orientata agli oggetti e la mancanza di progettazione.

La versione orientata agli oggetti

La soluzione è dare a ogni operazione il suo metodo. La classe `Consumi` offre un metodo per leggere il file, uno per inserire una lettura scartando i duplicati, uno per la somma, uno per il massimo, uno per svuotare i dati e due metodi che restituiscono i risultati. La figura mostra la scomposizione con la notazione UML delle classi, che studieremo nel dettaglio: il segno più indica un membro pubblico, il segno meno un membro privato.

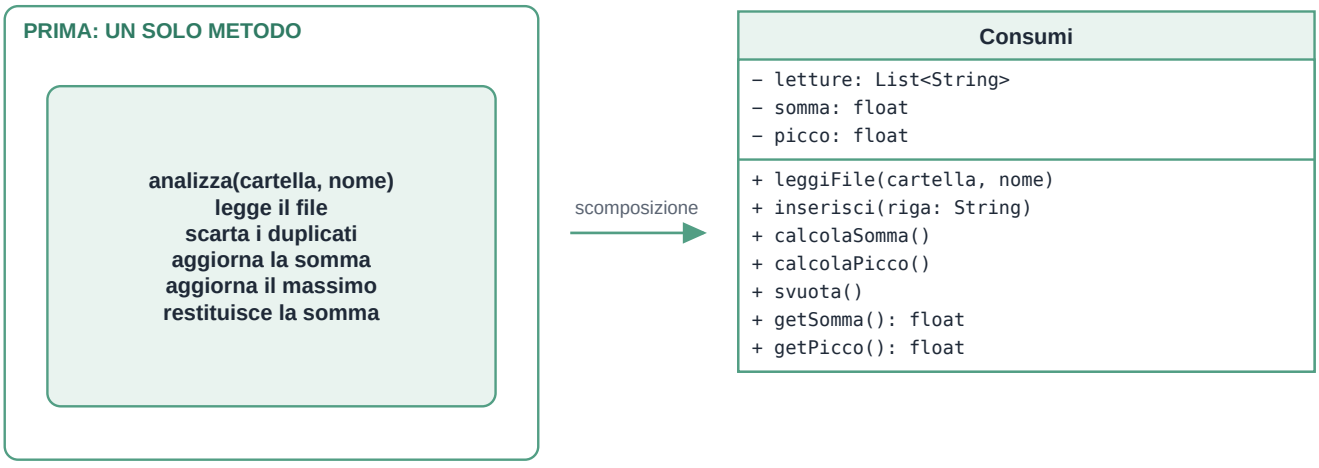


Figura 1.2: un unico metodo che fa tutto, scomposto in una classe con un metodo per ogni responsabilità.

```
public void inserisci(String riga) {          // una sola responsabilita'
    if (!letture.contains(riga)) letture.add(riga);
}

public void calcolaPicco() {                  // command: modifica lo stato
    picco = Float.NEGATIVE_INFINITY;          // valore iniziale sicuro
    for (String v : letture)                  // enhanced for
        if (picco < Float.parseFloat(v)) picco = Float.parseFloat(v);
}

public float getPicco() {                     // query: restituisce, non modifica
    return picco;
}
```

Il chiamante decide l'ordine delle operazioni: crea un oggetto `Consumi`, chiama `leggiFile`, poi `calcolaSomma` e `calcolaPicco`, e legge i risultati con `getSomma` e `getPicco`. I vantaggi sono evidenti.

- Ogni metodo ha una sola, piccola responsabilità, quindi è facile da capire e da verificare singolarmente.
- Il flusso delle chiamate è indipendente dai singoli algoritmi: si può cambiare il calcolo del massimo senza toccare la lettura.
- I servizi offerti dai metodi si possono riusare semplicemente richiamandoli.

Il ciclo `for (String v : letture)` è il cosiddetto **enhanced for**: a ogni passata fornisce un elemento della lista. Si può usare con tutti i tipi che implementano l'interfaccia `Iterable`, come le liste.

DEFINIZIONE

Un **metodo query** restituisce un risultato (lo si vede dal tipo di ritorno) e **non modifica** lo stato dell'oggetto. Un **metodo command** (o modificatore) **cambia lo stato** ma non restituisce alcun valore.

I metodi query si possono chiamare liberamente, senza preoccuparsi degli effetti; i metodi command vanno chiamati con più attenzione, perché modificano i dati. Tenere separate le due categorie rende il codice più prevedibile.

Le librerie Java usate

TIPO O METODO	COMPORTAMENTO
List	Interfaccia che rappresenta una sequenza ordinata di elementi.
ArrayList	Implementazione di List basata su un array che cresce automaticamente.
add(e)	Metodo di List: aggiunge l'elemento in fondo alla lista.
contains(e)	Metodo di List: restituisce true se la lista contiene l'elemento.
Float.parseFloat(s)	Converte la stringa s in float; se la stringa non contiene un numero lancia una NumberFormatException.
readLine()	Metodo di LineNumberReader: restituisce la riga successiva del file, oppure null alla fine del file.

La differenza fra `List` e `ArrayList` è quella fra un'interfaccia, che dice *che cosa* si può fare, e una sua implementazione, che dice *come* viene fatto. Per questo si scrive `List<String> letture = new ArrayList<>()`: il resto del codice dipende solo dall'interfaccia, e l'implementazione si può cambiare in un solo punto.

Misurare il codice: le metriche

Per confrontare le versioni si possono usare due **metriche** semplici: il numero di metodi e le **LOC** (lines of code, righe di codice). In un esempio come questo la versione monolitica ha un solo metodo di una ventina di righe, mentre quella orientata agli oggetti ha 7 metodi per una quarantina di righe in tutto, cioè circa 6 righe per metodo. Le righe complessive aumentano, ma ogni pezzo diventa piccolo e comprensibile. Come riferimento, ecco i valori approssimativi di due noti progetti open source.

SISTEMA	LOC	CLASSI	METODI	LOC PER METODO
JUnit	22 000	231	1 200	circa 18
JHotDraw	28 000	600	4 814	circa 6

La media si ottiene dividendo le LOC per il numero di metodi: 22 000 diviso 1 200 dà circa 18, e 28 000 diviso 4 814 dà circa 6. JHotDraw, progettato con largo uso di design pattern, ha metodi molto più brevi.

DA RICORDARE

La **correttezza** si verifica con i test, e un codice ben scritto si può verificare dall'esterno. Lo **spaghetti code** è un antipattern: un metodo che fa tutto non si verifica, non si riusa e non si modifica. La cura è assegnare a ciascun metodo **un unico compito** e separare i metodi **query** dai metodi **command**. Metodi brevi, misurati in LOC per metodo, sono un buon indicatore di progettazione orientata agli oggetti.

CAPITOLO 2

2. Qualità, test e principi di progettazione

Come rendere il codice verificabile, che cosa significa sviluppare guidati dai test e quali principi guidano la scelta delle responsabilità di metodi e classi.

Quattro miglioramenti del codice

Nel capitolo precedente abbiamo trasformato un metodo monolitico in una classe con molti metodi piccoli. Quella trasformazione si può descrivere come la somma di quattro miglioramenti, che valgono per qualunque codice.

1. **Separare le operazioni**, ognuna in un proprio metodo. Così ogni operazione si può verificare e riusare da sola.
2. **Costruire astrazioni** via via più utili: metodi di livello basso che fanno un passo elementare e metodi di livello alto che li richiamano. Lo stesso si fa con le classi.
3. **Separare command e query**: le operazioni che aggiornano lo stato (command) stanno in metodi diversi da quelle che restituiscono valori (query). Le query ottengono informazioni, accedendo ai dati o facendo calcoli; i command effettuano cambiamenti.
4. **Rendere osservabile lo stato** dell'oggetto con metodi opportuni, in modo da poterlo controllare nei test.

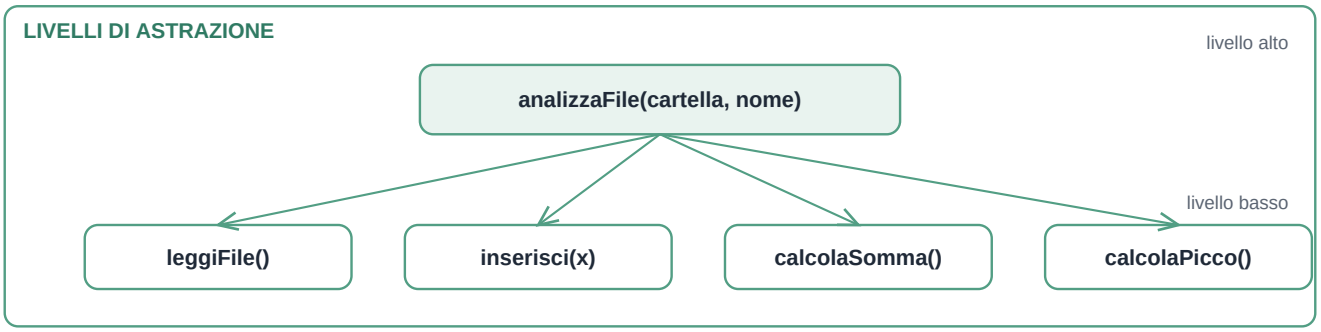


Figura 2.1: un metodo di livello alto si legge come una sequenza di passi, ciascuno affidato a un metodo di livello basso.

Verificare il codice con i test

Una volta che le operazioni sono separate e lo stato è osservabile, si può scrivere una seconda classe che controlla la prima. Ogni metodo di test prepara dei dati noti, chiama l'operazione da verificare e confronta il risultato con quello atteso, stampando un esito positivo o negativo. L'idea è semplice: se so che le letture sono 12, 7 e 30, la somma deve valere 49 e il massimo 30.

```
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");
    }
}
```

Ogni test chiama per prima cosa il metodo che prepara i dati. In questo modo i test sono **indipendenti** fra loro: l'esito di uno non dipende da ciò che ha fatto un altro, e possono essere eseguiti in qualunque ordine. I test sono anche **autovalutanti**: stabiliscono da soli se il risultato è giusto, senza che una persona debba leggere i numeri stampati. Conviene inoltre provare i **casi limite**, per esempio la lista vuota, dove somma e massimo devono avere un valore ben definito.

DA RICORDARE

I numeri in virgola mobile non si confrontano con l'uguaglianza esatta. La somma di valori float può differire dal valore atteso per un errore di arrotondamento minuscolo, e il test fallirebbe pur essendo il codice corretto. Si controlla invece che la differenza, in valore assoluto, sia minore di una piccola tolleranza, come fa `Math.abs(a - b) < 0.001`.

Query al posto degli attributi

Un ulteriore passo di pulizia consiste nel far restituire il risultato direttamente ai metodi di calcolo: `calcolaSomma()` e `calcolaPicco()` diventano query che restituiscono un float, e gli attributi `somma` e `picco` spariscono. Meno attributi significa meno stato condiviso, e quindi meno **effetti collaterali**: nessun altro metodo può lasciare un valore vecchio o sbagliato in un attributo che poi qualcuno legge. Conviene anche memorizzare le letture come `List<Float>` invece che come stringhe, così la conversione avviene una volta sola, in lettura, e i duplicati si riconoscono per valore e non per testo.

Sviluppo guidato dai test

DEFINIZIONE

Nel **Test Driven Development** (TDD) si scrive **prima il test** e poi il codice che soddisfa il requisito. Scrivendo il test lo sviluppatore sceglie i nomi delle classi e dei metodi e il modo in cui verranno chiamati: il test diventa così un primo atto di progettazione.

I test hanno più di un ruolo. Verificano la **correttezza**, perché dimostrano che il codice soddisfa i requisiti, e soddisfare e verificare i requisiti è ciò che dà qualità al sistema. **Documentano** le condizioni sotto le quali il codice funziona: chi legge un test capisce come si usa una classe. Infine **proteggono** il codice da modifiche indesiderate: se una modifica rompe qualcosa, un test fallisce e ce ne accorgiamo subito.

Responsabilità, coesione e astrazioni

DEFINIZIONE

Il **principio di singola responsabilità** (SRP, Single Responsibility Principle) afferma che ogni metodo e ogni classe devono avere un solo compito. La singola responsabilità è cruciale per la comprensione, per il riuso e per l'ereditarietà.

Quando i compiti sono suddivisi su vari metodi, e quindi su più classi, il codice diventa **coeso**: ogni unità contiene elementi che lavorano allo stesso scopo. Una classe coesa è come un ufficio con un compito preciso: sai sempre a quale sportello rivolgerti.

Lo sviluppatore orientato agli oggetti costruisce **astrazioni**, e il nome che dà a classi e metodi è importantissimo, perché deve descriverne l'obiettivo. Un metodo chiamato `calcolaPicco` dice cosa fa senza bisogno di leggerne il corpo; un metodo chiamato `elabora` non dice nulla.

Compilare ed eseguire un programma Java

Per sviluppare in Java servono gli strumenti di sviluppo (JDK) e l'ambiente di esecuzione. I passaggi sono tre.

1. Si salva la classe nel file `NomeClasse.java`. All'inizio del file le istruzioni `import` indicano dove trovare i tipi di libreria usati, per esempio `import java.io.File;`.
2. Si compila con `javac NomeClasse.java`. Se la compilazione va a buon fine non compare alcun messaggio e viene generato il file `NomeClasse.class`, che contiene il bytecode.
3. Si esegue con `java NomeClasse`. La classe passata al comando deve avere il metodo `public static void main(String[] args)`, da cui comincia l'esecuzione.

DA RICORDARE

Il codice si rende verificabile separando le operazioni, costruendo astrazioni, separando command e query e rendendo lo stato osservabile. I test devono essere **indipendenti** e **autovalutanti**. Nel **TDD** si scrive prima il test e poi il codice. Il **principio di singola responsabilità** porta a codice coeso, e i nomi di classi e metodi devono descriverne lo scopo. Si compila con `javac` e si esegue con `java` una classe che possiede il metodo `main`.

CAPITOLO 3

3. Test con JUnit

Lo strumento standard per scrivere ed eseguire test in Java: annotazioni, asserzioni, test parametrici, copertura del codice e organizzazione del progetto con Maven.

Che cos'è JUnit

Scrivere test a mano, con classi che stampano OK o FAILED, funziona ma è scomodo. **JUnit** è un **framework** per il test di applicazioni Java: fornisce un motore che esegue i test e una libreria (API) per scriverli. La versione più recente è JUnit 6. Con JUnit non serve più un metodo `main` che chiama i test uno per uno, né stampare gli esiti: il motore trova da solo i metodi di test, li esegue e riporta quali sono passati e quali no.

DEFINIZIONE

Un'**annotazione** Java è un metadato, scritto con il simbolo `@`, associato a un elemento del codice (classe, metodo, campo). Non cambia ciò che il codice fa, ma fornisce informazioni aggiuntive al compilatore o agli strumenti, come il motore di JUnit.

Le annotazioni principali

JUnit usa le annotazioni sui metodi della classe di test per sapere che ruolo ha ciascuno.

ANNOTAZIONE	SIGNIFICATO
@Test	Il metodo è un test e va eseguito dal motore.
@BeforeEach	Il metodo viene eseguito prima di ogni singolo test: serve a ripristinare dati noti, così i test restano indipendenti.
@BeforeAll	Il metodo viene eseguito una sola volta, prima di tutti i test; deve essere static.
@ParameterizedTest	Il test viene eseguito più volte con valori di ingresso diversi.
@CsvSource	Fornisce i valori per un test parametrico, una riga di valori separati da virgole per ogni esecuzione.

La differenza fra `@BeforeEach` e `@BeforeAll` è importante. Se un test modifica i dati (per esempio svuota la lista), con `@BeforeAll` i test successivi troverebbero i dati già alterati; con `@BeforeEach` invece ogni test riparte da zero.

Le asserzioni

Per verificare il comportamento del codice si usano i metodi statici della classe `Assertions`. Un'asserzione che non è soddisfatta fa fallire il test, e JUnit riporta il valore atteso e quello ottenuto.

ASSERZIONE	IL TEST PASSA SE
assertEquals(atteso, effettivo)	i due valori sono uguali; l'ordine conta: prima il valore atteso, poi quello calcolato.
assertEquals(atteso, effettivo, delta)	per i numeri in virgola mobile: la differenza è al massimo delta.
assertTrue(condizione)	la condizione è vera.
assertNull(oggetto)	il riferimento è null.

```
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 parametrici

Spesso lo stesso test va ripetuto con molti valori. Invece di copiare il metodo, lo si annota con `@ParameterizedTest` e si forniscono i valori con `@CsvSource`: ogni stringa è una riga di valori, che JUnit assegna ai parametri del metodo nell'ordine. In questo modo i valori non sono scritti dentro il codice del test.

```
@ParameterizedTest
@CsvSource({ "2, 3, 5", "10, 5, 15", "0, 0, 0" })   // a, b, risultato atteso
void somma(int a, int b, int atteso) {
    assertEquals(atteso, calcolatrice.somma(a, b));
}
```

Il metodo viene eseguito tre volte: con a uguale a 2 e b uguale a 3 si attende 5, poi con 10 e 5 si attende 15, infine con 0 e 0 si attende 0.

Copertura del codice

Durante un test si esegue solo una parte dell'applicazione. Per sapere quanta, si misura la copertura.

DEFINIZIONE

La **copertura del codice** (code coverage) è il rapporto, espresso in percentuale, fra le linee di codice eseguite dai test e le linee di codice totali.

$$\text{copertura} = (\text{linee eseguite} / \text{linee totali}) \times 100$$

Per esempio, se un'applicazione ha 200 linee e i test ne eseguono 140, la copertura è 140 diviso 200, cioè 0,7, moltiplicato per 100: il 70 per cento. Con il 100 per cento i test eseguono tutte le linee. Attenzione però: eseguire una linea non significa averla verificata in tutti i casi, quindi una copertura alta è necessaria ma non sufficiente. Per calcolarla si tracciano le linee eseguite, come fanno gli ambienti di sviluppo.

Organizzare il progetto con Maven

Apache Maven è uno strumento per la gestione dei progetti e l'automazione del **build**, cioè della costruzione del programma eseguibile. Standardizza compilazione, gestione delle dipendenze, test, impacchettamento e rilascio attraverso convenzioni e configurazioni. Un progetto Maven ha una struttura di cartelle prestabilita: il codice dell'applicazione sta in `src/main/java`, quello di test in `src/test/java`, e gli eventuali package si trovano dentro la cartella `java`.

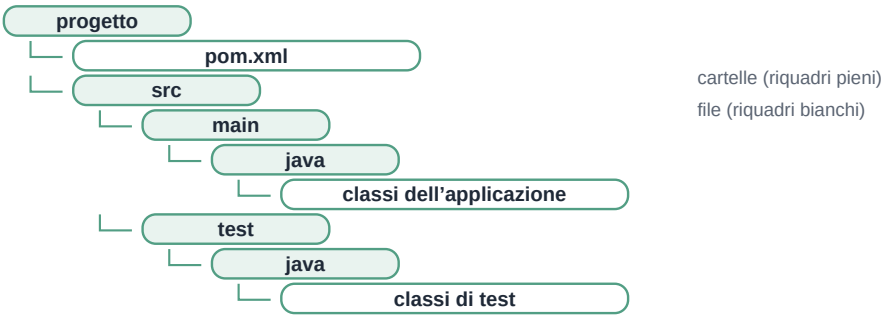


Figura 3.1: la struttura convenzionale di un progetto Maven.

Il file **pom.xml** (Project Object Model) descrive il progetto: nome e versione, versione di Java, dipendenze, plugin e configurazioni, racchiusi in tag XML predefiniti. Per usare una libreria, la si cerca nel repository **Maven Central** e si aggiunge il blocco `dependency` indicato da chi la fornisce; Maven scarica automaticamente la libreria e tutte le sue dipendenze. Per JUnit servono due dipendenze, l'API e il motore, con ambito `test`, cioè usate solo per i test e non incluse nel programma finale.

Un progetto si crea con l'ambiente di sviluppo oppure con il comando `mvn archetype:generate`. Maven lavora per **fasi**, e l'esecuzione di una fase esegue anche tutte le precedenti.

I comandi principali sono `mvn compile`, che compila il codice; `mvn test`, che esegue anche i test di unità; `mvn package`, che produce un file .jar distribuibile; `mvn verify`, che esegue anche i controlli di qualità configurati.

DA RICORDARE

JUnit esegue i test individuati dalle **annotazioni**: `@Test`, `@BeforeEach` (prima di ogni test), `@BeforeAll` (una volta, metodo static). Le verifiche si fanno con `assertEquals(atteso, effettivo)`, `assertTrue`, `assertNull`. I test parametrici usano `@ParameterizedTest` e `@CsvSource`. La **copertura** è la percentuale di linee eseguite dai test. Maven impone la struttura `src/main/java` e `src/test/java` e descrive il progetto nel `pom.xml`.

CAPITOLO 4

4. Refactoring

Come migliorare la struttura del codice senza cambiarne il comportamento, perché conviene farlo continuamente e tre tecniche fondamentali: Estrai Metodo, Sostituisci Temp con Query, Dividi Variabile Temp.

Che cos’è il refactoring

DEFINIZIONE

Il **refactoring** è il processo che modifica un sistema software in modo che il suo **comportamento esterno non cambi**, cioè i requisiti funzionali soddisfatti restano gli stessi, ma la sua **struttura interna migliori**.

È come riordinare un armadio: alla fine contiene gli stessi vestiti, ma trovarli e aggiungerne di nuovi diventa molto più facile. La trasformazione vista nei capitoli precedenti, dal metodo unico alla classe con un metodo per ogni operazione, è un refactoring; e anche eliminare gli attributi superflui, per prevenire effetti collaterali, lo è. Un esempio ancora più semplice è consolidare un frammento di codice duplicato in entrambi i rami di un `if`, portandolo fuori dalla condizione.

L’idea di fondo è riconoscere che è difficile ottenere fin dall’inizio una progettazione adeguata, e che quindi la progettazione va migliorata a poco a poco, durante e dopo l’implementazione. Il riferimento classico è il libro di Martin Fowler, *Refactoring: Improving the Design of Existing Code*.

Perché fare refactoring

- **Migliora la progettazione.** Senza refactoring la progettazione si deteriora man mano che si modifica il codice; con il refactoring diventa più facile incorporare requisiti nuovi o che cambiano.
- **Rende il codice più facile da capire**, perché la struttura migliora e il codice duplicato viene rimosso.
- **Aiuta a scoprire i bug**: ristrutturare il codice obbliga a comprenderlo a fondo, e questo fa trovare errori o anticipare quelli potenziali.
- **Fa programmare più velocemente**, perché una buona progettazione è più facile da cambiare.

Fra i vantaggi concreti: spesso la dimensione del codice si riduce, le strutture complicate diventano più semplici da capire e manutenere, e si evita di accumulare **debito tecnico**, cioè quel costo futuro che si paga quando si rimandano i miglioramenti e ogni modifica successiva diventa più faticosa.

Come si fa refactoring

Si applicano tecniche precise, catalogate da Fowler, e soprattutto si fanno **test costantemente**. Prima si scrivono i test, poi si fa una piccola modifica di struttura e si verifica che tutti i test passino ancora. Se un test non passa, il refactoring ha compromesso qualcosa che funzionava: lo si scopre subito e si interviene subito.

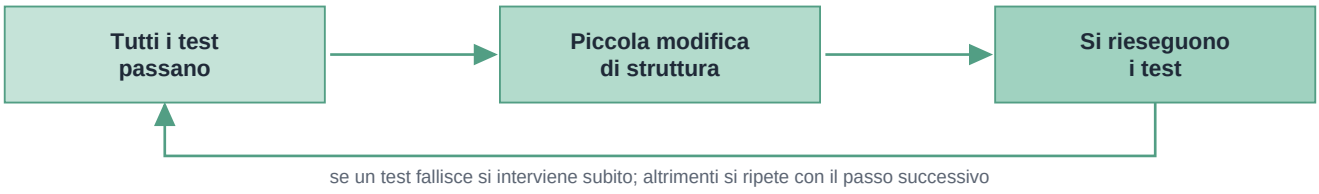


Figura 4.1: il ciclo del refactoring, fatto di piccoli passi sempre protetti dai test.

Estrai Metodo

DEFINIZIONE

Estrai Metodo: un frammento di codice che si può raggruppare diventa un metodo, il cui nome spiega lo scopo del frammento. Nel punto di partenza il frammento viene sostituito dalla chiamata al nuovo metodo.

MOTIVAZIONI

Si cercano metodi lunghi, oppure codice che ha bisogno di commenti per essere capito: il commento è il segnale che quel frammento merita un nome. Metodi piccoli hanno più probabilità di essere riusati da altri metodi, e con nomi ben scelti i metodi di alto livello, quelli che richiamano i piccoli, sembrano una sequenza di commenti. Inoltre ridefinire un comportamento in una sottoclasse (override) è più semplice se i metodi sono piccoli.

MECCANISMI

1. Creare un nuovo metodo con un nome che comunica la sua intenzione, cioè *che cosa* fa e non *come*.
2. Copiare il codice estratto dal metodo sorgente al nuovo metodo.
3. Se il codice estratto usa variabili locali del metodo sorgente, farle diventare parametri del nuovo metodo.
4. Le variabili usate solo dentro il codice estratto diventano variabili temporanee del nuovo metodo.
5. Se il codice estratto modifica una variabile locale del sorgente, provare a rendere il nuovo metodo una query e assegnare il suo risultato a quella variabile.
6. Sostituire nel sorgente il codice estratto con la chiamata al nuovo metodo, compilare e rieseguire i test.

Un esempio. Un metodo stampa la ricevuta di un cliente: intestazione, calcolo del totale e dettagli, separati da commenti.

```
public void stampaRicevuta() {
    // intestazione
    System.out.println("*** RICEVUTA ***");
    // calcolo del totale
    double totale = 0;
    for (Acquisto a : acquisti) totale += a.getPrezzo();
    // dettagli
    System.out.println("cliente: " + cliente);
    System.out.println("totale: " + totale);
}
```

Ogni commento indica un frammento da estrarre. Il totale è una variabile locale assegnata dal frammento centrale, quindi quel frammento diventa una query che lo restituisce; i dettagli usano il totale, quindi il metodo estratto lo riceve come parametro.

```
public void stampaRicevuta() {          // ora si legge come una sequenza di passi
    stampaIntestazione();
    double totale = calcolaTotale();
    stampaDettagli(totale);
}

private void stampaIntestazione() { System.out.println("*** RICEVUTA ***"); }
private double calcolaTotale() {        // query: restituisce il totale
    double totale = 0;
    for (Acquisto a : acquisti) totale += a.getPrezzo();
    return totale;
}

private void stampaDettagli(double totale) {   // la variabile locale diventa parametro
    System.out.println("cliente: " + cliente);
    System.out.println("totale: " + totale);
}
```

Sostituisci Temp con Query

DEFINIZIONE

Sostituisci Temp con Query: una variabile temporanea che tiene il risultato di un'espressione viene sostituita da un metodo che calcola quell'espressione. Tutti i riferimenti alla variabile diventano chiamate al metodo, che può essere richiamato anche altrove.

Le variabili temporanee sono locali: sono visibili solo dentro il metodo. Per questo spingono a scrivere metodi lunghi, perché l'unico modo di usarle è restare nello stesso metodo. Trasformandole in query, qualunque metodo può ottenere quel dato. Spesso questa tecnica si applica **prima** di Estrai Metodo, perché elimina le variabili locali che renderebbero difficile l'estrazione. È facile quando la variabile è assegnata una sola volta.

Il procedimento: si cerca una variabile assegnata una sola volta e la si dichiara final; se il compilatore non protesta, l'assegnazione è davvero unica. Si estrae la parte destra dell'assegnazione in un nuovo metodo, si sostituisce ogni riferimento alla variabile con la chiamata e si elimina la variabile; infine si compila e si testa.

```
// prima: due variabili temporanee
public double prezzoFinale() {
    double imponibile = biglietti * prezzoUnitario;
    double sconto;
    if (imponibile > 100) sconto = 0.90;
    else sconto = 0.97;
    return imponibile * sconto;
}

// dopo: due query riusabili
public double prezzoFinale() { return imponibile() * sconto(); }

private double imponibile() { return biglietti * prezzoUnitario; }

private double sconto() {
    if (imponibile() > 100) return 0.90;
    return 0.97;
}
```

Dividi Variabile Temp

DEFINIZIONE

Dividi Variabile Temp: se una variabile temporanea è assegnata più di una volta, ma non è una variabile di ciclo né un accumulatore, si usa una variabile separata per ciascuna assegnazione.

Alcune variabili sono legittimamente assegnate molte volte: il contatore di un ciclo, o un accumulatore che somma valori. Una variabile che invece tiene il risultato di un calcolo per usarlo dopo dovrebbe essere assegnata una sola volta. Se è assegnata due volte, ha due responsabilità, e usare lo stesso nome per due cose diverse confonde chi legge.

MECCANISMI

1. Rinominare la variabile nella dichiarazione e nella prima assegnazione, con un nome che dica cosa contiene.
2. Dichiararla final.
3. Aggiornare tutti i riferimenti fino alla seconda assegnazione.
4. Dichiarare una nuova variabile per la seconda assegnazione.
5. Compilare e testare; ripetere a piccoli passi per ogni assegnazione successiva.

```
// prima: temp ha due significati
double temp = 2 * (base + altezza);
System.out.println(temp);
temp = base * altezza;
System.out.println(temp);

// dopo: una variabile per ogni responsabilita'
final double perimetro = 2 * (base + altezza);
System.out.println(perimetro);
final double area = base * altezza;
System.out.println(area);
```

DA RICORDARE

Il **refactoring** cambia la struttura e non il comportamento esterno, procede a piccoli passi e ogni passo è verificato dai test. **Estrai Metodo** trasforma un frammento in un metodo con un nome che ne spiega lo scopo. **Sostituisci Temp con Query** trasforma una variabile locale in un metodo, e spesso si fa prima di Estrai Metodo. **Dividi Variabile Temp** dà una variabile diversa a ogni responsabilità.

CAPITOLO 5

5. Classi, ereditarietà e polimorfismo

Come riusare classi esistenti con l'ereditarietà, che ruolo hanno interfacce e classi astratte, come Java sceglie il metodo da eseguire e un primo design pattern che sfrutta tutto questo: Template Method.

Ereditarietà

Capita spesso di aver bisogno di classi simili a una già esistente, con qualche attributo o metodo in più o leggermente diverso. Copiare la classe e modificarla non è pratico: le classi si moltiplicano, il codice si duplica e ogni correzione va ripetuta. Il riuso deve invece avvenire **senza modificare il codice esistente**, che funziona, e in modo semplice per il programmatore. Lo strumento è l'ereditarietà.

DEFINIZIONE

Con l'**ereditarietà** si definisce una nuova classe, la **sottoclasse**, indicando solo ciò che ha in più o di diverso rispetto a una classe esistente, la **superclasse**. In Java si scrive `class Responsabile extends Dipendente`.

Una sottoclasse eredita tutti gli attributi e i metodi della superclasse e li può usare come se fossero suoi; può aggiungere nuovi metodi e attributi; può **ridefinire** (override) i metodi ereditati; non può invece eliminare metodi o attributi della superclasse. Nell'esempio della figura, `Responsabile` è un `Dipendente` che in più coordina un gruppo di persone, e ridefinisce `stampa()` per mostrare anche il suo gruppo. Dentro il metodo ridefinito, `super.stampa()` richiama la versione della superclasse, così si aggiunge comportamento senza riscrivere quello esistente.

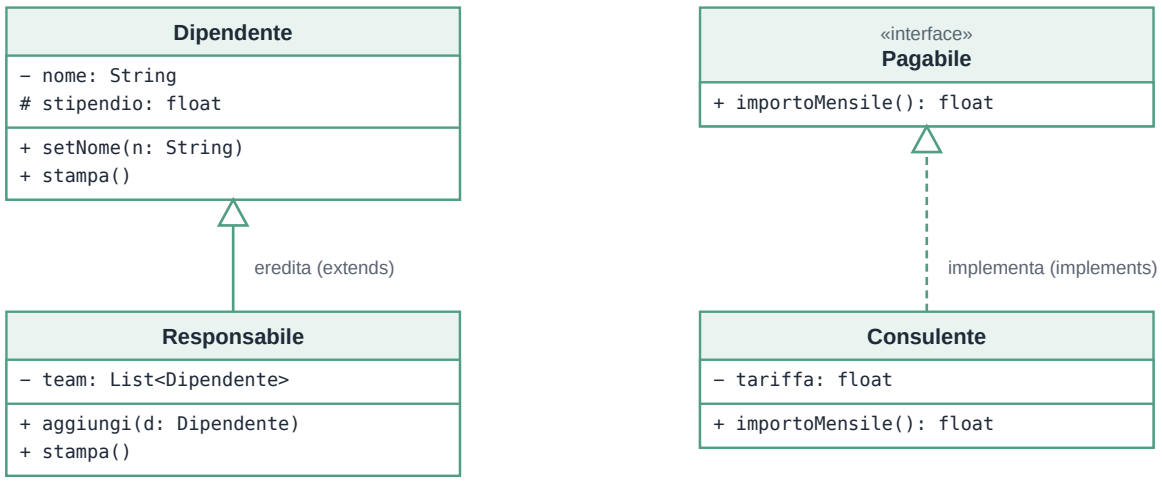


Figura 5.1: a sinistra una sottoclasse (freccia continua con triangolo vuoto), a destra una classe che implementa un'interfaccia (freccia tratteggiata).

VISIBILITÀ

MODIFICATORE	SIMBOLO UML	CHI VEDE IL MEMBRO
private	−	Solo la classe stessa, non le sottoclassi.
protected	#	La classe e le sue sottoclassi (in Java anche le classi dello stesso package).
public	+	Tutti, sottoclassi comprese.

Anche i nomi contano: il nome di una sottoclasse dovrebbe comunicare a quale classe somiglia e in che cosa differisce. Le classi alla radice di una gerarchia hanno nomi concisi, quelle più specializzate possono averne di più lunghi.

Interfacce e classi astratte

DEFINIZIONE

Un'**interfaccia** definisce un tipo elencando le signature dei metodi pubblici, senza il loro corpo. Non ha costruttori né attributi non inizializzati e non si può istanziare. Una classe **implementa** un'interfaccia (`implements`) quando fornisce il corpo di tutti i suoi metodi.

Una **classe astratta** è una classe implementata solo in parte: alcuni metodi hanno il corpo, altri sono `abstract`, cioè solo dichiarati. Neanche una classe astratta si può istanziare.

L'interfaccia è un contratto: dice ai **client**, cioè alle classi che la usano, che cosa possono invocare. Per creare oggetti e chiamare metodi si può usare una qualunque delle implementazioni disponibili, e un client che dipende solo dall'interfaccia **resta immutato** quando l'implementazione cambia. È come una presa elettrica: qualunque elettrodomestico con la spina giusta funziona, e la presa non deve sapere quale sia.

Un metodo astratto è utile per tre motivi: gli altri metodi della classe, già implementati, possono invocarlo; i client si aspettano di poterlo chiamare; e costringe le sottoclassi concrete a implementarlo.

Compatibilità fra tipi

L'ereditarietà definisce una classificazione di tipi: una sottoclasse è un **sottotipo** compatibile con la superclasse. Un `Responsabile` è anche un `Dipendente`, perché sa fare tutto ciò che fa un dipendente e qualcosa in più. Per questo un'istanza della sottoclasse può prendere il posto di un'istanza della superclasse; non vale il contrario.

```
Dipendente d = new Responsabile();   // lecito: un Responsabile e' un Dipendente
d.stampa();                          // esegue stampa() di Responsabile
d.aggiungi(x);                       // ERRORE di compilazione: Dipendente non ha aggiungi()
```

Qui entrano in gioco due tipi diversi. Il **tipo statico** (a tempo di compilazione) è quello con cui la variabile è dichiarata, `Dipendente`: determina quali metodi si possono chiamare, e il compilatore rifiuta `aggiungi()` anche se a runtime la variabile punterà a un responsabile. Il **tipo dinamico** (a runtime) è la classe dell'oggetto effettivamente creato, `Responsabile`: è sempre una classe, mai un'interfaccia, e non cambia mai per tutta la vita dell'oggetto.

Con un **cast** si può forzare un riferimento verso il sottotipo: `Responsabile r = (Responsabile) d;`. Il compilatore lo accetta, ma a runtime funziona solo se l'oggetto è davvero un responsabile; altrimenti, per esempio con `(Responsabile) new Dipendente()`, viene lanciata una `ClassCastException`.

Polimorfismo e late binding

Nei sistemi a oggetti possono esistere metodi con lo stesso nome e la stessa signature in classi diverse della gerarchia. Quando si chiama un metodo così, l'effetto dipende dall'oggetto su cui lo si chiama: il comportamento è **polimorfo**.

DEFINIZIONE

Il **polimorfismo** è la capacità di una stessa chiamata di metodo di produrre comportamenti diversi a seconda della classe dell'oggetto. Il **late binding** (collegamento dinamico) è il meccanismo che lo realizza: il compilatore verifica solo che il metodo esista per il tipo statico, mentre la scelta di quale implementazione eseguire avviene a runtime, in base al tipo dinamico.

Senza polimorfismo il chiamante dovrebbe contenere uno `switch` che controlla la classe di ogni oggetto e chiama il metodo giusto; ogni nuova sottoclasse costringerebbe a modificare tutti questi switch. Con il polimorfismo basta aggiungere la sottoclasse.

COME AVVIENE IL DISPATCH

Quando un metodo viene chiamato su un oggetto, Java cerca l'implementazione seguendo sempre lo stesso procedimento.

1. Si guarda il tipo dinamico dell'oggetto.
2. Si cerca il metodo in quella classe; se c'è, lo si esegue.
3. Se non c'è, si risale alla superclasse e si ripete la ricerca, fino a trovarlo.
4. Se il metodo trovato ne chiama un altro sullo stesso oggetto, la ricerca riparte **di nuovo dal tipo dinamico**, non dalla classe in cui ci si trova.

L'ultimo punto è quello che sorprende. Supponiamo che `Conto` abbia i metodi `preleva()` e `verifica()`, e che `ContoDeposito` ridefinisca solo `verifica()`, con una regola più severa. Chiamando `preleva()` su un oggetto `ContoDeposito` dichiarato come `Conto`: preleva non è in ContoDeposito, quindi si esegue quello di Conto; dentro, la chiamata a verifica riparte da ContoDeposito, dove il metodo c'è, e quindi si esegue la verifica severa.

Override e overload

	OVERRIDE (RIDEFINIZIONE)	OVERLOAD (SOVRACCARICO)
Dove	In una sottoclasse.	Nella stessa classe (o in una sottoclasse).
Signature	Stesso nome e stessi parametri del metodo ereditato.	Stesso nome ma parametri diversi per numero o tipo.
Quando si sceglie	A runtime, in base al tipo dinamico (late binding).	A compilazione, in base ai tipi degli argomenti.
Esempio	stampa() in Responsabile ridefinisce stampa() di Dipendente.	somma(int, int) e somma(float, float).

Il design pattern Template Method

Un **design pattern** è una soluzione collaudata a un problema di progettazione ricorrente, descritta in modo da poter essere riusata. Il primo che incontriamo usa proprio ereditarietà, metodi astratti e late binding.

DEFINIZIONE

Template Method definisce in un metodo della superclasse lo **scheletro di un algoritmo**, la sequenza dei passi, e lascia alle sottoclassi il compito di implementare alcuni di quei passi, senza poter cambiare la sequenza.

Pensiamo a una serie di report: tutti si generano stampando intestazione, corpo e chiusura, ma il corpo cambia da report a report. La classe astratta `Report` contiene il metodo modello `genera()`, dichiarato `final` perché nessuno possa alterare l'ordine dei passi, che chiama `intestazione()`, `corpo()` e `chiusura()`. `corpo()` è astratto e ogni sottoclasse lo implementa; gli altri passi hanno un'implementazione predefinita che le sottoclassi possono ridefinire se serve.

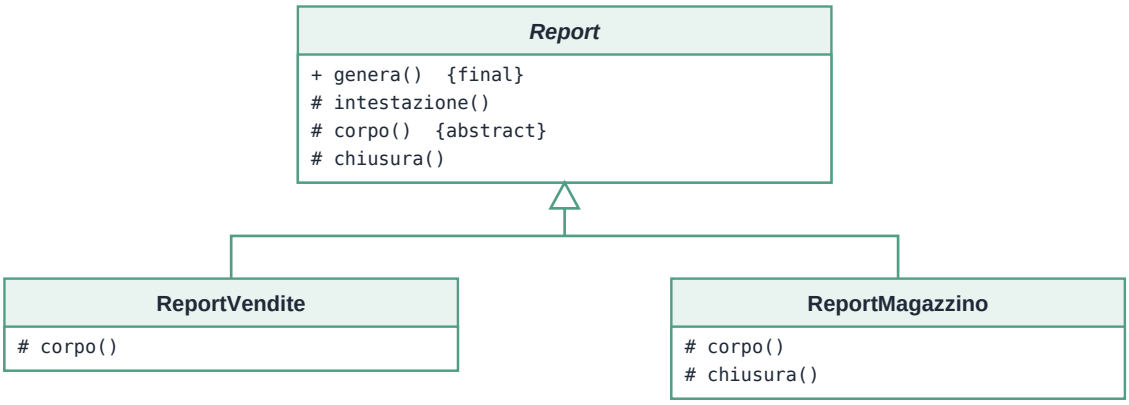


Figura 5.2: Template Method; il nome in corsivo indica una classe astratta, genera() fissa la sequenza dei passi.

Quando si chiama `genera()` su un `ReportVendite`, il metodo eseguito è quello di `Report`, ma le chiamate interne a `corpo()` finiscono, per late binding, nella sottoclasse. È la superclasse a chiamare i metodi della sottoclasse e non viceversa: questo rovesciamento del controllo è detto **principio di Hollywood**, "non chiamarci, ti chiameremo noi".

ASPETTO	TEMPLATE METHOD
Intento	Fissare la struttura di un algoritmo lasciando variare alcuni passi.
Partecipanti	Una classe astratta con il metodo modello e i passi; sottoclassi concrete che implementano i passi variabili.
Vantaggi	Il codice comune sta in un solo posto; la sequenza dei passi non può essere alterata; aggiungere una variante significa aggiungere una sottoclasse.
Svantaggi	Il legame con la superclasse è forte (ereditarietà); se i passi sono molti, il flusso diventa difficile da seguire.

DA RICORDARE

La sottoclasse eredita, aggiunge e ridefinisce, ma non elimina. Interfacce e classi astratte non si istanziano; un client che dipende da un'interfaccia non cambia quando cambia l'implementazione. Il **tipo statico** decide quali metodi si possono chiamare, il **tipo dinamico** decide quale implementazione viene eseguita (**late binding**). Nel dispatch ogni chiamata riparte dal tipo dinamico. **Template Method**: un metodo final della superclasse fissa i passi, le sottoclassi implementano quelli astratti.

CAPITOLO 6

6. Diagrammi UML

Come individuare le classi a partire dai requisiti e come descrivere un sistema con i diagrammi UML: delle classi, di sequenza, di collaborazione, delle attività, degli stati e dei casi d'uso.

UML (Unified Modeling Language) è il linguaggio grafico standard per descrivere la struttura e il comportamento di un sistema software. Il corso lo anticipa qui, rispetto all'ordine delle lezioni, perché i diagrammi delle classi servono per descrivere tutti i design pattern dei capitoli successivi.

Individuare le classi dai requisiti

Il punto di partenza è il testo dei requisiti, su cui si fa un'**analisi grammaticale**: i **sostantivi** suggeriscono classi o attributi, i **verbi** suggeriscono operazioni. Anche gli oggetti fisici menzionati nel testo sono buoni candidati a diventare classi. Attributi e operazioni vanno poi raggruppati nelle classi in modo da ottenere coesione e singola responsabilità, seguendo i principi KISS (Keep It Simple, Stupid) e SRP.

Proviamo con alcuni frammenti di requisiti di un gestionale di vendite: "dovrà essere possibile cercare un cliente e visualizzarne i dati anagrafici"; "per ciascun ordine si mostrano fornitore, cliente, prodotti e importo complessivo"; "dovrà essere calcolato il totale degli ordini di un cliente in un intervallo di date"; "il report mensile riporta per ogni cliente la provincia e il totale per fornitore".

ELEMENTO	CANDIDATI TROVATI NEL TESTO
Classi (sostantivi)	Cliente, Fornitore, Ordine, Prodotto, ReportMensile, SchedaCliente
Attributi (sostantivi)	nome e provincia del cliente, nome del fornitore, importo dell'ordine, data
Metodi (verbi)	cercare un cliente, mostrarne i dati, calcolare il totale degli ordini, selezionare gli ordini in un intervallo

L'analisi grammaticale non basta: il progettista deve capire da solo che servono anche classi che il testo non nomina, come una `Rubrica` che contiene i clienti e offre il metodo `cerca()`, o un `ElencoOrdini` con metodi brevi e separati come `selezionaPerCliente()`, `selezionaPerData()` e `getTotale()`. Tenerli separati permette di combinarli per ottenere selezioni diverse da quelle richieste oggi, il che tornerà utile quando i requisiti cambieranno.

Il diagramma delle classi

Il diagramma delle classi mostra la struttura statica del sistema. Ogni classe è un rettangolo diviso in tre parti: il nome, gli attributi e i metodi, ciascuno preceduto dal simbolo di visibilità (− privato, # protetto, + pubblico) e seguito dal tipo dopo i due punti. Un nome in corsivo indica una classe astratta; lo stereotipo «interface» indica un'interfaccia. Le classi sono collegate da relazioni.

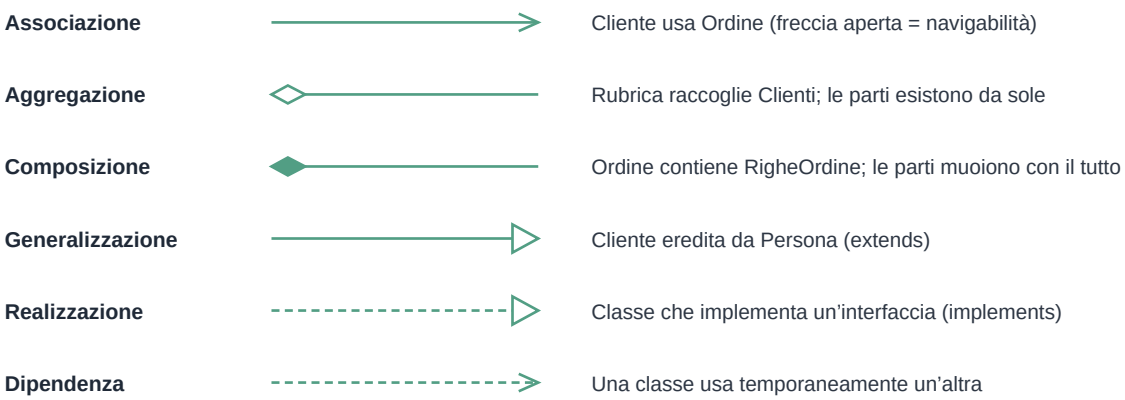


Figura 6.1: le relazioni del diagramma delle classi e la loro notazione.

Sulle associazioni si indica la **molteplicità**, cioè quante istanze partecipano: 1 (esattamente una), 0..1 (al più una), * (un numero qualsiasi), 1..* (almeno una). La differenza fra aggregazione e composizione è nella vita delle parti: in una composizione, il rombo pieno, le parti appartengono a un solo tutto e vengono distrutte con esso, come le righe di un ordine; in un'aggregazione, il rombo vuoto, le parti esistono anche da sole, come i clienti di una rubrica.

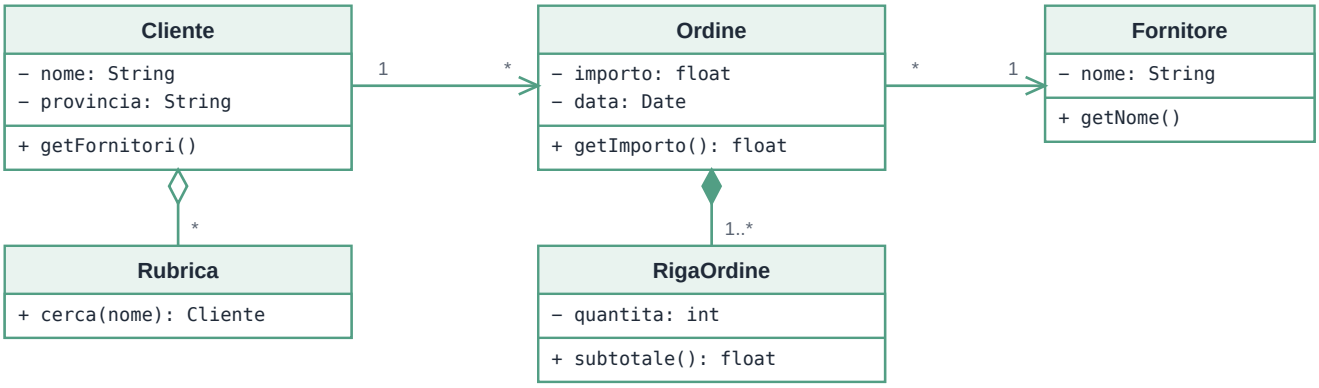


Figura 6.2: un diagramma delle classi per il gestionale degli ordini.

In Java un'associazione diventa un attributo: un `Ordine` che conosce il suo cliente ha un campo `private Cliente cliente;`, e una molteplicità "molti" diventa una lista o un array.

Il diagramma di sequenza

Il diagramma di sequenza mostra le **interazioni fra oggetti** nel tempo. In alto, in orizzontale, ci sono gli oggetti, indicati come `nome:Classe`; il tempo scorre in verticale verso il basso. Sotto ogni oggetto una linea tratteggiata, la **linea della vita**, indica che l'oggetto esiste; un rettangolo sottile, la **barra di attivazione**, indica che sta eseguendo un metodo. Una chiamata di metodo è una freccia piena dalla barra del chiamante a quella del chiamato; la creazione di un oggetto è una freccia etichettata `create` che punta al riquadro del nuovo oggetto, disegnato all'altezza in cui nasce. Un ritorno si può indicare con una freccia tratteggiata, e un riquadro etichettato `loop` racchiude le chiamate ripetute.

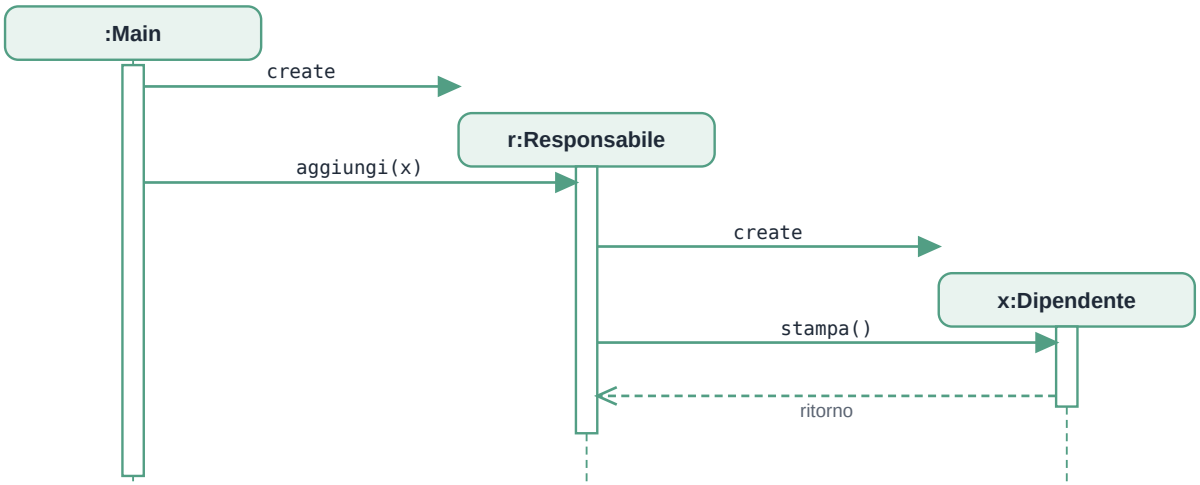


Figura 6.3: il main crea un responsabile, gli aggiunge un dipendente e questo, chiamato, stampa i suoi dati.

Il diagramma di collaborazione

Il diagramma di collaborazione mostra le stesse interazioni di un diagramma di sequenza, ma dispone gli oggetti liberamente nel piano, collegati dalle loro associazioni. I messaggi sono frecce accanto ai collegamenti, con un'etichetta che contiene un **numero progressivo**, che indica l'ordine, il **metodo** chiamato e, se serve, il **valore di ritorno**, per esempio `2: t := getTotale()`. Il sequenza mette in evidenza l'ordine temporale, il collaborazione chi è collegato a chi.

Il diagramma delle attività

Il diagramma delle attività è il punto iniziale della progettazione: serve a riflettere sulle attività che il sistema dovrà svolgere, senza dire ancora quali oggetti le svolgeranno, e può essere la base per ricavare poi i diagrammi di collaborazione. Gli elementi sono pochi.

SIMBOLO	SIGNIFICATO
Cerchio pieno	Stato iniziale.
Rettangolo arrotondato	Passo di elaborazione (attività).
Freccia continua	Flusso da un'attività alla successiva.
Rombo	Ramificazione condizionale (con le condizioni fra parentesi quadre) oppure merge, dove più rami si riuniscono.
Barra piena	Fork, una freccia entrante e più uscenti che procedono in parallelo; oppure join, dove i flussi paralleli si sincronizzano.
Cerchio pieno cerchiato	Stato finale.
Corsie (swimlane)	Partizioni verticali che indicano il responsabile (attore o modulo) delle attività; frecce tratteggiate collegano dati in ingresso o uscita.

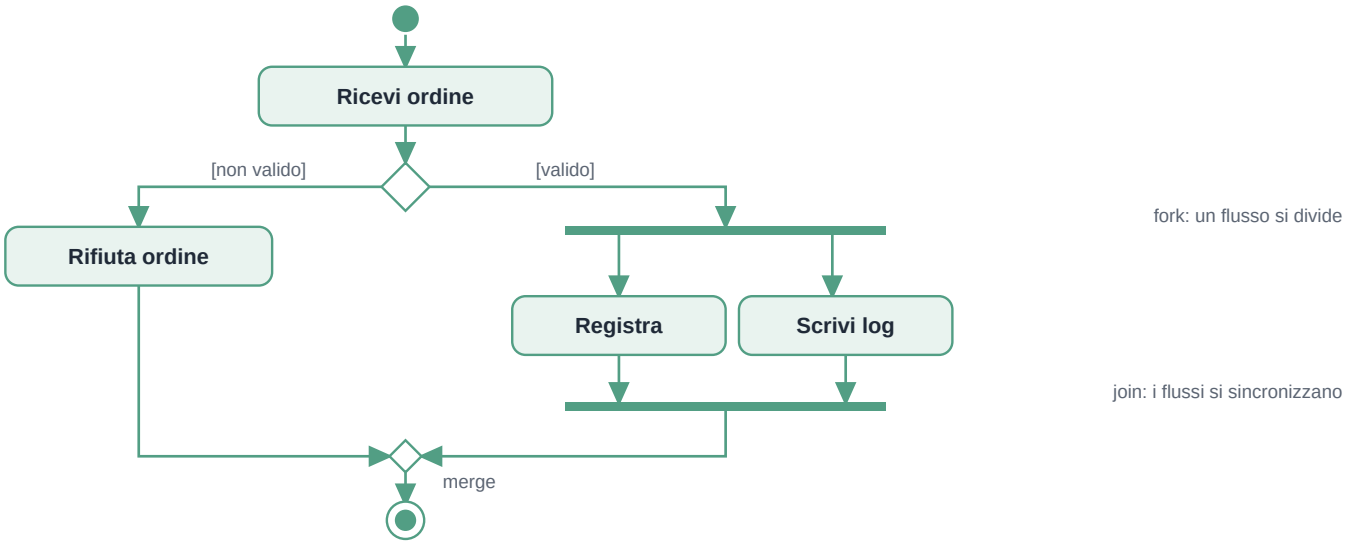


Figura 6.4: un ordine non valido viene rifiutato; uno valido viene registrato e annotato nel log in parallelo.

Il diagramma degli stati

Il diagramma degli stati descrive come cambia nel tempo **un singolo oggetto**. Uno **stato** è un intervallo della vita dell'oggetto, caratterizzato da certi valori dei suoi attributi, durante il quale l'oggetto attende certi eventi o svolge certe azioni. Una **transizione** fa lasciare uno stato in risposta a un evento ed è etichettata con la forma `evento [condizione] / azione`: l'evento la scatena, la condizione (facoltativa) deve essere vera, l'azione viene eseguita durante il passaggio allo stato di arrivo.

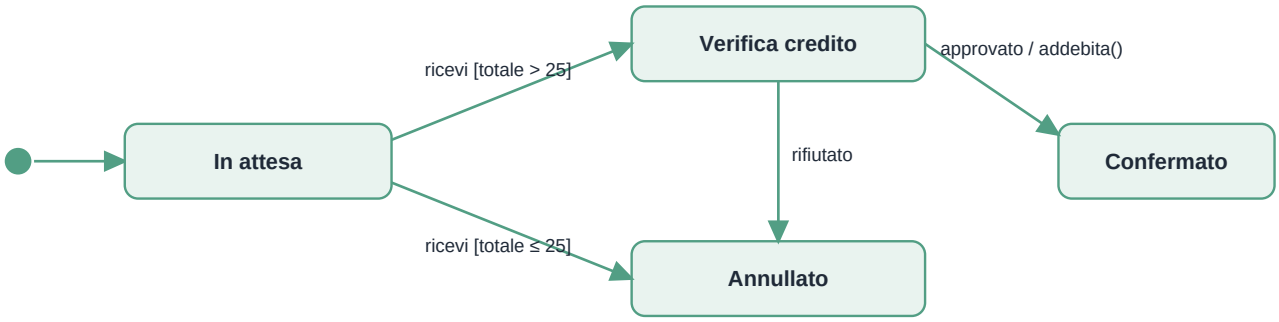


Figura 6.5: gli stati di un ordine; la stessa ricezione porta in stati diversi a seconda della condizione sul totale.

Uno stato **semplice** non ha sottostati; uno stato **composto** ne contiene altri. Se i sottostati sono **sequenziali**, uno solo alla volta può essere attivo; se sono **concorrenti**, più regioni sono attive insieme. Trovarsi nello stato esterno significa trovarsi in uno qualsiasi dei suoi sottostati, e le transizioni che entrano o escono da uno stato composto possono eseguire azioni di `entry` e di `exit`. Questo diagramma sarà il punto di partenza del design pattern State.

Il diagramma dei casi d'uso

Il diagramma dei casi d'uso descrive il sistema dal punto di vista di chi lo usa ed è tipico della fase di raccolta dei requisiti. Gli **attori**, disegnati come omini, sono persone o altri sistemi esterni; i **casi d'uso**, ovali dentro il rettangolo che delimita il sistema, sono i servizi che il sistema offre loro; una linea collega ogni attore ai casi d'uso a cui partecipa. Non descrive come i servizi sono realizzati, ma solo quali sono e per chi.

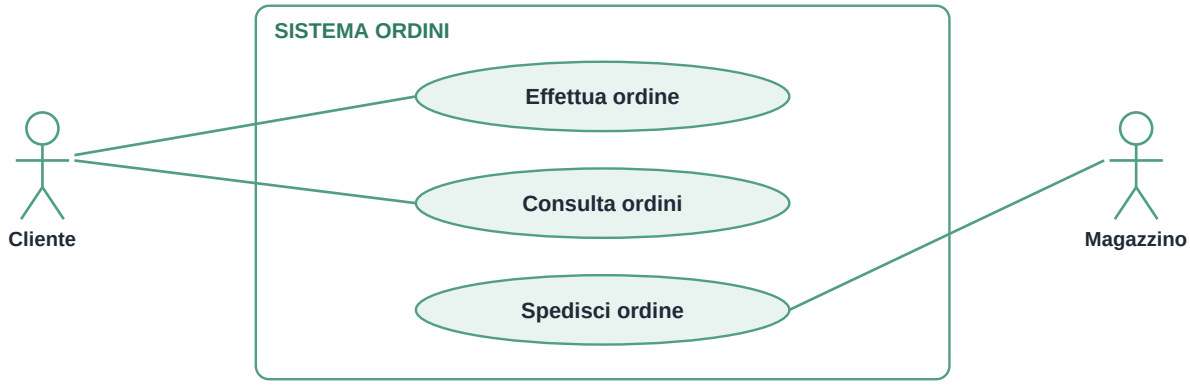


Figura 6.6: i casi d'uso di un sistema di ordini con due attori.

DA RICORDARE

Le classi si individuano con l'**analisi grammaticale** dei requisiti, ma il progettista deve aggiungere quelle non nominate. Diagramma delle **classi**: struttura statica, visibilità, relazioni e molteplicità. **Sequenza**: interazioni nel tempo, con linee della vita e barre di attivazione. **Collaborazione**: le stesse interazioni con messaggi numerati. **Attività**: il flusso del lavoro, con decisioni, fork e join. **Stati**: la vita di un oggetto, transizioni `evento [condizione] / azione`. **Casi d'uso**: attori e servizi offerti.

CAPITOLO 7

7. Design pattern e Singleton

Che cosa sono i design pattern, come si descrivono e come sono classificati; poi il primo pattern creazionale, Singleton, che garantisce che una classe abbia una sola istanza.

Che cosa sono i design pattern

DEFINIZIONE

Un **design pattern** descrive un problema di progettazione ricorrente, che si incontra in un contesto specifico, e presenta una soluzione collaudata, generica ma specializzabile. La soluzione è una piccola struttura software, una **microarchitettura** di poche classi, di cui si specificano i ruoli e le interazioni.

I pattern documentano soluzioni già applicate con successo e affinate nel tempo. Si riusa così non una singola classe, ma un'intera soluzione, spesso fatta di più classi. I vantaggi sono molti: aiutano i principianti ad agire come esperti, sostengono gli esperti nella progettazione su larga scala, evitano di reinventare soluzioni già note riducendo i costi, forniscono un **vocabolario comune** (dire "qui uso un Observer" basta a farsi capire) e analizzano le proprietà **non funzionali** della soluzione, cioè come una funzionalità è realizzata: affidabilità, modificabilità, sicurezza, testabilità, riuso. Durante la progettazione, infatti, le conseguenze delle scelte sulle classi possono non essere evidenti, e le classi rischiano di diventare difficili da riusare; un pattern dichiara quelle conseguenze in anticipo.

Come si descrive un pattern

La descrizione di un pattern ha cinque parti fondamentali.

PARTE	CONTENUTO
Nome	Identifica il pattern con una parola, permette di ragionare a un alto livello di astrazione e ne indica lo scopo.
Intento	Descrive brevemente funzionalità e scopo.
Problema	Motivazione e applicabilità: il problema affrontato e le condizioni per applicare il pattern.
Soluzione	Le classi che costituiscono il pattern, le loro responsabilità e le loro relazioni.
Conseguenze	Risultati, compromessi, vantaggi e svantaggi del suo uso.

Nella sezione Problema si parla di **forze**, come in fisica: obiettivi e vincoli, spesso in contrasto fra loro, presenti nel contesto del pattern. La soluzione è un equilibrio fra queste forze. A volte la descrizione aggiunge esempi di utilizzo e frammenti di codice.

La classificazione

Il catalogo più famoso è quello della "banda dei quattro" (GoF, Gang of Four: Gamma, Helm, Johnson, Vlissides), che organizza i pattern secondo il loro scopo. Esistono comunque cataloghi per molti altri contesti: sistemi concorrenti, distribuiti, in tempo reale.

CATEGORIA	RIGUARDANO	PATTERN
Creazionali	La creazione delle istanze: rendono il sistema indipendente da come gli oggetti sono creati, composti e rappresentati, nascondendo quali classi concrete vengono usate.	Singleton, Factory Method, Abstract Factory, Builder, Prototype
Strutturali	Il modo in cui classi e oggetti sono composti per formare strutture più grandi.	Adapter, Facade, Composite, Decorator, Bridge, Flyweight, Proxy
Comportamentali	Gli algoritmi e la distribuzione delle responsabilità: si passa da un insieme fisso di comportamenti a pochi comportamenti fondamentali che si compongono.	Template Method, Iterator, Mediator, Observer, State, Strategy, Chain of Responsibility, Command, Interpreter, Memento, Visitor

I pattern creazionali diventano importanti quando un sistema evolve verso la **composizione di oggetti** più che verso l'ereditarietà fra classi: invece di fissare i comportamenti nel codice, si combinano oggetti, e allora conta molto come quegli oggetti vengono creati.

Classi, oggetti e numero di istanze

Ogni oggetto è istanza di una classe, e ogni classe ha zero o più istanze. Le classi sono **statiche**: esistenza, significato e relazioni sono fissati prima dell'esecuzione. Gli oggetti invece sono creati e distrutti durante l'esecuzione, e la classe di un oggetto, una volta creato, non cambia. Le classi formano il vocabolario del dominio del problema; gli oggetti, interagendo, soddisfano i requisiti.

Una domanda di progettazione che si pone presto è: quante istanze servono di una certa classe? Per la classe `Consumi` dei capitoli precedenti potremmo volere un'istanza per ogni file letto, oppure un'unica istanza che accumula i valori di tutti i file, visto che `leggiFile()` aggiunge alla lista senza svuotarla. Se decidiamo che l'istanza deve essere una sola, come possiamo **vietare** che qualcuno ne crei altre con `new`? Questo è il problema risolto da Singleton.

Il design pattern Singleton

DEFINIZIONE

Intento: assicurare che una classe abbia **una sola istanza** e fornire un punto di accesso globale a essa.

MOTIVAZIONE

Alcune classi devono avere esattamente un'istanza in tutta l'applicazione: lo spooler di stampa, il file system, il gestore delle finestre, un registro dei messaggi, l'elenco dei clienti. Una variabile globale renderebbe l'oggetto accessibile ovunque, ma non impedirebbe di crearne altri. La soluzione è rendere la classe stessa responsabile di tenere traccia della sua unica istanza e del punto di accesso.

SOLUZIONE

1. Il **costruttore è privato**, così nessun'altra classe può usare `new`.
2. Un **attributo statico** privato tiene l'unica istanza.
3. Un **metodo statico** `getInstance()`, che appartiene alla classe e non a un oggetto, restituisce quell'istanza. È la classe stessa a crearla.

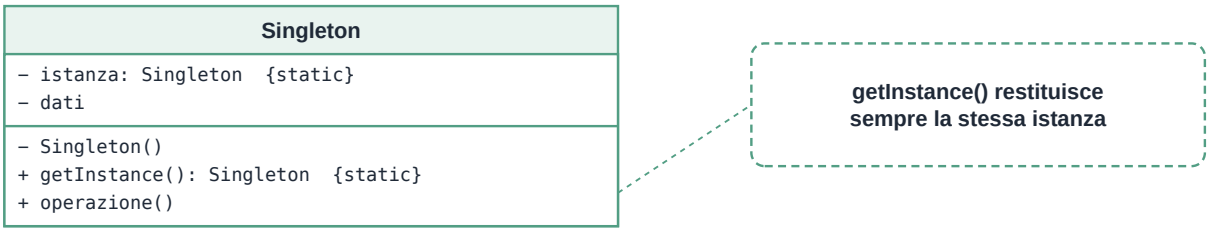


Figura 7.1: la struttura di Singleton; i membri statici sono indicati con {static}.

L'istanza si può creare in due momenti. Con l'**inizializzazione anticipata** (eager) l'attributo statico viene inizializzato direttamente nella dichiarazione, `private static Registro obj = new Registro();`, e `getInstance()` si limita a restituirlo. Con l'**inizializzazione pigra** (lazy) l'istanza nasce solo alla prima richiesta: `getInstance()` controlla se l'attributo è `null` e, in quel caso, crea l'oggetto. La seconda evita di creare un oggetto che magari non servirà mai, ma non è sicura con più thread: due chiamate simultanee potrebbero trovare entrambe null e creare due istanze; in quel caso si usa la versione anticipata o un metodo `synchronized`.

```
public class Registro {                    // ruolo Singleton
    private static Registro obj;           // l'unica istanza
    private List<String> messaggi = new ArrayList<>();

    private Registro() { }                 // costruttore privato: niente new dall'esterno

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

    public void scrivi(String m) { messaggi.add(m); }
    public String ultimo() { return messaggi.get(messaggi.size() - 1); }
}
```

Due variabili ottenute con `Registro.getInstance()` puntano allo stesso oggetto: ciò che scrive una lo legge l'altra; scrivere `new Registro()` altrove produce un errore di compilazione.

CONSEGUENZE

La classe ha pieno controllo su come e quando i client accedono alla sua unica istanza; si evitano variabili globali; il numero di istanze è controllato in un solo punto. È anche più flessibile che rendere static tutti i metodi e gli attributi: se in futuro servissero più istanze, basterebbe modificare la sola classe Singleton, mentre con tutto static si dovrebbero cambiare tutte le invocazioni nei client.

Quest'ultimo punto esprime il **principio delle conseguenze locali**: un cambiamento in un punto del codice non dovrebbe causare problemi in altri punti. La variante che ammette un numero finito di istanze si chiama **Multiton**: per esempio una classe che tiene un'istanza per ogni cartella, e che in `getInstance(cartella)` crea l'istanza solo se per quella cartella non esiste ancora, conservando le istanze in una mappa.

DA RICORDARE

Un design pattern ha **nome, intento, problema, soluzione, conseguenze**. Le categorie GoF sono **creazionali, strutturali, comportamentali**. **Singleton**: costruttore privato, attributo statico con l'unica istanza, metodo statico `getInstance()`; creazione anticipata o pigra. Controlla il numero di istanze in un solo punto (principio delle conseguenze locali); la variante **Multiton** ne consente un numero finito.

CAPITOLO 8

8. Factory Method e Abstract Factory

Come delegare la creazione degli oggetti a un metodo dedicato, per rendere il codice indipendente dalle classi concrete; le varianti, l'Object Pool, la Dependency Injection e la creazione di intere famiglie di oggetti con Abstract Factory.

Il problema della creazione

Scrivere `new ClasseConcreta()` lega il codice chiamante a quella classe: se domani la classe cambia, va modificato ogni punto in cui compare il `new`. Il problema diventa evidente in un **framework**, che usa classi astratte per definire le relazioni fra oggetti: il framework deve creare oggetti, ma conosce solo classi astratte, che non può istanziare. Serve un metodo responsabile dell'istanziazione, detto **factory** (fabbricatore), che incapsuli la conoscenza su quale classe creare.

Il design pattern Factory Method

DEFINIZIONE

Intento: definire un'interfaccia per creare un oggetto, lasciando decidere alle sottoclassi quale classe istanziare. Factory Method permette a una classe di rimandare l'istanziazione alle sottoclassi.

SOLUZIONE: I RUOLI

RUOLO	RESPONSABILITÀ
Product	Interfaccia comune degli oggetti creati dal metodo factory.
ConcreteProduct	Implementazione di Product; ce ne possono essere molte.
Creator	Dichiara `factoryMethod()`, che restituisce un Product; può averne un'implementazione predefinita e altri metodi che usano il prodotto creato.
ConcreteCreator	Implementa o ridefinisce `factoryMethod()`, sceglie quale ConcreteProduct istanziare e lo restituisce.

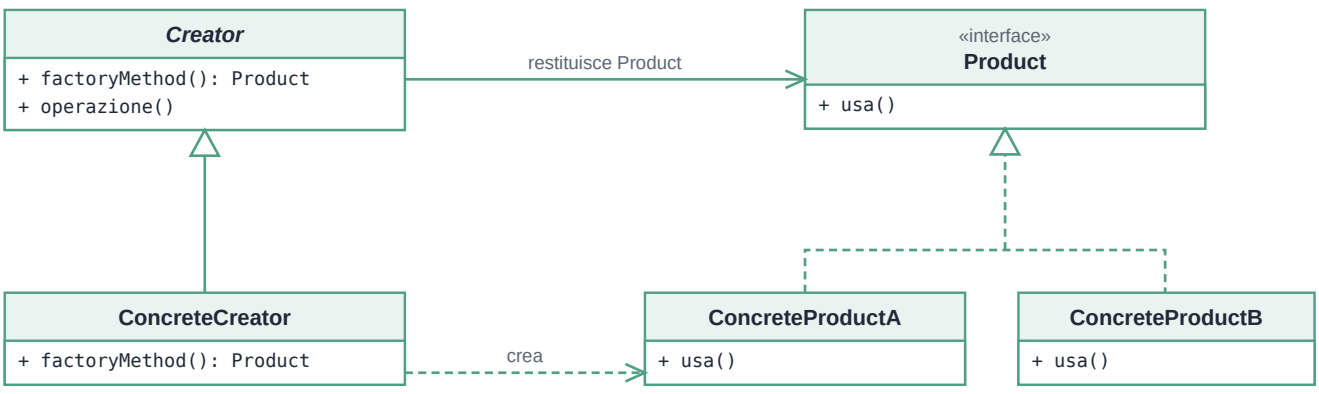


Figura 8.1: la struttura di Factory Method; il client riceve un Product senza sapere quale classe concreta sia stata istanziata.

L'interazione, come mostrerebbe un diagramma di sequenza, è questa: il client chiama un'operazione sul creator concreto; questa invoca `factoryMethod()`, che crea un ConcreteProduct e lo restituisce; il client usa poi il prodotto attraverso l'interfaccia Product. Un esempio: un sistema di biglietteria ha l'interfaccia `Biglietto` con le classi `BigliettoIntero` e `BigliettoRidotto`; un metodo `creaBiglietto(eta)` decide quale istanziare, e la cassa lavora solo con `Biglietto`.

VARIANTI

- I ruoli Creator e ConcreteCreator sono svolti dalla **stessa classe**.
- Il metodo factory è **static**, quindi si chiama sulla classe senza crearne un'istanza.
- Il metodo factory ha un **parametro** con cui il client suggerisce quale classe usare.
- Il metodo factory usa la **riflessione computazionale**: con `Class.forName(nome)` ottiene la classe a partire dal suo nome in una stringa e ne crea un'istanza con `getConstructor().newInstance()`. Così sparisce ogni dipendenza dai ConcreteProduct nel codice: la classe istanziata è nota solo a runtime, per esempio letta da un file di configurazione.

CONSEGUENZE

- Il codice dell'applicazione conosce solo l'interfaccia Product e funziona con qualsiasi ConcreteProduct, che diventano facilmente intercambiabili.
- Se si crea una sottoclasse di Creator per ogni ConcreteProduct, si ha una **proliferazione di classi**.

Object Pool

Un **object pool** è un deposito di istanze già create. Quando un client chiede un oggetto, il metodo factory ne preleva uno dal pool, e ne crea uno nuovo solo se il pool è vuoto; quando il client non ne ha più bisogno, lo restituisce al pool perché possa essere riusato. È utile quando creare un oggetto costa molto, come una connessione a un database. Il pool può crescere oppure avere dimensione fissa: in quel caso, se non ci sono oggetti disponibili, non se ne creano di nuovi.

Ci sono tre attenzioni: i client devono dichiarare esplicitamente quando un'istanza non serve più; lo stato di un'istanza riusata può dover essere reimpostato, perché conserva i valori del suo uso precedente; il pool dovrebbe essere unico, e quindi può essere a sua volta un Singleton.

Dependency Injection

DEFINIZIONE

Con la **Dependency Injection** un oggetto (il client) **riceve** dall'esterno gli altri oggetti di cui ha bisogno, detti **dipendenze**, invece di crearli da sé. La costruzione delle istanze è così separata dal loro uso.

Le dipendenze vengono passate come parametri del **costruttore** del client. Rispetto a impostarle con metodi set, questo evita oggetti incompleti e controlli continui sul fatto che le dipendenze non siano null, e il codice risulta più semplice. Il client conosce solo il tipo astratto della dipendenza, non i suoi sottotipi. Per esempio un `Ordine` riceve nel costruttore un `Notificatore`, che a runtime può essere un notificatore via email o via SMS. Un metodo factory può occuparsi dell'iniezione: è lui a collegare (fare il **wiring** di) le varie istanze, e in un unico posto si vedono tutte le connessioni fra gli oggetti.

Il design pattern Abstract Factory

DEFINIZIONE

Intento: fornire un'interfaccia per creare **famiglie di oggetti** correlati senza specificarne le classi concrete.

Il problema: il sistema dovrebbe essere indipendente dalle classi usate e configurabile con una fra più famiglie di classi, e le classi di una famiglia vanno usate in modo coerente. L'esempio classico è un'interfaccia grafica con più temi (look and feel): bottoni e finestre del tema chiaro non vanno mescolati con quelli del tema scuro.

RUOLO	RESPONSABILITÀ
AbstractFactory	Interfaccia con un metodo di creazione per ogni tipo di prodotto della famiglia.
ConcreteFactory	Implementa i metodi di creazione per una specifica famiglia.
AbstractProduct	Interfaccia per un tipo di prodotto (per esempio Bottone).
Product	Oggetto concreto creato da una ConcreteFactory, che implementa un AbstractProduct.
Client	Usa solo le interfacce AbstractFactory e AbstractProduct.

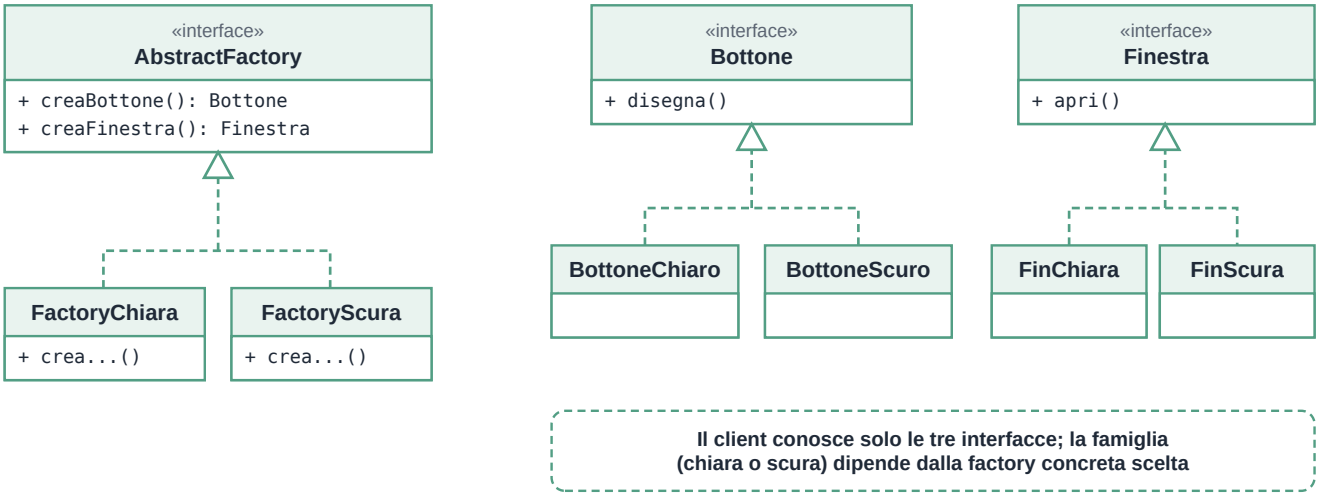


Figura 8.2: Abstract Factory; ogni factory concreta crea i prodotti di una sola famiglia.

CONSEGUENZE

- Le classi vengono usate in modo coerente, per famiglie.
- Le famiglie sono facilmente intercambiabili: basta cambiare la factory concreta, in un solo punto.
- Aggiungere un nuovo **tipo** di prodotto non è immediato: bisogna aggiungere un metodo all'AbstractFactory e a tutte le ConcreteFactory.

La differenza con Factory Method: Factory Method crea **un** prodotto e lascia la scelta a un metodo (spesso ridefinito in sottoclassi); Abstract Factory crea **famiglie** di prodotti tramite un oggetto factory, e di solito ogni suo metodo è a sua volta un factory method.

DA RICORDARE

Factory Method: un metodo crea il prodotto, il client conosce solo l'interfaccia Product; ruoli Product, ConcreteProduct, Creator, ConcreteCreator; varianti static, con parametro, con riflessione. **Object Pool**: il metodo factory riusa istanze già create. **Dependency Injection**: le dipendenze arrivano dal costruttore, il wiring sta in un solo posto. **Abstract Factory**: famiglie coerenti e intercambiabili, ma aggiungere un tipo di prodotto costa.

CAPITOLO 9

9. Prototype

Creare nuovi oggetti copiando un'istanza esistente, il prototipo, invece di istanziare una classe: struttura, conseguenze e la differenza fra copia superficiale e copia profonda.

Il design pattern Prototype

DEFINIZIONE

Intento: specificare i tipi di oggetti da creare usando un'istanza prototipo, e creare i nuovi oggetti **copiando** questo prototipo.

MOTIVAZIONE

A volte un nuovo oggetto deve partire da uno stato simile a quello di un oggetto esistente, uno stato che quest'ultimo ha raggiunto attraverso una serie di operazioni. Ricostruirlo da zero significherebbe ripetere tutte quelle operazioni. C'è poi un secondo problema: per creare un oggetto con `new` bisogna nominare una classe precisa, e così chi crea è strettamente accoppiato alla classe creata. Questo rende difficile aggiungere nuovi tipi; con Factory Method, per esempio, servirebbe una nuova sottoclasse di Creator per ogni tipo. Copiare un oggetto esistente risolve entrambi i problemi, come una fotocopia che riparte da un documento già compilato.

PARTECIPANTI

RUOLO	RESPONSABILITÀ
Prototype	Interfaccia delle operazioni comuni, che dichiara anche il metodo per clonare se stesso.
ConcretePrototype	Classe che implementa le operazioni, compresa la clonazione.
Client	Crea nuovi oggetti chiedendo al prototipo di clonarsi; conosce solo l'interfaccia Prototype.

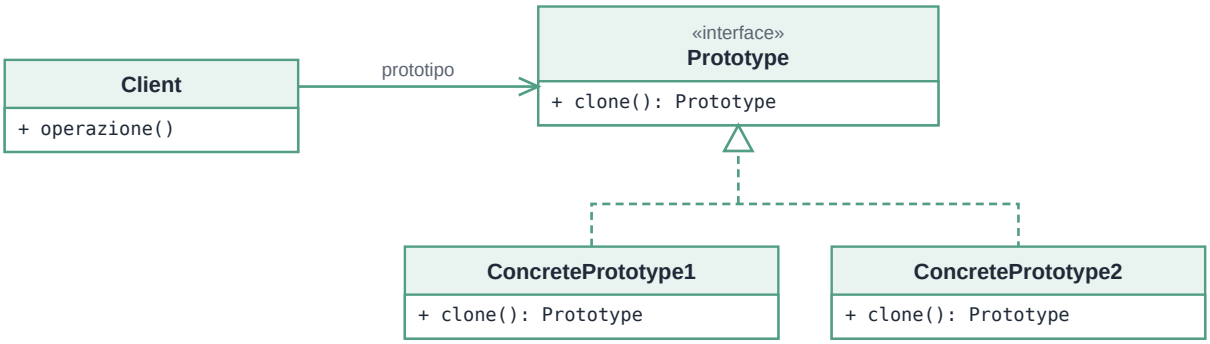


Figura 9.1: la struttura di Prototype.

Un esempio: un'`Agenda` di impegni è il Prototype, con le operazioni `aggiungi()`, `stampa()` e `clone()`. Le classi `Calendario` e `Taccuino` sono ConcretePrototype. Una nuova agenda ottenuta per clonazione contiene già tutti gli impegni inseriti nella precedente, e da quel punto le due possono evolvere separatamente.

CONSEGUENZE

- Il client usa l'interfaccia comune e non conosce la classe concreta: ha meno dipendenze e può usare sottoclassi di cui non conosce nemmeno il nome.
- Si possono registrare o rimuovere prototipi **a runtime**, anche con un **gestore dei prototipi**, cioè un registro da cui i client recuperano il prototipo prima di clonarlo.
- Si ottengono oggetti con valori diversi senza creare nuove classi, e si può creare una configurazione dinamicamente; un prototipo può anche contenere una struttura di oggetti, come in un Composite.
- Non serve una gerarchia di Creator come in Factory Method, perché ogni nuovo oggetto si ottiene per clonazione.
- Implementare la clonazione può essere la parte più difficile, soprattutto se l'oggetto contiene riferimenti circolari.

Copia superficiale e copia profonda

Il linguaggio può offrire un'operazione di clonazione predefinita, ma spesso questa esegue una **copia superficiale** (shallow copy): copia i valori degli attributi, e quando un attributo è un riferimento copia il riferimento, non l'oggetto puntato. Originale e clone finiscono così per **condividere** gli stessi oggetti interni: aggiungendo un impegno alla lista del clone, lo si aggiungerebbe anche all'originale. Nella **copia profonda** (deep copy) si duplicano anche gli oggetti interni, e le due istanze diventano davvero indipendenti.

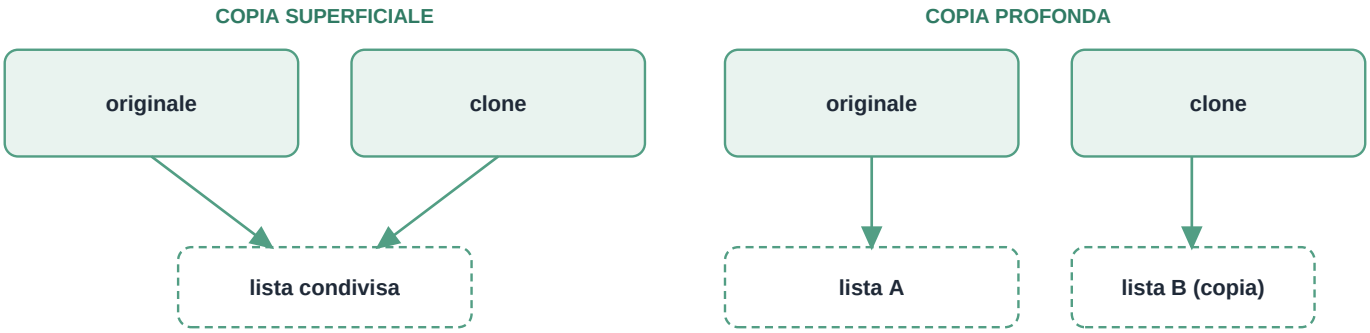


Figura 9.2: nella copia superficiale originale e clone condividono la lista; nella copia profonda ciascuno ha la sua.

Per l'agenda serve una copia profonda: il metodo `clone()` crea una nuova lista, vi copia tutti gli elementi della lista originale e la assegna al nuovo oggetto. Infine, i cloni possono aver bisogno di essere inizializzati, e per questo la classe prototipo può offrire operazioni per impostarne i dati.

DA RICORDARE

Prototype crea oggetti clonando un prototipo: il client conosce solo l'interfaccia con `clone()`, non serve una gerarchia di Creator, i prototipi si registrano a runtime. Attenzione alla **copia superficiale**, che condivide i riferimenti: di solito serve una **copia profonda**.

CAPITOLO 10

10. Builder

Come costruire passo per passo un oggetto complesso e immutabile, evitando costruttori con troppi parametri e metodi set che lasciano l'oggetto in uno stato incoerente.

Il problema dei molti parametri

Consideriamo una classe `Prenotazione` con molti attributi che, una volta fissati, non devono cambiare: sono `final`. Alcuni sono obbligatori, come il nome del cliente, altri facoltativi, come il numero di ospiti o la richiesta di colazione. Ci sono due soluzioni immediate, entrambe insoddisfacenti.

La prima è il **costruttore telescopico**: un costruttore con un parametro, uno con due, uno con tre, e così via, dove ognuno richiama il successivo con `this(...)` passando valori predefiniti per i parametri mancanti. Funziona, ma scrivere i client è difficile: chi chiama deve ricordare l'ordine esatto di molti parametri, spesso dello stesso tipo, e scambiarne due non produce alcun errore di compilazione.

La seconda sono i **metodi set separati**: un costruttore vuoto e un metodo set per ogni attributo. Ha due svantaggi. Gli attributi non possono più essere final, quindi l'oggetto non è immutabile. E non si sa quando, impostando i valori uno alla volta, l'oggetto ha raggiunto uno stato coerente; il controllo di coerenza non si può fare nel costruttore.

Il design pattern Builder

DEFINIZIONE

Intento: separare la costruzione di un oggetto complesso dalla sua rappresentazione, così che lo stesso processo costruttivo possa creare rappresentazioni diverse.

L'oggetto desiderato si chiede **indirettamente**, attraverso un costruttore dedicato, il builder. I passi sono tre.

1. Il client istanzia il builder, passando al suo costruttore i parametri obbligatori.
2. Il client chiama sul builder un metodo per ogni attributo facoltativo che vuole impostare. Ognuno restituisce il builder stesso (`return this`), e così le chiamate si possono concatenare.
3. Infine il client chiama il metodo `build()`, che verifica che lo stato sia coerente e solo allora crea l'oggetto, immutabile.

Le chiamate concatenate si leggono quasi come una frase: `new PrenotazioneBuilder("Bianchi").ospiti(2).colazione(true).build()`. Ogni parametro è accompagnato dal suo nome, e l'ordine non conta più.

PARTECIPANTI

RUOLO	RESPONSABILITÀ
Builder	Interfaccia per creare le parti di un Product.
ConcreteBuilder	Costruisce e assembla le parti implementando Builder, tiene traccia della rappresentazione che sta creando e offre il metodo per ottenere il Product.
Director	Costruisce un Product usando l'interfaccia Builder, decidendo la sequenza dei passi.
Product	L'oggetto complesso da costruire, con le sue parti.

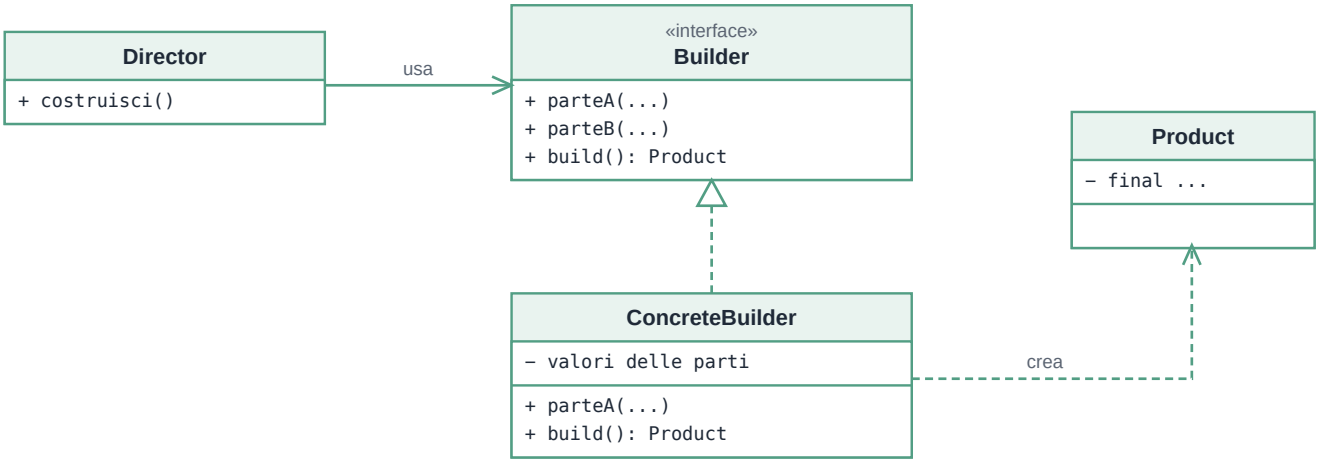


Figura 10.1: la struttura di Builder.

APPLICABILITÀ E CONSEGUENZE

Builder si usa quando l'algoritmo che crea un oggetto complesso deve essere indipendente dalle parti che lo compongono e da come sono assemblate, e quando lo stesso processo deve poter produrre rappresentazioni diverse.

- La rappresentazione interna del Product può cambiare: il builder la nasconde, e per cambiarla basta definire un nuovo builder.
- Il codice di costruzione è isolato da quello di rappresentazione: i client non sanno nulla delle classi che formano la struttura interna del prodotto, e ogni ConcreteBuilder contiene tutto il codice per creare un tipo di prodotto.
- C'è un controllo fine sul processo di costruzione: il prodotto viene costruito passo per passo e viene preso dal builder solo quando è completo.

DA RICORDARE

Contro il **costruttore telescopico** (ordine dei parametri difficile) e i **metodi set** (niente immutabilità, stato incoerente), **Builder** raccoglie i valori con metodi concatenati che restituiscono `this` e crea l'oggetto immutabile solo in `build()`, dopo averne verificato la coerenza. Ruoli: Builder, ConcreteBuilder, Director, Product.

CAPITOLO 11

11. Adapter

Come far collaborare classi con interfacce incompatibili inserendo un adattatore fra il chiamante e la classe di libreria, nelle due versioni Object Adapter e Class Adapter.

Il design pattern Adapter

DEFINIZIONE

Intento: convertire l'interfaccia di una classe in un'altra interfaccia, quella che i client si aspettano. Adapter permette a classi con interfacce incompatibili di lavorare insieme.

PROBLEMA

A volte una classe di libreria non si può usare perché la sua interfaccia non corrisponde a quella che l'applicazione si aspetta: nome del metodo, numero o tipo dei parametri non coincidono con le chiamate presenti nell'applicazione. Non si può modificare la libreria, perché non se ne ha il sorgente o comunque non conviene; e non si vuole modificare l'applicazione, anche perché si potrebbe voler cambiare in futuro il metodo effettivamente invocato senza che il chiamante se ne accorga.

Soluzione Object Adapter

RUOLO	RESPONSABILITÀ
Target	L'interfaccia che il client si aspetta.
Adaptee	La classe di libreria, con l'interfaccia incompatibile.
Adapter	Implementa Target e tiene un riferimento all'Adaptee; converte ogni chiamata del client nella chiamata corrispondente sull'Adaptee.
Client	Usa l'Adapter attraverso Target, come se fosse l'oggetto di libreria.

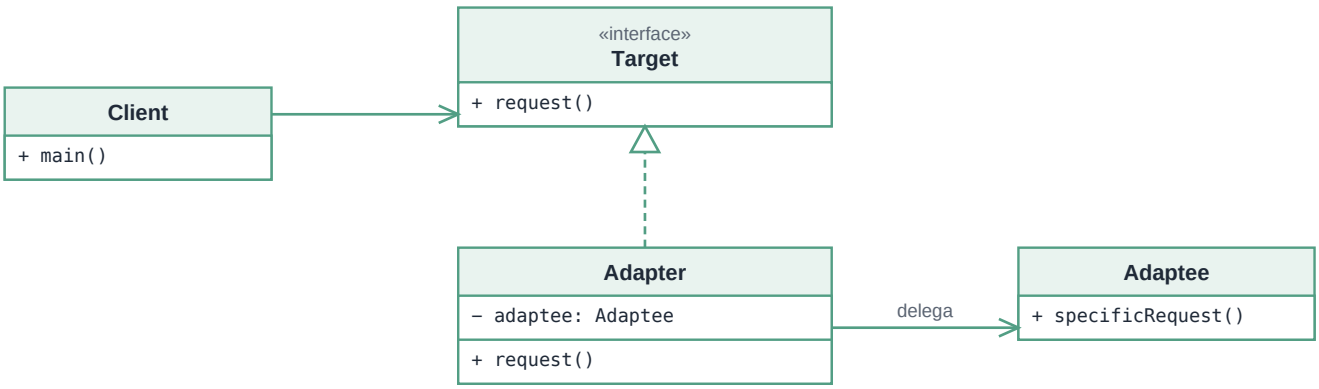


Figura 11.1: Object Adapter; l'adapter contiene l'adaptee e gli delega il lavoro.

Un esempio: l'applicazione chiede le etichette di un magazzino con l'interfaccia `Etichettatrice` e il metodo `prossimaEtichetta()`, mentre la libreria offre la classe `GeneratoreCodici` con il metodo `emettiCodice()`. L'adapter implementa `Etichettatrice`, tiene un riferimento a un `GeneratoreCodici` e, in

`prossimaEtichetta()`, chiama `emettiCodice()`. Può anche creare l'oggetto di libreria solo alla prima chiamata, quando serve davvero: è la tecnica della **lazy initialization**.

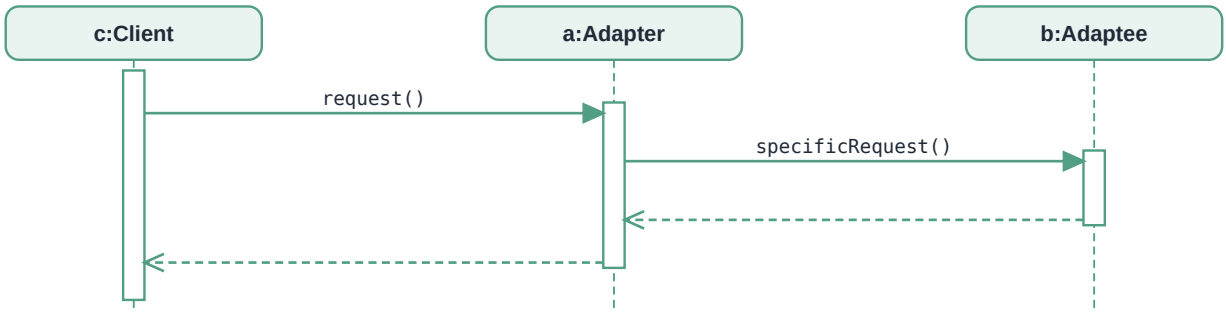


Figura 11.2: la sequenza delle chiamate: il client chiama request(), l'adapter la traduce in specificRequest().

Soluzione Class Adapter

Nella seconda versione l'Adapter non contiene l'Adaptee ma ne è una **sottoclasse**, e nello stesso tempo implementa Target. Il metodo `request()` chiama direttamente `specificRequest()`, ereditato. In Java, che non ha ereditarietà multipla fra classi, questo è possibile solo se Target è un'interfaccia.

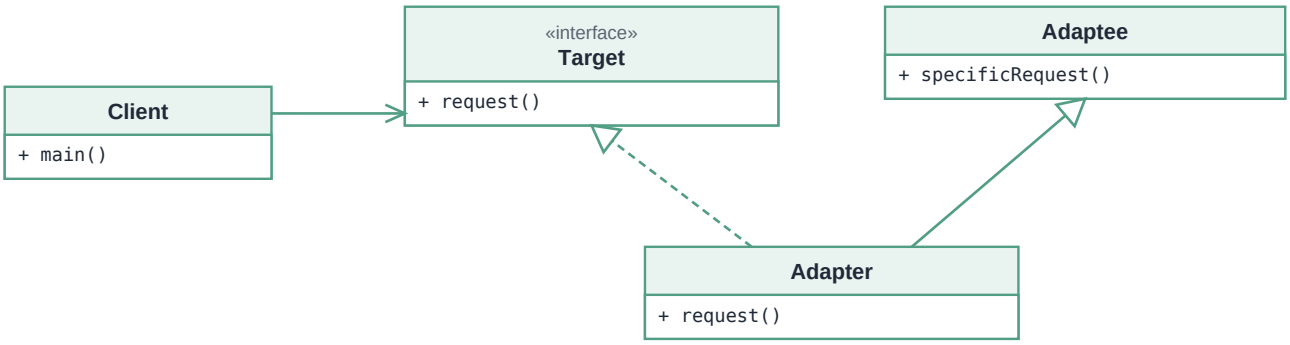


Figura 11.3: Class Adapter; l'adapter eredita dall'adaptee e implementa il target.

Poiché offre sia l'interfaccia di Target sia quella ereditata da Adaptee, il Class Adapter realizza la variante detta **Adapter a due vie**: lo stesso oggetto può essere usato da client che si aspettano l'una o l'altra interfaccia.

Conseguenze

Client e classe di libreria restano **indipendenti**: nessuno dei due viene modificato. L'adapter può cambiare il comportamento dell'adaptee e aggiungere controlli di **precondizioni** (ciò che deve valere prima dell'esecuzione) e **postcondizioni** (ciò che vale dopo, se tutto è andato bene). L'Object Adapter può usare la lazy initialization e funziona anche con le sottoclassi dell'adaptee; il Class Adapter è legato a una classe precisa ma può ridefinirne i metodi. Il prezzo è un **livello di indirezione**: ogni chiamata del client ne provoca un'altra, con un rallentamento trascurabile ma codice un po' più difficile da seguire.

DA RICORDARE

Adapter converte un'interfaccia in quella attesa dal client. **Object Adapter**: implementa Target e contiene un riferimento all'Adaptee (composizione). **Class Adapter**: implementa Target ed estende Adaptee (ereditarietà), ed è un adapter a due vie. Client e libreria restano indipendenti, al prezzo di un livello di indirezione.

CAPITOLO 12

12. Facade

Come offrire a chi usa un sottosistema complesso un unico punto di accesso semplificato.

Il design pattern Facade

DEFINIZIONE

Intento: fornire un'interfaccia unificata al posto dell'insieme di interfacce di un sottosistema, definendo un'interfaccia di alto livello che rende il sottosistema più facile da usare.

Facade in inglese è la facciata di un edificio: chi passa per strada vede una sola parete ordinata, non le stanze, i tubi e i cavi che stanno dietro. Allo stesso modo, la reception di un albergo è l'unico interlocutore del cliente, che non deve parlare direttamente con cucina, lavanderia e manutenzione.

PROBLEMA

Spesso molte classi svolgono funzioni correlate, e l'insieme delle loro interfacce è complesso: è difficile capire quale sia l'interfaccia essenziale per i client. Si vogliono ridurre le comunicazioni e le dipendenze dirette fra i client e il sottosistema.

SOLUZIONE

La classe Facade offre ai client un'unica interfaccia semplificata e nasconde gli oggetti del sottosistema; i suoi metodi invocano a loro volta i metodi degli oggetti nascosti, nell'ordine giusto. Il client interagisce solo con il Facade. Per rendere davvero inaccessibili le classi del sottosistema, le si può annidare dentro la classe Facade.

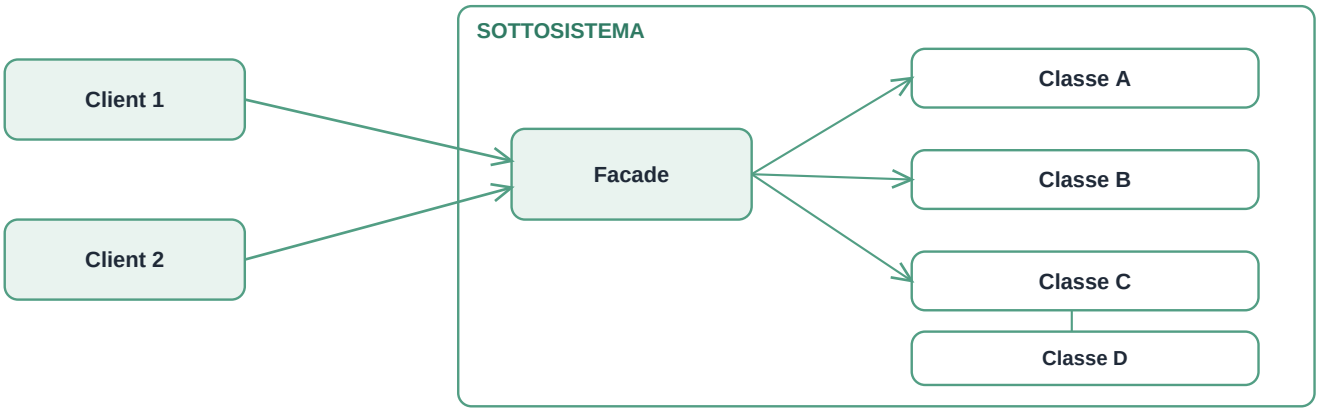


Figura 12.1: i client parlano solo con il Facade, che coordina le classi del sottosistema.

Un esempio: un sistema di traduzione ha una classe per il dizionario inglese e una per quello italiano. Per aggiungere una parola bisogna verificare che esista nel dizionario inglese, aggiungerla al testo inglese, trovarne la posizione e aggiungere al testo italiano la parola nella stessa posizione. Il Facade `Traduttore` offre il solo metodo `aggiungi(parola)`, che esegue tutti questi passi, e il metodo `stampaTutto()`; il client non vede i due dizionari.

Conseguenze

- Nasconde ai client l’implementazione del sottosistema e riduce il numero di oggetti con cui devono interagire.
- Promuove l’**accoppiamento debole** fra sottosistema e client: il sottosistema può cambiare senza toccare i client.
- Riduce le dipendenze di compilazione nei sistemi grandi: se cambia una classe del sottosistema si ricompila fino al Facade, non tutti i client.
- Non impedisce, quando serve, che client più esigenti accedano direttamente agli oggetti del sottosistema.

Facade e Adapter si somigliano perché entrambi avvolgono altre classi, ma hanno scopi diversi: Adapter **converte** un’interfaccia esistente in un’altra già attesa dal client, Facade **semplifica** definendo un’interfaccia nuova per un insieme di classi.

DA RICORDARE

Facade offre un’interfaccia unica e semplificata a un sottosistema: i client parlano solo con il Facade, che invoca le classi nascoste. Riduce dipendenze e accoppiamento, non vieta l’accesso diretto quando serve. Adapter converte, Facade semplifica.

CAPITOLO 13

13. Stream Java: lambda e operazioni di base

Lo stile funzionale introdotto da Java 8: espressioni lambda, interfacce funzionali, riferimenti a metodi e le operazioni fondamentali sugli stream, filter, map e reduce.

Dallo stile imperativo allo stile funzionale

I linguaggi tendono a semplificare il lavoro del programmatore spostando nelle librerie parte del codice che prima andava scritto a mano. Java 5 ha introdotto il for avanzato, che scorre una lista senza indici; Java 7 l'inferenza del tipo nella creazione delle collezioni, `new ArrayList<>()`. Java 8 (2014) ha fatto un passo più grande: pur restando un linguaggio imperativo, ha introdotto caratteristiche della **programmazione funzionale**, che in genere è più concisa, più espressiva e più facile da parallelizzare.

Consideriamo il compito di verificare se una lista di nomi contiene "Luca". In stile **imperativo** si scrive un ciclo, una variabile booleana che fa da segnalino e un `break`: il programmatore controlla ogni dettaglio, ma deve scrivere molte righe e chi legge deve esaminare il corpo del ciclo per capire lo scopo. L'iterazione è **esterna**, fuori dal codice della lista. In stile **dichiarativo** si scrive `nomi.contains("Luca")`: si dice *che cosa* si vuole e non *come* ottenerlo, i dettagli stanno nella libreria e l'iterazione è **interna**.

DEFINIZIONE

Lo stile **funzionale** è uno stile dichiarativo che aggiunge le **funzioni di ordine superiore**: funzioni che ricevono, creano o restituiscono altre funzioni. Da Java 8 si possono passare funzioni ai metodi, crearle dentro i metodi e restituirle. Un metodo appartiene a una classe; una funzione non è associata ad alcuna classe.

Dalle classi anonime alle espressioni lambda

Supponiamo che una classe `Allarme` riceva nel costruttore un oggetto che implementa l'interfaccia `Avviso`, con l'unico metodo `invia(String s)`. Fino a Java 7 avevamo due modi per fornirlo: scrivere una classe con nome che implementa l'interfaccia, oppure, da Java 1.1, una **classe anonima**, cioè una classe senza nome definita nel punto stesso in cui si crea l'oggetto. Entrambe richiedono molto codice di contorno (boilerplate) per esprimere una sola riga di comportamento. Con Java 8 basta un'espressione lambda.

```
// classe anonima: tanto contorno per una sola riga utile
Allarme a1 = new Allarme(new Avviso() {
    public void invia(String s) { System.out.println("Attenzione: " + s); }
});

// espressione lambda: solo il comportamento
Allarme a2 = new Allarme(s -> System.out.println("Attenzione: " + s));
```

DEFINIZIONE

Un'**espressione lambda** è una funzione anonima: a sinistra della freccia ci sono i parametri, a destra il corpo. Come un metodo ha parametri, corpo e tipo di ritorno, ma non ha nome e si può passare come argomento.

FORMA	SIGNIFICATO
`s → ...`	un solo parametro: le parentesi si possono omettere
`(x, y) → x + y`	più parametri: servono le parentesi; il valore dell'espressione è il risultato
`() → ...`	nessun parametro: parentesi vuote
`(x, y) → { ...; return x + y; }`	più istruzioni: servono le graffe e il return

Nel testo della dispensa la freccia delle lambda è resa con il simbolo →; nel codice si scrive con i due caratteri `->`. I tipi dei parametri sono dedotti dal compilatore: nell'esempio, `s` è una String perché tale è il parametro di `invia`. Nei linguaggi funzionali puri una lambda è una **funzione pura**, il cui risultato dipende solo dagli ingressi; in Java una lambda può anche accedere a uno stato esterno, e quindi non essere pura.

Interfacce funzionali e riferimenti a metodi

DEFINIZIONE

Un'**interfaccia funzionale** è un'interfaccia con **un solo metodo astratto**. Una lambda può essere usata ovunque sia atteso un oggetto di un'interfaccia funzionale: la lambda diventa l'implementazione di quel metodo. L'annotazione `@FunctionalInterface` chiede al compilatore di verificare che il metodo astratto sia uno solo, così nessuno potrà aggiungerne altri in futuro.

INTERFACCIA	METODO	CHE COSA RAPPRESENTA	ESEMPIO
Predicate<T>	test(T): boolean	una condizione su un valore	`x → x >= 0`
Function<T, R>	apply(T): R	una trasformazione da T a R	`s → s.length()`
Supplier<T>	get(): T	un fornitore di valori, senza ingressi	`() → "ciao"`
BinaryOperator<T>	apply(T, T): T	combina due valori in uno	`(a, b) → a + b`
Comparator<T>	compare(T, T): int	un criterio di ordinamento	`(a, b) → a.compareTo(b)`

Un Supplier non contiene il valore, ma il codice per produrlo, che viene eseguito a ogni chiamata di `get()`. Per le funzioni già esistenti come metodi si può usare un **riferimento a metodo**, con la sintassi `::`: `Integer::sum` equivale a `(a, b) → a + b`, `Prodotto::getPrezzo` equivale a `p → p.getPrezzo()`, `System.out::println` equivale a `x → System.out.println(x)`, e `Prodotto::new` è un riferimento al costruttore.

Collection, metodi di default e stream

Le librerie Java offrono molte interfacce e classi collection. `Collection` definisce metodi come `add()`, `remove()`, `size()`, `contains()`; `List` aggiunge `get()`, `indexOf()`, `set()`. Le implementazioni hanno costi diversi.

CLASSE	GET	ADD	CONTAINS	STRUTTURA
ArrayList	costante	costante (in media)	O(n)	array che cresce del 50 per cento quando è pieno; elementi contigui
LinkedList	O(n)	costante	O(n)	lista in cui ogni elemento punta al successivo e al precedente
TreeSet		O(log n)	O(log n)	insieme ordinato basato su un albero
Vector	costante	costante (in media)	O(n)	come ArrayList, ma con metodi synchronized

Java 8 ha introdotto anche i **metodi di default**: metodi con implementazione dentro un'interfaccia. Grazie a essi si sono potuti aggiungere metodi nuovi a interfacce già esistenti senza rompere le applicazioni scritte in precedenza. A differenza delle classi astratte, che possono avere costruttori e attributi, i metodi di default non agiscono su uno stato. Il metodo di default più importante è `stream()` di Collection.

DEFINIZIONE

`stream()` restituisce uno **Stream<T>**, una sequenza di elementi di tipo T presi dalla collezione, su cui si possono applicare operazioni in cascata. Lo stream non modifica la collezione di origine: ogni operazione produce un nuovo stream.

Le operazioni sugli stream sono di due tipi. Le **intermedie** restituiscono un altro stream e sono **lazy** (pigre): non vengono eseguite nel punto in cui sono scritte, ma solo quando serve. Le **terminali** producono un risultato che non è uno stream e sono **eager**: forzano l'esecuzione di tutte le operazioni precedenti. Una catena di operazioni si chiama **pipeline**.

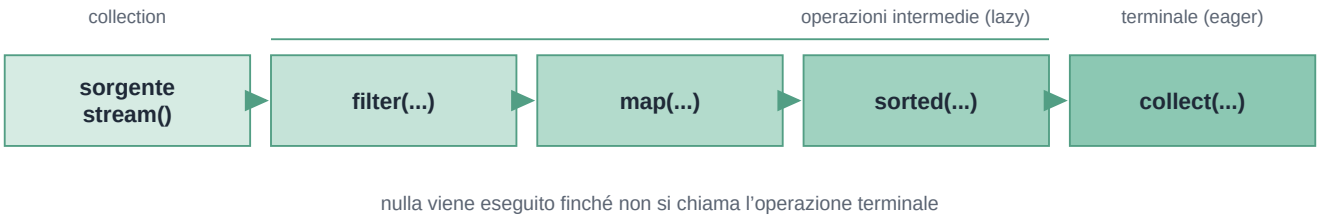


Figura 13.1: una pipeline di stream; le intermedie descrivono il lavoro, la terminale lo fa eseguire.

filter

`filter(Predicate<T> p)` è un'operazione intermedia che restituisce uno stream con i soli elementi che soddisfano il predicato, cioè per cui la funzione passata restituisce true. Per esempio, per contare i nomi di cinque lettere: `nomi.stream().filter(s → s.length() == 5).count()`. Qui `count()` è l'operazione terminale che conta gli elementi rimasti. Un predicato si può anche salvare in una variabile, `Predicate<Integer> positivo = x → x >= 0;`, e passare a `filter(positivo)`.

Seguiamo passo per passo `Stream.of("mela", "kiwi", "arancia", "pera").filter(s → s.length() == 4).count()`.

ELEMENTO	LENGTH()	PREDICATO	ESITO
"mela"	4	4 == 4 vero	passa
"kiwi"	4	4 == 4 vero	passa
"arancia"	7	7 == 4 falso	scartato
"pera"	4	4 == 4 vero	passa

Restano tre elementi, quindi `count()` restituisce 3.

map

`map(Function<T, R> f)` è un'operazione intermedia che applica la funzione a ogni elemento e restituisce lo stream dei risultati: ogni elemento viene **trasformato**, non scartato. `map(x → x * 2)` raddoppia ogni numero; `map(Prodotto::getPrezzo)` trasforma uno stream di prodotti nello stream dei loro prezzi.

reduce e Optional

`reduce(T identita, BinaryOperator<T> accumulatore)` è un'operazione terminale che riduce tutti gli elementi a un **unico valore** dello stesso tipo, per esempio la loro somma. Il primo argomento è il valore iniziale dell'accumulatore; il secondo è una funzione che riceve l'accumulatore e l'elemento corrente e restituisce il nuovo valore dell'accumulatore.

Seguiamo passo per passo `Stream.of(3, 5, 2).reduce(0, (acc, v) → acc + v)`.

PASSO	ACC PRIMA	V	ACC + V
inizio	0 (identità)		
1	0	3	3
2	3	5	8
3	8	2	10

Il risultato è 10. Con il riferimento a metodo si scrive in modo equivalente `reduce(0, Integer::sum)`. Esiste anche la forma con il solo accumulatore, `reduce(Integer::sum)`: senza valore iniziale, se lo stream è vuoto il risultato non esiste, e per questo restituisce un **Optional**.

DEFINIZIONE

Optional<T> rappresenta un valore che può esserci o no. `isPresent()` dice se il valore c'è, `get()` lo estrae (va chiamato solo se il valore è presente), `isEmpty()` dice se manca. Evita di restituire null e costringe chi chiama a considerare il caso vuoto.

Combinando le tre operazioni si scrive in poche righe ciò che in stile imperativo richiede un ciclo con condizioni annidate. Per trovare il prodotto più caro fra quelli sotto i 20 euro: `prodotti.stream().filter(p → p.getPrezzo() < 20).reduce(Prodotto::piuCaro)`, dove `piuCaro(a, b)` è un metodo statico che restituisce il più caro dei due. `filter` separa gli elementi che soddisfano la condizione, `reduce` li confronta a due a due tenendo il vincitore, e il risultato è un Optional, vuoto se nessun prodotto costa meno di 20.

max, Comparator e collect

Lo stesso risultato si ottiene con `max(Comparator)`, operazione terminale che restituisce un Optional con l'elemento massimo secondo il comparatore. Un **Comparator** implementa `compare(a, b)`, che restituisce zero se i due elementi sono equivalenti, un valore positivo se a viene dopo b, un valore negativo se a viene prima. Invece di scriverlo a mano si usa `Comparator.comparing(Prodotto::getPrezzo)`, che riceve una funzione che estrae la chiave di confronto e costruisce il comparatore.

`collect(Collector)` è un'operazione terminale che raccoglie gli elementi dello stream in una struttura. La classe `Collectors` offre collector pronti: `Collectors.toList()` accumula gli elementi in una List. Da Java 16 esiste anche la scorciatoia `stream.toList()`. Per esempio, `prodotti.stream().map(Prodotto::getPrezzo).collect(Collectors.toList())` produce la lista dei prezzi.

DA RICORDARE

Una **lambda** è una funzione anonima che implementa il metodo di un'**interfaccia funzionale** (Predicate, Function, Supplier, BinaryOperator, Comparator). Le operazioni **intermedie** (filter, map) sono lazy e restituiscono uno stream; le **terminali** (count, reduce, max, collect) sono eager e producono il risultato. **filter** scarta, **map** trasforma, **reduce** accumula in un solo valore. **Optional** rappresenta un risultato che può mancare.

CAPITOLO 14

14. Stream Java: operazioni avanzate e parallelismo

Ordinare, eliminare i duplicati, cercare con cortocircuito, generare stream, lavorare con gli interi, fare debug con peek e capire quando uno stream può essere eseguito in parallelo.

Ordinare, eliminare duplicati, cercare

OPERAZIONE	TIPO	CHE COSA FA
sorted(Comparator)	intermedia, stateful	Restituisce gli elementi ordinati secondo il comparatore.
distinct()	intermedia, stateful	Restituisce gli elementi senza duplicati.
limit(n)	intermedia	Tronca lo stream ai primi n elementi.
forEach(azione)	terminale	Esegue un'azione su ogni elemento, per esempio la stampa.
findAny(), findFirst()	terminale, cortocircuito	Restituiscono un Optional con un elemento qualsiasi o con il primo.

Esempio: data una lista di prodotti, stampare in ordine alfabetico i nomi di quelli economici della categoria "frutta".

```
prodotti.stream()
        .filter(p -> p.getPrezzo() < 3)                 // solo quelli economici
        .filter(p -> p.getCategoria().equals("frutta")) // solo la frutta
        .sorted(Comparator.comparing(Prodotto::getNome))  // ordine alfabetico
        .forEach(p -> System.out.print(p.getNome() + " "));
```

Per elencare le categorie presenti, ciascuna una volta sola, si trasforma ogni prodotto nella sua categoria e si eliminano i duplicati: `prodotti.stream().map(Prodotto::getCategoria).distinct().forEach(...)`.

`findAny()` è un'operazione di **cortocircuito** (short circuiting): per produrre il risultato non serve esaminare tutto lo stream, appena un elemento arriva in fondo alla pipeline la ricerca si ferma, e alcune parti del lavoro non vengono mai eseguite. Su uno stream parallelo `findAny()` può restituire un elemento qualsiasi, mentre `findFirst()` garantisce il primo nell'ordine dello stream.

Operazioni stateless e stateful

Le operazioni si distinguono anche per la memoria di cui hanno bisogno, e la distinzione conta per il parallelismo.

- **Stateless**, come `map()` e `filter()`: non hanno stato interno, prendono un elemento e producono zero o un risultato, indipendentemente dagli altri elementi.
- **Con accumulatore limitato**, come `reduce()` e `max()`: accumulano un risultato di dimensione fissa, indipendente dal numero di elementi; il risultato di una passata entra nella passata successiva.

- **Stateful**, come `sorted()` e `distinct()`: per lavorare devono conoscere anche gli altri elementi dello stream, e quindi devono memorizzarli.

Generare stream

Uno stream non deve per forza nascere da una collezione. `Stream.of(a, b, c)` lo crea dai valori elencati. `Stream.iterate(seme, f)` crea uno stream **infinito** e ordinato applicando ripetutamente una funzione a partire da un seme: seme, f(seme), f(f(seme)), e così via; ogni valore è calcolato dal precedente, quindi l'operazione è sequenziale. `Stream.generate(supplier)` crea uno stream infinito chiamando ogni volta un Supplier, senza legare un valore al precedente. Gli stream infiniti si usano sempre con `limit(n)`.

Seguiamo `Stream.iterate(3, n → n * 2).limit(5)`: il primo elemento è il seme 3; il secondo è 3 per 2, cioè 6; il terzo è 6 per 2, cioè 12; poi 24 e 48. Lo stream contiene 3, 6, 12, 24, 48. Invece `Stream.generate(() → Math.round(Math.random() * 10)).limit(5)` produce cinque numeri casuali fra 0 e 10.

Stream di interi

IntStream è uno stream di valori int primitivi, con operazioni numeriche dedicate.

OPERAZIONE	EFFETTO	ESEMPIO
IntStream.rangeClosed(a, b)	gli interi da a a b, estremi inclusi	rangeClosed(1, 5) dà 1, 2, 3, 4, 5
sum()	somma dei valori (terminale)	rangeClosed(1, 5).sum() dà 15
mapToInt(f)	da uno Stream di oggetti a un IntStream	mapToInt(String::length)
boxed()	da IntStream a Stream<Integer>	rangeClosed(1, 10).boxed()
mapToObj(f)	da IntStream a uno Stream di oggetti	mapToObj(i → lista.get(i))

Per esempio, `IntStream.rangeClosed(1, 4).map(x → x * x).sum()` calcola i quadrati 1, 4, 9, 16 e li somma: 1 più 4 fa 5, più 9 fa 14, più 16 fa 30. `mapToObj` è molto utile per lavorare sugli **indici** di una lista: `IntStream.rangeClosed(0, 2).mapToObj(i → lista.get(i))` produce lo stream dei primi tre elementi, e con gli indici si possono confrontare elementi vicini, cosa che uno stream di valori da solo non permette.

Debug con peek

Per capire che cosa produce ogni operazione non si può usare `forEach()`, perché è terminale e consuma gli elementi. Si usa invece `peek(azione)`, un'operazione intermedia che esegue l'azione su ogni elemento e lo lascia passare invariato. Inserendo un `peek` che stampa dopo ogni passo, si scopre un fatto importante: gli elementi non attraversano la pipeline "una operazione alla volta", ma **uno alla volta**, ciascuno fino in fondo.

Prendiamo la lista 2, 3, 4, 5 con `map(x → x + 17)`, poi `filter(x → x % 2 == 0)`, poi `limit(3)`.

ELEMENTO	DOPO MAP	FILTER (PARI?)	ESITO
2	19	no	scartato
3	20	sì	passa limit, raccolto
4	21	no	scartato
5	22	sì	passa limit, raccolto

Il risultato è 20, 22: `limit(3)` non arriva a fermare nulla, perché gli elementi pari sono solo due. `peek` serve al debug: usarlo per modificare lo stato degli oggetti è sconsigliato, perché su una pipeline pigra o con cortocircuito l'azione potrebbe non essere eseguita su tutti gli elementi.

Funzioni pure ed effetti collaterali

Consideriamo il problema di verificare se una lista di interi è ordinata. Una prima soluzione genera gli indici da 0 a n meno 2 e filtra quelli in cui un elemento è maggiore del successivo: `IntStream.rangeClosed(0, lista.size() - 2).filter(i → lista.get(i) > lista.get(i + 1)).findAny().isEmpty()`. Se non si trova alcuna coppia fuori ordine, la lista è ordinata; appena se ne trova una, `findAny` ferma la ricerca. La lambda passata a filter legge due valori e restituisce un booleano, senza modificare nulla: è una **funzione pura**, priva di effetti collaterali.

Una seconda soluzione scorre i valori e tiene in un attributo `prec` l'elemento precedente: la lambda legge `prec`, lo aggiorna con il valore corrente e restituisce true se il valore corrente è minore del precedente. Funziona in sequenza, ma la lambda modifica uno **stato condiviso**: ha un effetto collaterale e non è pura. Le lambda possono accedere al contesto esterno, ma una modifica di stato condiviso impedisce l'esecuzione parallela, perché il risultato dipenderebbe dall'ordine in cui i thread elaborano gli elementi.

Stream paralleli

I processori attuali hanno più core, ma la programmazione parallela tradizionale, con la classe Thread, synchronized, wait e notify, è difficile e soggetta a corse critiche. Collection offre il metodo di default `parallelStream()`, che restituisce, quando possibile, uno stream eseguito in parallelo su più core, senza scrivere thread né sincronizzazione.

Questo funziona perché `map` e `filter` sono stateless e lo stream di origine non viene modificato: il risultato non dipende dall'ordine in cui gli elementi vengono prelevati. Se però una lambda aggiorna uno stato globale, l'operazione non è più stateless e il parallelismo diventa scorretto. Le prestazioni dipendono dal numero di elementi, dalle operazioni, dall'hardware e dal tipo di collezione: su milioni di elementi migliorano con ArrayList, Vector e TreeSet, poco o nulla con LinkedList, che si può percorrere solo in sequenza e quindi è difficile da suddividere fra i core.

Factory Method con Supplier

Le interfacce funzionali semplificano anche i design pattern. Un Factory Method parametrico di solito contiene uno `switch` sul nome del prodotto. Con i Supplier si costruisce invece una **mappa** dal nome al riferimento del costruttore, `Map.of("intero", BigliettoIntero::new, "ridotto", BigliettoRidotto::new)`, e il metodo factory diventa `return mappa.getOrDefault(nome, BigliettoIntero::new).get();`. Aggiungere un prodotto significa aggiungere una voce alla mappa. Il Supplier funziona per costruttori senza parametri; per quelli con parametri serve un'altra interfaccia funzionale, per esempio una Function.

DA RICORDARE

sorted e **distinct** sono intermedie stateful; **findAny** e **findFirst** fanno cortocircuito. **iterate** e **generate** creano stream infiniti, da troncare con **limit**. **IntStream** offre rangeClosed, sum, mapToObj per lavorare sugli indici. **peek** serve al debug e mostra che gli elementi attraversano la pipeline uno alla volta. Solo con lambda **pure** e operazioni stateless un **parallelStream** è corretto.

CAPITOLO 15

15. Composite

Come rappresentare strutture ad albero, in cui contenitori e singoli elementi vengono trattati allo stesso modo.

Il design pattern Composite

DEFINIZIONE

Intento: comporre oggetti in **strutture ad albero** per rappresentare gerarchie parte e tutto. Composite permette ai client di trattare **uniformemente** oggetti singoli e composizioni di oggetti.

MOTIVAZIONE

Spesso bisogna raggruppare elementi semplici per formarne di più grandi, che a loro volta possono essere raggruppati. Su un disco ci sono file e cartelle; una cartella contiene file e altre cartelle, e così via, ma il client vuole creare, copiare o cancellare senza distinguere fra i due. In un editor grafico ci sono figure semplici, come linee e rettangoli, e gruppi di figure, e sia le une sia gli altri si devono poter disegnare e spostare. In un negozio online un ordine contiene prodotti e confezioni regalo, che a loro volta contengono prodotti.

Se le classi degli elementi semplici e quelle dei contenitori sono distinte, il codice che le usa deve trattarle in modo diverso, con continui controlli sul tipo, e diventa complicato. Composite descrive invece una **composizione ricorsiva**, in cui i client non devono distinguere i tipi di elementi.

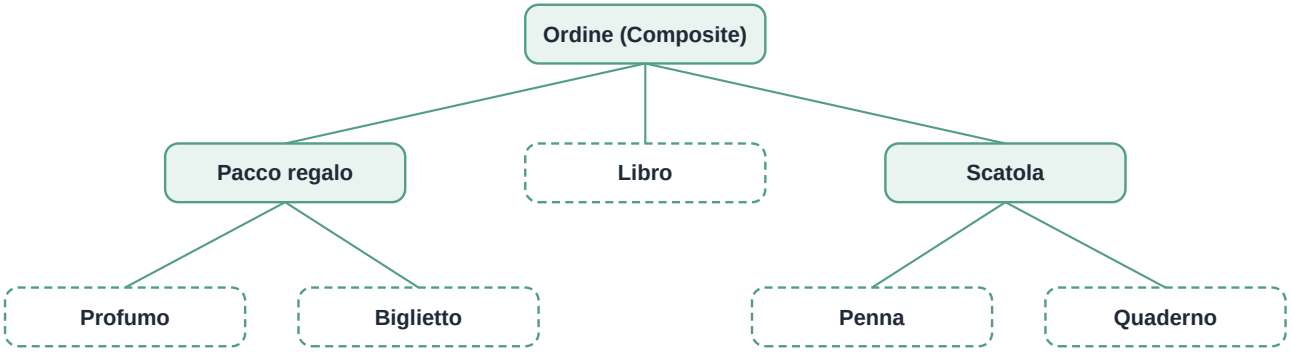


Figura 15.1: un ordine come albero; i nodi pieni sono composti, quelli tratteggiati sono foglie.

SOLUZIONE

RUOLO	RESPONSABILITÀ
Component	Interfaccia (o classe astratta) comune a elementi semplici e composti: dichiara le operazioni della composizione ed eventualmente quelle per gestire i figli e per risalire al padre.
Leaf	Elemento semplice, senza figli; implementa direttamente il comportamento.
Composite	Elemento contenitore: tiene i riferimenti ai figli, implementa le operazioni per gestirli e definisce il comportamento dell'aggregato, di solito inoltrando la richiesta a ciascun figlio.
Client	Manipola gli oggetti solo attraverso l'interfaccia Component.

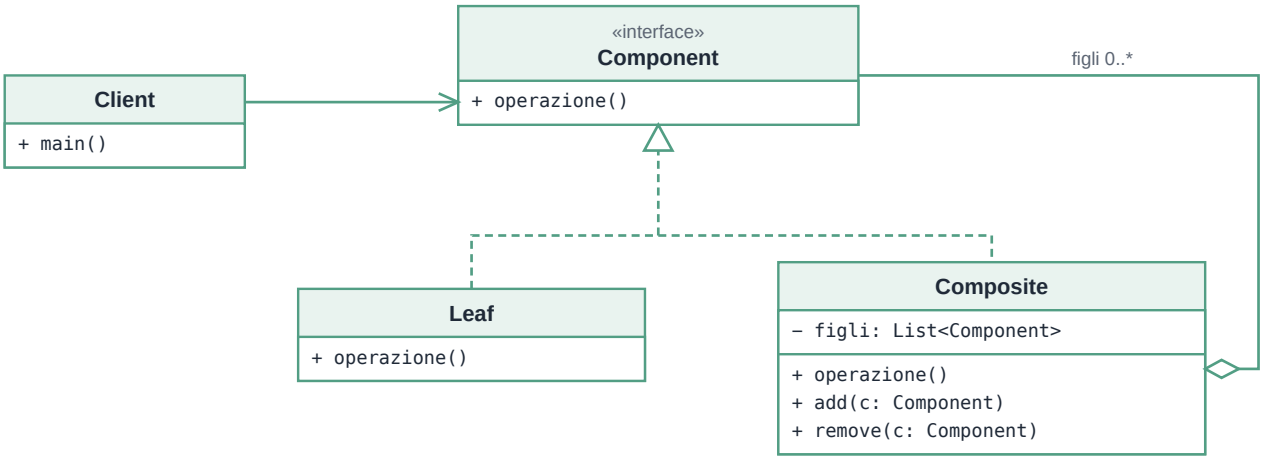


Figura 15.2: la struttura di Composite; un Composite contiene altri Component, foglie o composti.

Le collaborazioni sono semplici: se il destinatario di una richiesta è una foglia, la gestisce direttamente; se è un composite, la inoltra ai suoi figli, eventualmente con operazioni aggiuntive prima e dopo. Per calcolare il prezzo di un ordine, il composite somma i prezzi restituiti dai figli; un figlio composto fa lo stesso con i suoi, e la ricorsione termina sulle foglie.

DOVE DICHIARARE ADD E REMOVE

È la scelta più delicata del pattern. Se si dichiarano in **Component** si ottiene **trasparenza**: il client tratta tutti gli elementi allo stesso modo. Ma per le foglie quelle operazioni non hanno senso, e la **sicurezza** si perde: un client potrebbe chiamare `add()` su una foglia, e ignorare silenziosamente la richiesta nasconderebbe un difetto del programma. Se invece si dichiarano solo in **Composite** si ottiene sicurezza, perché il compilatore impedisce di chiamarle su una foglia, ma si perde la trasparenza. Una via di mezzo è un metodo `getComposite()` in Component che restituisce null, ridefinito in Composite per restituire se stesso; il client deve però comunque distinguere il risultato.

DETTAGLI DI IMPLEMENTAZIONE

- La lista dei figli va definita in Composite: in Component sprecherebbe spazio in ogni foglia.
- Per facilitare la navigazione, i figli possono tenere un riferimento al padre, dichiarato in Component.
- Component dovrebbe definire quante più operazioni comuni possibile, ridefinite da Leaf e Composite.
- L'ordine dei figli può essere importante; un composite può tenere una cache dei risultati, che i figli devono poter invalidare.

CONSEGUENZE

Gli elementi si compongono ricorsivamente; dove il client si aspetta un elemento semplice può riceverne uno composto; i client restano semplici perché non sanno se trattano una foglia o un composite; nuovi tipi di foglie o composite funzionano subito con i client esistenti. Lo svantaggio è che non si può vincolare in fase di progettazione un composite a contenere solo certi tipi di componenti: servono controlli a runtime.

DA RICORDARE

Composite: struttura ad albero in cui foglie e contenitori condividono l'interfaccia Component; il composite inoltra le richieste ai figli. Gestione dei figli in Component significa **trasparenza**, in Composite significa **sicurezza**.

CAPITOLO 16

16. Decorator

Come aggiungere responsabilità a un singolo oggetto, a runtime, avvolgendolo in altri oggetti con la stessa interfaccia.

Il design pattern Decorator

DEFINIZIONE

Intento: aggiungere dinamicamente ulteriori responsabilità a un oggetto. I decorator sono un'alternativa flessibile alle sottoclassi per estendere le funzionalità.

MOTIVAZIONE

A volte si vogliono aggiungere responsabilità a **singoli oggetti** e non a un'intera classe. Con l'ereditarietà la responsabilità vale per tutte le istanze della sottoclasse, e il client non può decidere come e quando applicarla: se `TestoConBordo` estende `Testo`, non si possono avere testi di quella classe senza bordo. Inoltre, con molte estensioni combinabili, le sottoclassi esploderebbero: bordo, ombra e scorrimento richiederebbero una classe per ogni combinazione.

L'approccio flessibile è **racchiudere** (wrap) il componente in un altro oggetto che aggiunge la responsabilità: il decorator. Il decorator ha la stessa interfaccia del componente che decora, quindi la sua presenza è trasparente per chi lo usa; inoltra le richieste al componente e aggiunge le sue attività prima o dopo. I decorator si possono annidare, come gli strati di una cipolla, e togliere a runtime.

SOLUZIONE

RUOLO	RESPONSABILITÀ
Component	Interfaccia degli oggetti a cui si possono aggiungere responsabilità.
ConcreteComponent	L'oggetto base da decorare.
Decorator	Implementa Component, tiene un riferimento a un Component e gli inoltra le richieste.
ConcreteDecorator	Aggiunge la responsabilità, con stato o comportamento propri, prima o dopo l'inoltro.

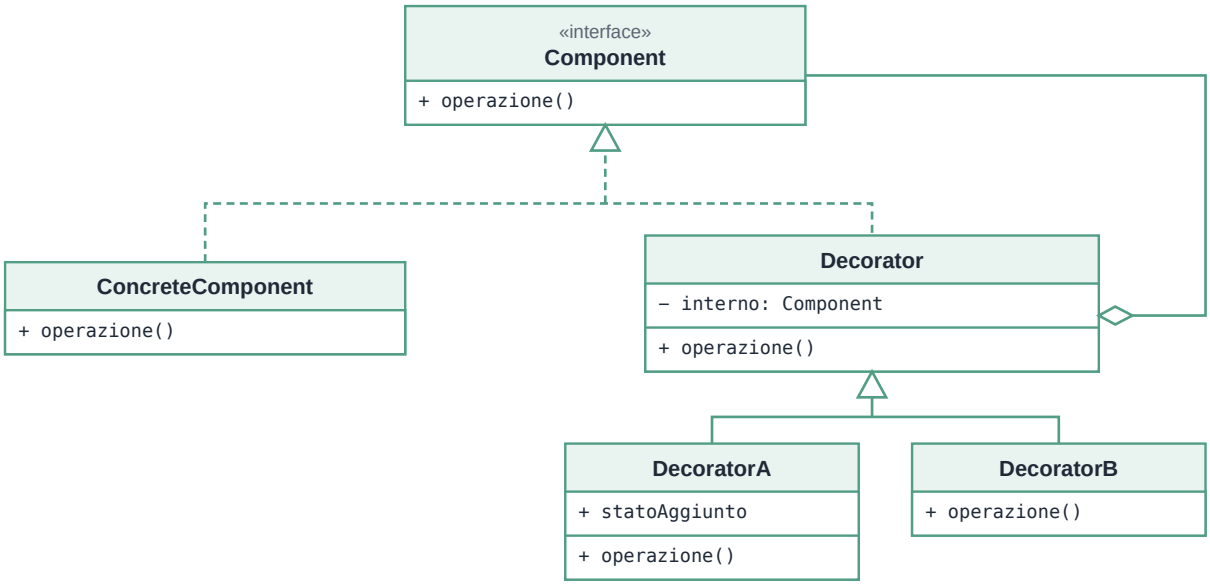


Figura 16.1: la struttura di Decorator; il decorator è un Component e contiene un Component.

Un esempio: un `Messaggio` ha il metodo `testo()`. Il decorator `Maiuscolo` chiama `testo()` sull'oggetto interno e restituisce il risultato in maiuscolo; il decorator `Firma` aggiunge una firma in fondo. Scrivendo `new Firma(new Maiuscolo(new Messaggio("ciao")))` la chiamata entra dallo strato più esterno, attraversa gli strati fino al messaggio e il risultato risale trasformato da ciascuno.

Figura 16.2: decorator annidati come strati; ogni strato inoltra e arricchisce.

CONSEGUENZE

- Più flessibilità dell'ereditarietà: le responsabilità si aggiungono e tolgono a runtime, e la stessa si può aggiungere due volte usando due istanze dello stesso decorator.
- Le classi in cima alla gerarchia non devono prevedere tutte le responsabilità possibili: i decorator sono indipendenti e si aggiungono in seguito.
- L'identità del decorator non è quella del componente decorato: confrontare i riferimenti è un errore.
- Si ottengono molti piccoli oggetti simili, che differiscono per come sono collegati, e il sistema può essere più difficile da capire.

DA RICORDARE

Decorator avvolge un oggetto in un altro con la stessa interfaccia, che inoltra e aggiunge comportamento. Si annidano e si combinano a runtime, sui singoli oggetti, evitando l'esplosione di sottoclassi. Ruoli: Component, ConcreteComponent, Decorator, ConcreteDecorator.

CAPITOLO 17

17. Bridge

Come separare un’astrazione dalla sua implementazione, perché le due gerarchie possano crescere in modo indipendente.

Il design pattern Bridge

DEFINIZIONE

Intento: disaccoppiare un’astrazione dalla sua implementazione, così che le due possano variare indipendentemente.

MOTIVAZIONE

Quando un’astrazione può avere più implementazioni si usa di solito l’ereditarietà. Supponiamo di dover disegnare rettangoli e cerchi (le astrazioni) su due piattaforme grafiche, P1 e P2 (le implementazioni), che offrono primitive diverse: P1 disegna le linee con un metodo, P2 con un altro. Con l’ereditarietà servirebbe una sottoclasse per ogni combinazione.

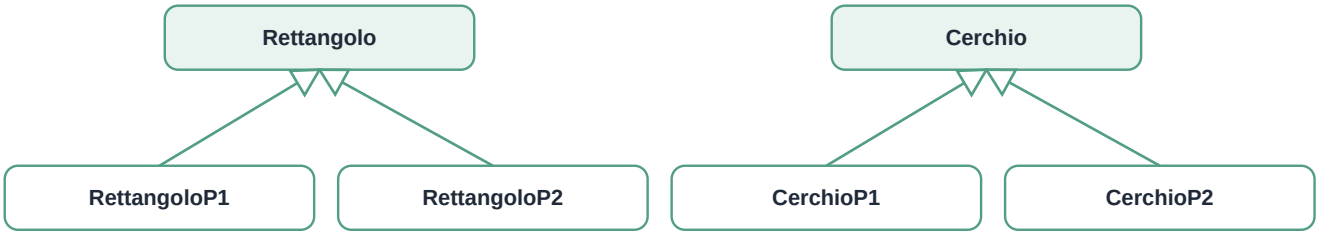


Figura 17.1: senza Bridge, due forme per due piattaforme richiedono quattro classi concrete.

Contiamo le classi concrete. Con 2 forme e 2 piattaforme sono 2 per 2, cioè 4. Aggiungendo una piattaforma P3 diventano 2 per 3, cioè 6. Aggiungendo poi una terza forma diventano 3 per 3, cioè 9. La crescita è **moltiplicativa**, mentre vorremmo che ogni nuova variante aggiungesse una sola classe. C’è anche un secondo problema: un oggetto `RettangoloP1` è legato per sempre a P1, e non può passare a un’altra piattaforma.

SOLUZIONE

RUOLO	RESPONSABILITÀ
Abstraction	Definisce l’interfaccia per i client e tiene un riferimento a un Implementor, a cui inoltra le richieste.
RefinedAbstraction	Estende l’interfaccia di Abstraction (per esempio Rettangolo, Cerchio).
Implementor	Interfaccia delle classi di implementazione; di solito offre operazioni primitive, su cui Abstraction costruisce operazioni di livello più alto.
ConcreteImplementor	Implementa Implementor per una piattaforma specifica.

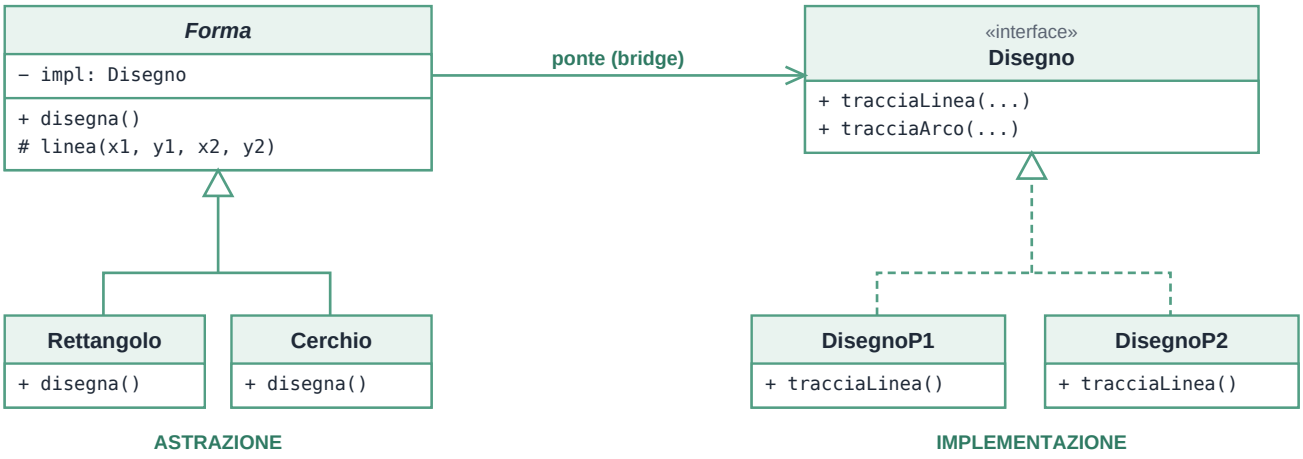


Figura 17.2: con Bridge le due gerarchie sono separate e collegate da un solo riferimento.

Ora `Rettangolo.disegna()` chiama `linea()` della superclasse `Forma`, che chiama `tracciaLinea()` sull'implementor ricevuto, per esempio un `DisegnoP1`, che a sua volta usa la primitiva della piattaforma. Una nuova forma, come un rombo, è solo una nuova sottoclasse di Forma; una nuova piattaforma è solo una nuova implementazione di Disegno. Le classi crescono in modo **additivo**: 3 forme e 3 piattaforme richiedono 3 più 3, cioè 6 classi concrete invece di 9.

CONSEGUENZE

- L'implementazione non è legata in modo permanente all'astrazione: si può configurare e anche cambiare a runtime.
- Si può cambiare l'implementazione senza ricompilare Abstraction e client.
- I client non conoscono Implementor; solo alcuni strati del software devono conoscere entrambe le gerarchie.
- Le gerarchie di Abstraction e Implementor evolvono in modo indipendente.

DA RICORDARE

Bridge separa due dimensioni di variazione in due gerarchie collegate da un riferimento (il ponte): le classi crescono in modo additivo e non moltiplicativo, e l'implementazione si sceglie anche a runtime. Ruoli: Abstraction, RefinedAbstraction, Implementor, ConcreteImplementor.

CAPITOLO 18

18. Processi di sviluppo del software

Le fasi in cui si articola lo sviluppo, dai requisiti alla manutenzione, e i principali modelli di processo: a cascata, evolutivo, incrementale, basato su componenti, a spirale e agile.

Che cos'è un processo software

DEFINIZIONE

Un **processo software** descrive le attività (o task) necessarie per sviluppare un prodotto software e il modo in cui queste attività sono collegate fra loro.

Le attività, o fasi, sono sempre le stesse: **analisi dei requisiti**, che produce le specifiche; **progettazione**; **implementazione**, che produce il codice; **convalida**, o testing, che porta all'approvazione; **manutenzione**. I diversi processi si distinguono per come le ordinano, le ripetono e le sovrappongono.

Analisi dei requisiti

L'analisi dei requisiti è la fase che stabilisce i servizi richiesti al software e i vincoli a cui deve sottostare, arrivando a definirne le specifiche.

TERMINE	SIGNIFICATO
Requisito	Ciascuna delle caratteristiche che il software deve avere. I requisiti tendono a essere granulari: molti e piccoli.
Specifica	Descrizione rigorosa delle caratteristiche del software.
Feature	Insieme di requisiti correlati che permette all'utente di raggiungere un obiettivo, per esempio "navigazione privata" o "gestione delle password" in un browser.
Requisito funzionale	*Che cosa* il sistema deve fare: le funzionalità.
Requisito non funzionale	*Come* il sistema lo fa: affidabilità, efficienza, prestazioni, manutenibilità, sicurezza.

L'ingegneria dei requisiti procede in quattro passi: **studio di fattibilità**, per capire se il sistema conviene ed è realizzabile; **analisi** dei requisiti; **specifica**, cioè la loro scrittura; **convalida**, per verificare che descrivano davvero ciò che il cliente vuole.

RACCOLTA DEI REQUISITI (ELICITAZIONE)

Raccogliere i requisiti significa scoprirli insieme ai portatori di interesse (stakeholder): clienti, utenti finali, esperti del dominio. Le tecniche principali sono le **interviste**, aperte o con domande prefissate; gli **scenari** e i **casi d'uso**, che descrivono esempi concreti di interazione con il sistema; l'**osservazione** del lavoro reale degli utenti (etnografia), utile per scoprire requisiti impliciti che nessuno pensa di dire; i **prototipi**, che fanno reagire gli utenti a qualcosa di concreto. Spesso gli stakeholder non sanno esprimere ciò che vogliono, usano il proprio gergo o hanno richieste in conflitto, e i requisiti cambiano durante l'analisi stessa.

TIPI DI REQUISITI E COME SCRIVERLI

Si distinguono i **requisiti utente**, scritti in linguaggio naturale con eventuali diagrammi per il cliente, dai **requisiti di sistema**, più dettagliati e destinati agli sviluppatori. I requisiti non funzionali possono riguardare il **prodotto** (prestazioni, affidabilità, usabilità), l'**organizzazione** (standard di processo, linguaggi imposti) o vincoli **esterni** (leggi, privacy). Per scriverli bene valgono alcune regole.

- Usare un formato standard e uniforme per tutti i requisiti.
- Distinguere i requisiti obbligatori ("il sistema *deve*") da quelli desiderabili ("il sistema *dovrebbe*").
- Evitare il gergo tecnico e le frasi ambigue; spiegare quando serve il motivo del requisito.
- Renderli **verificabili**: "il sistema deve essere veloce" non si può testare, "la ricerca deve rispondere entro 2 secondi" sì.

Progettazione, implementazione e convalida

La **progettazione** stabilisce la struttura software che realizza le specifiche: suddivide i requisiti, identifica i sottosistemi (cioè progetta l'architettura), specifica le responsabilità dei sottosistemi e progetta interfacce, componenti, strutture dati e algoritmi. Ogni attività produce un documento con un modello: degli oggetti, di sequenza, delle transizioni di stato, strutturale, dei flussi di dati.

L'**implementazione** traduce i modelli in un programma eseguibile e ne rimuove gli errori: i programmatori fanno dei test per scoprire i bug, localizzano l'errore nel codice, lo correggono prima nel modello e poi nel codice, e ripetono il test. Progettazione e implementazione sono attività strettamente legate e spesso si alternano.

La **convalida**, detta anche Verifica e Validazione (V e V), vuole mostrare che il sistema è conforme alle specifiche e che soddisfa le aspettative del cliente. Si svolge con revisioni e test, eseguendo il sistema in condizioni derivate dalle specifiche e con dati reali.

LIVELLO DI TEST	CHE COSA SI VERIFICA
Test di unità (componenti)	I singoli componenti, cioè funzioni, oggetti o loro gruppi, testati indipendentemente.
Test di sistema	Il sistema intero, con particolare attenzione alle proprietà emergenti, che nascono solo dall'integrazione.
Test di accettazione (alfa)	Condotti con i dati del cliente per verificare che il sistema soddisfi le sue esigenze.
Beta test	Condotti da alcuni clienti sul prodotto quasi completo.

Alle fasi di test corrispondono le versioni del prodotto: la versione **alfa** è integrata ma con difetti e parti mancanti, la **beta** include tutte le feature ma ha ancora difetti, la **gold** è quella definitiva. Il quadro complessivo collega i documenti: dai requisiti (R) si ottiene il documento di specifica dei requisiti (SRS); dall'SRS il design del sistema (DS); dal DS il design dettagliato (DD); dal DD codice e test di unità; da DS e DD l'integrazione e i test di sistema; da R e SRS i test di accettazione.

Il processo a cascata

Il primo modello di processo, degli anni Settanta (Royce, 1970), deriva dalle altre ingegnerie. Le fasi si susseguono in ordine e **ciascuna comincia solo quando la precedente è completa**: prima si specifica tutto, poi si progetta tutto, poi si implementa tutto, poi si testa tutto. È focalizzato sul prodotto completo ed è un processo statico con molta documentazione.

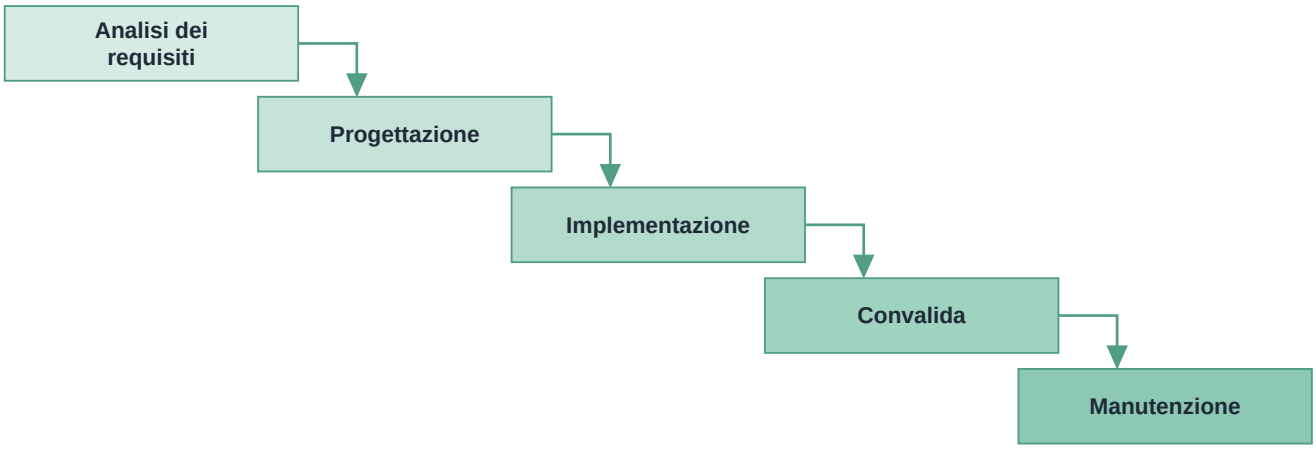


Figura 18.1: il processo a cascata; ogni fase parte quando la precedente è conclusa.

VANTAGGI	SVANTAGGI
Coerenza fra gli artefatti e ampia documentazione.	Tempi lunghi prima di avere un prodotto.
Utile se i requisiti sono stabili e ben definiti.	Poche interazioni con il cliente, solo all'inizio.
Adatto a sistemi grandi, complessi e critici e a team numerosi; codice di alta qualità.	Difficile introdurre i cambiamenti richiesti dal cliente.

Evoluzione e processo evolutivo

Il software è intrinsecamente flessibile, e quando i requisiti cambiano, perché cambia il mercato, l'hardware o l'ambiente, deve evolvere se vuole restare utile. Il **processo evolutivo** asseconda questa natura e ha due varianti. Nello **sviluppo per esplorazione** gli sviluppatori lavorano con i clienti e, partendo da specifiche iniziali chiare, arrivano al sistema finale per trasformazioni successive, aggiungendo le caratteristiche chieste dal cliente. Nel **build and fix** si costruisce una prima versione e la si modifica finché il cliente è soddisfatto: la documentazione è quasi assente, la progettazione pure, la comprensione del sistema è limitata e il codice di bassa qualità.

I problemi del processo evolutivo sono tempi lunghi, sistemi difficili da capire e modificare e probabilmente non corretti, e la mancanza di una visione d'insieme. È applicabile a sistemi piccoli, a singole parti di sistemi grandi (come l'interfaccia utente) e a sistemi di vita breve, come i prototipi.

Altri processi

Nel **processo incrementale** si implementano prima le funzionalità di base o prioritarie, poi si aggiunge codice per un altro gruppo di funzionalità, e si ripete fino al completamento. Nel processo **CBSE** (Component Based Software Engineering), basato su **COTS** (Components Off The Shelf, componenti già pronti), si analizzano i componenti esistenti, si adattano se possibile i requisiti, si progetta riusando i componenti e si sviluppa integrandoli. Nei processi **agili**, che vedremo nei prossimi capitoli, la prima versione arriva in poco tempo, c'è molta interazione con il cliente e ogni fase produce codice testato e integrato.

Il processo a spirale

Il processo a spirale (Boehm, 1988) è focalizzato su tanti prodotti parziali e soprattutto sul **rischio**. Ogni giro della spirale è una fase del progetto, per esempio la fase dei requisiti, e attraversa quattro settori.

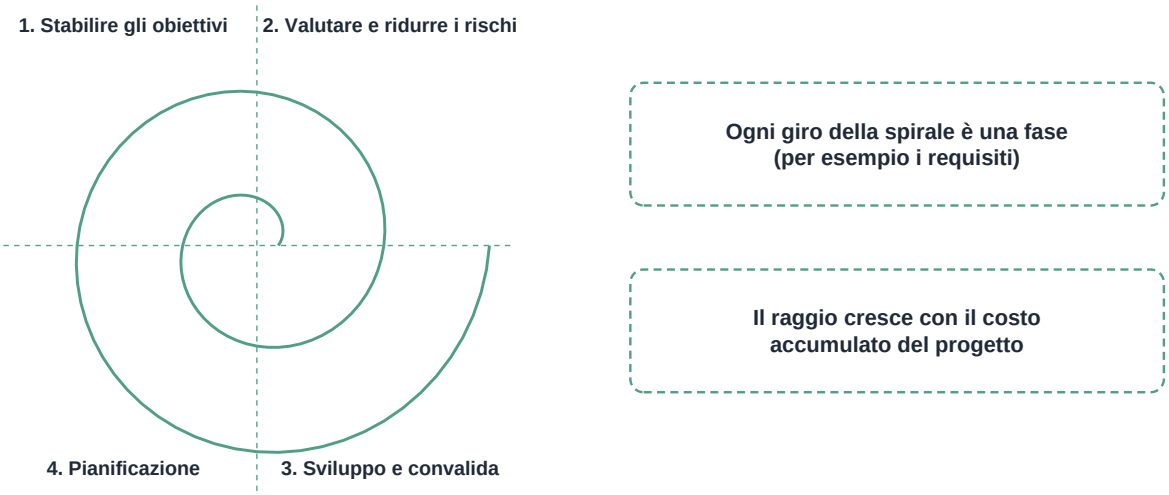


Figura 18.2: il processo a spirale e i suoi quattro settori.

1. **Stabilire gli obiettivi** specifici della fase corrente.
2. **Valutare e ridurre i rischi**: si valutano i rischi del progetto e si intraprendono attività per ridurre quelli più importanti.
3. **Sviluppo e convalida**: si produce una parte del sistema e la si convalida, seguendo uno dei modelli precedenti.
4. **Pianificazione**: si revisiona il progetto e si pianifica il giro successivo.

DEFINIZIONE

Un **rischio** è qualcosa, non ancora noto con certezza, che può impedire il successo del progetto, dove successo significa soddisfare tutti i requisiti. Un rischio ha due attributi: la **probabilità** che si verifichi e l'**impatto** sul progetto, cioè la gravità del danno peggiore.

DA RICORDARE

Fasi: requisiti, progettazione, implementazione, convalida, manutenzione. Requisiti **funzionali** (che cosa) e **non funzionali** (come), da scrivere in modo verificabile. Test di unità, di sistema, di accettazione, beta; versioni alfa, beta, gold. **Cascata**: fasi in sequenza, adatta a requisiti stabili. **Evolutivo**: esplorazione o build and fix, per sistemi piccoli o prototipi. **Incrementale**, **CBSE** con COTS. **Spirale**: ogni giro è una fase, guidata dalla gestione dei rischi.

CAPITOLO 19

19. Il processo Extreme Programming

I valori dello sviluppo agile e le dodici pratiche di XP: story card, pianificazione, piccole release, design semplice, CRC card, test, programmazione a coppie, refactoring e le altre.

Lo sviluppo agile

Lo sviluppo agile (Cockburn, 2002) mette al primo posto l'auto organizzazione, la collaborazione e la comunicazione fra i membri del team e la capacità di adattare il prodotto, più che l'ordine e la coerenza delle attività pianificate. I suoi valori sono espressi come preferenze.

SI PRIVILEGIANO	RISPETTO A
gli individui e le interazioni	processi e strumenti
il software funzionante	la documentazione
la collaborazione con il cliente	la negoziazione dei contratti
la risposta ai cambiamenti	l'esecuzione di un piano

Essere agili significa anche accogliere positivamente le richieste di cambiamento, persino in fase avanzata; rilasciare frequentemente software funzionante; costruire i sistemi con persone motivate; prestare attenzione continua all'eccellenza tecnica.

Extreme Programming

XP (Beck, 2000) è un processo agile basato sullo sviluppo e la consegna di **piccoli incrementi** di funzionalità, con circa due settimane per incremento. Le sue caratteristiche: gruppi piccoli, da 2 a 12 sviluppatori; miglioramento costante del codice; poca documentazione, sostituita da story card e CRC card; comunicazione diretta fra le persone; iterazioni brevi e di durata costante; coinvolgimento di sviluppatori, clienti e manager; prodotti testabili e testati fin dall'inizio. È adatto quando i requisiti non sono stabili, perché XP è fortemente adattativo, e quando i rischi sono alti, per esempio con tempi di consegna brevi o un software innovativo per gli stessi sviluppatori.

I principi di XP sono: avere feedback rapidamente, assumere la semplicità, procedere per cambiamenti incrementali, supportare i cambiamenti, produrre lavoro di qualità. Da questi principi discendono dodici pratiche.

PRATICA	PRATICA	PRATICA
Gioco di pianificazione	Piccole release	Metafora
Design semplice	Testing	Refactoring
Programmazione a coppie	Possesso collettivo del codice	Integrazione continua
Settimana di 40 ore	Cliente in sede	Standard di codifica

Story card e gioco di pianificazione

Le **story card** (storie utente) sono casi d'uso leggeri: due o tre frasi su un cartoncino di circa 12 per 7 centimetri. Sono importanti per il cliente e **scritte dal cliente**, si possono testare, permettono di stimare il tempo di sviluppo e si possono ordinare per priorità. Una story card contiene data, numero, priorità, tempo stimato, riferimenti, la descrizione del requisito, la lista dei task che lo sviluppatore dovrà svolgere e delle note. Le card vengono appese a una **story board** divisa per settimane, con le colonne "da fare" e "fatto".

STORY CARD N. 7

Data: 3 ottobre Priorità: alta

Stima: 2 giorni

Come cliente voglio salvare il carrello

per completare l'acquisto più tardi.

Task: tabella carrelli, pulsante, test

Note: valido 30 giorni

CRC CARD

Classe: Carrello

Responsabilità	Collaboratori
aggiunge prodotti	Prodotto
calcola il totale	Listino
si salva	Archivio

Figura 19.1: una story card inventata e una CRC card con responsabilità e collaboratori.

Nel **gioco di pianificazione** gli utenti scrivono le storie, che sono i requisiti; gli sviluppatori stimano il tempo di ciascuna; se una storia è troppo complessa da stimare, si torna dal cliente e la si fa dividere. Gli utenti dividono, uniscono e ordinano per priorità le storie e riempiono tre settimane scegliendole, senza preoccuparsi delle dipendenze. Chi si occupa del business decide date di rilascio e priorità; si pianifica grossolanamente l'intera release e in dettaglio solo l'iterazione corrente, senza guardare troppo avanti. Per la release attuale gli sviluppatori dividono ogni storia in task, li stimano e ciascuno si impegna a realizzarne alcuni, cominciando dai più rischiosi.

Le altre pratiche

PRATICA	IN CHE COSA CONSISTE
Piccole release	Ogni release è il più piccola possibile, circa due settimane, con un design sufficiente solo per essa. Dà feedback rapido, senso di risultato, rischio ridotto, fiducia del cliente e possibilità di aggiustare i requisiti.
Metafora	Un'immagine unica che guida il progetto e rappresenta l'architettura, per esempio "l'interfaccia è una scrivania"; semplifica le discussioni e deve essere chiara anche al cliente.
Design semplice	Il design giusto passa i test, non ha duplicazioni, esprime ogni intenzione importante e ha il minor numero di classi e metodi. Si fa la cosa più semplice che può funzionare; il design si documenta con le CRC card.
Testing	Si testa tutto ciò che può andare male, in modo automatico e più volte al giorno. Unit test scritti dagli sviluppatori, anche prima del codice (TDD); test funzionali scritti dagli utenti, che sono una specifica dei requisiti in forma eseguibile.
Programmazione a coppie	Due programmatori a un computer: uno usa tastiera e mouse e pensa a come implementare il metodo, l'altro si chiede se l'approccio funzionerà, pensa ai test e a soluzioni più semplici. Le coppie ruotano; la pratica aiuta la disciplina e diffonde la conoscenza.
Possesso collettivo	Chiunque può modificare qualunque parte del codice; chi trova un problema lo risolve; tutti sono responsabili dell'intero sistema, protetto dagli unit test.

PRATICA	IN CHE COSA CONSISTE
Integrazione continua	Il codice testato si integra ogni poche ore, al massimo una volta al giorno, e tutti i test devono passare; se un test fallisce la coppia che ha scritto il codice lo ripara, o butta il codice e ricomincia.
Settimana di 40 ore	Le ore di straordinario sono il sintomo di un problema serio; lavorare 40 ore lascia la mente fresca.
Cliente in sede	Il cliente sta con il team: scrive i test funzionali, stabilisce le priorità, risponde alle domande.
Standard di codifica	Tutti usano le stesse convenzioni per indentazione, parentesi, nomi e commenti, così il codice appare uniforme e si legge facilmente.
Refactoring	Si migliora la struttura del codice a piccoli passi senza cambiarne il comportamento, puntando a eliminare le ripetizioni; farlo in coppia dà più coraggio.

DEFINIZIONE

Una **CRC card** (Class, Responsibility, Collaboration) è un cartoncino che descrive una classe: il nome, le sue responsabilità e le classi con cui collabora per svolgerle. Aiuta a ragionare in termini di oggetti e dà una visione complessiva del sistema.

DA RICORDARE

XP si concentra sul codice e sulle persone: si fanno solo le cose che velocizzano la produzione di codice, la conoscenza passa attraverso la comunicazione, il processo è leggero e la qualità viene da test rigorosi. Iterazioni di circa due settimane, **story card** scritte dal cliente, **CRC card** per il design, **TDD**, **programmazione a coppie**, **integrazione continua**, **refactoring**, cliente in sede.

CAPITOLO 20

20. Scrum

Una struttura agile basata su iterazioni di durata fissa: principi, pilastri, ruoli, eventi e artefatti.

Principi di Scrum

La prima guida a Scrum è stata scritta nel 2010 da Schwaber e Sutherland. Scrum è una struttura di base, volutamente semplice e incompleta, che aiuta le persone a trovare soluzioni a problemi complessi; sopra di essa si possono usare processi e tecniche diverse. Si basa sull'**empirismo**, per cui la conoscenza viene dall'esperienza e le decisioni si prendono in base a ciò che si osserva, e sul pensiero **lean** (snello), che riduce gli sprechi e si concentra sull'essenziale. L'approccio è iterativo e incrementale, per ottenere prevedibilità e controllo del rischio.

Il ciclo è questo: il **Product Owner** elenca il lavoro da svolgere in un **Product Backlog**; lo **Scrum Team** trasforma una parte di quel lavoro in un **incremento** durante uno **Sprint**; il team e gli stakeholder, cioè chi ha interesse nel prodotto, valutano i risultati e si adattano per lo Sprint successivo; e si ricomincia.

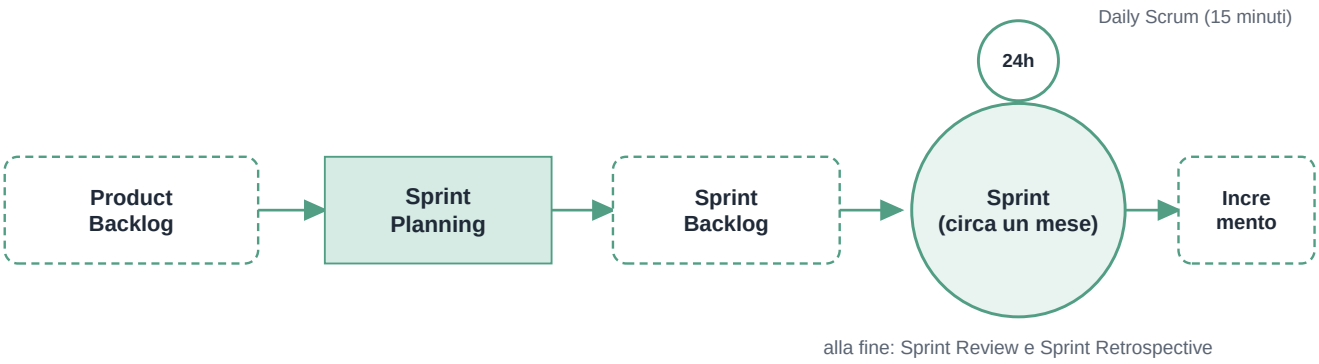


Figura 20.1: il ciclo di Scrum, dal Product Backlog all'incremento.

I tre pilastri

PILASTRO	SIGNIFICATO
Trasparenza	Processo e lavoro sono visibili a chi lavora e a chi riceve i risultati; le decisioni si basano sui tre artefatti (Product Backlog, Sprint Backlog, Incremento). La trasparenza rende possibile l'ispezione.
Ispezione	Artefatti e progressi verso l'obiettivo si controllano spesso e con cura, per scoprire i problemi; Scrum ne dà la cadenza con i suoi eventi. L'ispezione rende possibile l'adattamento.
Adattamento	Se il processo o il prodotto si allontanano da ciò che è accettabile, si correggono subito, per evitare che la divergenza cresca.

Gli eventi

EVENTO	SCOPO
Sprint	L’iterazione, di lunghezza fissa, tipicamente un mese, che contiene tutti gli altri eventi.
Sprint Planning	Pianifica il lavoro da svolgere nello Sprint.
Daily Scrum	Riunione di 15 minuti, ogni giorno alla stessa ora e nello stesso posto: ispeziona i progressi, adatta lo Sprint Backlog, migliora la comunicazione, fa emergere gli ostacoli.
Sprint Review	Penultimo evento: ispeziona i risultati dello Sprint e decide gli adattamenti; dura al massimo 4 ore per uno Sprint di un mese.
Sprint Retrospective	Il team esamina come è andato lo Sprint (persone, interazioni, processi, strumenti, Definition of Done) per migliorare qualità ed efficacia.

DEFINIZIONE

La **Definition of Done** descrive lo stato che l’incremento deve avere quando soddisfa le misure di qualità richieste per il prodotto: finché non la rispetta, il lavoro non è “fatto”.

DA RICORDARE

Scrum: empirismo e lean, approccio iterativo e incrementale. Ruoli: Product Owner e Scrum Team, con gli stakeholder. Artefatti: **Product Backlog**, **Sprint Backlog**, **Incremento**. Pilastri: **trasparenza, ispezione, adattamento**. Eventi: Sprint (circa un mese), Sprint Planning, Daily Scrum (15 minuti), Sprint Review, Sprint Retrospective.

CAPITOLO 21

21. Observer e Model View Controller

Come tenere aggiornati automaticamente gli oggetti che dipendono da un altro, la versione asincrona Publisher Subscriber di Java e l'architettura Model View Controller.

Il design pattern Observer

DEFINIZIONE

Intento: definire una dipendenza **uno a molti** fra oggetti, così che quando un oggetto cambia stato tutti gli oggetti che dipendono da esso siano notificati e aggiornati automaticamente. È noto anche come **Publish Subscribe**.

Gli oggetti chiave sono il **Subject**, l'oggetto osservato, e gli **Observer**, che dipendono da esso. È come l'iscrizione a una newsletter: chi è interessato si iscrive, e quando esce un numero nuovo tutti gli iscritti lo ricevono, senza che l'editore debba sapere chi siano.

MOTIVAZIONE E APPLICABILITÀ

Un sistema diviso in classi che cooperano deve mantenere la coerenza fra oggetti collegati, senza però accoppiarli strettamente. L'esempio tipico separa i dati dalla loro presentazione: un foglio di calcolo e un grafico mostrano gli stessi dati in modi diversi e devono aggiornarsi quando i dati cambiano, ma le classi dei dati e quelle di presentazione devono restare separate e riusabili. Observer si applica quando un'astrazione ha due aspetti dipendenti, da incapsulare in oggetti separati; quando il cambiamento di un oggetto richiede di cambiarne altri, senza sapere quanti; quando un oggetto deve avvisare altri oggetti senza fare ipotesi su chi siano.

SOLUZIONE

RUOLO	RESPONSABILITÀ
Subject	Conosce i suoi osservatori, in numero qualsiasi; offre le operazioni per aggiungerli (attach), toglierli (detach) e notificarli.
Observer	Interfaccia comune agli oggetti da notificare, con l'operazione update().
ConcreteSubject	Eredita da Subject, tiene lo stato che interessa agli osservatori e li notifica quando cambia.
ConcreteObserver	Implementa Observer, tiene un riferimento al ConcreteSubject e uno stato da mantenere coerente con il suo; dopo la notifica può interrogare il subject per ottenere il nuovo dato.

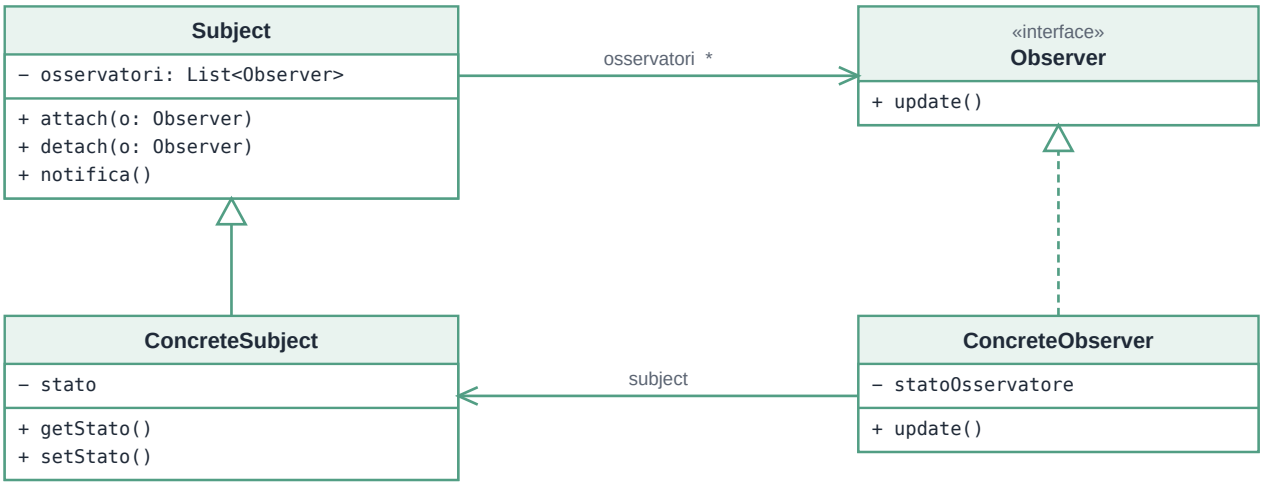


Figura 21.1: la struttura di Observer.

La sequenza è questa: l'observer si registra con `attach()`; quando lo stato del subject cambia, il subject chiama `notifica()`, che chiama `update()` su ogni observer registrato; ogni observer, se gli serve, legge il nuovo stato con `getStato()`. Un flag `cambiato`, impostato da un metodo `setChanged()`, può evitare notifiche inutili quando lo stato non è cambiato davvero.

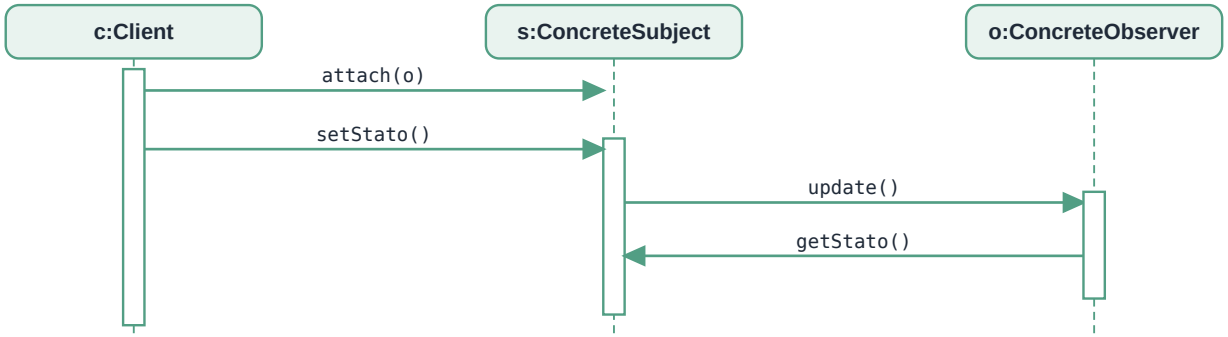


Figura 21.2: registrazione, cambiamento di stato, notifica e lettura del nuovo stato.

Un esempio: una rubrica è il subject; un componente che salva su file e uno che aggiorna l'elenco a schermo sono observer. Quando si inserisce un contatto, la rubrica notifica entrambi, e ciascuno fa il suo lavoro senza che la rubrica sappia quale.

CONSEGUENZE E AVVERTENZE

- Il subject conosce solo l'interfaccia Observer, non le classi concrete: subject e observer concreti non sono accoppiati e sono più facili da riusare e modificare.
- La notifica arriva a tutti gli iscritti, quanti che siano, e gli observer si possono aggiungere o rimuovere in ogni momento; ciascuno sceglie se gestire o ignorare la notifica.
- Un aggiornamento può scatenare a cascata molte operazioni; la notifica di base non dice che cosa è cambiato, segnala solo che un'operazione del subject si è conclusa.
- Nel modello **push** il subject passa le informazioni come argomenti di update(); nel modello **pull** l'observer le legge dopo la notifica. Un observer che osserva più subject riceve il riferimento al mittente come parametro di update().
- Gli observer possono registrarsi solo per certi eventi; quando un subject viene eliminato, gli observer vanno avvisati perché cancellino il riferimento.

Observer nelle librerie Java

Il problema è così comune che `java.util` offriva la classe `Observable`, nel ruolo di Subject con il flag impostato da `setChanged()` e il metodo `notifyObservers()`, e l'interfaccia `Observer` con il solo `update()`. Da Java 9 sono mantenute solo per compatibilità e il loro uso è sconsigliato. Il motivo è che `update()` viene eseguito sul thread del chiamante: il subject resta bloccato finché ogni observer non ha finito.

I **Reactive Streams** risolvono il problema di passare elementi da un produttore a un consumatore senza bloccare il produttore e senza sommergere il consumatore. Java 9 offre in `java.util.concurrent` le interfacce `Publisher`, `Subscriber`, `Subscription` e la classe `SubmissionPublisher`.

TIPO	RUOLO
Publisher<T>	Fornisce gli elementi; il metodo subscribe() registra un Subscriber.
SubmissionPublisher<T>	Implementa Publisher: submit(elemento) consegna l'elemento ai subscriber in modo asincrono, su un thread dedicato, chiamando il loro onNext(); si blocca solo se un subscriber non può ricevere.
Subscriber<T>	Riceve gli elementi con onSubscribe(), onNext(), onError(), onComplete().
Subscription	Il collegamento fra i due: request(n) chiede i prossimi n elementi, cancel() interrompe l'invio. Così è il consumatore a regolare il ritmo (contropressione).

Model View Controller

MVC è considerato un pattern architetturale per le applicazioni interattive. Nasce dal fatto che le interfacce utente cambiano spesso, al cambiare di funzionalità, dispositivi e piattaforme; che le stesse informazioni vengono mostrate in finestre diverse, per esempio come tabella e come grafico; che le visualizzazioni devono aggiornarsi subito quando i dati cambiano; e che il supporto alle diverse visualizzazioni non deve toccare le funzionalità principali.

COMPONENTE	RESPONSABILITÀ
Model	Incapsula dati e funzionalità principali, indipendente da come i dati sono mostrati e da come arrivano gli input; registra View e Controller e li avvisa quando i dati cambiano. È il Subject.
View	Mostra i dati all'utente leggendoli dal Model; ci possono essere più View, ognuna associata a un Controller che inizializza. È un Observer.
Controller	Riceve gli input dell'utente (mouse, tastiera) come eventi e li traduce in richieste di servizio al Model o in avvisi alla View.

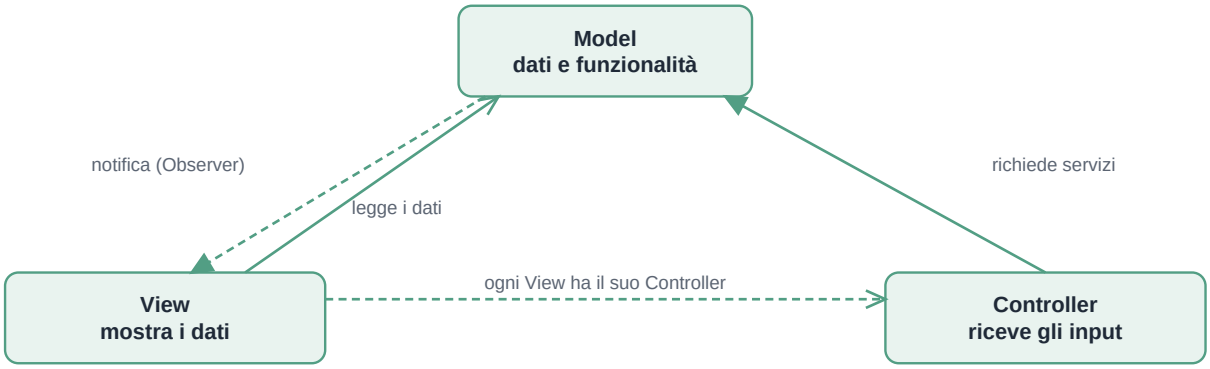


Figura 21.3: Model View Controller; il Model notifica le View come un Subject notifica i suoi Observer.

DA RICORDARE

Observer: dipendenza uno a molti; il Subject tiene la lista degli Observer e su ogni cambiamento chiama update() su ciascuno. Accoppiamento debole, modelli push e pull. In Java moderno: **Publisher**, **Subscriber**, **Subscription** e **SubmissionPublisher**, asincroni. **MVC**: Model (dati, Subject), View (presentazione, Observer), Controller (input).

CAPITOLO 22

22. State

Come far cambiare comportamento a un oggetto quando cambia il suo stato interno, sostituendo i rami condizionali con il polimorfismo.

Il design pattern State

DEFINIZIONE

Intento: permettere a un oggetto di modificare il proprio comportamento quando il suo stato interno cambia. L'oggetto sembrerà aver cambiato classe.

PROBLEMA

Il comportamento di un oggetto dipende dal suo stato e deve cambiare a runtime. Lo stato è spesso rappresentato da una o più variabili con valori costanti enumerati, e le operazioni contengono grandi rami condizionali, `if` o `switch`, che dipendono da quei valori; spesso più operazioni ripetono la stessa struttura condizionale.

SOLUZIONE

L'idea è mettere **ogni ramo condizionale in una classe separata**.

RUOLO	RESPONSABILITÀ
Context	Definisce l'interfaccia che interessa ai client e tiene un'istanza del ConcreteState che rappresenta lo stato corrente.
State	Interfaccia che incapsula il comportamento associato a uno stato del Context.
ConcreteState	Ogni sottoclasse implementa il comportamento di uno stato.

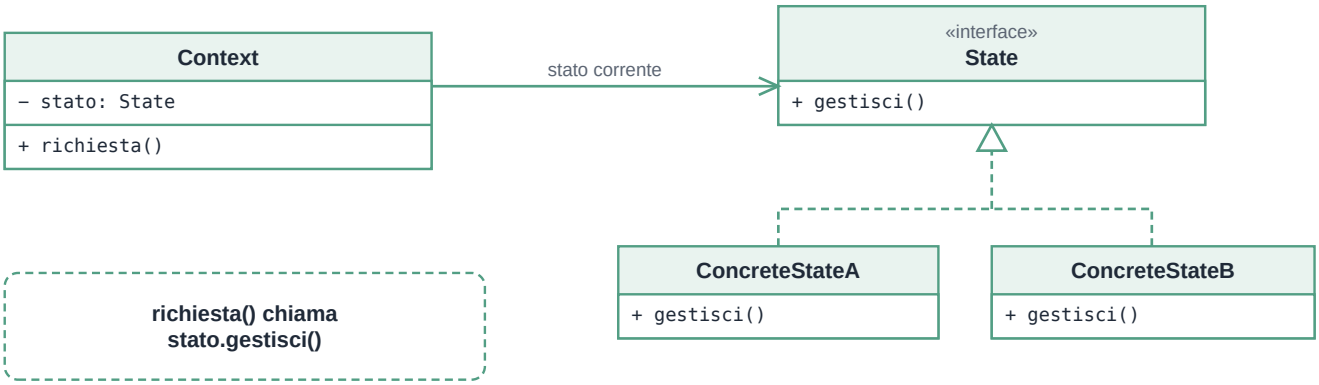


Figura 22.1: la struttura di State; il Context delega allo stato corrente.

Il Context inoltra le richieste dipendenti dallo stato all'oggetto stato corrente, eventualmente passando se stesso come argomento, perché lo stato possa accedervi. La decisione su quale sia lo stato successivo, e in quali circostanze, può spettare al Context oppure agli stessi ConcreteState.

Un esempio: dal codice con switch a State

Requisiti: un sistema permette di prenotare e acquistare un biglietto. Si può annullare la prenotazione ma non l'acquisto. Ogni biglietto ha codice, prezzo, data di acquisto e intestatario, da indicare al momento della prenotazione. Dall'analisi emergono la classe Biglietto, i suoi attributi, le operazioni prenota, compra, cancella e mostra, e tre stati: disponibile, prenotato, venduto.

Figura 22.2: il diagramma degli stati del biglietto.

Una prima implementazione usa un attributo di tipo enumerativo per lo stato, e ogni metodo contiene uno switch con un caso per stato: `prenota()` intesta il biglietto e passa a prenotato se era disponibile, cambia solo il nome se era già prenotato, rifiuta se era venduto; e così via per gli altri metodi. Il codice funziona, ma ogni metodo ha tanti rami quanti sono gli stati, la logica condizionale è ripetuta e difficile da modificare, e i comportamenti dei diversi stati sono mescolati negli stessi metodi.

La presenza ripetuta di switch sullo stesso valore è un sintomo che suggerisce il polimorfismo. Il refactoring **Sostituisci i condizionali con il polimorfismo** (Replace Conditional with Polymorphism), che in questo caso prende il nome di **Sostituisci il codice di tipo con State**, procede così.

1. Si definisce l'interfaccia State, `StatoBiglietto`, con un metodo per ogni operazione: `mostra()`, `intesta(nome)`, `paga()`, `cancella()`. Ognuno restituisce lo stato successivo.
2. Si crea una classe per ogni stato, `Disponibile`, `Prenotato`, `Venduto`, e in ciascun metodo si sposta il ramo dello switch che corrispondeva a quello stato.
3. Ogni metodo restituisce `this` se lo stato non cambia, oppure una nuova istanza dello stato successivo: per esempio `Disponibile.intesta()` restituisce un nuovo `Prenotato`.
4. Il Context `Biglietto` tiene l'attributo `stato` e ogni suo metodo diventa una riga: `stato = stato.intesta(nome);`.

Un'interfaccia i cui metodi restituiscono l'oggetto con cui proseguire si dice **fluente** (fluent). Il risultato: nessuna istruzione condizionale, metodi di due o tre righe, ogni stato in una classe che si può capire e modificare da sola, e ciascuno stato che avvia le proprie transizioni.

CONSEGUENZE

Il comportamento di ogni stato è localizzato in una sola classe, e aggiungere stati e transizioni significa aggiungere sottoclassi; incapsulare le azioni di uno stato impone una struttura e chiarisce lo scopo del codice; la logica dei cambi di stato non è sparsa in tanti if, e questo aiuta a evitare stati incoerenti. Le classi sono di più, ma ciascuna è più semplice.

DA RICORDARE

State: ogni stato diventa una classe che implementa l'interfaccia State; il Context delega allo stato corrente e, con un'interfaccia fluente, lo sostituisce con quello restituito. Elimina gli switch ripetuti (Replace Conditional with Polymorphism). Ruoli: Context, State, ConcreteState.

CAPITOLO 23

23. Mediator

Come evitare che molti oggetti si conoscano tutti fra loro, centralizzando le loro interazioni in un mediatore.

Il design pattern Mediator

DEFINIZIONE

Intento: definire un oggetto che incapsula il modo in cui un gruppo di oggetti interagisce. Il Mediator promuove l'**accoppiamento debole**, perché evita che gli oggetti interagiscano direttamente, e permette di modificare le loro interazioni indipendentemente da essi.

MOTIVAZIONE

Distribuire le responsabilità fra molti oggetti può creare moltissime connessioni: nel caso peggiore ogni oggetto conosce tutti gli altri. Il sistema diventa di fatto monolitico, e cambiarne il comportamento è difficile perché è sparso fra gli oggetti. Pensiamo a una finestra di ricerca con un campo di testo, due elenchi di categorie e i pulsanti Cerca e Annulla: ogni elemento deve informare gli altri dei suoi cambiamenti, e senza un mediatore ciascuna classe chiamerebbe i metodi di tutte le altre. Come in un aeroporto, dove gli aerei non parlano fra loro ma tutti con la torre di controllo, gli oggetti comunicano solo con un mediatore.

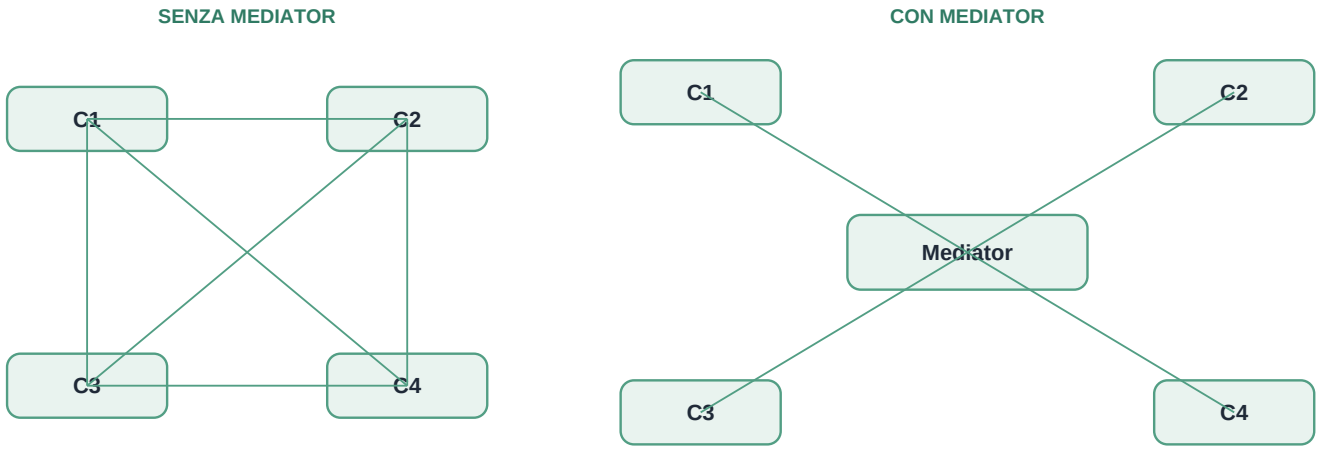


Figura 23.1: con quattro oggetti servono sei collegamenti diretti; con il Mediator ne bastano quattro.

Con n oggetti che si conoscono tutti, i collegamenti sono n per (n meno 1) diviso 2: con 4 oggetti 4 per 3 diviso 2, cioè 6; con 10 oggetti già 45. Con il mediatore sono solo n.

APPLICABILITÀ E SOLUZIONE

Mediator si usa quando un insieme di oggetti comunica in modo ben definito ma complesso, con interdipendenze non strutturate; quando riusare un oggetto è difficile perché comunica con troppi altri; quando un comportamento distribuito fra molte classi deve poter cambiare senza creare sottoclassi.

RUOLO	RESPONSABILITÀ
Mediator	Interfaccia, il punto d’incontro, per comunicare con gli oggetti Colleague.
ConcreteMediator	Implementa il comportamento cooperativo e coordina i Colleague, inoltrando le richieste a quelli giusti.
Colleague	Ogni Colleague conosce il Mediator e comunica con lui ogni volta che avrebbe comunicato con un altro Colleague.

Un esempio: un componente legge dati dalla tastiera e li consegna a un mediatore `Centrale`, che li smista, in base al tipo, ai componenti interessati: uno che raccoglie i numeri, uno che li somma, uno che raccoglie i nomi. Quando un componente riconosce una condizione di arresto chiama `stop()` sul mediatore, che avvisa tutti gli altri. Nessun componente conosce gli altri.

CONSEGUENZE

- La maggior parte della complessità delle dipendenze si sposta dagli oggetti al Mediator, e gli oggetti diventano più facili da implementare e mantenere.
- I Colleague sono più riusabili, perché la loro funzione principale non è mescolata con la gestione delle dipendenze.
- Il codice del Mediator invece in genere non è riusabile, perché è specifico di un’applicazione, e rischia di diventare una classe molto complessa.

DA RICORDARE

Mediator centralizza le interazioni: i Colleague parlano solo con il Mediator, che coordina. Riduce i collegamenti da circa n al quadrato a n e rende i Colleague riusabili; il Mediator invece è specifico e può diventare complesso.

CAPITOLO 24

24. Chain of Responsibility

Come passare una richiesta lungo una catena di oggetti finché uno di essi la gestisce, senza che il mittente sappia quale.

Il design pattern Chain of Responsibility

DEFINIZIONE

Intento: evitare di accoppiare il mittente di una richiesta al suo destinatario, dando a più oggetti la possibilità di gestirla. Gli oggetti riceventi sono concatenati e la richiesta passa lungo la catena finché uno la gestisce.

Pensiamo all'aiuto contestuale di un'interfaccia grafica: l'utente chiede aiuto su un bottone; se il bottone ha un messaggio specifico lo mostra, altrimenti va mostrato quello del contesto più vicino, per esempio la finestra, e se neanche questa lo ha, quello dell'applicazione. Il problema è che l'oggetto che fornirà l'aiuto non è noto a chi fa la richiesta. È come un reclamo in un'azienda: l'operatore risponde se può, altrimenti passa al responsabile, che a sua volta può passare al direttore.

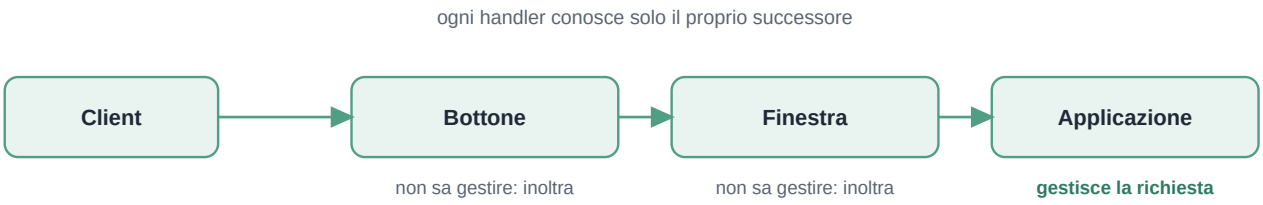


Figura 24.1: la richiesta di aiuto percorre la catena finché un oggetto la gestisce.

PARTECIPANTI

RUOLO	RESPONSABILITÀ
Handler	Definisce l'interfaccia per gestire le richieste ed eventualmente il collegamento al successore.
ConcreteHandler	Gestisce le richieste di cui è responsabile; può accedere al successore e gli inoltra la richiesta se non sa gestirla.
Client	Invia la richiesta al primo ConcreteHandler della catena.

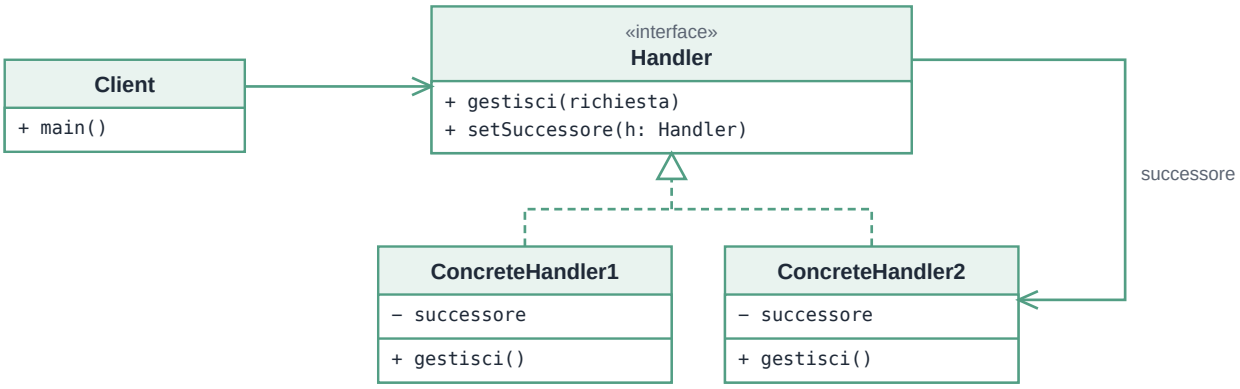


Figura 24.2: la struttura di Chain of Responsibility.

CONSEGUENZE E IMPLEMENTAZIONE

- Riduce l'accoppiamento: il mittente non conosce il destinatario e nessun oggetto deve conoscere la struttura della catena; ogni handler tiene solo il riferimento al successore, non una lista.
- Dà flessibilità: si cambiano o aggiungono responsabilità modificando la catena a runtime.
- Non c'è garanzia che la richiesta venga gestita: potrebbe arrivare in fondo alla catena senza che nessuno se ne occupi.
- I collegamenti al successore si definiscono in Handler o nei ConcreteHandler; si può anche riusare una gerarchia già esistente, come i collegamenti figlio e padre di un Composite.
- Ogni tipo di richiesta può essere un'operazione distinta dell'Handler, oppure un codice passato a un'unica operazione: in questo caso l'insieme delle richieste non è fissato, ma servono più controlli a runtime.

DA RICORDARE

Chain of Responsibility: la richiesta scorre lungo una catena di handler, ognuno la gestisce o la passa al successore. Mittente e destinatario disaccoppiati, catena modificabile a runtime, ma nessuna garanzia di gestione.

CAPITOLO 25

25. Command

Come trasformare una richiesta in un oggetto, per poterla memorizzare, accodare, registrare e annullare.

Il design pattern Command

DEFINIZIONE

Intento: incapsulare una richiesta in un oggetto, permettendo di parametrizzare i client con richieste diverse, di accodare o registrare le richieste e di supportare l'annullamento delle operazioni.

A volte bisogna inviare una richiesta senza sapere nulla dell'operazione richiesta né di chi la riceverà: un bottone di un menù non sa che cosa farà il comando a cui è collegato. Se la richiesta diventa un oggetto, la si può memorizzare e passare come qualunque altro oggetto. È come la comanda di un ristorante: il cameriere non cucina, scrive l'ordine su un foglio e lo passa in cucina, dove il cuoco sa come eseguirlo; il foglio si può mettere in coda o annullare.

PARTECIPANTI

RUOLO	RESPONSABILITÀ
Command	Interfaccia per eseguire un'operazione, con execute() ed eventualmente unexecute().
ConcreteCommand	Collega un Receiver a un'azione: execute() chiama le operazioni corrispondenti sul Receiver.
Client	Crea il ConcreteCommand e ne imposta il Receiver.
Invoker	Tiene il comando e gli chiede di eseguire la richiesta, per esempio quando si preme un bottone.
Receiver	Sa come svolgere le operazioni della richiesta; qualunque classe può esserlo.

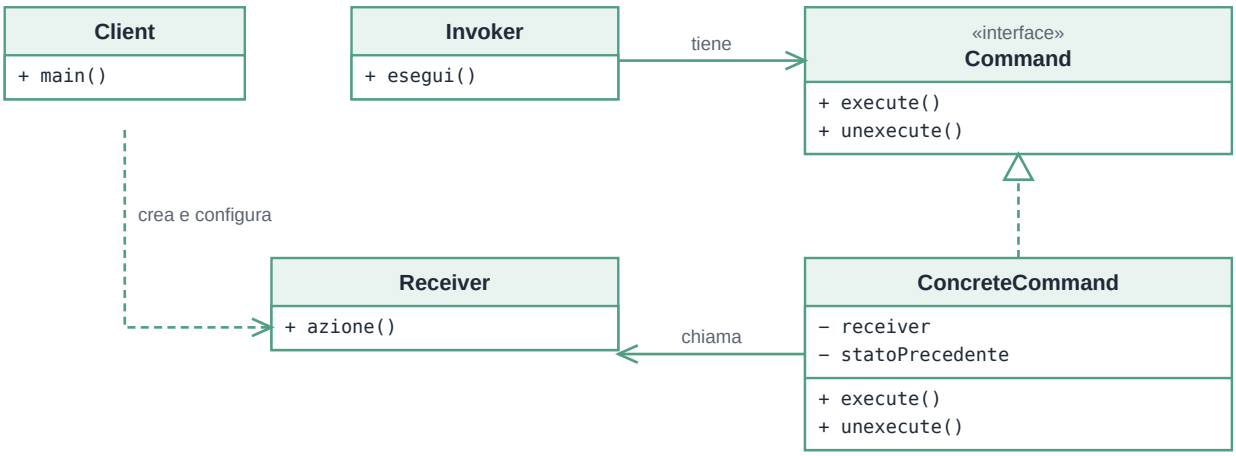


Figura 25.1: la struttura di Command.

La collaborazione: il client crea il comando e gli associa il receiver; l'invoker conserva il comando; quando serve, l'invoker chiama `execute()`, e il comando chiama le operazioni del receiver. Se il comando è reversibile, prima di

eseguire conserva lo stato necessario per tornare indietro con `unexecute()`. Un esempio: un calendario (Receiver) con note da aggiungere o togliere tramite un menù (Invoker); il comando `AggiungiNota` ricorda quale nota ha aggiunto, così il suo `unexecute()` la può togliere.

CONSEGUENZE E IMPLEMENTAZIONE

- Disaccoppia l'oggetto che invoca l'operazione da quello che sa come eseguirla.
- I comandi sono oggetti: si possono manipolare, estendere, accodare e registrare in un log.
- Si possono assemblare in un comando composto (MacroCommand), che è un'istanza del pattern Composite.
- Aggiungere nuovi comandi è facile, perché non si modificano le classi esistenti.
- Un comando può limitarsi a legare receiver e azione, oppure fare tutto da sé senza receiver.
- Per un solo livello di annullamento basta ricordare l'ultimo comando eseguito; per più livelli di annulla e ripristina (undo e redo) serve una lista dei comandi eseguiti, e il receiver deve offrire le operazioni per tornare allo stato precedente.

DA RICORDARE

Command: la richiesta diventa un oggetto con execute() e unexecute(). Ruoli: Command, ConcreteCommand, Client, Invoker, Receiver. Permette code, log, undo e redo, comandi composti (con Composite).

CAPITOLO 26

26. Progettare con i design pattern

Le cause più comuni di riprogettazione e i pattern che le prevengono.

Progettare per preservare

Una buona progettazione anticipa i cambiamenti; i design pattern aiutano a evitare le cause più comuni di riprogettazione.

CAUSA DI RIPROGETTAZIONE	PERCHÉ È UN PROBLEMA	PATTERN UTILI
Creare oggetti nominando la classe	Lega il codice a un'implementazione invece che a un'interfaccia.	Factory Method, Abstract Factory, Prototype
Dipendere da operazioni specifiche	Fissa un solo modo di soddisfare una richiesta.	Chain of Responsibility, Command
Dipendere da una piattaforma	Le API specifiche ostacolano porting e aggiornamenti.	Abstract Factory, Bridge
Dipendere da rappresentazioni	Il client cambia quando cambia l'oggetto; meglio nascondere le informazioni.	Abstract Factory, Bridge, Memento, Proxy
Dipendere da algoritmi	Gli algoritmi cambiano spesso e vanno isolati.	Builder, Iterator, Strategy, State, Template Method, Visitor
Accoppiamento stretto	Il sistema diventa monolitico, difficile da modificare e riusare.	Abstract Factory, Bridge, Chain, Command, Facade, Mediator, Observer
Estendere solo con sottoclassi	Serve conoscere a fondo la superclasse; le classi si moltiplicano.	Bridge, Chain, Composite, Decorator, Observer, Strategy
Non poter modificare le classi	Sorgente non disponibile o modifiche a cascata.	Adapter, Decorator, Visitor

Composizione invece di ereditarietà

Estendere con le sottoclassi è un riuso **a scatola bianca** (white box): bisogna conoscere a fondo la superclasse e ridefinire un'operazione può costringere a ridefinirne altre. La **composizione** di oggetti è invece un riuso **a scatola nera** (black box): si usa un oggetto solo attraverso la sua interfaccia: conviene limitare i livelli delle gerarchie e preferire la composizione.

Fra i pattern citati c'è **Strategy**: incapsula algoritmi intercambiabili dietro un'interfaccia comune, scelti dal client; ha la struttura di State, ma l'algoritmo non cambia con lo stato.

DA RICORDARE

Le cause di riprogettazione sono **dipendenze** (da classi concrete, operazioni, piattaforme, rappresentazioni, algoritmi) e **accoppiamento stretto**. Preferire la **composizione** (scatola nera) all'**ereditarietà** (scatola bianca).

CAPITOLO 27

27. Evoluzione del software e metriche

Che cosa succede al software dopo la consegna, le leggi di Lehman sull'evoluzione e le metriche per misurare complessità, dimensione e qualità della progettazione a oggetti.

Manutenzione ed evoluzione

DEFINIZIONE

La **manutenzione** è il processo di modifica di un prodotto software dopo la consegna al cliente. Il termine **evoluzione** è spesso usato come sinonimo, oppure indica un singolo passo di manutenzione: modifica, rilascio di una correzione (patch), rimozione di funzioni.

I costi di manutenzione rappresentano dal 67 all'80 per cento del costo complessivo del software. I cambiamenti si raggruppano in quattro categorie.

CATEGORIA	SCOPO	QUOTA
Correttivi	Rimuovere errori.	17 per cento
Adattativi	Adattare il sistema a un nuovo ambiente.	18 per cento
Perfettivi	Migliorare e aggiungere funzionalità.	60 per cento
Preventivi	Modifiche interne per prevenire problemi futuri.	5 per cento

La parte maggiore è quindi l'aggiunta di funzionalità: per questo è buona pratica anticipare i cambiamenti già in progettazione, con parametrizzazione e incapsulamento.

Le leggi di Lehman

Le dinamiche di evoluzione sono i processi di cambiamento di un sistema. Lehman e Belady le hanno studiate empiricamente a partire dal 1968 su grandi sistemi sviluppati da grandi aziende, e studi recenti le hanno confermate.

LEGGE	ENUNCIATO
Cambiamento continuo (1974)	Un sistema deve essere adattato continuamente, altrimenti diventa progressivamente meno soddisfacente.
Aumento della complessità (1974)	Quando un sistema evolve la sua struttura diventa più complessa, a meno che si lavori per preservarla o semplificarla.
Autoregolazione (1974)	Attributi come dimensione, intervallo fra le release e numero di errori per release sono approssimativamente invarianti.
Stabilità organizzativa (1978)	Durante la vita di un sistema il tasso di sviluppo è circa costante e indipendente dalle risorse impiegate.
Conservazione della familiarità (1978)	In media la crescita fra una release e l'altra resta costante o diminuisce.

LEGGE	ENUNCIATO
Crescita continua (1978)	Le funzionalità devono crescere continuamente per mantenere la soddisfazione degli utenti.
Diminuzione della qualità (1994)	La qualità diminuisce se non è gestita e adattata con rigore durante i cambiamenti.

Valgono soprattutto per grandi sistemi di grandi organizzazioni; per prodotti piccoli o basati su COTS non è chiaro.

Costi e modelli di manutenzione

Il costo si riduce se il team di sviluppo partecipa alla manutenzione, ma spesso gli sviluppatori non ne sono responsabili e non progettano pensando ai cambiamenti; intanto la struttura si degrada a ogni modifica. I modelli di manutenzione sono il **quick fix** (cambiamenti diretti sul codice), il **miglioramento iterativo** (cambiamenti basati sull'analisi del sistema, che preservano il design) e il **riuso** (nuovo sistema che riusa il più possibile).

Le modifiche possono essere di tre tipi: il **refactoring** (o restructuring) cambia la struttura interna senza alterare il comportamento; il **reverse engineering** analizza un sistema per estrarne informazioni su comportamento e struttura; il **reengineering** lo ricostruisce in un'altra forma.

Le metriche

Una **metrica** definisce un insieme di misure per un sistema software. Le metriche servono a monitorare il prodotto mentre si costruisce, a stabilire livelli accettabili per ogni misura e a intervenire quando non sono rispettati. Solo gli attributi **interni**, come la dimensione, si misurano direttamente; quelli **esterni**, come la manutenibilità, si ricavano indirettamente.

METRICHE TRADIZIONALI

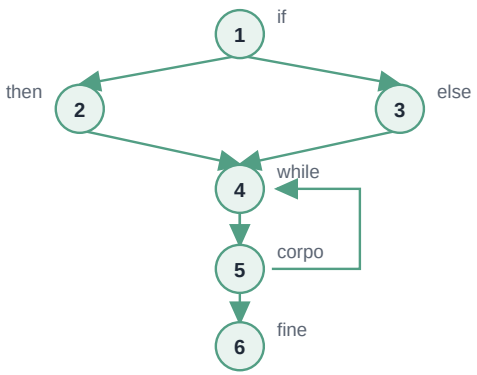
La **dimensione** si misura in LOC (lines of code) oppure in NCNB (non comment non blank, righe che non sono né commenti né vuote). La **percentuale di commenti** rispetto alle LOC è consigliata intorno al 30 per cento. La **complessità ciclomatica** valuta la complessità di un algoritmo.

DEFINIZIONE

La **complessità ciclomatica** (cc) è il numero di percorsi indipendenti nel grafo di flusso di un programma, e quindi il numero di test necessari per esercitarli tutti. Una sequenza di istruzioni richiede un test, una condizione due.

$$cc = archi - nodi + 2$$

Esempio: un metodo con un `if else` seguito da un `while`.



nodi N = 6
archi E = 7
cc = E − N + 2
cc = 7 − 6 + 2 = 3
servono 3 test per coprire
i percorsi indipendenti

Figura 27.1: il grafo di flusso del metodo e il calcolo della complessità ciclomatica.

1. Si contano i nodi: if, then, else, while, corpo del ciclo, fine. Sono 6.
2. Si contano gli archi: da if a then e a else (2), da then e da else al while (2), dal while al corpo e alla fine (2), dal corpo di nuovo al while (1). Sono 7.
3. Si applica la formula: 7 meno 6 più 2 fa 3. Servono 3 test: uno per il ramo then, uno per il ramo else, uno che entra nel ciclo.

METRICHE CK PER SISTEMI A OGGETTI

La suite di Chidamber e Kemerer (CK) contiene metriche pensate per i sistemi a oggetti.

METRICA	DEFINIZIONE	INTERPRETAZIONE
WMC, Weighted Methods per Class	Somma delle complessità dei metodi della classe; con metodi di pari complessità è il numero di metodi.	Alto: molto lavoro di manutenzione, classe molto specifica e poco riusabile.
DIT, Depth of Inheritance Tree	Massimo numero di livelli fra la classe e la radice della gerarchia (la radice vale 0).	Alto: comportamento difficile da capire per i tanti metodi ereditati, sistema più complesso, ma più riuso.
NOC, Number Of Children	Numero di sottoclassi dirette della classe.	Alto: grande influenza della classe sul sistema e molto riuso.
CBO, Coupling Between Object classes	Numero di classi con cui la classe interagisce.	Alto: forte dipendenza, poco riuso, classe difficile da capire, modificare e correggere.
RFC, Response For a Class	Numero di metodi che possono essere eseguiti in risposta a un messaggio: i metodi della classe più quelli che essi invocano.	Alto: grande complessità, difficoltà di comprensione e di test.
LCOM, Lack of Cohesion Of Methods	Mancanza di coesione: misura quanto i metodi usano attributi diversi (vedi sotto).	Basso, cioè alta coesione: semplicità, singola responsabilità, riuso.

Per LCOM si contano *a*, il numero di attributi, *m*, il numero di metodi, e per ogni attributo A_i il numero $m(A_i)$ di metodi che lo usano. Il rapporto fra la somma degli $m(A_i)$ e il prodotto *a* per *m* misura la coesione, e LCOM ne è il complemento a 1.

$$LCOM = 1 - (m(A_1) + ... + m(A_a)) / (a \times m)$$

Con 3 attributi e 3 metodi: se ogni metodo usa tutti e 3 gli attributi, ogni attributo è usato da 3 metodi, la somma è 3 più 3 più 3 uguale a 9, il rapporto è 9 diviso 9 uguale a 1 e LCOM vale 0: coesione massima. Se ogni metodo usa 2 attributi, la somma è 6, il rapporto 6 diviso 9 cioè 2/3, e LCOM vale 1/3. Se ogni metodo usa un solo attributo, diverso dagli altri, la somma è 3, il rapporto 1/3 e LCOM vale 2/3: la classe sta facendo tre cose separate e andrebbe probabilmente divisa.

DA RICORDARE

La manutenzione costa dal 67 all'80 per cento; le modifiche sono correttive, adattative, perfettive (la maggioranza) e preventive. **Leggi di Lehman**: cambiamento continuo, aumento della complessità, crescita continua, diminuzione della qualità, e le altre. Metriche: LOC, commenti, **cc = archi − nodi + 2**; CK: **WMC, DIT, NOC, CBO, RFC, LCOM**.

CAPITOLO 28

28. Verifica, validazione e test

Come si progettano e si implementano i test: tipi di difetti, strategie black box e white box, classi di equivalenza, test dei percorsi, copertura, test di stress e di regressione.

Difetti, verifica e validazione

Il software con difetti è un grande problema, e i difetti sono comuni. Ci accorgiamo di un difetto perché sappiamo, grazie a qualcosa che non è il codice, cioè una **specifica**, che cosa il programma dovrebbe fare: un comportamento anomalo è il modo in cui il software ci comunica i suoi difetti, che non devono passare inosservati.

	DOMANDA	SIGNIFICATO
Verifica	Stiamo costruendo il prodotto nel modo giusto?	Il sistema è conforme alle sue specifiche.
Validazione	Stiamo costruendo il prodotto giusto?	Il sistema fa ciò che l'utente ha realmente richiesto.

Il processo di V e V si applica in ogni fase dello sviluppo e ha due obiettivi: scoprire i difetti e valutare se il sistema è usabile in una situazione operativa. I difetti si classificano in base alla fase in cui nascono: **di specifica**, quando la descrizione del prodotto è ambigua, contraddittoria o imprecisa; **di progettazione**, negli algoritmi (una divisione per zero), nelle strutture dati (un campo mancante o di tipo sbagliato), nelle interfacce fra moduli (parametri incoerenti); **di codice**, per scarsa comprensione del progetto o del linguaggio (overflow, conversioni di tipo, precedenza degli operatori, variabili non inizializzate); **di test**, nei casi e nei piani di test stessi. Non sempre è facile distinguere un difetto di progettazione da uno di codice.

Test e debugging

DEFINIZIONE

Il **test** ha l'obiettivo di rivelare l'esistenza di difetti. Può mostrare la **presenza** di errori, mai la loro **assenza**; per questo un test **ha successo** quando scopre uno o più errori. Il **debugging** invece localizza e corregge gli errori, formulando ipotesi sul comportamento del programma e verificandole.

TERMINE	SIGNIFICATO
Dati di test	Input scelti per testare il sistema.
Caso di test	Dati di input e output attesi se il sistema funziona secondo le specifiche; gli input comprendono anche file, eccezioni e stato del sistema necessari per eseguire il test.
Test suite	Insieme di casi di test.

Solo un test **esaustivo** potrebbe dimostrare che un programma non ha difetti, ma è impraticabile. Facciamo il conto per una funzione con due parametri int.

1. Ogni int può assumere 2^{32} valori, quindi le coppie possibili sono 2^{32} per 2^{32}, cioè 2^{64}, circa 1,8 per 10^{19}.
2. Se ogni esecuzione dura un nanosecondo, cioè 10^{-9} secondi, servono 1,8 per 10^{19} per 10^{-9}, cioè 1,8 per 10^{10} secondi.
3. Un anno dura circa 3 per 10^{7} secondi; 1,8 per 10^{10} diviso 3 per 10^{7} fa circa 600 anni.

Bisogna quindi scegliere bene i test. Le priorità: i test devono mostrare le capacità del software più che eseguire singoli componenti; testare le vecchie funzionalità è più importante che testare le nuove; testare le situazioni tipiche è più importante che testare quelle limite. Quando l'obiettivo è trovare difetti, un buon caso di test è quello con buona probabilità di rivelarne di non noti; deve contenere i risultati attesi e va sviluppato sia per input validi sia per input non validi. Infine, i difetti tendono a raggrupparsi: la probabilità che un componente ne contenga altri è proporzionale a quelli già trovati, e un codice molto complesso indica una progettazione scadente.

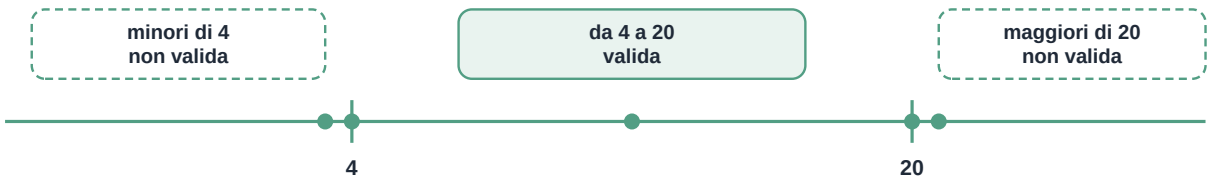
Strategie black box e white box

Nel test **black box** (a scatola nera) il sistema si testa senza conoscerne la struttura né il codice: i casi di test si progettano a partire dalle specifiche, e quindi si possono preparare già nelle prime fasi dello sviluppo. Nel test **white box** (a scatola trasparente, o strutturale) ci si concentra sulla struttura del software, ed è necessario avere il codice sorgente o uno pseudocodice. I due approcci si usano insieme.

PARTIZIONAMENTO IN CLASSI DI EQUIVALENZA

Nel black box, un buon modo per scegliere gli input è dividerli in **classi di equivalenza**: gruppi di valori per cui ci si aspetta che il programma si comporti allo stesso modo. Testare un valore di una classe equivale a testarli tutti, e così un piccolo insieme di valori copre un dominio enorme. Si scelgono valori da ogni classe, comprese le classi **non valide**, che rappresentano input inattesi o errati, e i valori ai **confini** fra le classi, dove gli errori sono più frequenti.

Esempio: una funzione accetta solo numeri da 4 a 20. Le classi sono tre: minori di 4, da 4 a 20, maggiori di 20.



valori di test: 3, 4, 12, 20, 21 (i valori ai bordi sono i test di confine)

Figura 28.1: le tre classi di equivalenza e i valori di test scelti, compresi quelli di confine.

TEST DEI PERCORSI

Nel white box, il test dei percorsi vuole che ogni percorso indipendente del programma sia eseguito almeno una volta. Si rappresenta il programma con un grafo di flusso e si calcola la complessità ciclomatica, che indica il numero di test necessari, come visto nel capitolo precedente. Così si eseguono tutti i percorsi, ma non tutte le loro combinazioni.

Copertura del codice

Fino a quando continuare a testare? Un criterio è la **copertura**: si divide il programma in unità (costrutti, condizioni, comandi), si fissa la copertura che la suite deve raggiungere, per esempio il 60 per cento, e ci si ferma quando la si raggiunge. La copertura è il numero di unità eseguite diviso il numero di unità del programma. Le parti critiche possono richiedere una copertura maggiore, e la misura serve anche a scoprire che cosa manca alla suite. Una copertura del 100 per cento però non garantisce l'assenza di difetti, perché dipende dai dati scelti.

CRITERIO	PERCENTUALE DI
Funzioni	funzioni eseguite.
Costrutti (statement)	istruzioni eseguite.
Archi	archi del grafo di flusso di controllo percorsi.
Rami	rami condizionali eseguiti, per esempio sia il blocco if sia il blocco else.
Condizioni	sottoespressioni booleane valutate sia vere sia false; in `if (x && y)` sia x sia y devono assumere entrambi i valori.
Decisioni	combinazione di funzioni e rami: ogni punto di ingresso e di uscita eseguito e ogni espressione booleana valutata con tutti i risultati possibili.

Stress, trend dei difetti e regressione

Il **test di stress** esegue il sistema oltre il carico massimo previsto, per far emergere i difetti: oltre i limiti il sistema non dovrebbe fallire in modo catastrofico, perdendo servizio o dati. È particolarmente importante nei sistemi distribuiti, che degradano con le condizioni della rete. La metrica **bug trend** misura la frequenza con cui si trovano difetti: quando tende a zero, o non ci sono più difetti, o trovarli è diventato molto costoso. Nella pratica si eseguono i test spesso, su versioni compilate ogni notte (nightly build), e si fanno i **test di regressione**: si rieseguono tutti i test già scritti per evitare che ricompaiano difetti già eliminati. Ci si ferma in base alla copertura e al bug trend.

Progettare i test a partire dai requisiti

Un esempio completo. Requisiti: un'associazione registra i prestiti di libri fra i soci; ogni socio può avere un solo libro in prestito alla volta, e di ogni libro esiste una sola copia. In progettazione si individuano le classi `Socio` e `Libro`, che contengono i dati, e `Prestiti`, che tiene la tabella dei prestiti come coppie socio e libro e offre le operazioni `registra` e `restituisci`. Dai requisiti si ricavano i casi di test, ciascuno con il risultato atteso.

CASO DI TEST	RISULTATO ATTESO
Registrare un prestito a un socio	la registrazione va a buon fine
Registrare due libri allo stesso socio	il secondo prestito è bloccato
Registrare lo stesso libro a due soci	il secondo prestito è bloccato
Registrare un prestito a un socio null	la registrazione è bloccata
Registrare un prestito di un libro null	la registrazione è bloccata

Ogni caso diventa un metodo di test, preceduto da un metodo che ricrea dati nuovi, così i test restano indipendenti: gli ultimi due casi appartengono alle classi di equivalenza non valide.

DA RICORDARE

Verifica: prodotto costruito bene, conforme alle specifiche. **Validazione**: prodotto giusto per l'utente. Un test ha successo se trova errori, e può mostrarne la presenza, non l'assenza. Test **black box** con classi di equivalenza e valori di confine; **white box** con test dei percorsi (cc). **Copertura** di funzioni, costrutti, archi, rami, condizioni, decisioni. Test di **stress**, **bug trend**, test di **regressione**.