Astrazione

Astrazione in senso lato

L'astrazione, nel suo significato più generale, è un'**attività di eliminazione dei dettagli operativi e irrilevanti** per concentrarsi sulle linee generali del problema da risolvere o degli oggetti da descrivere.

Il processo di astrazione

Quando affronti un problema reale, l'astrazione ti permette di:

1. **Identificare gli aspetti essenziali** - Capire quali caratteristiche e comportamenti sono veramente importanti per il tuo sistema
2. **Ignorare i dettagli superflui** - Tralasciare tutto ciò che non è rilevante per il contesto specifico
3. **Creare un modello semplificato** - Rappresentare la realtà in modo gestibile

Esempio pratico

Supponiamo di dover modellare una "Persona" per un sistema universitario:

Dettagli da INCLUDERE (rilevanti):

- Nome, cognome
- Matricola
- Corso di laurea
- Esami sostenuti

Dettagli da ESCLUDERE (irrilevanti per questo contesto):

- Colore degli occhi
- Peso, altezza
- Gruppo sanguigno
- Marca del telefono

Astrazione in senso stretto

L'astrazione in senso stretto è un **meccanismo che permette di separare la definizione di ciò che vogliamo ottenere ("cosa") dal modo in cui lo otterremo ("come")**.

La separazione tra interfaccia e implementazione

Questo principio fondamentale della programmazione orientata agli oggetti ti permette di:

- **Definire COSA** un oggetto può fare (le sue funzionalità, il suo contratto)
- **Nascondere COME** lo fa (i dettagli implementativi)

```
// COSA - Definisco l'interfaccia (il contratto)
abstract class Forma {
    abstract double calcolaArea();
    abstract double calcolaPerimetro();
}

// COME - Implementazioni specifiche
class Cerchio extends Forma {
    private double raggio;

    double calcolaArea() {
        return Math.PI * raggio * raggio; // implementazione specifica
    }

    double calcolaPerimetro() {
        return 2 * Math.PI * raggio; // implementazione specifica
    }
}

class Rettangolo extends Forma {
    private double base, altezza;

    double calcolaArea() {
```

```
        return base * altezza; // implementazione diversa
    }

    double calcolaPerimetro() {
        return 2 * (base + altezza); // implementazione diversa
    }
}
```

```
// Chi usa queste classi sa COSA fanno, non COME
Forma f1 = new Cerchio();
Forma f2 = new Rettangolo();

// Posso chiamare i metodi senza sapere come sono implementati
double area1 = f1.calcolaArea(); // non so che usa PI*r²
double area2 = f2.calcolaArea(); // non so che usa base*altezza
```

Lo sapevi?

Il **future proofing** è una strategia di progettazione che mira a rendere un prodotto resistente ai cambiamenti tecnologici, normativi o di mercato nel tempo.

Come si collega all'astrazione?

L'astrazione è uno degli strumenti principali per realizzare il future proofing nel software:

```
// Interfaccia stabile nel tempo
interface SistemaPagamento {
    boolean processaPagamento(double importo);
}

// Implementazioni che possono cambiare
class PagamentoCartaCredito implements SistemaPagamento { ... }
class PagamentoCriptovaluta implements SistemaPagamento { ... }
class PagamentoBancaDigitale implements SistemaPagamento { ... }
```

Se domani emerge una nuova tecnologia di pagamento, aggiungi semplicemente una nuova implementazione senza toccare il codice esistente.

2. Adattabilità alle normative

```
abstract class GestorePrivacy {
    abstract void ottieniConsenso();
    abstract void eliminaDati();
}

// Oggi: GDPR europeo
class GestorePrivacyGDPR extends GestorePrivacy { ... }

// Domani: nuove normative internazionali
class GestorePrivacyCCPA extends GestorePrivacy { ... }
```

Il valore del future proofing

Progettando con astrazione, crei software che può evolversi senza costose riscritture complete. Il codice che usa le astrazioni rimane valido anche quando le tecnologie sottostanti cambiano radicalmente.

Classi Astratte e Classi Concrete: Una Spiegazione Teorica

Il Concetto Fondamentale

Quando progettiamo software orientato agli oggetti, ci troviamo di fronte a una distinzione importante: alcune entità rappresentano concetti **completi e concreti** che esistono nella realtà o nel nostro dominio applicativo, mentre altre rappresentano **idee generali e incomplete** che servono come fondamenta per costruire altre entità.

Le Classi Concrete

Le classi concrete sono quelle che rappresentano oggetti **realmente istanziabili**. Pensa a una classe come a un progetto per costruire qualcosa: una classe concreta è un progetto completo, con tutte le istruzioni necessarie per creare effettivamente l'oggetto.

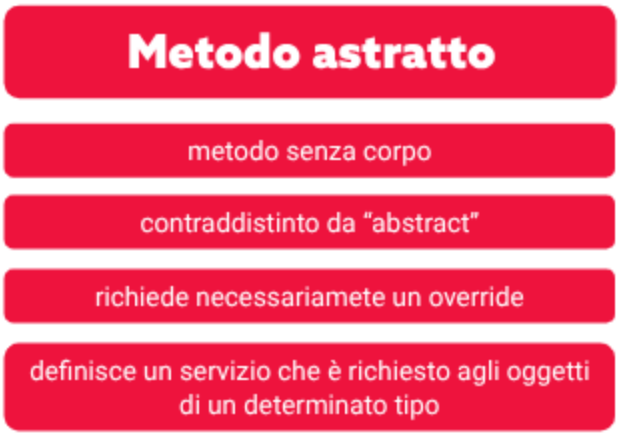
Quando scrivi una classe concreta, stai dicendo: "Questo è un tipo di oggetto che esiste davvero nel mio sistema, e so esattamente come funziona in ogni suo aspetto". Ogni metodo ha una sua implementazione, ogni comportamento è definito.

Le Classi Astratte

Le classi astratte, invece, rappresentano concetti **incompleti per natura**. Sono idee generali che catturano caratteristiche comuni a più entità concrete, ma che da sole non hanno senso di esistere come oggetti indipendenti.

Una classe astratta ti permette di dire: "So che tutti gli oggetti di questo tipo devono avere certi comportamenti, ma non so ancora esattamente come questi comportamenti verranno realizzati - dipenderà dal tipo specifico di oggetto”

Il Ruolo dei Metodi Astratti



I metodi astratti sono il cuore delle classi astratte. Rappresentano una **promessa**: "Qualsiasi classe che eredita da me deve fornire un'implementazione per questo comportamento". Non dicono come fare qualcosa, ma solo che quella cosa deve essere fatta.

È come un contratto: la classe astratta stabilisce gli obblighi, le classi concrete li adempiono. La classe astratta dice "devi saper volare", la classe concreta "Aereo" spiega come vola un aereo, la classe concreta "Uccello" spiega come vola un uccello.

Non puoi mai richiamare un metodo astratto "puro”. Quello che fai in realtà è **richiamare l'implementazione concreta** del metodo astratto attraverso un riferimento alla classe astratta.

```
// 1. Definisco la classe astratta con il metodo astratto
abstract class Employee {
    public abstract int getYearlyRetribution();
}

// 2. Creo una classe concreta che implementa il metodo
class Teacher extends Employee {
    private double stipendioMensile = 2000;

    // Questa è l'implementazione REALE
    public int getYearlyRetribution() {
        return (int)(stipendioMensile * 13);
    }
}

// 3. Uso il metodo
public static void main(String[] args) {
    // Non posso fare questo:
    // Employee e = new Employee(); // ERRORE!

    // Devo creare un oggetto concreto:
    Teacher t = new Teacher();

    // Quando chiamo il metodo, eseguo l'implementazione di Teacher
    int retribuzione = t.getYearlyRetribution();
    System.out.println(retribuzione); // 26000
```

```
}
```

Posso usare un **riferimento alla classe astratta** per puntare a un **oggetto concreto**:

```
// Riferimento di tipo Employee, oggetto di tipo Teacher
Employee dipendente = new Teacher();

// Chiamo il metodo attraverso il riferimento Employee
int retribuzione = dipendente.getYearlyRetribution();
```

La distinzione tra classi astratte e concrete riflette il modo in cui pensiamo naturalmente alla realtà. Nel mondo reale, ragioniamo sia in termini di **categorie generali** (gli animali respirano, i veicoli si muovono, le forme hanno un'area) sia in termini di **istanze specifiche** (il mio cane Fido, la mia auto Toyota, questo particolare cerchio di raggio 5).

Il Vantaggio della Generalizzazione

Quando usi classi astratte, stai praticando la **generalizzazione**. Invece di scrivere codice separato per ogni tipo specifico, identifichi le somiglianze e le estrai in una classe astratta. Questo ti dà diversi vantaggi.

- eviti la duplicazione
- crei un vocabolario comune.
- faciliti l'estensibilità

Quando scrivi `dipendente.getYearlyRetribution()`, Java fa questo ragionamento:

1. **Tempo di compilazione**: "Ok, `dipendente` è dichiarato come `Employee`, e `Employee` ha un metodo `getYearlyRetribution()`. Verifico che la chiamata sia sintatticamente corretta. ✓"
2. **Tempo di esecuzione**: "Ma l'oggetto a cui punta `dipendente` è effettivamente un `Teacher`. Quindi eseguo l'implementazione di `Teacher.getYearlyRetribution()`"

Questo meccanismo si chiama **dynamic binding** o **late binding**: Java decide quale versione del metodo eseguire solo a runtime, in base al tipo **effettivo** dell'oggetto, non al tipo del riferimento.

Il Polimorfismo e le Classi Astratte

Le classi astratte abilitano uno dei meccanismi più potenti della programmazione orientata agli oggetti: il **polimorfismo**. Puoi scrivere codice che lavora con la classe astratta, e quel codice funzionerà automaticamente con qualsiasi classe concreta che la estende.

Quando Usare l'Una o l'Altra

Usi una classe concreta quando hai un concetto **completo e istanziabile**. Se ha senso creare direttamente oggetti di quel tipo, allora è una classe concreta.

Usi una classe astratta quando hai un concetto **incompleto o troppo generico** per esistere da solo, ma che rappresenta caratteristiche comuni a più classi concrete. Se serve solo come base per altre classi e non ha senso crearla direttamente, allora è una classe astratta.

La scelta dipende dal tuo dominio e da come hai deciso di modellarlo. Non esiste una risposta giusta in assoluto, ma solo scelte più o meno appropriate al contesto specifico.