



# Capitolo 1



## Regole base

### Introduzione

Questo capitolo è dedicato all'illustrazione di un insieme iniziale di linee guida di carattere generale relativo alla programmazione. L'obiettivo fondamentale è favorire la realizzazione di programmi di migliore qualità. In particolare, in questo primo capitolo l'attenzione è focalizzata sulla produzione di codici più facilmente leggibili e quindi più facilmente comprensibili, mantenibili e riusabili. L'enfasi, pertanto, è quasi interamente conferita a caratteristiche "stilistiche" della programmazione, mentre considerazioni relative al miglioramento delle performance, a un più efficiente utilizzo della memoria e al miglioramento della trasportabilità sono rimandate al capitolo successivo.

Benché questa trattazione sia esplicitamente orientata a linguaggi di programmazione basati sul paradigma Object Oriented (OO), e in particolare al linguaggio Java, le direttive proposte si prestano ad essere facilmente adattate ad altri linguaggi di programmazione anche non necessariamente basati sul paradigma OO.

Considerata la caratteristica di generalità di questo primo capitolo, vi compaiono sia linee guida relative a concetti basilari come le dimensioni ottimali di classi e dei metodi, ad una migliore selezione dei nomi dei vari elementi del linguaggio di programmazione (attributi, metodi, interfacce e classi), sia altre di carattere più squisitamente OO, come coesione, accoppiamento, etc. Benché queste ultime regole presentino un livello concettuale molto diverso da



quelle più immediate presenti nel primo gruppo, si è comunque deciso di illustrarle in quanto rappresentano concetti fondamentali della programmazione, frequentemente nominati e discussi.

Pertanto, benché l'obiettivo principale del libro sia quello di presentare aspetti molto pratici riducendo al minimo la teoria, non è infrequente il caso in cui la trattazione si orienti verso tematiche più astratte. Ciò si è reso necessario sia per evitare un'eccessiva costrizione della trattazione che finirebbe per porre un eccessivo limite, sia per fornire indicazioni a tutti colori che intendano approfondire tematiche meno pratiche a maggior grado di astrazione.

La piena comprensione di quanto riportato in questo capitolo non richiede particolari prerequisiti, sebbene una minima esperienza di programmazione Java sia sicuramente di aiuto.

Quantunque la maggior parte delle direttive presentate in questo capitolo possano, a tratti, sembrare basilari e quasi scontate, nella pratica lavorativa esse vengono non di rado trascurate.

## Obiettivi

L'obiettivo principale di questo capitolo consiste nel supportare la produzione di codice più facilmente comprensibile. Si tratta di una caratteristica fondamentale del software ed è prerequisito irrinunciabile per un'altra caratteristica fondamentale: la manutenibilità. Quest'ultima ha assunto un ruolo fondamentale nei moderni processi di sviluppo del software essenzialmente per due motivi. In primo luogo perché è ormai universalmente accettato il fatto che i requisiti del sistema siano un'entità dinamica e quindi in continua evoluzione. Ciò rende impossibile e/o non conveniente "congelare i requisiti". La logica conseguenza è che ogni software di successo deve essere continuamente rivisto al fine di adeguarlo al perenne cambiamento dei requisiti. In secondo luogo perché la quasi totalità dei moderni processi di sviluppo del software include approcci di carattere iterativo e incrementale al fine di controllare più efficacemente i rischi progettuali. Ciò fa sì che la versione "finale" del sistema sia ottenuta attraverso una serie di iterazioni, ognuna delle quali aggiunge un determinato incremento alla versione precedente che, tipicamente, implica l'aggiornamento del codice prodotto precedentemente.

In ogni modo, produrre codici facilmente leggibili è importante per i seguenti motivi:

- permette di capire più velocemente, e spesso in maniera più completa, il problema che il codice risolve: quindi semplifica l'attività di test e di individuazione di eventuali errori;
- semplifica l'utilizzo di approcci iterativi e incrementali che spesso richiedono, a persone diverse, di modificare parti di programma realizzate in precedenza;
- facilita la comunicazione in termini degli algoritmi selezionati, delle scelte operate, e così via: ciò è particolarmente importante sia per l'attività di revisione del codice, sia in contesti di progetti di media-grande difficoltà;
- semplifica l'adeguamento del codice al perenne cambiamento dei requisiti.

## Direttive

### 1.1 Selezionare nomi significativi per le classi

La selezione dei nomi delle classi dovrebbe avvenire principalmente durante la fase di disegno del sistema. Tuttavia, in alcuni contesti molto ben limitati e definiti, come per esempio circoscritte investigazioni (pratica comunemente nota con il nome di *speak programming*), può risultare opportuno procedere con la codifica a partire da un documento di disegno molto essenziale. In scenari di questo tipo, il programmatore si trova nella situazione di dover disegnare parte del software direttamente codificando. Inoltre, anche in scenari più formali, l'attività di disegno non dovrebbe mai giungere a un eccessivo livello di dettaglio. Ciò, probabilmente, risulterebbe in un cattivo utilizzo del tempo a disposizione e condurrebbe alla mortificazione del team di sviluppo. Comunque sia, è abbastanza frequente la creazione di classi, inizialmente non previste, direttamente nella fase di sviluppo e vi è quindi la necessità di includere questa serie di regole in questo libretto fortemente orientato alla programmazione.

#### 1.1.1 Selezionare nomi (relativamente) brevi

La prima regola per la selezione di un nome opportuno da assegnare a una classe consiste nello scegliere un nome breve ma significativo. Si faccia attenzione che in questo contesto con il termine "nome" si intende proprio l'elemento grammaticale: *la particella linguistica che indica esseri viventi, oggetti, idee, fatti o sentimenti*. Pertanto, è necessario scegliere una stringa breve, in grado di indicare le responsabilità principali della classe. Tipicamente, i nomi più efficaci sono quelli attinti dal dominio che il programma intende automatizzare.

Alcune letterature, non eccessivamente moderne, indicano in 15 lettere il valore ideale per la lunghezza dei nomi di classi. Probabilmente, si tratta di un'indicazione eccessivamente restrittiva visti i servizi esposti dai moderni IDE (*Integrated Development Environment*, Ambienti di Sviluppo Integrato), ma che comunque fornisce un'idea dell'ordine di grandezza.

Alcuni esempi di validi nomi di classe sono riportati nella tabella 1.1.

Dominio	Esempi
Agenzia di viaggi	Flight, Travel, Booking, Customer, Itinerary, GeographicUnit, Nation, City, TimeZone, ShoppingCart, etc.
Istituzione di investimento	Currency, Price, StreamPrice, Quote Trade, Counterparty, Settlement, HolidayCalendar, Transfer, Instrument, etc.
Biblioteca	Book, Paper, Article, Author, Picture, Editor, etc.
Università	Lecture, Topic, Teacher, Student, Thesis, Department, etc.

Tabella 1.1 – Esempi di nomi di classi derivanti dal dominio di riferimento.

Come si può notare, i programmi dovrebbero essere scritti utilizzando la lingua Inglese. Questo per garantirne la massima audience possibile. Secondo le direttive [JavaCC] il nome delle classi deve essere scritto con la prima lettera in maiuscolo e le rimanenti in minuscolo. Qualora, il nome della classe sia composto da più termini, la prima lettera di ciascuna parola componente deve essere scritta in maiuscolo.

### 1.1.2 Valutare attentamente il ricorso alle abbreviazioni

Qualora la selezione di un nome breve sia ottenibile solo introducendo improbabili abbreviazioni, è consigliato rilassare la regola sul contenimento della lunghezza di un nome e quindi abbandonarsi a nomi più lunghi. In alcune sporadiche occasioni, tuttavia, il ricorso ad abbreviazioni è assolutamente necessario per evitare di introdurre nomi troppo lunghi. In questo caso, occorre utilizzare abbreviazioni in modo oculato, e soprattutto coerente, al fine di evitare confusione.

Per esempio, per il calcolo del fattore di rischio di diversi *trade* è necessario consultare specifici valori di rischio denominati *Future Fluctuation Risk Factors*. Questi elementi sono indicati con l'acronimo FFR dagli stessi operatori del business. Quindi, per la relativa implementazione, è possibile e conveniente dar luogo a classi denominate *FFR*: *FfrVO*, *FfrBS*, etc.

### 1.1.3 Porre attenzione all'utilizzo di "verbi" per i nomi di classi

I nomi delle classi dovrebbero essere costituiti da nomi e non da verbi. Questo perché le classi dovrebbero rappresentare entità, oggetti del dominio e non azioni. Pertanto, ad eccezione di alcuni casi molto limitati, qualora si abbia la necessità di denominare una classe con un verbo, bisognerebbe interrogarsi circa le responsabilità della classe, se abbia veramente senso creare una classe, e su quali siano proprietà le fondamentali, come coesione e accoppiamento. Nel caso in cui questo controllo abbia esito positivo, è importante ricordare che quasi sempre è possibile passare da un verbo ad un nome. Come esempio si consideri l'applicazione del pattern Command.

Il Command, molto brevemente, è un pattern che permette di richiedere a un oggetto l'esecuzione di determinate azioni, appositamente incapsulate in istanze di una predefinita classe, senza che l'istanza richiedente sia al corrente dei dettagli dell'operazione da eseguire né del concreto oggetto che la eseguirà. L'elemento fondamentale di questo pattern è la classe astratta, tipicamente denominata *Command*, che astrae il comportamento comune di tutte le classi che implementano comandi concreti. Queste, frequentemente, sono denominate con il nome del comando, che spesso è un verbo, come per esempio *Save*, *Read*, *Update*, etc. Anche in questo contesto, tuttavia, il nome può essere, frequentemente, ottenuto da una composizione, come per esempio *FileSaver*, *FileReader*, etc. Ed ecco che il nome della classe torna a essere un nome.

### 1.1.4 Aggiungere al nome della classe un suffisso relativo allo strato di appartenenza

Sebbene al momento in cui viene scritto questo libro, non ci sia un accordo nella comunità informatica circa il significato della parola "architettura", una sua caratteristica su cui è possi-

Strato	Dizione inglese	Esempi
strato di presentazione	<i>presentation layer</i>	TrolleyPage, TrolleyHandler
strato di business	<i>Business Service layer</i>	TrolleyBS
strato di business object	<i>Business Object layer</i>	TrolleyBO
strato di integrazione	<i>integration layer</i>	TrolleyDAO

Tabella 1. 2 – Strati di una classica architettura multi-tiered.

bile individuare un accordo unanime è che le architetture dovrebbero essere multi-strato (*multi-layered* o *multi-tiered*): organizzate in una serie di strati (*layer* o *tier*) adiacenti e comunicanti. Pertanto, in queste architetture è frequente che diverse versioni di una stessa classe esistano in diversi strati con diverse responsabilità. In questi circostanze è conveniente aggiungere al nome delle classi, in maniera coerente, opportuni suffissi atti a identificarne lo strato di appartenenza (tabella 1.2).

Altri utili suffissi sono VO e DTO (TrolleyVO, TrolleyDTO) relativi, rispettivamente, a *Value Object* e *Data Transfer Object*; questi non appartengono necessariamente a un solo strato ma, anzi, vengono tipicamente utilizzati per trasportare informazioni tra strati differenti.

Questa convenzione offre il grande vantaggio di poter derivare molte informazioni relative a una specifica entità già dalla semplice lettura del nome. Qualora si decida di utilizzare una simile convenzione, tuttavia, è di fondamentale importanza utilizzarla con tassativa coerenza onde evitare inutili confusioni.

### 1.1.5 Valutare l'utilizzo di appositi prefissi per evidenziare interfacce e classi astratte

Quantunque non si tratti di una strategia utilizzata dalla Sun, spesso risulta conveniente aggiungere, al nome delle classi, un prefisso, composto da un solo carattere, per evidenziare se la classe sia astratta o concreta e si tratti o meno di un'interfaccia. Pertanto la convenzione consiste nell'utilizzare:

- prefisso I per le interfacce. Per esempio IRepository, IConfigurationManager, etc.
- prefisso A per denotare classi astratte. Per esempio AltemsRepository, ADoubleLinkedList.

Il vantaggio offerto da questa nomenclatura consiste nella possibilità di risalire ad alcune informazioni utili circa una determinata classe o interfaccia già dalla sola lettura del nome.

Qualora si decidesse di utilizzare questa convenzione, è fondamentale applicarla consistentemente altrimenti si correrebbe il rischio di generare inutile confusione. Per esempio, la lettura di un nome di classe non prefissato dalla lettera I o A, porterebbe alla legittima conclusione che si tratti di una classe concreta e quindi il programmatore sarebbe portato a trattarla di conseguenza, mentre, invece potrebbe trattarsi di una classe astratta o di un'interfaccia denominata in maniera incorretta.

### 1.1.6 Utilizzare il suffisso "Exception" per indicare le classi eccezione



L'utilizzo del termine Exception per indicare classi atte ad incapsulare eccezione è un'utile convenzione utilizzata della Sun che dovrebbe essere sempre rispettata. Per esempio: NullPointerException, IllegalArgumentException, etc.

## 1.2 Selezionare nomi significativi per attributi/variabili e parametri

Variabili e attributi sono elementi fondamentali della programmazione, tanto che Dijkstra sosteneva che "una volta che un programmatore ha compreso l'utilizzo delle variabili, ha compreso l'essenza della programmazione". Gli attributi, poi, sono a tutti gli effetti variabili incapsulate in oggetti, utilizzate per mantenere lo stato degli stessi. Pertanto, una corretta ed efficace definizione dei loro nomi costituisce una condizione irrinunciabile per ottenere software di qualità.

### 1.2.1 Selezionare nomi (relativamente) brevi

Coerentemente a quanto specificato per i nomi delle classi, anche i nomi di attributi/variabili e parametri devono essere mnemonici e brevi (anche in questo caso diverse letterature, non eccessivamente moderne, identificano in 15 lettere il valore della lunghezza massima del nome di un attributo). Coerentemente con l'analoga direttiva valida per le classi, anche per i nomi di attributi, variabili e parametri è necessario cercare di utilizzare sostantivi (nomi). Inoltre, anche per gli attributi, quando possibile, è utile ricorrere a nomi attinti dal dominio di riferimento.

Per esempio, una classe Currency, potrebbe disporre dei seguenti attributi: mainUnitName (per esempio Euro), secondaryUnitName (per esempio Cent), symbol, currencySign, decimalPositions, etc.

Se usassimo l'italiano, una classe Libro, potrebbe disporre dei seguenti attributi: titolo, autori, numeroPagine, caseEditrici, prezzoConsigliato, etc.

Da notare che mentre gli attributi come titolo, numeroPagine e prezzoConsigliato, sono veri e propri attributi, caseEditrici, autori sono relazioni con altri oggetti. Queste distinzioni però, sebbene siano fondamentali in termini di disegno, lo sono molto meno in termini implementativi.

Per quanto concerne il nome delle variabili, esso dovrebbe ricordarne l'utilizzo. Alcuni esempi sono: boolean, skipWhiteSpace, int, length, char, currentChar, byte[], buffer, etc.

Una convenzione efficace per la dichiarazione dei parametri, molto utile per i metodi costruttori e modificatori, consiste nell'aggiungere al nome un prefisso formato dall'articolo indeterminativo. Ciò evita eventuali conflitti con gli attributi di classe, che comunque si risolvono inserendo la parola chiave this per identificare gli attributi di classe.

*Listato 1. Esempio di un costruttore della classe Byte. (L'implementazione Sun utilizza la convenzione **this.value = value**).*

```
/**
```

```
 * Constructs a newly allocated <code>Byte</code> object that
```

```
* represents the specified <code>byte</code> value.
*
* @param value the value to be represented by the
*           <code>Byte</code>.
*/
public Byte(byte aValue) {
    value = aValue;
}
```

Un'altra tecnica molto interessante consiste nell'aggiungere al nome della proprietà il suffisso `new`. Per esempio si consideri il metodo riportato di seguito atto a impostare il limite del Buffer, nella classe `java.nio.Buffer`.

*Listato 2. Metodo `limit` della classe `java.nio.Buffer`.*

```
public final Buffer limit(int newLimit) {
    if ((newLimit > capacity) || (newLimit < 0))
        throw new IllegalArgumentException();

    limit = newLimit;
    if (position > limit) position = limit;
    if (mark > limit) mark = -1;

    return this;
}
```

### 1.2.2. Valutare attentamente il ricorso alle abbreviazioni

I nomi di variabili spesso richiedono la combinazione di due o più termini. Le abbreviazioni, in linea di principio, tendono a ridurre il grado di leggibilità del codice. Tuttavia, qualora il nome di un attributo tenda a divenire eccessivamente lungo, è consigliabile valutare opportune abbreviazioni: l'importante però è che queste siano comprensibili e soprattutto utilizzate coerentemente. Per esempio:

```
/** If the next character is a line feed, skip it */
private boolean skipLF = false;

/** default char buffer size */
private static int defaultCharBufSize = 8192;
```

### 1.2.3 Considerare l'utilizzo delle lettere i, j, k per le variabili di ciclo

Le lettere `i`, `j`, `k`, grazie anche ad un retaggio matematico, risultano ottimi nomi per le variabili che regolano cicli come `for` e `while`. Inoltre, per cicli annidati è consigliabile seguire l'ordine alfabetico.

Qualora, il ciclo sia semplice, è possibile utilizzare anche nomi come `counter`. Infine, qualora si abbia la necessità di eseguire computazioni su matrici bidimensionali, potrebbe risultare conveniente utilizzare i seguenti indici: `row`, `col`.

Listato 3 – Esempio di utilizzo di variabili contatore.

```
for (int row=0; row < MAX_ROWS; row++) {  
  
    for (int col=0; col < MAX_COLS; col++) {  
        <instruction>  
        ...  
        <instruction>  
    }  
}
```

#### 1.2.4 Evitare di utilizzare nomi simili nell'ambito dello stesso scope

Molti linguaggi di programmazione prevedono sintassi *case-sensitive*: sensibili al carattere. Pertanto, caratteri maiuscoli e minuscoli sono considerati diversi (per esempio: `Count` è diverso da `count`). In questo caso, è necessario evitare nomi simili che differiscano semplicemente per la modalità di scrittura, per esempio: `currencyISOCode`, `currencySOCCode`. In generale è sempre opportuno evitare nomi simili nell'ambito del medesimo *scope* (campo d'azione).

Nel listato 4 sono mostrate due implementazioni del metodo `rehash` della classe `java.util.Hashtable`. L'implementazione di sinistra è stata volutamente resa meno leggibile utilizzando nomi di variabile molto simili tra loro.

Il metodo `rehash` serve per incrementare la capacità di un oggetto `Hashtable` e quindi per meglio organizzare gli elementi riducendo il numero di conflitti e, conseguentemente, aumentandone l'efficienza.

#### 1.2.5 Evitare di utilizzare la stessa variabile per scopi diversi

Non è infrequente analizzare implementazioni di metodi in cui una stessa variabile sia riutilizzata, all'interno di uno stesso metodo, per scopi completamente diversi. Questa pratica dovrebbe essere evitata in quanto tende a ridurre il grado di leggibilità del codice e, frequentemente, a confondere il lettore. Inoltre, "ottimizzazioni" di questo tipo raramente portano un vantaggio reale e anzi finiscono per degradare la leggibilità del codice. Chiaramente, un discorso completamente diverso vale per tentativi di "riciclare" oggetti, soprattutto in contesti di programmazione concorrente.

#### 1.2.6 Valutare attentamente il ricorso al carattere sottolineato come prefisso delle variabili di classe

La convenzione di utilizzare il carattere sottolineato basso (*underscore*, il segno `_`) per indicare attributi di classe è una prassi molto utilizzata nella programmazione C++. In Java ciò non è assolutamente necessario, giacché il linguaggio dispone della parola chiave `this`.



Listato 1.4 – Due versioni del metodo `rehash` della classe `java.util.Hashtable`, in cui quella di sinistra è stata resa meno leggibile per via dell'utilizzo di nomi di variabili simili.

#### Codice sconsigliato

```
protected void rehash() {
    int cap      = table.length;
    Entry[] map  = table;

    int capacity = cap * 2 + 1;
    Entry[] mapp = new Entry[capacity];

    modCount++;
    threshold = (int) (capacity * loadFactor);
    table = mapp;

    for (int i = cap ; i-- > 0 ; ) {
        for (Entry<K,V> mapn = map[i]; mapn!=null ; ) {
            Entry<K,V> e = mapn;
            mapn = mapn.next;

            int i1 = (e.hash & 0x7FFFFFFF) %
                capacity;
            e.next = mapp[i1];
            mapp[i1] = e;
        }
    }
}
```

#### Codice più leggibile

```
protected void rehash() {
    int oldCapacity = table.length;
    Entry[] oldMap = table;

    int newCapacity = oldCapacity * 2 + 1;
    Entry[] newMap = new Entry[newCapacity];

    modCount++;
    threshold = (int)(newCapacity * loadFactor);
    table = newMap;

    for (int i = oldCapacity ; i-- > 0 ; ) {
        for (Entry<K,V> old = oldMap[i] ; old != null ; ) {
            Entry<K,V> e = old;
            old = old.next;

            int index = (e.hash & 0x7FFFFFFF) %
                newCapacity;
            e.next = newMap[index];
            newMap[index] = e;
        }
    }
}
```

Listato 5 – Costruttore della classe `java.lang.Boolean`.

```
public Boolean(boolean value) {
    this.value = value;
}
```

Inoltre, come visto in precedenza, in questi casi è sufficiente assegnare al parametro un prefisso costituito dall'articolo indeterminativo: `aValue`.

### 1.2.7 Qualora si decida di utilizzare la notazione ungherese, la si utilizzi coerentemente

La notazione ungherese (*Hungarian Notation* in Inglese, il cui nome è un omaggio al suo ideatore Charles Simonyi di origine appunto ungherese) è una convenzione nata in casa Microsoft per l'attribuzione di nomi di parametri, variabili e attributi particolarmente popolare nella comunità

C e C++. L'idea base consiste nell'assegnare un prefisso ai nomi al fine di specificarne il tipo. L'obiettivo consiste nel permettere agli sviluppatori di codice di capire il tipo di una variabile semplicemente dal relativo nome. Si tratta di una convenzione spesso utile, in quanto evita di dover effettuare continui salti di pagina dovuti al fatto che l'utilizzo di variabili, frequentemente, avviene in punti distanti dalla relativa dichiarazione. Per esempio: `iCounter` rappresenta un contatore intero, `bFound` rappresenta una variabile di tipo booleana, `oAccount` rappresenta il riferimento a un oggetto di tipo account, e così via.

La notazione ungherese è oggetto di diversi dibattiti tra chi la considera molto utile e altri che invece le addebitano di portare alla generazione di codice meno leggibile.

In questo contesto non viene presa alcuna posizione, ad eccezione del fatto che è fondamentale preservare la coerenza. Pertanto, qualora si decidesse di usare la notazione ungherese, è necessario utilizzarla in maniera costante e consistente. Nella tabella 1.3 viene proposto un elenco standard di prefissi.

Infine, qualora si debba intervenire per modificare del codice scritto da altri, è consigliabile mantenere la convenzione utilizzata dall'autore del codice onde evitare confusione.

## 3 Selezionare nomi significativi per i metodi

### 3.1 Selezionare nomi brevi

I nomi dei metodi devono essere sufficientemente brevi ma significativi, in modo da riuscire a illustrare chiaramente il servizio che forniscono. I metodi rappresentano azioni e pertanto i loro nomi dovrebbero includere un verbo. Qualora poi non si tratti di metodi di servizio, i nomi dovrebbero essere facilmente riconducibili al linguaggio business.

Prefisso	Tipo
b	Boolean
c	Char
by	Byte
s	Short
i	Int
l	Long
f	Float
d	Double
o	Object
e	Exception

Tabella 1.3 – Prefissi standard da utilizzarsi per la notazione ungherese.

Da tener presente che i vari metodi, tipicamente, eseguono qualche azione relativa alla classe cui appartengono. Pertanto, il nome della classe dovrebbe essere omesso soprattutto per i metodi non privati.

Per esempio, qualora si consideri una classe carrello della spesa (Trolley) per il nome del metodo atto a verificarne la consistenza è sufficiente considerare `check` invece che `checkTrolley`. Altri esempi di metodi della classe "carrello della spesa" sono: `addItem`, `deleteItem`, `empty`, etc.

### 3.2 Valutare attentamente il ricorso alle abbreviazioni

Utilizzare abbreviazioni per nomi dei metodi tende a diminuirne il grado di comprensione e a creare confusione. Pertanto il loro utilizzo dovrebbe essere sempre limitato a casi in cui non ricorrere alle abbreviazioni genererebbe nomi di metodi troppo lunghi. Qualora si decida di introdurre delle abbreviazioni, come di consueto, è opportuno utilizzarle in modo standard.

Per esempio, una delle operazioni necessarie per eseguire il *settlement* (gestione del pagamento) di un trade consiste nell'individuare tra le istruzioni di pagamento (Standard Settlement Instructions) predefinite dalla controparte quella più opportuna. In questo caso, un metodo potrebbe essere `getMatchingSSI`. Da notare che l'acronimo SSI è di uso comune anche tra gli operatori business.

### 3.3 Qualora un metodo svolga più azioni logicamente distinte, aggiungere le particelle "and" o "or" nel nome

Spesso l'implementazione di un metodo richiede di includere diverse azioni di alto livello che è opportuno evidenziare chiaramente già dal nome. Non è infrequente il caso in cui due azioni distinte debbano appartenere allo stesso metodo, perché, per esempio debbano condividere una stessa transazione o perché è conveniente avere una sezione protetta in comune. Qualora queste azioni siano eseguite sequenzialmente, allora è necessario specificare la congiunzione `and` tra i nomi di queste azioni, mentre qualora le azioni siano eseguite in alternativa, allora è necessario utilizzare la congiunzione `or`. Per esempio: `updateOrInsertItem`, `saveAndPublishMessage`,

#### 1.3.4 Non ripetere il nome della classe nei nomi dei metodi

Come visto in precedenza, per la selezione dei nomi dei metodi si dovrebbe limitare l'attenzione ai soli verbi, sebbene sia frequentemente necessario ricorrere a nomi composti. Spesso però si compone il nome dei metodi utilizzando anche il nome della classe di appartenenza. Sebbene ciò non sia un problema serio, si tratta comunque di una ripetizione inutile.

Si consideri per esempio la classe `java.util.Vector`. Alcuni dei suoi metodi sono: `addElement`, `insertElementAt`, `elementAt`, `size`, `firstElement`, `lastElement`, `indexOf`, etc.

#### 1.3.5 Utilizzare una convenzione chiara nella selezione dei nomi dei metodi

Nel disegno di classi succede spesso di dover realizzare metodi ricorrenti come per esempio "inserimento", "eliminazione", "ricerca" di elementi e così via. In questo caso si consiglia di selezionarli coerentemente i nomi per questi metodi. Per esempio, analizzando le classi Java è possibile evidenziare le convenzioni riportate nella tabella 1.4.

Nome metodo	Utilizzo
add	Inserimento di un elemento in una lista.
addAll	Inserimento di una collezione in una lista.
clear	Rimuove tutti gli elementi di una lista.
get	Reperimento di un elemento specifico di una lista.
remove	Rimozione di un elemento.
size	Restituzione del numero degli elementi di una lista.

Tabella 1.4 – Tabella con la convenzione Java per alcuni nomi di metodi ricorrenti.

## 1.4 Implementare classi orientate agli oggetti

### 1.4.1 Evitare l'implementazione di classi di grandi dimensioni

Classi di grandi dimensioni sono sempre da evitare in quanto tendono a diventare difficilmente comprensibili e quindi manutenibili, a complicare i test, a rendere problematico lo sviluppo in parallelo da parte di team di diverse persone e a ridurre il riutilizzo.

Classi di grandi dimensioni, spesso, sono il risultato dell'inglobamento di diverse classi solamente in una e pertanto si prestano a dare luogo a implementazioni a bassa coesione che, come i principi dell'OO insegnano, andrebbero evitati. La tendenza naturale di sistemi OO è quella di generare molte classi di dimensioni medio-piccole, facili da comprendere, mantenere, verificare e riusare. Ciò, tra l'altro, favorisce il naturale processo di apprendimento della mente umana che richiede, in ogni momento, di concentrarsi su un insieme limitato di aspetti. Chiaramente, un numero eccessivo di classi è ugualmente sbagliato e tende a creare una serie di problemi legati a un elevato accoppiamento.

Ogni qualvolta la dimensione di una classe cominci a passare i limiti, dovrebbe essere naturale interrogarsi se si stiano inglobando più concetti distinti in una sola classe; e, se non è così, è comunque bene chiedersi se l'introduzione di opportune classi "helper" possano facilitarne la comprensione.

### 1.4.2 Utilizzare il livello di accesso più ristretto possibile

Il livello di accesso di un elemento (classe, interfaccia, metodo e attributo) determina quali altre classi appartenenti allo stesso sistema possono accedere all'elemento in questione.

Una delle leggi fondamentali dell'OO, l'incapsulamento (nota anche come principio dell'*information hiding*), prescrive che le classi celino al mondo esterno la propria organizzazione in termini di struttura e logica interna. Il principio fondamentale è che nessuna parte di un sistema debba dipendere dai dettagli interni di una sua parte. Questo, come è lecito attendersi, rende possibile modificare la struttura interna delle classi di un sistema evitando che si generino ripercussioni su altre parti dello stesso.

Per esempio, non è infrequente accorgersi, durante la fase di test, che determinati oggetti danno luogo a un eccessivo consumo di memoria oppure eseguono specifiche operazioni con

performance insoddisfacenti. In questi casi, qualora si sia utilizzato con intelligenza il principio dell'incapsulamento, è possibile reingegnerizzare le classi da cui derivano tali oggetti, senza dover modificare altre parti del sistema.

I linguaggi di programmazione OO tendono a prevedere i seguenti quattro livelli di accesso, riportati in ordine di accesso crescente: privato, amichevole o di default, protetto e pubblico. Al fine di evitare qualsiasi confusione, in tabella 1.5 è riportato uno schema dei livelli di accesso Java.

Da notare che per quanto concerne il livello di accesso "amichevole", in Java questo è dichiarato omettendo il livello di accesso di un elemento. Inoltre, un elemento il cui livello di accesso è amichevole non è accessibile a classi che non si trovino nello stesso package, anche qualora la classe che desideri accedere all'elemento sia una classe figlio.

L'applicazione della legge dell'incapsulamento, per quanto concerne gli attributi, implica di utilizzare prevalentemente un livello di accesso privato. Qualora una classe possa prevedere delle specializzazioni (classi ereditanti) che, per loro natura, siano fortemente legate alla classe antenata (come per esempio la classe `java.util.AbstractCollection`), allora potrebbe risultare molto utile dichiarare alcuni attributi con un livello di accesso protetto. Il livello di accesso di default, invece potrebbe risultare utile nella realizzazione di package grafici. Infine il livello di accesso pubblico dovrebbe essere riservato esclusivamente alla dichiarazione di costanti.

Dato che una buona implementazione OO prevede che tutti gli attributi siano privati, ne segue che l'interfaccia propria di una classe sia data dai suoi metodi, la cui invocazione rappresenta l'unico modo con cui i vari oggetti possano interagire tra loro. Pertanto, i metodi dovrebbero essere dichiarati privati (meglio se protetti, al fine di consentirne l'accesso a possibili classi ereditanti), a meno che non si tratti di metodi appartenenti all'interfaccia della classe (metodi che possano essere invocati da altre classi). In questo caso, il livello di accesso da scegliere è pubblico (se invocabile da qualsiasi classe) o di default (se invocabile da classi appartenenti allo stesso package).

<b>Livello di accesso</b>	<b>Stessa classe</b>	<b>Stesso package</b>	<b>Classe ereditante</b>	<b>Tutte le altre</b>
Privato <code>private</code>	✓	✗	✗	✗
Amichevole non specificato	✓	✓	✗	✗
Protetto <code>protected</code>	✓	✓	✓	✗
Pubblico <code>public</code>	✓	✓	✓	✓

Tabella 1.5 – Livelli di accesso in Java.

### 1.4.3 Massimizzare la coesione interna delle classi

"Un modulo presenta un'elevata coesione quando tutti i componenti collaborano fra loro per fornire un ben preciso comportamento" (Booch). Dal punto di vista delle classi, la coesione è la misura della correlazione delle proprietà strutturali (attributi e relazioni con le altre classi) e comportamentali di una classe (metodi). Nella letteratura informatica è possibile individuare diverse formule matematiche atte a determinare il grado di coesione di una classe. Una delle più note è:

$$c = \frac{(m - \sum(m_i) / a)}{(m - 1)}$$

dove

$m$  è il numero di metodi della classe

$a$  è il numero di attributi della classe.

$m_i$  il numero di metodi che accedono all'attributo  $i$ -mo.

$\sum(m_i)$  è la sommatoria di tutti i contributi  $m_i$  calcolati per tutti gli attributi della classe.

Da notare che minore è il valore di  $c$  e maggiore è il grado di coesione della classe. In sostanza, maggiore è il numero di metodi che accedono a diversi attributi della classe, e superiore è il grado di coesione. Questa formula considera anche casi in cui più classi siano state inglobate in una sola, caratterizzati dall'esistenza di diversi gruppi (logici) di attributi, acceduti ciascuno da un insieme distinto e ben definito di metodi. Pertanto la classe espone servizi che eseguono compiti non relazionati.

I problemi tipici generati da classi a bassa coesione sono relativi alla difficoltà di comprensione del codice e quindi di manutenzione, di laboriosità nell'eseguire i vari test, impossibilità nel riutilizzo, difficoltà di isolare elementi soggetti a variazioni, ecc. In casi estremi si giunge alla perdita di controllo dell'intera applicazione caratterizzata da servizi sparpagliati in classi in cui non dovrebbero appartenere e conseguente incremento del grado di accoppiamento. Un esempio di classe a bassa coesione è relativa alla rappresentazione di un utente del sistema (*User*) ove attributi del tipo *name*, *surname*, *dateOfBirth*, *gender*, etc. siano combinati con altri del tipo *currency*, *value*, etc. Lo stesso vale nel caso in cui in una classe di tipo carrello della spesa (*Trolley*), si trovassero dei metodi del tipo *empty*, *addItem*, *verifyContent*, etc. e altri del tipo *placeOrder*, *findUsers*, etc.

Per valutare il grado di coesione di una determinata classe, non sempre è necessario applicare un approccio formale basato sulle formule. Infatti, problemi di scarsa coesione possono essere facilmente rilevati da una serie di segnali di allarme. Il più evidente è connesso alle dimensioni delle classi. Qualora una classe contenga troppi attributi, oppure troppe relazioni con altre classi, oppure un numero eccessivo di metodi, molto probabilmente il livello di coesione di questa non è molto elevato. Pertanto è probabile si abbia a che fare con una classe che ne ingloba altre. Un altro segnale è rappresentato dalla presenza, nella medesima classe, di attributi partizionabili in insiemi distinti e non relazionati, e dall'aver metodi che non accedono mai ad attributi appartenenti a diversi insiemi. Un altro indicatore di scarsa coesione è dato da situazioni in cui non si riesca ad identificare un nome preciso per una classe oppure questo risulta troppo generico.

#### 1.4.4 Minimizzare l'accoppiamento tra le classi

Un principio di importanza fondamentale nel disegno e nell'implementazione delle classi è relativo al grado di accoppiamento che, chiaramente, deve essere minimizzato. Massima coesione e minimo accoppiamento sono due proprietà imprescindibili e fortemente relazionate dell'OO. Il minimo accoppiamento è particolarmente ricercato in quanto capace di generare tutta una serie di vantaggi (del tutto equivalenti a quelli generati dalla massima coesione), quali maggiore comprensione e facilità di manutenzione del codice, aumento della probabilità di riutilizzo, semplificazione delle attività di manutenzione, ecc. Il livello di accoppiamento delle classi deve essere minimizzato ma, ovviamente, non eliminato: non esiste un reale disegno OO senza accoppiamento. Ogni volta che un oggetto interagisce con un altro si ha una manifestazione di accoppiamento. La mancanza totale di accoppiamento porterebbe generare l'effetto contrario, ossia la produzione di codice poco leggibile dovuto a classi (scarsamente coese) omnicomprehensive.

L'accoppiamento è definita come la "misura della dipendenza tra componenti software (classi, package, componenti veri e propri, ecc.) di cui è composto il sistema". Sia ha una dipendenza tra due classi, quando un elemento (client) per espletare le proprie responsabilità accede alle proprietà comportamentali (metodi) e/o strutturali (attributi) di un altro (*supplier*, fornitore). Ciò implica che il funzionamento del componente stesso dipende dal corretto funzionamento (e quindi dall'implementazione) degli altri componenti "acceduti"; pertanto cambiamenti in questi ultimi generano ripercussioni sul componente client. Come scritto pocanzi, il livello di accoppiamento deve essere ridotto al minimo, ma non eliminato.

## 1.5 Porre attenzione alla scrittura dei metodi

### 1.5.1 Implementare metodi che eseguono un solo compito

Come principio di carattere generale bisognerebbe implementare metodi che svolgano una sola funzione ben definita. Ciò permette di realizzare codice di maggiore leggibilità, quindi più facilmente comprensibile, manutenibile, riusabile, e così via. Ciò, inoltre, concorre ad aumentare il livello di coesione.

*Listato 6 – Metodi che eseguono un solo compito ben definito.*

```
/**
 * expands the capacity of an AbstractStringBuilder object
 * @param minimumCapacity minimum capacity requested to this object.
 */
void expandCapacity(int minimumCapacity) {
    int newCapacity = (value.length + 1) * 2;

    if (newCapacity < 0) {
        newCapacity = Integer.MAX_VALUE;
    } else if (minimumCapacity > newCapacity) {
```

```
        newCapacity = minimumCapacity;
    }

    char newValue[] = new char[newCapacity];
    System.arraycopy(value, 0, newValue, 0, count);
    value = newValue;
}
```

### 1.5.2 Evitare l'implementazione di metodi lunghi

Metodi la cui implementazione risulti maggiore di circa 15 righe risultano di difficile comprensione e quindi di difficile test, manutenzione e riutilizzo. Una tecnica decisamente più conveniente consiste nello scrivere metodi la cui intera implementazione sia visualizzabile in una pagina di un normale monitor evitando così di dover eseguire continui scroll della pagina. Questo approccio è poi confacente alla caratteristica tipica della mente umana di concentrarsi, in ogni momento, su un insieme ristretto di concetti.

Ogni qualvolta l'implementazione di un metodo ecceda le dimensioni suddette, è meglio tentare di ridurlo nella composizione di metodi più piccoli di più facile comprensione.

### 1.5.3 Non scrivere metodi con molti parametri

Ogni qualvolta la firma di un metodo contenga troppi parametri (tipicamente più di quattro o cinque), è necessario verificare se sia il caso di incapsulare questi parametri in un'apposita classe o permetterne l'impostazione attraverso opportuni metodi. Questa regola può essere rilassata per i metodi costruttori di oggetti disegnati puramente per trasportare valori, per esempio Value Object (VO) e Data Transfer Object (DTO). Anche in questi casi però, disegnare metodi costruttori con diversi parametri può creare non pochi problemi in presenza di gerarchie di oggetti. Infatti, la variazione del costruttore di una classe antenata (per esempio l'aggiunta di un nuovo parametro) finisce con il ripercuotersi in tutte le classi ereditanti.

Il problema è che metodi la cui firma contiene una lunga lista di parametri sono difficili da utilizzare, il loro utilizzo richiede di controllare continuamente il significato e la posizione dei parametri, etc. Pertanto, metodi di questo tipo possono essere causa di frequenti errori.

Si consideri come esempio di metodo con molti parametri, il costruttore della classe `SimpleTimeZone`: riportato nel listato 7.

*Listato 7 – Un costruttore della classe `java.util.SimpleTimeZone`.*

```
public SimpleTimeZone(int rawOffset, String ID,
                    int startMonth, int startDay, int startDayOfWeek, int startTime,
                    int endMonth, int endDay, int endDayOfWeek, int endTime)
```

### 1.5.4 Eseguire il controllo del valore dei parametri appena possibile

È bene che le prime linee di codice dell'implementazione di ogni metodo siano relative al controllo della validità del valore dei parametri forniti. Se il metodo è destinato a fallire è meglio che fallisca prima possibile. Ciò è particolarmente importante per i metodi costruttori.



Le tipiche eccezioni (Java standard) considerate per comunicare questo tipo di errore sono `NullPointerException` e `IllegalArgumentException`. Si consideri l'implementazione di uno dei costruttori della classe `java.util.Vector`.

*Listato 8 – Costruttore della classe `java.util.Vector`.*

```
public Vector(int initialCapacity, int capacityIncrement) {
    super();

    if (initialCapacity < 0)
        throw new IllegalArgumentException("Illegal Capacity: "+initialCapacity);

    this.elementData = new Object[initialCapacity];
    this.capacityIncrement = capacityIncrement;
}
```

### 1.5.5 Estrarre il comportamento comune

Nell'implementare metodi di una classe, non è infrequente il caso in cui una determinata sezione di codice sia ripetuta in diversi metodi, magari con minime differenze. Questa evenienza, tipicamente, porta ad implementare metodi lunghi e quindi meno leggibili. Inoltre, è naturale che una stessa porzione di codice sia ripetuta in diverse parti. Ciò fa sì che eventuali modifiche relative a tale sezione debbano essere ripetute diverse volte con il concreto rischio di tralasciarne qualcuna. In questi casi è opportuno verificare la possibilità di raggruppare il comportamento comune, eventualmente parametrizzandolo, in appositi metodi privati. Si consideri il codice del listato 9.

*Listato 9 – Improbabile implementazione del metodo `setLength` della classe `java.lang.AbstractStringBuilder`.*

```
public void setLength(int newLength) {
    if (newLength < 0)
        throw new StringIndexOutOfBoundsException(newLength);
    if (newLength > value.length) {
        int newCapacity = (value.length + 1) * 2;
        ensureCapacity(newCapacity);

        if (newCapacity < 0) {
            newCapacity = Integer.MAX_VALUE;
        } else if (newLength > newCapacity) {
            newCapacity = minimumCapacity;
        }

        char newValue[] = new char[newCapacity];
        System.arraycopy(value, 0, newValue, 0, count);
        value = newValue;
    }
}
```

```

if (count < newLength) {
    for (; count < newLength; count++)
        value[count] = '\0';
} else {
    count = newLength;
}
}

```

resetArray

### 1.5.6 Intervallare il codice con apposite righe vuote

Dall'analisi del codice dei metodi è possibile notare come alcuni gruppi di istruzioni eseguano compiti ben definiti e quindi risultino fortemente interconnesse. Questi gruppi formano le particelle dell'implementazione del metodo. Pertanto costituisce una buona norma evidenziare questi gruppi a maggiore coesione separandoli dagli altri tramite righe vuote.

Codici organizzati in questo modo sono più facili da leggere e da comprendere.

*Listato 10 – Due versioni del metodo equals presente nella classe java.util.AbstractList. In quella di sinistra (sconsigliata) le istruzioni sono scritte in sequenza senza separare i diversi blocchi. In quella di destra, invece, sono state aggiunte delle righe vuote per assegnare più enfasi a gruppi logici di istruzioni.*

Codice sconsigliato

```

public boolean equals(Object o) {
    if (o == this)
        return true;
    if (!(o instanceof List))
        return false;
    ListIterator<E> e1 = listIterator();
    ListIterator e2 = ((List) o).listIterator();
    while (e1.hasNext() && e2.hasNext()) {
        E o1 = e1.next();
        Object o2 = e2.next();
        if (!(o1==null ? o2==null : o1.equals(o2)))
            return false;
    }
    return !(e1.hasNext() || e2.hasNext());
}

```

Codice consigliato

```

public boolean equals(Object o) {
    if (o == this)
        return true;

    if (!(o instanceof List))
        return false;

    ListIterator<E> e1 = listIterator();
    ListIterator e2 = ((List) o).listIterator();

    while (e1.hasNext() && e2.hasNext()) {
        E o1 = e1.next();
        Object o2 = e2.next();

        if (!(o1==null ? o2==null : o1.equals(o2)))
            return false;
    }

    return !(e1.hasNext() || e2.hasNext());
}

```

### 1.5.7 Valutare l'utilizzo delle parentesi graffe anche quando non è strettamente necessario

L'utilizzo delle parentesi graffe anche quando non strettamente necessario (per esempio nel caso di cicli con una sola istruzione) è una buona tecnica per migliorare lo stile del proprio codice. In particolare, semplifica la lettura e la manutenzione del codice, specie qualora il codice contenga una serie di costrutti annidati, e agevola la manutenzione del codice qualora sia necessario aggiungere ulteriori istruzioni in un costrutto che originariamente ne conteneva una sola.

*Listato 11 – Due versioni di un metodo costruttore della classe java.io.File. Quello di destra (originale) utilizza le parentesi graffe anche in situazioni in cui non è strettamente necessario al fine di favorire la leggibilità del codice.*

Codice sconsigliato

```
public File(String parent, String child) {
    if (child == null)
        throw new NullPointerException();
    if (parent != null)
        if (parent.equals(""))
            this.path = fs.resolve(fs.getDefaultParent(),
                fs.normalize(child));
        else
            this.path = fs.resolve(fs.normalize(parent),
                fs.normalize(child));
    else
        this.path = fs.normalize(child);
    this.prefixLength = fs.prefixLength(this.path);
}
```

Codice consigliato

```
public File(String parent, String child) {
    if (child == null) {
        throw new NullPointerException();
    }
    if (parent != null) {
        if (parent.equals("")) {
            this.path = fs.resolve(fs.getDefaultParent(),
                fs.normalize(child));
        } else {
            this.path = fs.resolve(fs.normalize(parent),
                fs.normalize(child));
        }
    } else {
        this.path = fs.normalize(child);
    }
    this.prefixLength = fs.prefixLength(this.path);
}
```

### 1.5.8 – Implementare metodi a minimo accoppiamento

In prima analisi, i metodi di ciascuna classe possono essere suddivisi in tre macro-categorie:

1. metodi che forniscono un servizio semplicemente elaborando i dati di input senza ricorrere all'utilizzo di altri dati e senza utilizzare lo stato dell'oggetto (per esempio in Java i metodi della classe `java.lang.Math`, come `Math.abs()`);
2. metodi che comunicano una porzione dello stato interno di un oggetto, oppure elaborano risultati dipendenti da esso, senza modificarlo (per esempio i famosi metodi `getX()`);

3. metodi che aggiornano lo stato interno di un oggetto (per esempio i metodi `setX()`).

Per quanto concerne la prima tipologia di metodi, essi presentano un accoppiamento nullo quando risultano privi di effetti collaterali (i famosi *side effects*), ottenuti mutando lo stato di un oggetto, e operano dunque esclusivamente sui parametri di input. Qualora questi metodi utilizzino altri dati, magari privati all'oggetto, di cui però si abbia strettamente bisogno, si ha ancora un accoppiamento minimo. Per quanto concerne i risultati generati, il metodo deve produrre unicamente un dato atomico o eventualmente un altro oggetto di cui è fornito il riferimento in memoria. Per mantenere un accoppiamento nullo, il metodo, durante la propria esecuzione, non deve poi delegare ad altri parte del proprio processo (non deve invocare altri metodi). Da quanto riportato, è evidente che non sempre un accoppiamento nullo è assolutamente indispensabile e desiderabile. Anzi spesso, sono accettabilissimi alcuni compromessi, purché l'accoppiamento resti minimo, magari al fine di soddisfare altre proprietà di qualità del software, come per esempio rendere i metodi più leggibili, manutenibili, riusabili, etc. magari derogando parte del proprio lavoro ad altri metodi.

Nel caso di metodi del secondo tipo, si ha un accoppiamento minimo quando il metodo, per generare i risultati della propria elaborazione, utilizza i parametri di input e accede ai soli attributi e metodi della classe (sia statici che non). Ancora una volta restituisce un valore atomico o un riferimento a un apposito grafo di oggetti o, eventualmente, genera un'eccezione per comunicare uno stato di errore. Metodi di questo tipo, pertanto, accedono allo stato dell'oggetto senza però modificarlo e utilizzano esclusivamente proprietà (metodi e attributi) della classe o dell'oggetto stesso.

Per i metodi dell'ultimo tipo, la materia non varia di molto. La differenza è che la propria esecuzione altera lo stato dell'oggetto. Chiaramente un accoppiamento minimo non prevede la variazione dello stato di altri oggetti.

## 1.6 Utilizzare correttamente l'ereditarietà

L'ereditarietà è probabilmente la legge più nota dell'OO e, verosimilmente, anche quella più abusata. Brevemente, l'ereditarietà è un meccanismo attraverso il quale un'entità più specifica incorpora struttura e comportamento definiti da entità più generali. Se da una parte è vero che, qualora utilizzata correttamente, essa permette una migliore ristrutturazione gerarchica (raggruppare il comportamento condiviso da più classi in una versione generalizzata incapsulata in un'apposita classe antenata), dall'altro bisogna tenere in mente che l'ereditarietà non è esente da controindicazioni come per esempio il forte legame di dipendenza che si instaura tra classe antenata e quelle discendenti (modifiche eseguite su una classe antenata tendono a ripercuotersi su tutte le classi discendenti) e la staticità e rigidità della struttura gerarchica. In particolare, una volta che una classe sia incastonata in questo tipo di organizzazione, non ne potrà più uscire.

### 1.6.1 Ereditare il tipo di una classe e non solo attributi/metodi

Utilizzare la relazione di ereditarietà per dar luogo ad una "specializzazione" dipendente dal tipo di una classe e non semplicemente per condividere pochi attributi e/o metodi. Con il tipo di una classe, brevemente, si intende la sua "interfaccia implicita" (elenco delle proprietà strut-

turali e comportamentali esposte ad altre classi). Il problema è che non è raro il caso di relazioni di ereditarietà realizzate semplicemente per "ereditare" opportune porzioni di implementazione. Ciò dovrebbe costituire una conseguenza e non un principio.

### 1.6.2 Non utilizzare l'ereditarietà per oggetti che possono "cambiare tipo"

L'ereditarietà presenta una serie di problemi qualora un'istanza di una determinata classe abbia necessità, in qualche modo, di "trasmutare tipo" durante il proprio ciclo di vita. In questi casi, un'alternativa migliore consiste nel rappresentare questo comportamento per mezzo di opportune versioni della relazione di associazione (composizione) e non con legami di generalizzazione. Per chiarire quanto espresso, si consideri l'esempio classico relativo alla classificazione dei ruoli degli "attori" di una compagnia aerea. In prima analisi alcuni programmatori sarebbero portati a definire una classe base, probabilmente astratta denominata *Person*, e quindi a specializzarla con classi del tipo *Pilot*, *Crew*, *Passenger*, etc. Questo è un chiaro esempio di errato utilizzo della relazione di ereditarietà: alcune entità possono cambiare il loro "tipo" durante il relativo ciclo di vita. Per esempio, un pilota frequentemente è anche un passeggero. Pertanto, una soluzione migliore consiste nel realizzare comunque la classe *Person*, questa volta concreta e associarla, attraverso apposita associazione, con una classe astratta denominata *Role*, le cui specializzazioni sono appunto le classi *Pilot*, *Crew*, *Passenger*, etc.

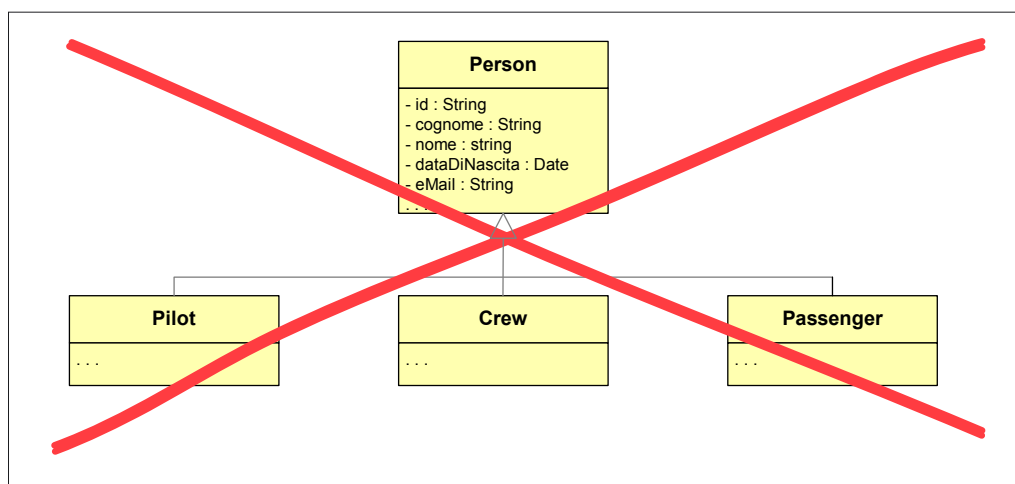


Figura 1.1 – Esempio di utilizzo errato della relazione di ereditarietà.

