

Un'introduzione al paradigma ad oggetti attraverso lo schema "Kernel-Modulo"

Matteo Baldoni

Dipartimento di Informatica
Università degli Studi di Torino
C.so Svizzera, 185 — I-10149 Torino (Italy)
Tel. +39 011 6706756 — Fax. +39 011 751603
E-mail: baldoni@di.unito.it
URL: <http://www.di.unito.it/~baldoni>

Abstract. La programmazione orientata agli oggetti ha avuto una notevole diffusione negli ultimi anni, questo è merito soprattutto del linguaggio Java e della sua vasta libreria di programmi disponibili. Molte delle classi presenti in queste librerie sono organizzate secondo uno schema ricorrente. Tale schema sfrutta completamente le caratteristiche principali dei linguaggi orientati agli oggetti, l'*astrazione sui dati*, il *binding dinamico*, il *polimorfismo*, l'*overriding* e l'*ereditarietà*. Un'analisi di questo schema è utile per acquisire una maggiore consapevolezza dell'uso dei linguaggi orientati agli oggetti e a comprendere perchè sia facile e conveniente realizzare librerie per tali linguaggi.

1 Lo schema Kernel-Modulo

Uno schema ricorrente dell'organizzazione di classi nelle librerie di Java è quello rappresentato nella Figura 1. Una classe (che chiameremo *ClasseKernel*) definisce

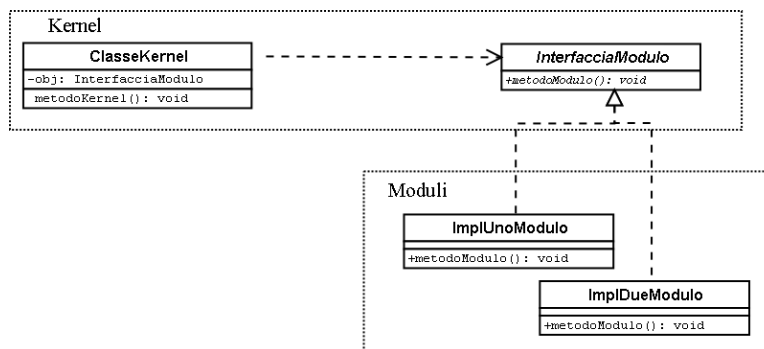


Fig. 1. Lo schema "Kernel-Modulo Dinamico".

un metodo (*metodoKernel*) nel cui corpo si utilizza un oggetto (*obj*) di un'altra

classe (*InterfacciaModulo*) invocando un suo metodo (*metodoModulo*). Molto spesso la classe dell'oggetto utilizzato, *InterfacciaModulo*, è un'interfaccia o una classe astratta implementata (estesa) da una o più classi (nella Figura 1, *ImplUnoModulo* e *ImplDueModulo*).

```
/* File: ClasseKernel.java */
public class ClasseKernel {
    private obj: InterfacciaModulo;
    public void metodoKernel() {
        ...
        obj.metodoModulo();
        ...
    }
    ...
}

/* File: InterfacciaModulo.java */
public interface InterfacciaModulo {
    public void metodoModulo();
}

/* File: ImplUnoModulo.java */
public class ImplUnoModulo implements InterfacciaModulo {
    ...
    public void metodoModulo() {
        ...
    }
    ...
}
```

Questo è lo schema seguito per la gestione degli eventi nella libreria Swing e per la realizzazione dei metodi per l'ordinamento di array e collezioni di oggetti in genere nella libreria Collection (si veda la Figura 2). Ad esempio, nella gestione dell'evento "pressione di un bottone", *ClasseKernel* è rappresentata dalla classe *JButton* dove *metodoKernel* è il metodo *notifyAction* che l'interprete di Java invoca automaticamente per notificare l'occorrenza dell'evento ad ogni oggetto registrato (*obj*) come ascoltatore di quel bottone (*ActionListener*, cioè il nostro *InterfacciaModulo*). Questo invoca a sua volta il metodo *actionPerfomed* (il nostro *metodoModulo*). Quest'ultimo metodo (più in generale l'interfaccia *ActionListener*) è poi effettivamente realizzato in una classe definita dal programmatore (il nostro *ImplUnoModulo*) che contiene il codice da eseguire in relazione alla pressione del bottone.

Chiameremo lo schema rappresentato in Figura 1 "Kernel-Modulo Dinamico" (*KMD* in breve) dove la combinazione delle classi *ClasseKernel* e *InterfacciaModulo* rappresentano il "Kernel" e le classi che implementano *InterfacciaModulo*, nell'esempio *ImplUnoModulo* e *ImplDueModulo*, rappresentano i "Moduli". Il Kernel può essere compilato in modo indipendente dai Moduli mentre per la

<i>ClasseKernel</i>	<i>metodoKernel</i>	<i>InterfacciaModulo</i>	<i>metodoModulo</i>
JButton	notifyAction	ActionListener	actionPerformed
Arrays	sort	Comparable	compareTo
Observable	notifyObservers	Observer	update

Fig. 2. Lo schema “Kernel-Modulo” nelle librerie di Java.

compilazione di un Modulo sono necessarie le informazioni contenuto nella classe *InterfacciaModulo*.

Lo schema Kernel-Modulo è utilizzato anche nella realizzazione di librerie in linguaggi non orientati agli oggetti, per esempio il linguaggio C. Si consideri una funzione che richieda una funzione esterna: in C il prototipo della funzione esterna (nome più tipo di dato restituito più tipi degli argomenti) è contenuto in un file header (“.h”), incluso dal programma principale. A un file header corrisponde un file sorgente (“.c”) contenente il codice della funzione esterna. Chiameremo questo schema “*Kernel-Modulo Statico*” (*KMS* in breve). Gli schemi grafici di schema KMD e KMS sono molto simili (si veda la Figura 3).

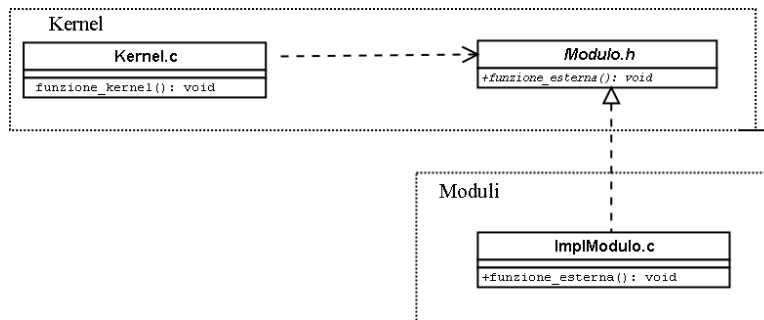


Fig. 3. Lo schema “Kernel-Modulo Statico”.

2 Astrazione sui dati, binding dinamico, polimorfismo, overriding ed ereditarietà

A differenza dei linguaggi procedurali, lo schema KMD nei linguaggi orientati agli oggetti contiene e sfrutta le caratteristiche principali della programmazione orientata agli oggetti: l'*astrazione sui dati*, il *binding dinamico*, il *polimorfismo*, l'*overriding* e l'*ereditarietà*. La combinazione di queste caratteristiche permette di perseguire l'obiettivo del riuso del software in maniera più semplice ed efficace rispetto al *KMS*. Vediamo come con un esempio.

La seguente classe definisce gli oggetti di tipo *Prisma*:



Fig. 4. A destra un parallelepipedo, a sinistra un cilindro. Per entrambi il volume è calcolato mediante la formula area di base per altezza.

```

/* File: Prisma.java */
public class Prisma {
    private Poligono base;
    private double altezza;
    public Prisma(Poligono base, double altezza) {
        this.base = base;
        this.altezza = altezza;
    }
    public double volume(){
        // Chiamata al metodo esterno area
        return base.area() * altezza;
    }
}

```

Un prisma è un oggetto che ha un *Poligono* come base ed un *double* rappresentante l'altezza. Il volume di un prisma è definito dal metodo *volume* come area della base per altezza. Si noti che a questo livello non indichiamo quale tipo di poligono costituirà la base di un prisma.

Il poligono alla base di un prisma è definito tramite un'interfaccia che richiede che ogni *Poligono* contenga un metodo di nome *area*, utilizzato per calcolare l'area del poligono stesso:

```

/* File: Poligono.java */
interface Poligono
{
    public double area();
}

```

Il nome della classe o dell'interfaccia può essere utilizzato come tipo nella dichiarazione di nuovi identificatori o come valore restituito da un metodo. Nell'esempio la variabile *base* nella classe *Prisma* è di tipo *Poligono*. La classe *Prisma* e l'interfaccia *Poligono* costituiscono il Kernel (vedi anche Figura 2).

Se desideriamo costruire un oggetto di tipo *Prisma* è necessario fornire la definizione concreta (un'*implementazione*) del tipo *Poligono*. Tale classe costitu-

isce un Modulo del nostro schema *KMD*. Ad esempio, la seguente classe definisce il poligono *Rettangolo*:

```
/* File: Rettangolo.java */
public class Rettangolo implements Poligono {
    private double base;
    private double altezza;
    public Rettangolo(double base, double altezza) {
        this.base = base;
        this.altezza = altezza;
    }
    public double area() {
        return base * altezza;
    }
}

/* File: UsaPrismi.java, prima versione */
public class UsaPrismi {
    public static void main(String args[]) {
        Prisma parallelepipedo =
            new Prisma(new Rettangolo(3, 4), 4);
        System.out.println(parallelepipedo.volume());
    }
}
```

Dopo aver compilato le varie classi, eseguendo la classe *UsaPrismi* (*java UsaPrismi*) avremo come output sul terminale il numero 48, cioè il volume del prisma con base un rettangolo (di base 3 per 4) e altezza 4 (in effetti, un parallelepipedo). A differenza dello schema *KMS* un Modulo *KMD* non è il semplice file sorgente per i prototipi dei metodi definiti dall'interfaccia nel Kernel, bensì il tipo introdotto dalla classe Modulo è in relazione di *sottotipo* (o tipo più specifico) con il tipo definito dall'interfaccia. Nell'esempio significa che il tipo *Rettangolo* è sottotipo di *Poligono* e quindi ogni oggetto di tipo *Rettangolo* è anche di tipo *Poligono*. Questo significa che il campo *base* di tipo *Poligono* nella classe *Prisma* può contenere anche un oggetto di tipo *Rettangolo* (più precisamente un riferimento ad un oggetto *Rettangolo*). Questa proprietà è nota come *polimorfismo*¹.

Si supponga ora di voler definire un cilindro, cioè un prisma con base un cerchio, in Java sarà sufficiente introdurre la seguente definizione di classe e modificare il *main* di *UsaPrismi* nel seguente modo:

```
/* File: Cerchio.java */
public class Cerchio implements Poligono {
```

¹ Più precisamente si fa riferimento al particolare tipo di polimorfismo detto di *inclusione*. Infatti dal punto di vista insiemistico l'insieme degli elementi di un tipo *B*, sottotipo di *A*, è *incluso* nell'insieme degli elementi di tipo *A*.

```

    private double raggio;
    public Cerchio(double raggio) {
        this.raggio = raggio;
    }
    public double area() {
        return raggio * raggio * Math.PI;
    }
}

/* File: UsaPrismi.java, seconda versione*/
public class UsaPrismi {
    public static void main(String args[]) {
        Prisma parallelepipedo =
            new Prisma(new Rettangolo(3, 4), 4);
        Prisma cilindro = new Prisma(new Cerchio(2), 4);
        // Nella seguente verra' utilizzato il
        // metodo area di Rettangolo!
        System.out.println(parallelepipedo.volume());
        // Nella seguente verra' utilizzato il
        // metodo area di Cerchio!
        System.out.println(cilindro.volume());
    }
}

```

Ad una nuova esecuzione della classe *UsaPrismi* l'output sarà il precedente valore 48 (il volume del parallelepipedo) ed il valore 50.24, cioè il volume del cilindro. La cosa interessante è che per calcolare questi valori si è utilizzato lo stesso metodo *volume* presente nella classe *Prisma* contenente la stessa identica chiamata al metodo esterno *area*. In altre parole, il metodo *area* descritto nell'interfaccia *Poligono* è polimorfo e può essere definito per più di una classe, assumendo all'occorrenza (come nell'esempio sopra) diverse implementazioni in ciascuna di esse (*overriding* o *sovrascrittura*).

L'effettivo codice del metodo *area* che viene eseguito viene determinato solo a tempo di esecuzione, cioè nel momento in cui potrà essere valutato l'effettivo tipo dell'oggetto puntato dalla variabile *base* in *Prisma*.

La differenza fondamentale tra lo schema KMD e KMS è quindi il momento in cui si effettua il legame tra il nome del metodo esterno (o funzione esterna) utilizzato all'interno del Kernel e l'effettivo codice da eseguire.

Tale legame, nel caso del linguaggio C, è realizzato dal compilatore, ovvero il *linker* nel caso che sia il Kernel che il Modulo siano già stati compilati separatamente. Nel caso dei linguaggi orientati agli oggetti la soluzione adottata è invece di ritardare il più possibile, in pratica a tempo di esecuzione, la risoluzione del legame tra il nome di un metodo e il suo codice.

Il vantaggio è che mentre nello schema KMS ad ogni header corrisponde un solo modulo, nello schema KMD ad ogni interfaccia o classe astratta possono corrispondere più implementazioni usate anche contemporaneamente, ottenendo una maggiore flessibilità.

Se vi sono più definizioni per lo stesso metodo esterno, sarà possibile scegliere quale codice effettivamente eseguire in base al tipo di oggetto su cui è invocato (nell'esempio, il metodo per il calcolo l'area del rettangolo o del cerchio) senza la necessità di un esplicito test nella classe Kernel. Si noti che è anche possibile aggiungere classi Modulo senza alcuna necessità di ricompilare le classi Kernel (nel nostro esempio abbiamo aggiunto la definizione della classe Cerchio). Tale meccanismo prende il nome di *binding dinamico* e si contrappone al *binding statico* dei linguaggi procedurali tipo C.

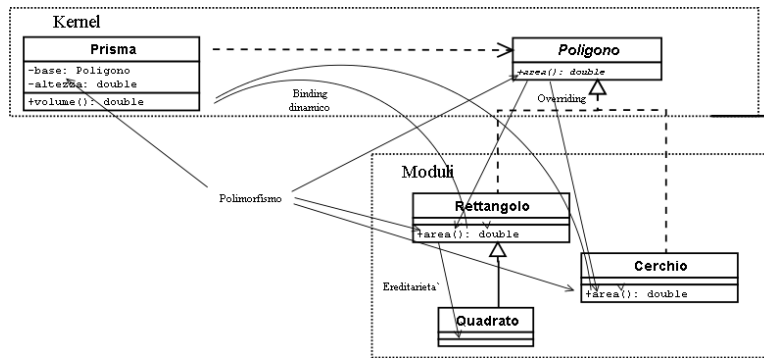


Fig. 5. L'astrazione sui dati, il binding dinamico, il polimorfismo e l'ereditarietà nello schema KMD.

Infine supponiamo di voler introdurre nel nostro esempio la definizione del poligono "quadrato". Un quadrato è un rettangolo in cui la base e l'altezza hanno la stessa lunghezza. In Java e in più generale nei linguaggi orientati agli oggetti è possibile definire classi come *estensioni* di altre classi senza la necessità di riscrivere il codice in comune ma semplicemente *ereditandolo*. Nell'esempio il metodo per il calcolo dell'area nella classe *Quadrato* è ereditato dalla classe *Rettangolo* (si veda la Figura 5). Possiamo così scrivere una terza versione del *main* di *UsaPrismi* introducendo un terzo prisma, un cubo:

```
/* File: Quadrato.java */
public class Quadrato extends Rettangolo {
    public Quadrato(double lato) {
        super(lato, lato);
    }
}
```

```

/* File: UsaPrismi.java, terza versione*/
public class UsaPrismi {
    public static void main(String args[]) {
        Prisma parallelepipedo =
            new Prisma(new Rettangolo(3, 4), 4);
        Prisma cilindro = new Prisma(new Cerchio(2), 4);
        Prisma cubo = new Prisma(new Quadrato(4), 4);
        System.out.println(parallelepipedo.volume());
        System.out.println(cilindro.volume());
        System.out.println(cubo.volume());
    }
}

```

3 Typechecking statico come documentazione interna al programma

La componente Kernel nello schema *KMD* può essere compilata separatamente dalla componente Modulo.

Solitamente la componente Kernel è fornita come libreria ed il programmatore completa lo schema fornendo il proprio o i propri Moduli. In Java spesso si ricorre al meccanismo di ereditarietà per stabilire un'interfaccia comune piuttosto che per ereditare effettivamente del codice.

A differenza di altri linguaggi orientati agli oggetti, come *Python*, il controllo dei tipi in Java è eseguito a tempo di compilazione (*typechecking statico*). Quindi durante la compilazione del Kernel il compilatore si preoccupa di verificare che l'oggetto *obj* di tipo *InterfacciaModulo* utilizzato nella classe *ClasseKernel* disponga effettivamente del metodo *metodoModulo* invocato su di esso. Nel caso che *InterfacciaModulo* sia un'interfaccia o una classe astratta, e quindi *metodoModulo* in tale classe è una semplice dichiarazione di segnatura (o prototipo), Java si riserverà di controllare durante la compilazione di una componente Modulo che *metodoModulo* venga effettivamente implementato in modo da garantire a tempo di esecuzione l'esistenza del codice da associare alla chiamata di *metodoModulo* nella classe Kernel.

L'uso dell'interfaccia *InterfacciaModulo* provvede una forma di *documentazione "interna al programma"*, i metodi che devono essere realizzati da una classe per potersi definire una componente Modulo per un certo Kernel sono quelli specificati nell'interfaccia stessa.

La corretta compilazione della componente Kernel garantisce che tutti i riferimenti a metodi esterni siano stati "documentati" nell'interfaccia che fa parte del Kernel mentre la corretta compilazione di una componente Modulo garantisce che quest'ultima fornisca un'implementazione per ogni metodo esterno utilizzato nel Kernel.

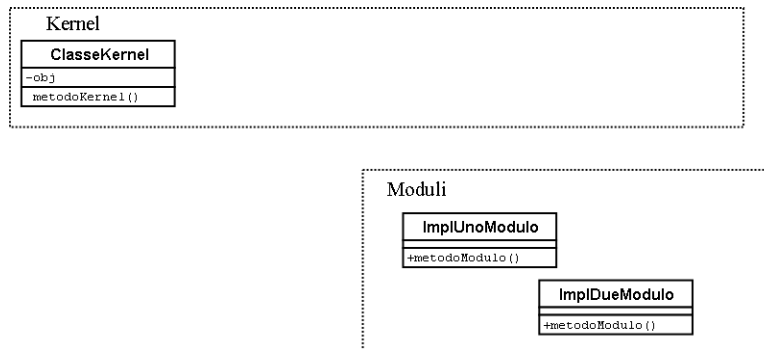


Fig. 6. Lo schema “Kernel-Modulo Dinamico” in Python.

In altri linguaggi orientati agli oggetti, come ad esempio Python, il controllo dei tipi è effettuato a tempo di esecuzione e così pure il controllo dell’esistenza di un metodo associato ad un certo oggetto (*typechecking dinamico*). Per questo motivo i progettisti di Python non hanno sentito la necessità di introdurre un costrutto del linguaggio per definire interfacce o classi astratte. Nel caso di Python lo schema *KMD* si presenta come in Figura 6, cioè priva dell’interfaccia *InterfacciaModulo*:

```
# File: ClasseKernel.py
class ClasseKernel:
    ...
    def metodoKernel():
        ...
        (self.obj).metodoModulo()
        ...
    ...

# File: ImplUnoModulo.py
class ImplUnoModulo:
    ...
    def metodoModulo():
        ...
    ...
```

Le classi Modulo non condividono più un’interfaccia comune ed in generale non è garantita alcuna relazione tra esse. L’unica relazione che dovrebbe accumunarle è l’implementazione del metodo esterno utilizzato in *ClasseKernel* che però è descritto, documentato, esclusivamente all’interno del codice stesso della classe *ClasseKernel*. L’unico modo per aiutare i realizzatori delle classi Modulo è di fornire una documentazione che accompagni il codice del Kernel, una documentazione “*esterna al programma*” che descriva le signature dei metodi richiesti dal Kernel.

In Python, all'atto dell'esecuzione non c'è nessuna garanzia che l'implementazione di un certo Modulo rispetti le richieste del Kernel e quindi nessuna garanzia che a tempo di esecuzione non si verifichino errori.

Il controllo statico dei tipi permette invece di eliminare in fase di compilazione questa classe di errori, assai frequenti in fase di sviluppo.

4 L'altra faccia della medaglia: il downcasting

Nelle sezioni precedenti abbiamo sottolineato la natura polimorfa delle variabili definite in Java: una variabile di un certo tipo può contenere i riferimenti ad oggetti dello stesso tipo o di un sottotipo (ma mai di un sovratipo!).

Si consideri lo schema *KMD* supponendo che la classe Modulo *ImplUnoModulo* disponga oltre all'implementazione del metodo esterno *metodoModulo* utilizzato in *ClasseKernel* anche del metodo *altroMetodo* il cui prototipo non è presente in *InterfacciaModulo* (si veda la Figura 7). Nella precedente sezione

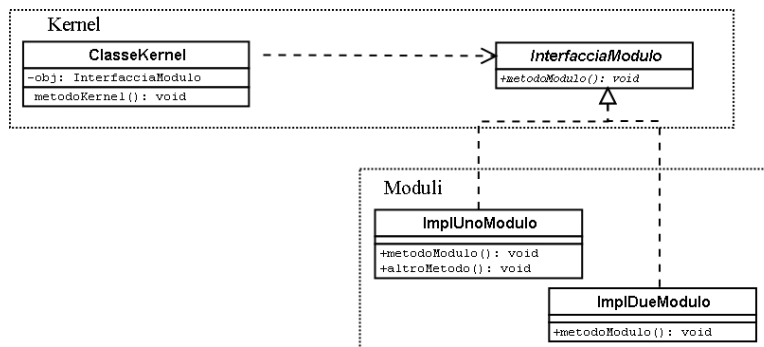


Fig. 7. Lo schema “Kernel-Modulo Dinamico”, una prima variante.

abbiamo chiarito il fatto che in Java viene verificata la validità di una invocazione di un metodo su di un oggetto durante la compilazione. Questo fa sì che non conoscendo effettivamente il tipo di oggetto a cui una certa variabile farà riferimento durante l'esecuzione, l'unico criterio accettabile per verificare la correttezza della chiamata sarà quello di basarsi sul tipo (statico) della variabile stessa. Consideriamo il seguente frammento di codice di un metodo della *ClasseKernel*.

```
/* File: ClasseKernel.java */
public class ClasseKernel {
    public InterfacciaModulo obj;
    ...
    public void unMetodoInKernel() {
        ...
    }
}
```

```

        // quello qui di seguito e' un upcast
        obj = new ImplUnoModulo();
        // la seguente istruzione genera un errore
        // a tempo di compilazione
        obj.altroMetodo();
        // quello qui di seguito e' un downcast
        ((ImplUnoModulo)obj).altroMedoto();
        ...
    }
    ...
}

```

Nell'esempio l'istruzione `obj.altroMetodo()` causerà un errore di compilazione poichè la variabile `obj` ha tipo `InterfacciaModulo` e non esiste nessun metodo di nome `altroMetodo` per tali tipo di oggetti (sebbene sia facile capire dal codice del programma che durante l'esecuzione tale variabile farà sicuramente riferimento ad un oggetto di tipo `ImplUnoModulo`). In questo caso sarà necessario "rassicurare" il compilatore che a tempo di esecuzione quella data variabile conterrà un oggetto di tipo `ImplUnoModulo` effettuando un *downcast* nel seguente modo:

```
((ImplUnoModulo)obj).altroMetodo();
```

ritardando a tempo di esecuzione il controllo di tipo².

Il downcast può essere considerato come un modo per ovviare taluni limiti del typechecking statico che, ovviamente, non può conoscere cosa avverrà a tempo di esecuzione introducendo una sorta di typechecking dinamico localizzato.

Consideriamo un altro esempio che dimostra come in Java non tutti gli errori di tipo possono essere catturati dal typechecker statico. Il seguente frammento di codice presenta un possibile metodo della `ClasseKernel`. Supponiamo che `InterfacciaModulo` sia una classe anziché una interfaccia (è quindi possibile creare istanze, oggetti, di essa), in questo caso la classe `ImplUnoModulo` estende la classe `InterfacciaModulo` anziché implementarla:

```

/* File: ClasseKernel.java */
public class ClasseKernel {
    public InterfacciaModulo obj;
    ...
    public unAltroMetodoInKernel() {
        ...
        // Un array di elementi di tipo InterfacciaModulo

```

² Si noti che nel caso che se a tempo di esecuzione la variabile `obj` non farà riferimento ad un oggetto di tipo `ImplUnoModulo` verrà segnalato un errore (sollevata un'eccezione di tipo `ClassCastException`).

```

        // che a tempo di esecuzione saranno
        // effettivamente di tipo ImplUnoModulo
        InterfacciaModulo[] array = new ImplUnoModulo[10];
        // Staticamente il tipo di array[0] e'
        // InterfacciaModulo ma dinamicamente
        // avra' tipo ImplUnoModulo e la seguente
        // istruzione solleva un'eccezione a run-time!!
        array[0] = new InterfacciaModulo();
    }
    ...
}

```

Nel metodo *unAltroMetodoInKernel* viene dichiarata una variabile *array* di tipo “array di elementi di tipo *InterfacciaModulo*” ma sarà inizializzata, a tempo di esecuzione, con un oggetto “array di dieci elementi di tipo *ImplUnoModulo*”. Questo è possibile poichè in Java gli array sono oggetti e un array di oggetti di tipo *ImplUnoModulo* è anche un array di oggetti di tipo *InterfacciaModulo* poichè *ImplUnoModulo* estende *InterfacciaModulo*. L’istruzione di assegnamento al primo elemento di *array*:

```
array[0] = new InterfacciaModulo();
```

a tempo di compilazione non desta nessun problema poichè il compilatore assume che ogni elemento dell’array stesso sia una variabile di tipo *InterfacciaModulo*. Purtroppo però, in fase di esecuzione, verrà sollevata un’eccezione di tipo *ArrayStoreException* perchè quella che si credeva, a tempo di compilazione, una variabile di tipo *InterfacciaModulo* risulta a tempo di esecuzione una variabile di tipo *ImplUnoModulo* e non adatta quindi a contenere riferimenti ad oggetti di sopraclassi (si veda la Figura 8). Questo tipo di errore non è evitabile con

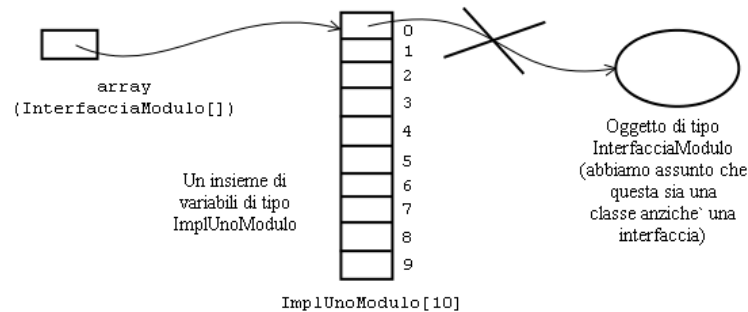


Fig. 8. Il problema degli array in Java.

un cast di qualche tipo. Infatti, siamo in presenza di una sorta di dichiarazione dinamica di variabili tipate: quando si crea un oggetto di tipo array di elementi di un certo tipo *A* è come se si creassero dinamicamente (a tempo di esecuzione)

un certo numero di variabili del tipo A . Il typechecker statico non può quindi tenerne conto.

5 Typechecking statico: sovraccarico (overloading) e ridefinizione (overriding)

In Java il controllo dei tipi è effettuato a tempo di compilazione (typechecking statico), questo significa che il compilatore si preoccupa di verificare che un certo oggetto disponga effettivamente del metodo invocato su di esso. Il controllo avviene sulla base del tipo della variabile che conterrà il riferimento all'oggetto e non sul tipo effettivo dell'oggetto stesso (eventualmente più specifico, cioè un sottotipo) che sarà noto solo a tempo di esecuzione. Se l'oggetto puntato dalla variabile durante l'esecuzione ha tipo più specifico rispetto al tipo della variabile stessa e per esso è disponibile una ridefinizione del metodo associato durante la compilazione, il meccanismo del binding dinamico assicura che quest'ultimo sia quello utilizzato. In altre parole il binding dinamico permette di scegliere a tempo di esecuzione tra le possibili *ridefinizioni* di un metodo quella da applicare. Il punto chiave per comprendere e riuscire a sfruttare bene dal punto di vista pratico questo meccanismo è capire cosa si intende esattamente per ridefinizione.

In Java, a differenza di altri linguaggi orientati agli oggetti, un metodo di una sottoclasse ridefinisce un metodo di una sua sovraclassa se ha il suo stesso nome e identica lista di tipi di parametri.

Per approfondire questo tema, consideriamo un'altra semplice variante nello schema *KMD* precedentemente illustrato. Supponiamo che il metodo esterno *metodoModulo* utilizzato in *ClasseKernel* abbia un parametro formale c di tipo *ClasseUno* e che la classe *Modulo ImplUnoModulo* disponga, oltre che dell'implementazione del metodo esterno *metodoModulo* con parametro di tipo *ClasseUno*, anche di un altro metodo avente sempre nome *metodoModulo* ma con un parametro di tipo *ClasseDue* e il cui prototipo non è presente in *InterfacciaModulo*. Tecnicamente si dice che il metodo *metodoModulo* in *ImplUnoModulo* è sovraccarico (*overloaded*), si veda la Figura 9.

Supponiamo, infine, che *ClasseDue* sia un'estensione di *ClasseUno* (*ClasseDue* è sottotipo di *ClasseUno*):

```
/* File: ClasseUno.java */
public class ClasseUno {
    ...
}

/* File ClasseDue.java */
public class ClasseDue extends ClasseUno {
    ...
}
```

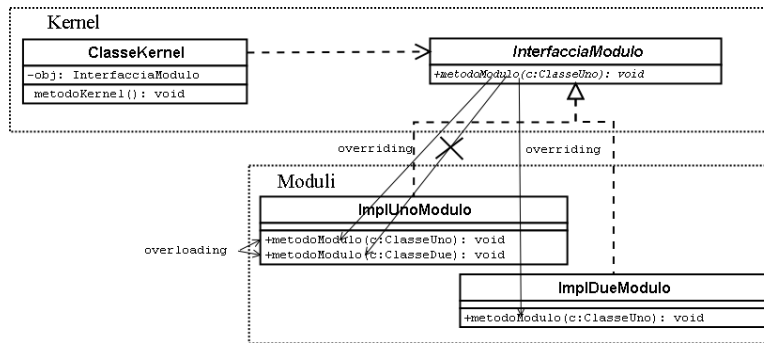


Fig. 9. Lo schema “Kernel-Modulo Dinamico”, un’altra variante.

e consideriamo l’istruzione *obj.metodoModulo(c)* nel seguente frammento di codice di un metodo della *ClasseKernel*:

```

/* File: ClasseKernel.java */
public class ClasseKernel {
    public InterfacciaModulo obj;
    ...
    public metodoKernel() {
        ClasseUno c;
        ...
        obj.metodoModulo(c);
        ...
    }
    ...
}

```

In fase di compilazione il typechecker verificherà che essendo *obj* di tipo *InterfacciaModulo* e *c* di tipo *ClasseUno*, esista una definizione del metodo *metodoModulo* che si applica al caso, cosa che per l’esempio è vera in quanto esiste proprio un metodo con quel nome e con parametro formale di tipo *ClasseUno* nella definizione di *InterfacciaModulo*. Osserviamo che nel nostro caso esistono ben due implementazioni dell’interfaccia in questione, quindi vi sono due differenti implementazioni di *metodoModulo* applicato a *ClasseUno*. A tempo di esecuzione il meccanismo di binding dinamico farà sì che il codice associato alla chiamata sia quello del metodo ridefinito nella classe del tipo dell’oggetto puntato da *obj*, ad esempio in *ImplUnoModulo*. Ma cosa succede se il parametro attuale della chiamata avesse tipo *ClasseDue*?

```

/* File: ClasseKernel.java */
public class ClasseKernel {
    public InterfacciaModulo obj;
    ...
    public metodoKernel() {

```

```

        ClasseDue c;
        ...
        obj.metodoModulo(c);
        ...
    }
    ...
}

```

Sarebbe ragionevole aspettarsi che durante l'esecuzione, venga associato alla chiamata il codice del metodo *metodoModulo* avente parametro formale di tipo *ClasseDue*. Questo però non accade e il metodo selezionato sarà quello dell'esempio precedente. Il motivo è semplice, questo secondo metodo nella classe *ImplUnoModulo* non è una ridefinizione del prototipo specificato in *InterfacciaModulo* e quindi il binding dinamico non la considera. L'unica maniera per effettuare l'associazione desiderata sarebbe effettuare un esplicito downcast durante la chiamata:

```
((ImplUnoModulo) obj).metodoModulo(c)
```

In questo modo fin dalla fase di compilazione il metodo con parametro formale di tipo *ClasseDue* risulterebbe associato alla nostra chiamata. Un'alternativa a questa soluzione "sporca" (che tra l'altro non è consigliabile come soluzione visto che costringe alla modifica diretta del codice del Kernel!) consiste nell'introduzione della segnatura di questo nuovo metodo nell'interfaccia *InterfacciaModulo*; questo risolverebbe il problema senza la necessità di un downcast specifico al tipo della classe *Modulo* utilizzata.

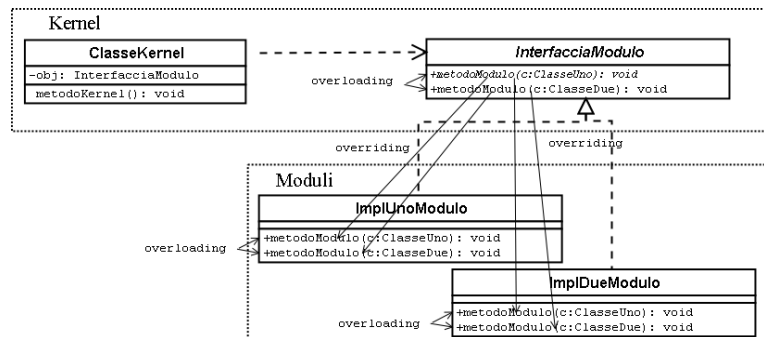


Fig. 10. Lo schema "Kernel-Modulo Dinamico", un'altra variante ancora.

Complichiamo ulteriormente la situazione al fine di capire esattamente come agisce il binding dinamico in Java e come questo è in relazione con il typechecking statico. Nella Figura 10 è presentata una ulteriore variante dello schema in cui i metodi esterni utilizzabili nella *ClasseKernel* e definiti in *InterfacciaModulo* sono due, entrambi hanno lo stesso nome ma tipi di parametri diversi (*overloading*). I

due metodi definiti in *InterfacciaModulo* differiscono tra di loro dalla presenza di un parametro di tipo *ClasseUno* per il primo e un parametro di tipo *ClasseDue* per il secondo, dove la classe *ClasseDue* estende la classe *ClasseUno*. Ovviamente, in questo caso, ogni classe *Modulo*, come nell'esempio *ImplUnoModulo* e *ImplDueModulo*, dovranno necessariamente implementare entrambi i metodi che ridefiniranno i prototipi in *InterfacciaModulo* (*overriding*). Si noti che, seguendo quanto detto prima sul concetto di ridefinizione in Java, ognuno dei metodo nel *Modulo* ridefinisce il corrispondente prototipo in *InterfacciaModulo*, come mostrato in Figura 10. Consideriamo il seguente frammento di codice di un possibile metodo della *ClasseKernel*. In questo caso la variabile che conterrà il parametro attuale è di tipo *ClasseUno* ma a tempo di esecuzione sicuramente farà riferimento ad un oggetto di tipo *ClasseDue*.

```

/* File: ClasseKernel.java */
public class ClasseKernel {
    public InterfacciaModulo obj;
    ...
    public metodoKernel() {
        ClasseUno c;
        ...
        c = new ClasseDue();
        ...
        obj.metodoModulo(c);
        ...
    }
    ...
}

```

Quale codice viene associato a tempo di esecuzione alla chiamata dell'istruzione *obj.metodoModulo(c)*?

Il fatto che abbiamo indicato che l'oggetto effettivamente puntato dal parametro attuale *c* a tempo di esecuzione sia un oggetto di tipo *ClasseDue* non cambia il ragionamento presentato precedentemente. Durante la compilazione il compilatore determina il tipo del parametro attuale *c* della chiamata (*ClasseUno*) e il tipo della variabile *obj* (*InterfacciaModulo*). Quindi verifica che in *InterfacciaModulo* esista un metodo di nome *metodoModulo* con un parametro di tipo *ClasseUno* e questo viene trovato. A tempo di esecuzione, per il meccanismo del binding dinamico, si associa il codice contenuto in una possibile ridefinizione del metodo determinato a tempo di compilazione presente nella classe che definisce tipo a tempo di esecuzione dell'oggetto puntato da *obj*. Poichè per ridefinizione si intende un metodo con stesso nome e stessa lista di tipi di parametri il metodo associato a tempo di esecuzione sarà il metodo *metodoModulo* della classe *ImplUnoModulo* con parametro formale un oggetto di tipo *ClasseUno* (anche se intuitivamente avremmo pensato a quello con parametro formale un oggetto di tipo *ClasseDue*). Per capire quale metodo verrà usato bisogna considerare il tipo, a tempo di esecuzione, dell'oggetto su cui il metodo è invocato; quindi occorre cercare nella classe così determinata un metodo avente nome identico a

quello invocato e lista dei tipi degli argomenti corrispondente alla lista dei tipi determinati a tempo di compilazione per quella chiamata.

Il binding dinamico determina il codice da associare ad una chiamata di un metodo tenendo conto del tipo dell'oggetto su cui è invocato il metodo stesso durante l'esecuzione e del tipo statico associato ai suoi parametri attuali durante la compilazione.

Vi è quindi un differente trattamento per quello che è considerato il *parametro implicito* (l'oggetto a cui si invia il messaggio) e i *parametri espliciti* di un metodo. Tale differenza di trattamento può spesso causare problemi, infatti è come se avessimo una chiamata di una funzione *metodoModulo(obj, c)*, con riferimento all'esempio, in cui il tipo di *obj* è determinato dinamicamente (a tempo di esecuzione) e quello di *c* staticamente (a tempo di compilazione)! In Python, ad esempio, la scelta di non introdurre tipi espliciti e non permettere l'overloading porta a non avere tali problemi; tuttavia al contempo si perdono tutti i vantaggi, costituiti dall'eliminazione di tutta una serie di errori già a tempo di compilazione nonché della forma di documentazione "interna al programma" tramite l'uso delle interfacce.

Ringraziamenti

L'autore desidera ringraziare *Cristina Baroglio, Ferruccio Damiani e Luca Roversi* per i loro preziosi consigli durante la stesura di questo lavoro.

Bibliografia

1. Bruce Eckel, "*Thinking in Java*", Edizione Italiana, Apogeo, 2002.
<http://www.mindview.net/Books>
2. Bruce Eckel, "*Thinking in Python*", <http://www.mindview.net/Books>, 2001.
In preparazione.
3. Cay S. Horstmann, "*Concetti di informatica e fondamenti di Java 2*", Apogeo, 2000.
4. Cay S. Horstmann e Gary Cornell, "*Java 2: i fondamenti*", McGraw-Hill, 2001.
5. Cay S. Horstmann e Gary Cornell, "*Java 2: tecniche avanzate*", McGraw-Hill, 2000.
6. Meilir Page-Jones, "*Progettazione a oggetti con UML*", Apogeo, 2002.
7. Mark Pilgrim, "*Dive Into Python*". <http://diveintopython.org>
8. *Python*. <http://www.python.org>
9. Dave Schmidt, "*Programming Principles in Java: Architectures and Interfaces*", <http://www.cis.ksu.edu/schmidt/CIS200>