Appunti sulla documentazione di un progetto software object oriented in linguaggio Java

Marco Liverani*

Luglio 2006

1 Introduzione

Ogni progetto informatico è sicuramente incompleto fino a quando non viene corredato da una documentazione tecnica esauriente, tale da consentire ad altre persone di intervenire a diversi livelli sullo stesso progetto informatico, per modificarne il comportamento o, semplicemente, per comprenderne esattamente le finalità e gli aspetti architetturali e funzionali. L'importanza della documentazione che correda un progetto software diventa ancora maggiore nel momento in cui ci si prefigge l'obiettivo di rendere portabile e riusabile il software prodotto nell'ambito del progetto. Questa è una delle finalità principali per cui è stato sviluppato il paradigma della programmazione ad oggetti: garantire un livello elevato di riuso delle componenti software e semplificare il processo di *porting* del software su piattaforme differenti, isolando specifiche funzionalità o servizi all'interno di appositi package o classi progettate appositamente, in modo tale da poter modificare singole componenti del progetto, senza doversi preoccupare dell'impatto che tali modifiche possono avere sulle restanti componenti software.

La documentazione di un progetto *object oriented* avviene pertanto a diversi livelli, con punti di vista differenti a seconda dello specifico aspetto che si intende descrivere: i **diagrammi di flusso** (o *flow chart*), tipici della programmazione strutturata, consentiranno di documentare in modo efficace brevi algoritmi implementati come metodi delle classi del progetto; i **diagrammi delle classi** (o *class diagram*) permetteranno di elencare in modo sintetico le classi che compongono il progetto, mettendo in evidenza gli attributi che le caratterizzano ed i metodi resi disponibili per operare su di esse, ma consentendo anche di esplicitare le relazioni esistenti tra le classi stesse o il processo di generalizzazione e di estensione che viene implementato attraverso la realizzazione di sotto-classi. Con il diagramma delle classi si ottiene pertanto una fotografia "statica" del progetto *object oriented*: ne vengono esplicitate le componenti, ma non viene descritto il modo in cui tali componenti potranno interagire fra loro durante il ciclo di vita di ciascun oggetto ottenuto come istanza di una specifica classe. Per fornire quindi una vista "dinamica" sul progetto e sull'interazione tra le componenti, è possibile utilizzare i **diagrammi di sequenza** (o *sequence diagram*), che consentono di visualizzare efficacemente il ciclo di vita di ciascun oggetto ed i metodi con cui ogni singolo oggetto invia dei "messaggi" ad altri oggetti, determinandone la creazione, la rimozione da parte del *garbage collec-*

*E-mail: liverani@mat.uniroma3.it; ultimo aggiornamento: 12 luglio 2006

tor della *Java Virtual Machine* o, più comunemente, l'esecuzione di uno specifico metodo reso visibile ad altri oggetti della stessa classe o di classi esterne.

Un altro livello di documentazione del progetto software, elementare e spesso trascurato dai meno esperti, consiste nella adozione di uno stile di programmazione e di scelta dei nomi di variabili, metodi e classi che sia significativo e consistente con il contesto in cui si sta sviluppando un determinato software: spesso un nome "sbagliato" o inadatto, o retaggio di precedenti versioni di una determinata procedura, è ben più dannoso per la comprensione del software stesso e per la sua riusabilità e manutenibilità di un manuale scritto malamente. A questo proposito si rimanda all'ottimo testo di B. Kernighan e R. Pike [4].

Naturalmente la leggibilità del software può (e deve) essere migliorata utilizzando estensivamente la tecnica dei **commenti** inseriti direttamente nel codice sorgente del programma. Tali commenti dovranno contenere, oltre a riferimenti utili per identificare l'autore di una specifica procedura o di una sua successiva modifica ed integrazione, anche la versione e la data di ultima modifica della procedura stessa e tutte le "istruzioni" per poter utilizzare facilmente un determinato metodo o una classe richiamandoli da altri punti del medesimo programma. In linguaggio Java i commenti inseriti nel sorgente del programma assumono un connotato ancora più interessante, dal momento che con la *utility* `javadoc` è possibile produrre un documento informativo in formato ipertestuale HTML su una classe o su un intero package rielaborando automaticamente i soli commenti inseriti con una apposita sintassi all'interno del codice sorgente.

In questa breve nota sono riportati alcuni riferimenti pratici per la realizzazione di diagrammi delle classi in formato UML e per la generazione della documentazione automatica mediante `javadoc`. Per quanto attiene al formalismo grafico introdotto da UML si rimanda principalmente al testo di Fowler e Scott [3], mentre per i dettagli sull'utilizzo di `javadoc` si può fare riferimento al testo di Bertacca e Guidi [1] o alla "pagina di manuale" in linea per il comando `javadoc`[1].

2 Diagrammi delle classi

Unified Modeling Language è un complesso insieme di notazioni sviluppate a partire dagli anni '80 e consolidate in un apparato standard alla fine degli anni '90 con il nome, appunto, di UML. I principali artefici di UML sono Grady Booch, Ivar Jacobson e James Rumbaugh che, oltre a definire le linee guida di UML hanno anche stabilito i principi su cui si basa il *Rational Unified Process*, ossia un processo che delinea la modalità e la sequenza di attività e fasi che eseguite sequenzialmente e reiterate più volte, portano alla realizzazione di un progetto di sviluppo software complesso.

UML è dunque un paradigma con cui è possibile documentare con un "linguaggio grafico" alcuni degli aspetti più rilevanti di un progetto software *object oriented*. Il diagramma delle classi è uno strumento che consente di produrre un'immagine "statica" del progetto, identificando le classi che lo compongono e le relazioni che intercorrono tra di esse. Pertanto è uno strumento assai utile "a posteriori", per documentare un programma ormai realizzato, ma anche "a priori", per supportare il processo di progettazione e di identificazione delle componenti (classi, oggetti, package, ecc.) in cui è opportuno suddividere il programma che si intende realizzare.

[1] In ambiente UNIX è possibile eseguire il comando "`man javadoc`" per visualizzare una *pagina di manuale* sul comando `javadoc`.

Utilizzando UML ognuno può decidere quale sia il livello di dettaglio che intende adottare: non sempre infatti spingersi verso la descrizione più dettagliata è una scelta efficace. Specialmente nelle fasi preliminari della progettazione è più opportuno soffermarsi sulla elencazione degli oggetti e delle relazioni che intercorrono tra di essi, piuttosto che entrare nel merito dei loro attributi e dei metodi che espongono.

2.1 Classi, attributi e metodi

In estrema sintesi un diagramma delle classi consente di raffigurare le classi del progetto, eventualmente raggruppandole in uno o più package ed evidenziando per ciascuna classe il nome che la identifica, l'elenco degli attributi e l'elenco dei metodi. Per la rappresentazione di una classe si utilizza un riquadro rettangolare suddiviso verticalmente in tre aree: nella prima si riporta il nome della classe, nella seconda l'elenco degli attributi e nella terza l'elenco dei metodi (vedi Figura 1).

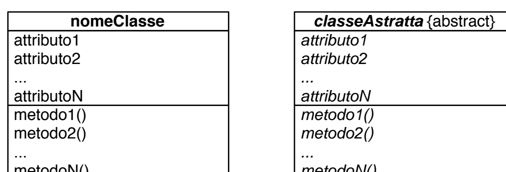

Figura 1: Il box con cui si rappresenta una classe nel diagramma delle classi

Naturalmente si può anche decidere di omettere l'elenco degli attributi o dei metodi, ovvero di spingersi nel livello di dettaglio fino a riportare la visibilità (*private*, *protected*, *public*, ecc.) e il tipo degli attributi e dei metodi. È legittimo anche riportare un elenco non esaustivo degli attributi e dei metodi di una classe, limitandosi ad elencare soltanto quelli interessanti nell'ambito di un diagramma in cui si vuole mettere in evidenza un particolare aspetto del progetto.

Se la classe è *astratta*, allora il nome sarà rappresentato in corsivo, eventualmente riportando la dicitura "`{abstract}`" accanto al nome.

2.2 Relazioni tra classi

È possibile collegare fra loro due o più classi attraverso apposite linee, creando così una sorta di grafo. In questo modo si rappresentano le relazioni tra oggetti che caratterizzano ogni progetto *object oriented*. È importante distinguere ed evidenziare le differenze tra diversi tipi di relazione che possono intercorrere tra le classi, osservando innanzi tutto che tali relazioni spesso non sono simmetriche: ad esempio una classe ne può estendere un'altra, ma allora non è vero il viceversa.

Le linee che collegano coppie di classi possono essere prive di verso, ovvero dotate di un verso rappresentato dalla punta di una freccia: in questo caso si vuole mettere in evidenza il fatto che una classe sa dell'esistenza dell'altra (e la utilizza), ma non è vero il viceversa. Ad esempio rappresentando con due classi distinte il concetto di grafo e quello di vertice, è facile immaginare che mentre la classe `grafo` conosce ed utilizza gli oggetti della classe `vertice`, gli oggetti di quest'ultima classe non conoscono e non utilizzano la classe `grafo`. Possiamo rappresentare questa re-

lazione come in Figura 2; in questo caso si dice che il diagramma è *navigabile* dalla classe `grafo` verso la classe `vertice`.

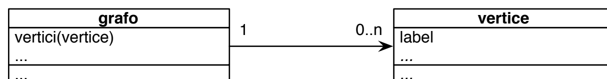

Figura 2: La classe `grafo` utilizza gli oggetti della classe `vertice`

Come si vede nel diagramma rappresentato in figura sugli archi che collegano le classi è possibile anche esprimere la "molteplicità" della relazione: nell'esempio i numeri riportati sullo spigolo che collega la classe `grafo` alla classe `vertice` indicano che le due classi sono legate da una relazione "1 a molti"; ogni grafo infatti può contenere 0 o più vertici, mentre un vertice può appartenere ad un unico grafo.

Una relazione molto comune tra le classi è quella della **associazione**: in questo caso una classe definisce un oggetto che è composto da uno o più elementi di un'altra classe. Facendo riferimento all'esempio precedente possiamo dire che un oggetto della classe `grafo` è costituito, tra le altre cose, da zero o più elementi della classe `vertice`. L'associazione può essere "forte" o "debole", a seconda che il ciclo di vita dell'oggetto contenitore determini o meno il ciclo di vita dell'oggetto contenuto. Nel primo caso, quando la relazione di associazione è forte, si parla di **composizione**: sullo spigolo che collega le due classi si aggiunge un rombo pieno, sull'estremo dal lato della classe contenitore; nel secondo caso, quando l'associazione è debole, si parla di **aggregazione**: questo tipo di relazione viene evidenziato ponendo un rombo vuoto (bianco) sull'estremo dal lato della classe contenitore.

Ad esempio potremmo ipotizzare di progettare un insieme di classi per implementare algoritmi che operano sui grafi, in modo tale che il ciclo di vita di un oggetto della classe `vertice` sia determinato completamente dal ciclo di vita dell'oggetto della classe `grafo`. In questo caso rappresenteremo la relazione di tipo "forte" che intercorre tra le due classi con lo spigolo dotato di un verso, di una molteplicità e con un estremo contrassegnato da un rombo pieno (vedi Figura 3).

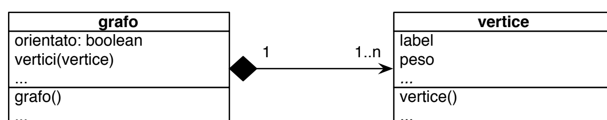

Figura 3: La classe `grafo` aggrega uno o più oggetti della classe `vertice`

Mediante gli spigoli che collegano due o più classi è possibile evidenziare l'ereditarietà tra le classi stesse. Questa relazione è evidenziata con una punta di freccia triangolare bianca posta sull'estremo dello spigolo incidente sulla classe "base"; da questa punta triangolare possono uscire uno o più spigoli che collegano la classe base con le classi derivate. Facendo riferimento ancora una volta ad una ipotetica implementazione di classi per la rappresentazione di grafi e vertici, potremmo definire la classe base `vertice` e la classe derivata `verticePesato` che ne eredita le proprietà aggiungendo un attributo relativo al peso associato al vertice.

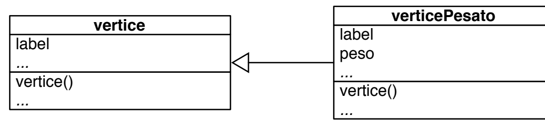

Figura 4: La classe `verticePesato` estende la classe `vertice`

Possiamo dire che esiste una **dipendenza** fra due elementi se una modifica ad uno di essi ha un impatto sull'altro. In generale è bene limitare, per quanto possibile, le dipendenze fra le classi. Le dipendenze si rappresentano con una linea tratteggiata con una freccia orientata dalla classe dipendente verso quella da cui questa dipende (vedi Figura 5).

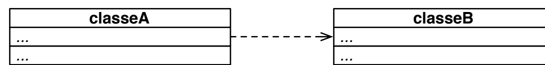

Figura 5: `classeA` dipende da `classeB`

In uno stesso diagramma è possibile rappresentare classi che appartengono a package differenti; in questi casi, per rappresentare l'appartenenza di una o più classi al medesimo **package** si utilizza un box con una "linguetta" per identificare il package e si tracciano delle linee che uniscono le classi al package di cui queste fanno parte, oppure racchiudendo le classi nel box del package che le contiene (vedi Figura 6).

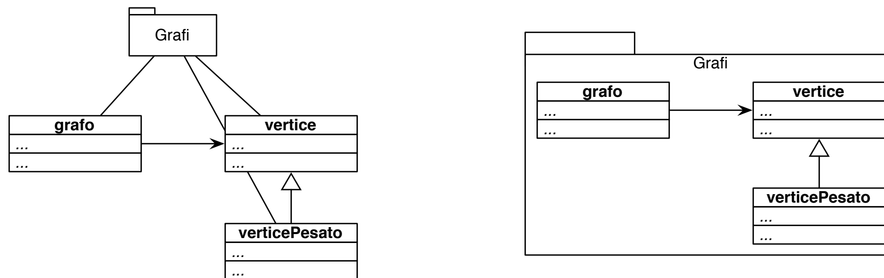

Figura 6: Due diverse modalità per indicare che le classi fanno parte del package `Grafi`

Se le dipendenze tra le classi sono numerose, allora, per rendere più leggibile il diagramma, è possibile rappresentare la dipendenza tra una o più classi di un determinato package dalle classi di un altro package, riportando un'unica linea tratteggiata tra i box che identificano i due package. Se i package coinvolti nel progetto sono numerosi, è possibile rappresentare le relazioni in un diagramma in cui sono rappresentati solo i package (e non le classi) e le relazioni esistenti tra di essi.

Infine è possibile inserire delle **note** e dei **commenti** all'interno del diagramma delle classi, per chiarire meglio alcuni aspetti del diagramma stesso. Tipicamente i

commenti sono inseriti all'interno di un rettangolo con un angolo "piegato" ed una linea collega il commento all'elemento del diagramma cui si riferisce (vedi Figura 7).

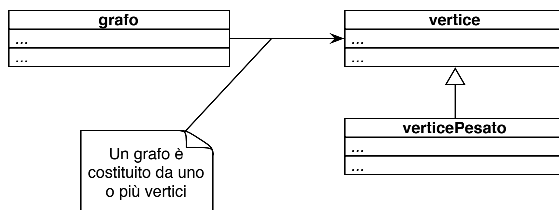

Figura 7: Un commento inserito per chiarire la relazione esistente tra due classi

3 Javadoc

Il programma `javadoc` è una *utility* distribuita insieme al Java Development Kit (JDK) per consentire ai programmatori di produrre facilmente della documentazione ipertestuale sulle classi realizzate nell'ambito di un determinati progetto software. Di fatto `javadoc` è un programma che legge in input uno o più file con il codice sorgente scritto in linguaggio Java, ne interpreta i commenti introdotti con una apposita sintassi, e produce in output dei file in formato HTML con la documentazione relativa alle classi presenti nel programma sorgente.

Il programma può essere eseguito per generare la documentazione relativa ad un singolo file sorgente Java, ad un intero package o a più package contemporaneamente. L'output prodotto è un insieme di file HTML con lo stesso formato e la stessa struttura della documentazione standard delle API Java disponibile *on-line* ed allegata al JDK.

3.1 Esecuzione del programma

Il programma `javadoc` non ha un'interfaccia utente grafica a finestre, ma è costituito da un eseguibile che deve essere lanciato con un comando da una *shell* UNIX o dal "prompt dei comandi" di Microsoft Windows. L'uso del comando è estremamente semplice:

```
javadoc [opzioni] [packages] [sorgenti]
```

La forma più semplice prevede semplicemente l'indicazione del nome del package sulla riga di comando, senza alcuna opzione e senza specificare il nome di nessun file sorgente. In questo modo sarà lo stesso `javadoc` ad individuare nel CLASSPATH la directory con i file del package da cui estrarrà tutti i sorgenti Java da elaborare. L'output viene prodotto di *default* nella directory corrente.

Il programma ammette numerose opzioni per le quali si rimanda alla documentazione presente nella pagina di manuale del comando stesso. In questa sede ci limitiamo ad osservare che l'opzione "`-d`" seguita dal *path* di una directory, consente di specificare la directory di output in cui collocare i file HTML prodotti da `javadoc`; l'opzione "`-sourcepath`", seguita dal *path* di una directory, consente di

specificare la directory in cui si trovano i file sorgente, se diversa dalla directory corrente. Ad esempio, supponiamo di voler produrre la documentazione ipertestuale di tutte le classi che compongono il package `grafi`, i cui file sorgente si trovano in `/home/marco/src/grafi` e supponiamo di voler collocare la documentazione nella directory `/home/marco/doc/grafi`. In questo caso il comando sarà il seguente:

```
javadoc -d /home/marco/doc -sourcepath /home/marco/src grafi
```

Il comando produrrà una sottodirectory “`grafi`” nella directory `/home/marco/doc` (se non esiste, sarà creata anche questa directory); in `/home/marco/doc/grafi` saranno creati automaticamente una serie di file in formato HTML ed alcune sottodirectory. Per accedere alla pagina iniziale della documentazione, è sufficiente aprire all'interno di un *web browser* il file “`index.html`”.

3.2 Commenti per la generazione della documentazione

La documentazione viene prodotta sulla base di alcuni dei commenti inseriti nei file sorgente ed in base alla dichiarazione di classi, variabili e metodi. I commenti interpretati da `javadoc` devono iniziare con la sequenza “`/**`” e, naturalmente, devono chiudersi con la sequenza “`*/`”. I commenti su linea singola inseriti con la sintassi “`//`” sono ignorati da `javadoc`. Ad esempio un commento valido che sarà utilizzato da `javadoc` per produrre la documentazione è il seguente:

```
/**
 *   Riga di commento che puo' contenere ogni
 *   tipo di carattere ed anche i tag HTML per
 *   marcare alcune porzioni di testo.
 */
```

I commenti devono precedere immediatamente la dichiarazione dell'elemento a cui si riferiscono: la dichiarazione di una variabile, la dichiarazione di un metodo o di una classe.

I commenti possono contenere dei tag HTML per formattare il documento prodotto come output e possono contenere alcuni comandi identificati dal simbolo “`@`” seguito da una parola chiave, che verrà interpretato da `javadoc` per evidenziare con una notazione standard uno specifico elemento nella documentazione (ad esempio l'autore di un file, la versione, la data di ultima modifica, ecc.).

Di seguito riportiamo una breve descrizione di alcuni dei principali tag interpretati da `javadoc`:

`@author` ***`nome autore`*** Indicazione del nome dell'autore; può essere impiegato esclusivamente nella documentazione a livello di package, di classe o di interfaccia; es.: “`@author Marco Liverani (liverani@mat.uniroma3.it)`”. Il tag `@author` viene ignorato se sulla linea di comando non viene utilizzata l'opzione “`-author`”.

`@param` ***`parametro descrizione`*** Descrizione di un parametro nella chiamata di un metodo; es.: “`@param primo Riferimento al primo elemento della lista`”.

`@return` *`variabile descrizione`* Descrizione del valore restituito da un metodo; es.: "`@return max Massimo elemento del vettore`".

`@see` *`riferimento`* Citazione di documenti o articoli utili, riferimento ad altre risorse o ad altri package o classi; per riferirsi ad altri elementi documentati con `javadoc` si deve utilizzare come riferimento una stringa composta con la sintassi "`package.classe#metodo`" o anche "`package.classe#campo`"; questo tipo di riferimento può essere anche relativo a metodi e campi della classe corrente (in questo caso si possono omettere il nome del package e della classe), o ad una classe del medesimo package (si può omettere il nome del package); se la descrizione del riferimento non è costituita dal nome di un package, di una classe, di un metodo o di un campo, allora la descrizione deve essere riportata tra virgolette; es.: "`@see java.math`", oppure "`@see "Kernighan, Pike, <em>The Practice of Programming</em>"`".

`@throws` *`eccezione descrizione`* Consente di descrivere l'eccezione lanciata dalla classe; es.: "`@throws IOexception Errore di input/output`".

`@version` *`versione, gg/mm/aa`* Indicazione della versione del file o della porzione di codice sorgente; può essere impiegato esclusivamente nella documentazione a livello di package, di classe o di interfaccia; la versione in genere è un numero (es.: 1.5), separato con una virgola dalla data di rilascio, nel formato "gg/mm/aa"; es.: "`@version 1.5, 11/07/06`". In genere, partendo dalla versione 0 si aumenta di un punto decimale il numero di versione ogni volta che viene rilasciata una correzione o una integrazione minore, mentre si aumenta di un'unità il numero di versione ad ogni rilascio molto significativo; la prima versione definitiva del software è la 1.0, mentre le versioni minori di 1 rappresentano software in versione preliminare (versioni "beta"). Questo tag viene ignorato se sulla linea di comando non viene utilizzata l'opzione "`-version`".

Per concludere ed illustrare come possano essere utilizzati i tag di `javadoc` all'interno dei commenti, riportiamo un brevissimo esempio che documenta un metodo di una classe, aggiungendo la descrizione dei parametri e le indicazioni per altre classi correlate alla presente:

```
/**
 *  Costruttore della classe. Vengono impostate a zero le
 *  coordinate per la rappresentazione grafica sul piano.
 *
 *  @param a Label del nuovo vertice (valore del campo info).
 *  @see java.util.Vector
 */
public vertice(int a) {
   ...
}
```

Per inserire dei commenti nella documentazione a livello di package si deve creare un file sorgente Java denominato "`package-info.java`" nella directory del package stesso. In questo file si potranno inserire esclusivamente i commenti che si desidera aggiungere con la medesima sintassi vista in precedenza; non si deve aggiungere a questo file alcuna istruzione reale in linguaggio Java.

Riferimenti bibliografici

[1] Marco Bertacca, Andrea Guidi, *Introduzione a Java*, McGraw-Hill, 2000.

[2] Judy Bishop, *Java gently – Corso introduttivo*, Addison-Wesley, 1998.

[3] Martin Fowler, Kendall Scott, *UML Distilled*, Addison-Wesley, 2000.

[4] Brian W. Kernighan, Rob Pike, *The Practice of Programming*, Addison-Wesley, 1999.

[5] Sun Microsystems, *Javadoc Tool*, Sun Developer Network, documentazione on-line disponibile su Internet all'indirizzo http://java.sun.com/j2se/javadoc/.