WHITE PAPER

Java manuale di stile

Massimiliano Bigatti
max@bigatti.it

Pubblicato da Massimiliano Bigatti
max@bigatti.it
http://www.bigatti.it

Giugno 2003

 Si ringranzia Emanuele Giuliani (emanuele@giuliani.mi.it) per il supporto.

Contenuti

Introduzione

Perché un altro manuale di stile?

Su Internet e nelle librerie si possono trovare molti articoli o libri dedicati alle convenzioni da utilizzare per scrivere programmi, sui diversi stili da utilizzare o a suggerimenti per rendere più robusto il codice. Ci si potrebbe dunque chiedere il perché di un altro manuale di stile. In realtà, la documentazione che è possibile reperire è varia, per certi versi frammentata, e, cosa più importante, scritta in inglese per il mondo inglese. Molte convenzioni infatti devono essere "adattate" alla lingua che si utilizza per l'applicazione, perché in fin dei conti la maggior parte dei progetti poi parla italiano con i suoi utenti. Questa "impedenza", tra il linguaggio naturale dei linguaggi di programmazione, l'inglese, e, nel nostro caso, l'italiano, spinge a cercare convenzioni di codifica se non alternative, per lo meno integrative di quelle disponibili in rete.

Cosa troverete in questo testo

E' anche una risposta alla domanda "cosa è uno stile": uno stile di programmazione tratta di *come* scrivere il codice, e non di *cosa* scrivere. Alcune convenzioni prensenti in rete, come [3], oppure le stesse convenzioni di SUN [1], fondono i due concetti, proponendo stili di sviluppo e trucchi di programmazione all'interno dello stesso testo. Ad esempio, la determinazione del nome di una variabile è un aspetto

legato allo stile, come pure la posizione delle parentesi graffe in un ciclo; il suggerimento di utilizzare asserzioni, oppure su come scrivere una classe singleton sono aspetti legati al *cosa* scrivere. Questa confusione può anche nascere per "colpa" di uno dei libri migliori sull'argomento programmazione: *Code Complete* di McConnell [libro1]. Il libro dovrebbe essere letto da chiunque debba mettere mano a del codice, ed illustra un numero enorme di suggerimenti sul come scrivere correttamente i programmi; e dedica un paragrafo allo stile. Questi aspetti sono al di fuori dello scopo di questo testo, anche se potranno essere presi in considerazione in futuro oppure in un testo "gemello".

> **Noi per stile intendiamo solamente la formattazione del sorgente e la scelta dei nomi.**

Le indicazioni che trovate in questo testo sono il frutto dell'esperienza e della ricerca eseguita sulla documentazione reperibile in rete, elencata nella sezione "Bibliografia".

Perché stile?

Lo scopo di uno stile è quello di rendere il più possibile leggibile un programma (concetto differente rispetto all'*interpretabilità*). Applicare uno stile coerente all'interno della propria applicazione permettere di cogliere il funzionamento e l'impostazione del sistema a colpo d'occhio, riducendo gli sforzi necessari alla sua comprensione, e , di conseguenza, i costi.

Uno stile definitivo?

Le convenzioni di scrittura del codice sono spesso argomento che suscita guerre di religione; non è mia intenzione sollevarne di nuovi ed anzi, quanto qui descritto è questo emerso nella mia personale esperienza, ed è suscettibile di miglioramenti e di punti di vista diversi. E' per questo motivo che per ogni convenzione viene riportata la *ratio*, e cioè il ragionamento che sta dietro ad una scelta, in modo che chiunque possa valutare in prima persona la bontà della stessa.

Organizzazione del testo

In modo omogeneo rispetto all'organizzazione gerarchica del linguaggio Java, il testo affronta per primo lo stile dei package, poi delle classi, dei metodi ed attributi, e della struttura dei metodi.

Convenzioni

Il codice d'esempio contiene sia esempi di errori, che di corretta implementazione. Gli errori sono rappresentati con parole ~~barrate~~. Ad esempio:

```
~~sbagliato~~
~~sbagliato~~
giusto
```

In generale

Alcune regole generali che si applicano a nomi di package, classi, metodi, parametri ed attributi sono le seguenti:

Nomi descrittivi

Utilizzare nomi che descrivano lo scopo dell'elemento. Ad esempio, una classe che accede ad un database per leggere informazioni sul cliente:

```
~~RecuperoDati~~
CaricatoreDatiCliente
```

Oppure, in caso di un metodo che ritorna la descrizione di un oggetto

```
~~public String ritornaStringa();~~
public String getDescrizione();
```

Nomi pronunciabili

Utilizzare nomi facilmente pronunciabili, e di uso comune, che non abbiano doppie in posti critici.

Abbreviazioni

Non utilizzare abbreviazioni, ma parole complete. Ad esempio, negli attributi:

```
~~float valMedCamp;~~
float valoreMedioCampione;
```

Lunghezza medio-corta

Buoni nomi non sono necessariamente nomi lunghi; quelli migliori sono infatti di medio-corta lunghezza, ma sufficientemente descrittivi. Ad esempio, per le classi:

```
Cliente
ElencoProdotti
DocumentoPratica
```

Non sono invece molto leggibili:

```
ClienteCensitoNelWeb
DocumentoWorkflowStrutturatoMisto
```

Convenzioni esistenti

E' buona norma riutilizzare nomi che sono già in uso nella comunità informatica. Ad esempio, la libreria di base di Java fornisce delle buone linee guida. Un'altra fonte interessante sono i Design Pattern: nel momento che se ne implementi qualcuno nel sistema, le classi che lo realizzano devono utilizzare i nomi generici descritti nella documentazione del pattern stesso.

Nomi con senso positivo

E' più leggibile un nome espresso in positivo, che uno che contiene una negazione. Ad esempio:

```
~~boolean nonTrovato;~~
boolean trovato;
```

Omogeneità

In nomi utilizzati all'interno dell'applicazione dovrebbero essere il più omogeneo possibile. Se si comincia ad utilizzare un certo insieme di termini, si dovrebbe proseguire con quelli per tutta l'applicazione. Ad esempio, è sbagliato utilizzare il termine Prodotto come nome di classe e poi magari sviluppare un metodo nella classe Ordini che recita:

```
public void addArticolo( Prodotto prodotto );
```

è più corretto:

```
public void addProdotto( Prodotto prodotto );
```

Ottimizzazioni

Una regola principe nell'informatica è: "non ottimizzare in anticipo" (premature optimization is the root of all evil). Le ottimizzazioni del codice non devono comunque inficiare la leggibilità dello stesso, i quanto è solitamente meno costoso acquisire hardware più potente che realizzare codice estremamente complesso. In questo caso il costo della manutenzione è sicuramente superiore (le persone costano di più delle macchine). Ad ogni modo, questo non significa che non deve essere eseguita nessuna ottimizzazione: se si è costretti a sviluppare codice molto illeggibile, perché le estreme prestazioni sono indispensabili (non è la norma nelle applicazioni per gli affari), è perlomeno necessario documentarlo adeguatamente.

Package

Il nome del package dovrebbe essere univoco, scritto in tutte lettere minuscole e dovrebbe iniziare da domini di livello più alto (come gov, com, it). Ogni componente del nome è separato da un punto (.). Il nome dovrebbe includere il proprio dominio Internet, in modo da assicurarne l'univocità a livello di azienda. Ad esempio:

```
it.bigatti.iSafe
com.sun.internal
```

Classi

Una classe è contenuta all'interno di un file con estensione .java e segue questa struttura:

- commento con l'intestazione del file;
- dichiarazione del package;
- sezione import;
    - package interni;
    - package estensioni;
    - package libreria di terze parti;
    - package applicativi;
- dichiarazione della classe;
    - attributi pubblici;
    - attributi privati;
    - attributi protetti;
    - altri attributi;
    - costruttori;
    - altri metodi;
- classi interne;

Nel file è obbligatoria solo la definizione della classe; il resto degli elementi è opzionale. Ciascun elemento è separato dagli altri da una riga vuota, come ciascun metodo, ad esempio:

```
package it.bigatti.application;

import java.util.*;

```

```
class Cliente extends AbstractElemento
      implements Serializable {

      public static Cliente _instance;
      int id;
      //…

      public Cliente() {
      }

      public int getID() {
            return id;
      }

      public String getRagioneSociale() {
            return ragioneSociale;
      }
}
```

In merito all'implementazione:

- **lunghezza righe.** La lunghezza massima di una riga di codice è di 80 caratteri. Questo consente di evitare lo scorrimento orizzontale del testo, che richiede tempo ed affatica la lettura;
- **lunghezza file.** Il file sorgente non dovrebbe eccedere le 2000 righe; file più lunghi sono difficoltosi da scorrere e gestire;
- **indentazione.** L'indentazione deve essere di 4 spazi. In molti linguaggi (VB, C) si utilizza l'indentazione ad 8 spazi, mentre altri strumenti addirittura utilizzano solo due caratteri. Una buon compromesso è quindi 4 spazi. Bisogna anche tenere presente che in Java, per la sua organizzazioni a classi, utilizza sempre almeno una indentazione per la dichiarazione della stessa;

- **spazi e non tabulazioni.** Gli editor espandono le tabulazioni ad una quantità diversa di spazi: 2, 4 o 8. Per evitare che un codice ben formattato, appaia in modo errato su altri editor, è necessario utilizzare gli spazi al posto dei tab. Questo non significa che è necessario utilizzare la barra spaziatrice per allineare il codice: alcuni editor convertono in automatico le tabulazioni in spazi, rendendo possibile utilizzare comunque il tasto “tabulazione”.

Sezione import

La sezione relativa all’importazione dei package e delle classi dovrebbe seguire queste regole:

- **ordine alfabetico.** I package dovrebbero essere inseriti in ordine alfabetico;
- **raggruppati.** I package dovrebbero essere raggruppati come segue:
  - package della libreria di base (java.*);
  - package delle estensioni standard (javax.*);
  - package delle librerie aggiuntive (p.e. org.apache.*, org.w3c.dom.*);
  - package dell’applicazione che si sta sviluppando;
- **separati.** Tra un gruppo e l’altro deve essere presente una riga vuota;

Esempio:

```
import java.io.*;
import java.sql.*;
import java.text.*;
import java.util.*;

import java.xml.parsers.*;
import javax.xml.transform.*;

import org.s3c.dom.*;
import org.xml.sax.*;
import org.xml.sax.ext.*;

import it.bigatti.iSafe.*;
import it.bigatti.util.*;
```

Quanto importare

In merito allo stile da utilizzare per i singoli import, esistono alcune alternative:

- **nessuna importazione.** Alcuni suggeriscono di non utilizzare nessun import e di specificare per ciascuna classe il package completo, come `java.util.List`. Questa soluzione richiede di scrivere molto codice, e non utilizza una parola chiave del linguaggio, allontanandosi da esso;
- **importazione singola.** L'importazione delle singole classi consente di avere un controllo fine su cosa viene importato e quindi su quali classi l'applicazione si basa. Sebbene questo sia un suggerimento volto a rendere più semplice il codice da leggere da parte del programmatore inesperto, su classi mediamente complesse richiede la specifica di una sezione import troppo lunga, che riduce la leggibilità. Il programmatore

deve poi comunque riferirsi alla documentazione per sapere che cosa rappresenta una classe: il fatto di conoscere subito a che package appartiene, non è un grosso vantaggio, quando si può effettuare una ricerca nell'elenco completo delle classi di Javadoc.

- **importazione completa.** Questa tipologia comporta l'importazione di tutte le classi di un package (p.e. con import `java.util.*`) e consente di ridurre la sezione degli import snellendo il programma. Le collisioni tra i nomi infatti non si verificano così spesso da richiedere una importazione della singola classe.

Dichiarazione della classe

La dichiarazione della classe dovrebbe essere disposta su più righe, in modo da limitarne l'ampiezza. Una buona soluzione è spezzare la riga prima della parola chiave `implements`:

```
class ElencoProdotti extends AbstractElenco
      implements Storeable, Cacheable
```

L'indentazione della seconda riga è di 8 spazi;

Nomi

Il nome di una classe è composto da una serie di *sostaintivi* concatenati, la cui iniziale è scritta in maiuscolo. Ad esempio:

```
Cliente
ElencoClienti
```

Come accennato, uno dei problemi principali è legato all'utilizzo di lingue diverse dall'inglese. I linguaggi di programmazione utilizzano infatti esclusivamente questo linguaggio, anche se poi sono utilizzati in tutto il mondo.
Il nome dovrebbe essere corto ma descrittivo e se utilizza acronimi dovrebbero essere solo molto diffusi (come URL ed http).

Tutto in inglese

Una possibilità è dunque quella di scrivere tutto il programma in inglese, uniformandosi alla maggioranza anglofona. Questo comporta però dei problemi, in quanto non tutti hanno una conoscenza approfondita di questa lingua, che però è indispensabile per realizzare programmi complessi. E' facile trovare parole inglesi per descrivere concetti semplici, oppure ricorrere saltuariamente al dizionario per trovare quelle che non si conoscono, quando però si affrontano progetti complessi ci si scontra con alcuni problemi:

- **argomenti verticali**. Il progetto potrebbe trattare di elementi funzionali molto particolari, come l'erogazione di un mutuo, oppure la gestione delle promozioni commerciali, o del controllo statistico. Lo sviluppatore medio non conosce il vocabolario specifico di questo segmento applicativo (e spesso non lo conosce il traduttore professionista). Questo aspetto costringe a tenere al proprio fianco un dizionario;
- **complessità eccessiva**. Anche utilizzando un dizionario, chi crea una determinata classe, con un nome esotico, deve poi condividere questa parte del programma con il resto del gruppo

di lavoro che si troverebbe con lo stesso problema – non capirebbe cosa vuol dire quel particolare termine inglese. Si finirebbe a dotare ciascun sviluppatore di un dizionario, sempre considerando che lo scopo di un progetto, normalmente, è lo sviluppo dell'applicazione e non l'apprendimento di una lingua straniera;

- **disomogeneità**. L'interfaccia utente dell'applicazione rivolta al mercato italiano è sviluppata in italiano, dunque si viene a creare nel sistema una impedenza tra il modello dell'applicazione e l'interfaccia utente. Se nell'interfaccia si parla di cliente, mentre nel dominio si trova un Customer, la leggibilità del sistema nella sua integrità ne risulta minata;
- **rapporto con il cliente**. Anche nel rapporto con il cliente questa soluzione comporta dei problemi, in quanto vengono utilizzati termini differenti. Per esempio il cliente potrebbe parlare di "Impiegato" mentre nell'applicazion è implementata una classe "Employee". Questo caso può sembrare anche semplice, ma si consideri un intero progetto: diviene necessario riferirsi agli stessi concetti con due insiemi di termini differenti.

Come si nota, lo sviluppo completamente in inglese risulta in un procedimento goffo. Inoltre, è bene porre attenzione a questo aspetto all'inizio del progetto: si consideri infatti di cominciare lo sviluppo di un'applicazione per il calcolo statistico cominciando a creare entità come *mean* (media) o *standard deviation* (deviazione standard), si potrebbe arrivare ad un punto dove il dominio applicativo è pieno

zeppo di termini esotici e sconosciuti, complicando la proseguzione dello sviluppo.

Tutto in italiano

Una alternativa è dunque quella di scrivere tutte le classi in italiano; si presuppone infatti che questa lingua sia sufficentemente nota in Italia. Anche questa soluzione è problematica, perché porta a soluzioni goffe. Ad esempio, si immagini di creare una lista linkata ottimizzata, si potrebbe chiamare:

- **ListaOttimizzata.** Il problema di questo nome è che è in conflitto con lo standard delle strutture dati Java, presenti in java.util (p.e. ArrayList, LinkedList);
- **OttimizzataList.** Questo nome segue la convenzione delle API Java, ma che significa? L'ordine delle parole è tale da rendere il nome senza senso;
- **OptimizedList.** Questo è un nome corretto, che però utilizza un nome inglese.

Uno dei problemi con i nomi di classe in italiano è dunque il conflitto con la libreria di base di Java.

Soluzione mista

Il compromesso migliore che ho potuto trovare è il seguente: utilizzare termini completamente in italiano per le classi funzionali e completamente in inglese per quelli più a basso livello.
Per classi funzionali si intendono quelle classi relative al dominio dell'applicazione, come Cliente, Fornitore, Ordine; le classi tecniche a

livello inferiore possono riguardare il caching, l'accesso al database o all'infrastruttura Web (servlet/JSP). Questa differenziazione rende indispensabile la netta separazione tra classi di business (funzionali) e tecniche.

Un esempio di classe completamente funzionale:

```
class Cliente {
	int id;
	String ragioneSociale;

	//…
	String getId() {
		return id;
	}
}
```

Una classe completamente tecnica:

```
class Cache {
	Map elements = hew HashMap();

	//…
	public Cacheable get( String id ) {
		return (Cacheable)elements.get( id );
	}
}
```

Ovviamente le due tipologie ad un certo momento entreranno in comunicazione tra di loro:

```
class Ordine implements Cacheable {
	String id;
	static Cache _cache;

	//…
```

```
      static Ordine getOrdine( String id ) {
            Ordine ordine = _cache.get( id );
            if( ordine == null ) {
                  ordine = new Ordine( id );
                  _cache.put( ordine );
            }
            return ordine;
      }
}
```

La differenza di lingua aiuta a distinguere le classi funzionali da quelle di business.

> **Il ragionamento comporta il fatto che il gruppo di lavoro sia composto da sviluppatori che conoscano bene l'italiano e quanto basta l'inglese. Questo può non accadere ove esistano gruppi di lavoro che impiegano personale straniero; in tal caso sono necessari interventi correttivi, come ad esempio affidare lo sviluppo delle parti completamente in inglese a questi sviluppatori.**

Sfruttare i package

Il nome della classe non deve essere troppo lungo. Il package di appartenenza deve fornire il contesto utile per ridurne il nome. Ad esempio:

```
~~NodoPratiche~~
~~PraticheTreeModel~~
elencoPratiche.Nodo
elencoPratiche.TreeModel

~~NodoProposte~~
~~ProposteTreeModel~~
elencoProposte.Nodo
elencoProposte.TreeModel
```

I nomi troppo lunghi riducono la leggibilità del codice, e sono comunque indice di un non ottimale utilizzo del linguaggio, in quanto i package dovrebbero definire un contesto entro cui agire. I nomi delle classi dovrebbero dare solo l'elemento distintivo finale.

Parti standard del nome

Se necessario, è possibile indicare che una determinata implementazione è quella di default utilizzando il termine "Default". Ad esempio:

```
DefaultGruppo
DefaultUtente
```

Le classi astratte iniziano sempre per Abstract:

```
AbstractCliente
AbstractProdotto
```

Le classi che utilizzano il pattern model-view-controller devono terminare con un postfisso standard, che identifica la tipologia del componente. Ad esempio:

```
ListModel
TreeModel
PlainView
DefaultListController
```

Le eccezioni terminano invece sempre per Exception:

```
UtenteNonTrovatoException
GiacenzaInsufficienteException
```

Classi di elenco

Le classi che rappresentano un elenco di elementi devono avere un elemento distintivo. Una possibilità è utilizzare il plurale:

```
Clienti (elenco clienti)
Prodotti (elenco prodotti)
```

Questo approccio però tende a creare classi dal nome troppo simile a quello della classe di cui contengono l'elenco, dove cambia tendenzialmente una sola lettera. Una soluzione migliore è l'utilizzo del termine "Lista" o "Elenco". Ad esempio:

```
ElencoClienti
ListaProdotti
```

Interfacce

Il nome di una interfaccia in Java riflette le regole adottate per le classi. Non utilizzare prefissi o convenzioni particolari come:

```
~~IAccessibile~~
~~AccessibileInterface~~
```

ma:

```
Accessibile
```

Metodi

Anche per quanto riguarda i metodi è bene utilizzare metodi in italiano per le classi funzionali e metodi in inglese per quelle tecniche, ma si aggiungono problemi di impedenza, soprattutto in merito alle convenzioni per i Javabeans.

Nomi

Per i metodi normali infatti il problema non sussiste, si consideri ad esempio i seguenti metodi tutti in italiano:

```
elaboraRisultato();
caricaDati();
eseguiTransazione();
```

Sono sufficientemente leggibili, anche se generici (il contesto dovrebbe essere dato dalla classe in cui sono inseriti). I metodi dovrebbero essere verbi, dove ciascuna parola che compone il nome ha l'iniziale in maiuscolo, tranne la prima.

Javabeans

Le convenzioni per i componenti della piattaforma Java creano qualche problema. Queste prevedono infatti l'utilizzo di prefissi per indicare lo scopo di determinati metodi:

- **getXXX()**. Definisce un metodo che ottiene il valore di una proprietà (getter);
- **setXXX()**. Definisce un metodo che imposta il valore di una proprietà (setter);

Nel caso che il valore ritornato sia di tipo boolean, il getter non utilizza il prefisso "get", ma "is". Questo tipo di scelta è ovvio se si pensa alla maggiore leggibilità del codice, come ad esempio:

```
isWritable()
```

si legge "è scrivibile", mentre l'alternativa getWritable() non avrebbe reso la stessa leggibilità. In alternativa ad "is", le specifiche dei Javabeans indicano che è possibile utilizzare il prefisso "has", se il significato dell'attributo lo richiede, ad esempio:

```
hasChilds()
hasData()
```

Prefissi in italiano

Una possibilità è quella di tradurre in italiano i prefissi (get/set/is/has), ad esempio, ed utilizzare i metodi in italiano (tabella 1.1)

Tabella 1.1 – Esempio di prefissi in italiano

Inglese	Italiano
getCustomer()	ritornaCliente()
getID()	ritornaID()
setTimestamp()	impostaMomento()
setName()	impostaNome()
isWritable()	èScrivibile()
isUpdatable()	èAggiornabile()
isAvailable()	èDisponibile()
hasChild()	haFigli()
hasChild()	possiedeFigli()

Questa soluzione ha almeno due problemi distinti:

- **accentate.** L'italiano ci costringerebbe ad utilizzare caratteri accentati come "è". Sebbene Java sia un linguaggio UNICODE, spesso si decide di "non fidarsi", e di compilare i file sorgenti in ascii standard;
- **difformità.** Il codice realizzato è difforme dal codice Java esistente, come le librerie di base di Java o la parte più tecnica dell'applicazione, rendendo più difficoltosa le leggibilità del codice;
- **incompatibilità.** Le convenzione di codifica dei Javabeans sono state sviluppate per rendere utilizzabile un componente che ne segua i dettami all'interno di strumenti di sviluppo visuali. Sebbene questi strumenti abbiano un utilizzo non estensivo nello sviluppo Java, che spesso si attua lato server magari utilizzando

un editor, esistono anche altri strumenti che si basano su queste convenzioni, come analizzatori di codice o generatori.

Prefissi in inglese

In alternativa è possibile lasciare in inglese i prefissi, utilizzando l'italiano solo per la parte principale del nome (come detto, scrivere il metodo tutto in inglese non è un'opzione percorribile per metodi funzionali):

```
getCliente()
getID()
setTimestamp()
setNome()
isScrivibile()
isAggiornabile()
isDisponibile()
hasFigli()
```

Se si da per scontato l'utilizzo della convenzione per le classi sopra descritta, ci si trova presto, nelle classi funzionali, con metodi come:

```
class Ordine {
      Prodotto[] getProdotti() {
            //…
      }
}
```

L'accoppiamento tra le classi funzionali richiede dunque l'utilizzo di metodi che trattino direttamente di classi scritte in italiano. Attenzione all'utilizzo del singolare e del plurale: utilizzare il plurale se il metodo ritorna un elenco, sia questo un array, una lista o un enumeratore; se viene ritornato un oggetto singolo, viene utilizzato il singolare.

Parti standard del nome

Alcune volte il nome del metodo può essere costruito utilizzando altre convenzioni, utilizzando termini mutuati dal codice esistente nella libreria di base di Java, oppure suggerito da Design Pattern [libro2]. Ad esempio:

```
loadProdotto()
saveOrdine()

initializeCliente()
computeTotaleOrdine()
findProdotto()

addUtente()
removeUtente()
```

Anche in questo caso si hanno nomi ibridi, che richiedono un po' di attenzione all'inizio ma che poi aumentano la leggibilità.

Dichiarazione del metodo

In modo similare alle classi, i metodi vengono dichiarati anche su due righe, spezzando la prima riga prima della parola chiave throws. Ad esempio:

```
void loadProdotto( String gruppo, String id )
    throws ProdottoNonTrovatoException;
```

L'indentazione della seconda riga è di 4 spazi;

Se i parametri sono molti, è necessario spezzare l'elenco con righe indentate di 4 spazi:

```
public void saveProdotto( String gruppo, String descrizione,
    float costo, int quantita, boolean disponibile )
        throws AggiornamentoProdottoException;
```

Parametri

I parametri di un metodo dovrebbero rispettare una serie di convenzioni:

- **nomi descrittivi.** Il nome del metodo dovrebbe dare il senso del significato del parametro, come: codiceProdotto, tipoOperazione, ordineDalnoltrare;
- **variabili generiche.** In caso di variabili generiche, il nome della variabile è uguale a quello della classe, tranne per l'iniziale in minuscolo. Ad esempio:

```
aggiungi( Cliente cliente )
addMessaggio( Utente utente, Messaggio messaggio )
```

Valore di ritorno

Il valore da ritornare dovrebbe essere contenuto in una variabile, dichiarata ad inizio metodo e ritornata alla fine dello stesso. Ad esempio:

```
float getPrezzo() {
      float prezzo = 0.0;

      prezzo = costo - (costo * sconto) / 100;
      prezzo += prezzo * TabellaIVA.getValore( codiceIVA );

      //… altre elaborazioni
```

```
        return prezzo;
}
```

Non devono esistere altre istruzioni return a parte quella a fine metodo. Nel caso ne vengano utilizzati diversi, può non apparire chiaro, in una determinata esecuzione, quale di queste istruzioni venga invocata e di conseguenza risulta più complesso capire il codice.

Campi e variabili

Nomi

I nomi dei campi sono costruiti come i nomi dei metodi, concatenando nomi ed aggettivi, scrivendo in maiuscolo l'iniziale a partire dalla seconda parola. Ad esempio:

```
String descrizione;
int id;
String codiceProdotto;

List elencoProdotti;
```

Il nome deve rispettare due regole:

- **evitare la notazione ungherese.** La notazione ungherese, che prevede l'utilizzo di prefissi per descrivere il tipo di dato, non dovrebbe essere utilizzata. La motivazione è semplice: la notazione ungherese va bene per linguaggi che hanno tipi semplici, e dove è possibile creare un vocabolario di prefissi limitato. In linguaggi OOP i tipi primitivi hanno un uso più limitato, mentre sono gli oggetti a farla da padrone. Diventa dunque poco pratico, oltre che poco leggibile, utilizzare prefissi per ciascuna classe (ad esempio: prodottoUnProdotto, clienteUnCliente, ListUnaLista). A questo punto è meglio abbandonare la notazione ungherese anche per i tipi primitivi. Tra l'altro, mantenendo piccole le classi ed i metodi, il vantaggio di leggibilità di questa notazione è più limitato. Una sola eccezione riguarda le parti che utilizzano interfacce utente grafiche, dove è possibile implementare la notazione postposta, indicando il tipo

al termine del nome della variabile, ad esempio: confermaButton, nomeFileTextField o codiciProdottoList.

- **caratteri consentiti**. Sono a...z, A...Z, 0...9. Questo limite è dettato sia dalle specifiche del linguaggio, ma anche dalla necessità di creare nomi chiari e leggibili: l'utilizzo del solo alfabeto e dei numeri consente di raggiungere questo obbiettivo. Inoltre è possibile utilizzare l'underscore (_), sia nei nomi delle costanti (per separare le parole che compongono il nome), sia eventualmente come prefisso di variabili di istanza che potrebbero servire a scopi "interni" della classe.
- **posizione**. Dichiarare le variabili ad inizio blocco, sia questo un metodo o una classe, in modo da raccogliere in un unico punto tutte le dichiarazioni. L'unica eccezione è l'istruzione `for()`, descritta sopra.
- **una variabile per riga.** Utilizzare la dichiarazione per definire una sola variabile:

```
int numeroRighe, numeroColonne, numeroRecord;
int numeroRighe;
int numeroColonne;
int numeroRecord;
```

- **inizializzazione.** L'inizializzazione deve essere eseguita in fase di dichiarazione, impostando il valore di default o il risultato di un metodo. Se proprio non è possibile, in quanto il valore da impostare è il risultato di una elaborazione compiuta nel metodo stesso, inizializzare la variabile appena prima del suo utilizzo;

- **allineamento**. Allineare la dichiarazione delle variabili per renderle più leggibili, strutturandole in blocchi omogenei per contesto (e non per tipo di dato):

```
public int numeroRighe;
public int numeroColonne;

public Prodotto prodottoCorrente;
public Prodotto prodottoSuccessivo;

private List            listaProdotti;
private Map             mappaProdotti;
private DefaultCache    cacheProdotti;
```

- **spazi**. Utilizzare gli spazi per separare i diversi elementi in una espressione. Ad esempio:

```
int a = 5;
```

Costanti

Le costanti sono attributi marcati come final e che hanno nomi completamente in maiuscolo, eventualmente composti da più parole. Le singole parole sono separate da underscore (_). E' buona norma specificare almeno due termini, il primo definisce il contesto della costante. Ad esempio:

```
COLORE_ROSSO
PRIORITA_ALTA
```

Se esiste una sola tipologia di costanti all'interno di una specifica classe, è possibile omettere il prefisso, in quanto in contesto è già dato dalla classe:

```
Colore.ROSSO
Livello.MEDIO
```

Algoritmi

Dimensioni

La dimensione di un singolo algoritmo (o metodo) non dovrebbe eccedere la pagina, in modo che sia leggibile in una schermata e non richieda scorrimenti verticali. Dove l'algoritmo sia troppo complesso, eseguire un refactoring per separarlo in diversi sotto-metodi più semplici.

Blocchi di codice

Le parentesi graffe devono iniziare sulla riga relativa all'istruzione e terminare sulla colonna di inizio dell'istruzione stessa. Ad esempio:

```
if ( a == 5 ) {
    //…
}

while ( true ) {
    //…
}

do {
    //…
} while( !esci );
```

L'indentazione delle espressioni if dovrebbe essere di 8 spazi, in modo da staccare chiaramente dal codice all'interno delle graffe:

```
if( (prodotto.prezzo > Prodotto.prezzoMinimo ) &&
        prodotto.giacenza() != 0 ) {
```

```
    //…
}
```

Espressioni

Nelle espressioni, valutare la posizione di termine della riga, per spezzarla nel punto più appropriato, ad esempio in corrispondenza di parentesi:

```
float prezzoFinale = (prezzo
    - (prezzo * sconto) / 100);
```

Per quanto riguarda l'operatore ternario, esistono tre possibilità [1], la cui scelta dipende molto dalla lunghezza di *beta* e *gamma*:

```
alpha = (aLongBooleanExpression) ? beta : gamma;

alpha = (aLongBooleanExpression) ? beta
                                 : gamma;

alpha = (aLongBooleanExpression)
        ? beta
        : gamma;
```

Ogni riga dovrebbe contenere al massimo una espressione:

```
~~indiceProdotto++; elabora();~~

indiceProdotto++;
elabora();
```

Condizioni

L'istruzione if deve rispettare le seguenti regole:

- **sempre parentesi**. Utilizzare sempre le parentesi graffe, anche se segue solo una istruzione, allo scopo di ridurre eventuali errori di codifica nel momento che si desiderino aggiungere ulteriori istruzioni condizionate alla if. Inoltre, il blocco di codice risulta più leggibile nel momento che lo sviluppatore si abitua ad utilizzare questo standard.

Iterazioni

I cicli devono seguire le seguenti regole:

- **contatore interno.** Dichiarare il contatore direttamente nel ciclo, in quando si riduce l'area di attenzione che lo sviluppatore deve porre a questo aspetto del programma. Ad esempio: `for( int ii = 0; ii < 5; ii++ );`
- **contatori "cercabili".** Utilizzare i classici contatori "i" (derivati dalla prassi utilizzata dai matematici) per i cicli, complica le ricerche nel codice, in quanto la ricerca del carattere "i", ritorna sempre un numero elevato di occorrenze (vengono individuate anche le i delle istruzioni if, ad esempio). Per contro, contatori come indiceCiclo, sono troppo lunghi; una soluzione semplice è utilizzare contatori a due caratteri, come ii, kk, jj. Questa convenzione può essere applicata anche solo in classi di grossa dimensione;

Variabili

Le variabili dei metodi (ma le stesse regole si applicano alle variabili di instanza e di classe) devono rispettare le seguenti linee guida:

- **visibilità**. Utilizzare l'area di visibilità più stretta possibile, dichiarando le variabili appena prima del loro utilizzo. Questo consente di limitare il raggio di attenzione che lo sviluppatore deve porre.
- **semantica singola**. Una variabile non deve servire a due scopi, sebbene sia più compatto utilizzare una generica variabile contatore per due cicli diversi, è più leggibile dichiararne diverse in modo da specificare un nome più descrittivo. Ad esempio:

```
void controlla() {
      int numeroControlliFormali = 0;
      while( controlliFormali( numeroControlliFormali ) ) {
            numeroControlliFormali++;
      }
      setNumeroControlliFormali( numeroControlliFormali );

      int numeroControlliAggiuntivi = 0;
      while( controlliAggiuntivi( numeroControlliAggiuntivi ) ) {
            numeroControlliAggiuntivi++;
      }
      setNumeroControlliAggiuntivi( numeroControlliAggiuntivi );
}
```

Chiamata a metodi

Non utilizzare spazi dopo il nome del metodo. Ad esempio:

```
elabora (5,prodotto, “prova”);
elabora( 5, prodotto, “prova” );
```

Se l’elenco dei parametri è lungo, è possibile spezzarlo per renderlo più chiaro:

```
prodotto.aggiorna( gruppo, descrizione, costo, quantita,
      disponibilita, immagine );
```

o in alternativa:

```
prodotto.aggiorna(
      gruppo,
      descrizione,
      costo,
      quantita,
      disponibilta,
      immagine
);
```

La seconda versione è utile quando le variabili da passare siano lunghe oppure che il valore sia il risultato di una chiamata più complessa:

```
prodotto.aggiorna(
      gruppo,
      altroProdotto.getDescrizione(),
      Catalogo.getCatalogoDefault().getCosto( prodotto.id ),
      quantita,
      disponibilta,
      CommonUtils.loadImmagine( prodotto.id );
);
```

Documentazione

La documentazione nei programmi Java è redatta secondo lo standard Javadoc, che prevede la compilazione della documentazione direttamente all'interno del codice. In seguito, tramite il comando javadoc, questi vengono estratti e utilizzati per compilare la documentazione di progetto, solitamente in formato HTML. Per mantere un approccio agile allo sviluppo, è necessario trovare un bilanciamento tra la necessità di documentare, e quella di concentrarsi sullo sviluppo. Due sono le ulteriori forze in gioco: da una parte il codice deve essere utilizzato da terzi, sia per la manutenzione che un eventuale riutilizzo - e quindi ha la necessità di essere documentato – ma d'altra parte, la documentazione non deve offuscare il codice: la documentazione migliore è il codice ben scritto.

Alcune convenzioni:

- non documentare. Sembra un suggerimento assurdo ed al di fuori di ogni corrente di pensiero dominante. La versione estesa è però la seguente: non documentare subito le classi che vengono sviluppate, in quanto non è certo che la classe appena realizzata sopravviverà all'intereno dell'applicazione ed in quella forma. Un approccio agile è dunque quello di non documentare subito tutto, ma aspettare che una determinata porzione di codice sia consolidata. E' necessario però che congiuntamente si applichino le seguenti regole;

- **codice auto-documentante.** Il codice dovrebbe essere sempre il più possibile chiaro, in modo che il suo funzionamento sia ovvio senza la necessità di scrivere commenti;
- **codice complesso.** Codice particolarmente complesso dovrebbe essere riscritto e non commentato, tramite operazioni di refactoring. Ovviamente ciò non è sempre possibile, e dunque brevi commenti esplicativi sono indispensabili;
- **commento mirato.** Nei casi di codice particolarmente complesso e non ulteriormente semplificabile, oppure per quelle informazioni che lo sviluppatore ritiene di dover associare al codice, utilizzare un commento per descrivere l'informazione;
- **non offuscare.** La documentazione dovrebbe essere inserita in modo da non rendere più complessa la lettura del codice. Ad esempio:

```
Sbagliato:
if( a == 5 ) {
      /* commento */
      break;
}

Corretto:
/* commento */
if( a == 5 ) {
      break;
}
```

- **non commentare il linguaggio.** I commenti devono riportare informazioni sul perché viene eseguita una determinata operazione e non sulla sintassi del codice. Ad esempio:

```
Sbagliato:
```

```
i++; //incrementa i

Corretto:
i++; //elabora il prossimo cliente
```

- **italiano.** Nelle parti descrittive sarebbe buona norma utilizzare tutti termini italiani, per quanto possibile. Termini di uso comune inglesi, come file o directory possono essere mantenuti, mentre il plurale non utilizza la "s" finale (no files ma file) – queste convenzioni sono solitamente utilizzate negli articoli sulle riviste tecniche;
- **commenti cercabili.** Se una determinata porzione di codice richiede un intervento successivo, sia di ottimizzazione che di correzione, marcarlo con un commento opportuno, con un prefisso standard: `//TODO: descrizione`. La presenza della stringa costante "TODO" facilita la ricerca dei punti in sospeso.

Dove usare i diversi tipi di commenti

Se è necessario commentare una sola riga, utilizzare il commento a singola linea (//), sia che questo sia posizionato alla riga precedente (soluzione migliore), che a fine riga:

```
//elaboro il prodotto successivo
indiceProdotto++;

indiceProdotto++; //elaboro il prodotto successivo
```

Nel caso si debba commentare un blocco di codice, utilizzare /* ... */.

I commenti /** ... */ sono riservati alla documentazione Javadoc.

Evitare di costruire riquadri grafici, in quanto richiedono del tempo per essere costruiti e mantenuti allineati quando se ne modifica il contenuto:

```
/*******************
 * Classe Cliente *
 *                *
 * Implementa la  *
 * gestione di un *
 * cliente        *
 *******************/
```

Bibliografia

[1] Vari, *"Code Conventions for the Java Programming Language"*, http://java.sun.com/docs/codeconv/html/CodeConvTOC.doc.html

[2] Christie Badaux, *"Netscape Software Coding Standards Guide for Java",* http://developer.netscape.com/docs/technote/java/codestyle.html

[2] Scott Ambler, *"Writing Robust Java Code"*, www.ambysoft.com

[4] Marco Lamberto, *"Programmare? Questione di stile"*, http://www.siforge.org/articles/2002/09/04-style.p.html

[5] Geotechnical Software Servces, *"Java Programming Style Guidelines"*, http://geosoft.no/javastyle.html

[6] Paul Haahr, *"A Programming Style for Java"*, http://www.webcom.com/~haahr/essays/java-style/

[7] Java Ranch, *"Java Programming Style Guide"*, http://www.javaranch.com/style.jsp

[8] Catharina Candolin, *"Java Style Guide"*, http://www.cs.hut.fi/~candolin/java/styleguide.html

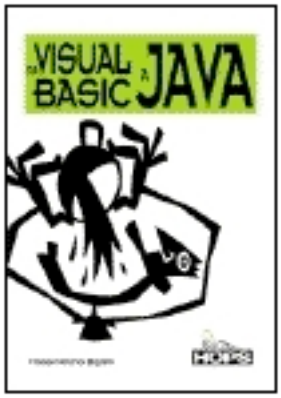
[9] Dinopolis Team, *"Dinopolis Java Coding Convention"*, http://www.dinopolis.org/documentation/conventions/dinopolis_coding_convention.html

Libri

[libro1] Steve Mc Connell, *"Code Complete: a pratical handbook of software construction"*, Microsoft Press 1993

[libro 2] Erich Gamma et.al., *"Design Patterns"*, Addison Wesley 1995

L'autore



Massimiliano Bigatti é autore del libro "Da Visual Basic a Java". E' certificato, tra le altre, come *SUN Certified Enterprise Architect for Java Platform, Enterprise Edition Technology*. Si occupa di architetture applicative basate su Java, technical writing e del portale http://javawebservices.it.

Leggi altre mie white paper:

- Extreme Programming e metodologie agili di sviluppo software: concetti, prodotti e risorse (www.bigatti.it);
- Web Services e SOAP (javawebservices.it/whitepaper)

Java Web Services.it

Il portale italiano dedicato ai Web Services in Java.