Capitolo 2

Nomi significativi

di Tim Ottinger

In questo capitolo

Introduzione

I nomi sono dappertutto nel software. Diamo un nome alle variabili, alle funzioni, agli argomenti, alle classi e ai package. Diamo un nome ai nostri file di codice sorgente e le directory che li contengono. Diamo un nome ai nostri file jar, war e ear. Diamo nomi e nomi e nomi. Dato che lo facciamo così spesso, meglio farlo bene. Ecco alcune semplici regole per scegliere buoni nomi.

Usate nomi "parlanti"

È facile dire che i nomi dovrebbero essere parlanti. Quello che vogliamo farvi capire è che questa è una faccenda *seria*. La scelta di un buon nome richiede tempo ma ne fa risparmiare molto di più. Pertanto abbiate cura dei vostri nomi e modificateli se ne trovate di migliori. Chiunque legga il vostro codice (voi compresi) ne sarà felice.
Il nome di una variabile, funzione o classe, deve dare una risposta a tutte le domande. Dovrà dirvi perché esiste, che cosa fa e come viene usato. Se un nome richiede un commento, significa che tale nome non rivela il proprio scopo.

```
int d; // tempo trascorso in giorni
```

Il nome `d` non dice nulla. Non evoca un senso del tempo trascorso e nemmeno dei giorni. Dovremmo scegliere un nome che specifichi che cosa viene misurato e l'unità di misura impiegata:

```
int elapsedTimeInDays;
int daysSinceCreation;
int daysSinceModification;
int fileAgeInDays;
```

La scelta di nomi parlanti aiuta molto a comprendere e modificare il codice. Qual è lo scopo del seguente codice?

```
public List<int[]> getThem() {
  List<int[]> list1 = new ArrayList<int[]>();
  for (int[] x : theList)
    if (x[0] == 4)
      list1.add(x);
  return list1;
}
```

Perché è difficile capire che cosa fa questo codice? Eppure non vi sono espressioni complesse. Le spaziature e l'indentazione sono corrette. Sono menzionate solo tre variabili e due costanti. Non vi è traccia di strane classi o di metodi polimorfici, solo una lista di array (o almeno così sembra).
Il problema non è la semplicità ma l'*implicità* del codice (ecco coniata una frase…): il livello in cui il contesto non è esplicitato nel codice. Il codice richiede implicitamente che noi conosciamo le risposte a domande come:

1. Che genere di cose si trovano in `theList`?
2. Qual è il significato dell'indice zero per un elemento di `theList`?
3. Qual è il significato del valore 4?
4. Come dovrei usare la lista restituita?

Le risposte a queste domande non sono presenti nel codice presentato, *mentre dovrebbero esservi*. Mettiamo che si tratti del classico giochino *Mine Sweeper*. Sappiamo che la tavola è una lista di celle che ora è chiamata `theList`. Rinominiamola in `gameBoard`.
Ogni cella della tavola da gioco è rappresentata da un semplice array. Scopriamo inoltre che l'indice zero è la posizione di un certo valore di stato e che un valore di stato pari a `4` significa "con bandiera", `flagged`. Già il fatto di convogliare questi concetti nei nomi migliora considerevolmente il codice:

```
public List<int[]> getFlaggedCells() {
  List<int[]> flaggedCells = new ArrayList<int[]>();
  for (int[] cell : gameBoard)
    if (cell[STATUS_VALUE] == FLAGGED)
      flaggedCells.add(cell);
  return flaggedCells;
}
```

Notate che la semplicità del codice non è cambiata. Ha ancora esattamente lo stesso numero di operatori e costanti, con esattamente lo stesso numero di livelli di annidamento. Ma ora il codice è molto più esplicito.
Possiamo spingerci oltre e scrivere una semplice classe per le celle invece di usare un array di `int`. La classe potrà includere una funzione, sempre parlante, (chiamata `isFlagged`) che nasconde i numeri magici. Il risultato è una nuova versione della funzione:

```
public List<Cell> getFlaggedCells() {
  List<Cell> flaggedCells = new ArrayList<Cell>();
  for (Cell cell : gameBoard)
    if (cell.isFlagged())
      flaggedCells.add(cell);
  return flaggedCells;
}
```

Con questi semplici cambi di nome, non è più difficile comprendere che cosa accade. Questa è la potenza dei "buoni nomi".

Evitate la disinformazione

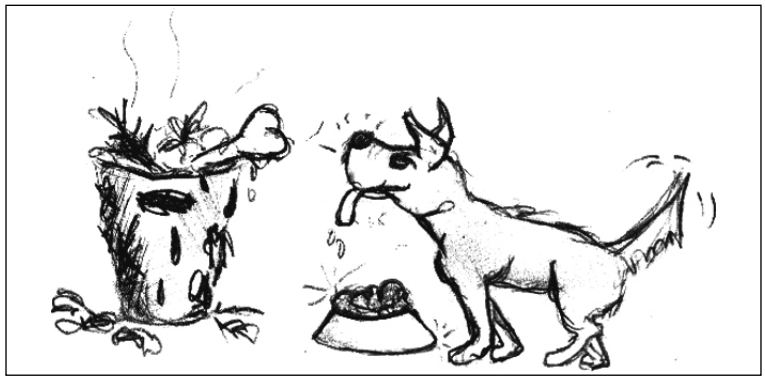
I programmatori devono evitare di lasciare falsi indizi che celano il significato del codice. Dovremmo evitare parole dai significati nascosti ambigui e lontani da quello principale e desiderato. Per esempio, `hp`, `aix` e `sco` sarebbero inadatti come nomi di variabili, perché sono anche nomi di piattaforme o varianti di Unix. Anche se dovete calcolare un'ipotenusa (*hypotenuse*) e `hp` sembrerebbe un'abbreviazione perfetta, tale nome sarebbe fuorviante.
Non chiamate un gruppo di account con il nome `accountList` a meno che si tratti effettivamente di una `List`. La parola "`list`" ha un significato ben preciso in programmazione.

Se il contenitore degli account non è una `List`, potrebbe portare a conclusioni errate. (Come vedremo, anche se il contenitore *è* una `List`, probabilmente è meglio non specificare nel nome il tipo scelto per il contenitore.) Per un semplice gruppo di account, è molto meglio usare i nomi `accountGroup` o `bunchOfAccounts` o anche semplicemente `accounts`. Attenzione anche a usare nomi solo leggermente diversi. Quanto tempo ci vuole per individuare la subdola differenza che c'è fra un `XYZControllerForEfficientHandlingOfStrings` in un modulo e un `XYZControllerForEfficientStorageOfStrings` magari poco distante? Le due parole hanno praticamente la stessa "forma".
Scrivere in modo simile dei concetti simili è un'*informazione*. Un uso incoerente dei nomi è *disinformazione*. Negli ambienti Java di oggi possiamo contare sul completamento automatico del codice. Scriviamo alcuni caratteri di un nome e premiamo una certa combinazione di tasti e otteniamo un elenco di tutti i possibili completamenti di tale nome. Questo è molto utile se i nomi di elementi molto simili si trovano in ordine alfabetico e se le differenze sono evidenti, perché lo sviluppatore può selezionare un oggetto per nome senza vedere tutti i commenti o anche la lista dei metodi forniti da tale classe. Un pessimo esempio di nomi disinformativi sarebbe l'uso come nomi di variabili delle lettere "L" minuscola o "O" maiuscola, in particolare se in combinazione. Il problema, naturalmente, è che la lettera "l" somiglia al numero "1" e che la lettera "O" somiglia al numero "0".

```
int a = l;
if ( O == l )
  a = O1;
else
  l = O1;
```

Potreste anche pensare che sia superfluo ricordarlo, ma abbiamo esaminato molte volte del codice in cui questi identificatori erano presenti in abbondanza. In un caso l'autore del codice suggeriva di usare un particolare font che aiutasse a evidenziare le differenze, una soluzione che sarebbe stato necessario comunicare poi a tutti i successivi sviluppatori in modo orale o in un documento scritto. Il problema si elimina definitivamente e senza creare altri prodotti esterni semplicemente rinominando gli identificatori.

Adottate distinzioni sensate



I programmatori creano problemi a se stessi quando scrivono codice con l'unica idea di soddisfare un compilatore o un interprete. Per esempio, poiché non potete usare lo stesso nome per far riferimento a due diverse cose nello stesso campo di visibilità (*scope*), potreste essere tentati di cambiare un nome in un modo arbitrario. Talvolta la "soluzione" consiste nello sbagliare di proposito l'ortografia di uno dei due identificatori, il che può portare a una sorprendente situazione: correggendo gli errori di ortografia il codice diventa incompilabile (si consideri, per esempio, la pratica veramente orribile di creare una variabile denominata `klass` solo perché il nome `class` era usato per qualcos'altro).
Non è sufficiente aggiungere dei numeri o delle parole di disturbo, pur di soddisfare il compilatore. Se i nomi devono essere differenti, devono anche avere significati differenti. La denominazione numerica (`a1`, `a2`, … `aN`) è l'opposto della denominazione intenzionale. Tali nomi non sono solo disinformativi, ma sono del tutto non-informativi: non forniscono alcun indizio sull'intenzione dell'autore. Considerate la seguente funzione:

```
public static void copyChars(char a1[], char a2[]) {
  for (int i = 0; i < a1.length; i++) {
    a2[i] = a1[i];
  }
}
```

Questa funzione si legge molto meglio se per gli argomenti si usano i nomi `source` e `destination`.
Le parole di disturbo rappresentano un'altra distinzione insensata. Immaginate di avere una classe `Product`. Se ne avete un'altra chiamata `ProductInfo` o `ProductData`, avete creato nomi differenti senza convogliare in essi un significato differente. `Info` e `Data` sono semplici parole di disturbo, come `a`, `an` e `the`.
Notate che non c'è niente di sbagliato nell'uso di un prefisso come `a` e `the`, sempre che essi rappresentino una distinzione parlante. Per esempio, potreste usare `a` per tutte le variabili locali e `the` per tutti gli argomenti di funzione (Uncle Bob lo faceva in C++, ma ha abbandonato la pratica perché gli IDE recenti lo rendevano inutile). Il problema sorge se decidete di chiamare una variabile `theZork` solo perché avete già un'altra variabile chiamata `zork`.
Le parole di disturbo sono ridondanti. La parola `variable` non dovrebbe mai comparire nel nome di una variabile. La parola `table` non dovrebbe mai comparire nel nome di una tabella. Perché mai `NameString` sarebbe meglio di `Name`? E `Name` potrebbe essere un numero in virgola mobile? In tal caso, viola la regola sulla disinformazione. Immaginate di trovare una classe chiamata `Customer` e un'altra chiamata `CustomerObject`. Cosa intende dire questa distinzione? Quale delle due rappresenta la soluzione migliore per la cronologia dei pagamenti di un cliente?
C'è un'applicazione in cui questo errore è perfettamente rappresentato. Abbiamo modificato i nomi per proteggere il colpevole, ma ecco l'aspetto esatto di tale errore:

```
getActiveAccount();
getActiveAccounts();
getActiveAccountInfo();
```

In quale modo i programmatori di questo progetto devono immaginare quale di queste funzioni richiamare?
In assenza di convenzioni specifiche, la variabile `moneyAmount` è indistinguibile da `money`, `customerInfo` è indistinguibile da `customer`, `accountData` è indistinguibile da `account` e `theMessage` è indistinguibile da `message`. Distinguete i nomi in modo che chi legge possa capire quali sono le differenze.

Usate nomi pronunciabili

Agli esseri umani piacciono le parole. Una parte significativa del nostro cervello è dedicata proprio alle parole. E le parole sono, per definizione, pronunciabili. Sarebbe un peccato non sfruttare questa parte del nostro cervello che si è evoluta proprio per gestire la parola. In altre parole, che i vostri nomi siano pronunciabili!
Se non riuscite a pronunciarlo, non potrete parlarne senza passare da idioti. "Proprio lì dove c'è *bi ci erre tre ci enne ti* abbiamo l'`int` *pi esse zeta cu*, lo vedi?" Non è una cosa da poco, perché la programmazione è un'attività sociale.
Una società che conosco usa l'identificatore `genymdhms` (data di generazione, anno, mese, giorno, ora, minuto e secondo) e così i programmatori vagano dicendo qualcosa come *gen ipsilon emme di acca emme esse*. Ho la fastidiosa abitudine di pronunciare tutto così come è scritto, così ho comindiato a chiamarlo *gen-yah-mudda-hims*. Dopo un po' veniva chiamata così anche da progettisti e analisti e sembravamo tutti un po' sciocchi. Ma stavamo tutti allo scherzo, ed era divertente. Comunque sia, la sostanza era che tolleravamo una cattiva scelta in termini di denominazione. Ai nuovi sviluppatori era necessario spiegare tale scelta di denominazione delle variabili e così essi finivano per pronunciarle usando strane parole invece di usare termini corretti del linguaggio naturale [in inglese, in questo caso, NdT]. Provate a confrontare…

```
class DtaRcrd102 {
  private Date genymdhms;
  private Date modymdhms;
  private final String pszqint = "102";
  /* ... */
};
```

con:

```
class Customer {
  private Date generationTimestamp;
  private Date modificationTimestamp;
  private final String recordId = "102";
  /* ... */
};
```

Questo rende possibile una conversazione intelligibile: "Ehi, Mario, dai un'occhiata a questo record! `generationTimestamp` ha la data di domani! Come è possibile?".

Usate nomi ricercabili

I nomi mono-lettera e le costanti numeriche presentano il problema di essere difficili da individuare all'interno di un testo.
È facile trovare `MAX_CLASSES_PER_STUDENT`, mentre il numero `7` può essere più problematico. Una ricerca può individuare la cifra dentro nomi di file, dentro le definizioni di altre costanti e in varie espressioni nelle quali il valore viene usato con scopi molto differenti. Ancora peggio: una costante può essere un numero lungo e qualcuno potrebbe scambiare due cifre, da un lato creando un bug e simultaneamente sfuggendo alla ricerca del programmatore.
Analogamente, il nome `e` è una cattiva scelta per qualsiasi variabile che debba essere ricercata da un programmatore. È una lettera eccezionalmente comune e probabilmente comparirà in ogni singola riga di testo di ogni programma. In buona sostanza, i nomi lunghi battono i nomi brevi e i nomi ricercabili battono le costanti seminate nel codice. La mia personale preferenza prevede di usare nomi mono-lettera SOLO come variabili locali all'interno di piccoli metodi. *La lunghezza di un nome dovrebbe corrispondere alle dimensioni del suo campo di visibilità* [N5]. Se una variabile o una costante può essere vista o usata in più punti del codice, è imperativo assegnarle un nome facile da ricercare. Ora confrontate…

```
for (int j = 0; j<34; j++) {
  s += (t[j]*4)/5;
}
```

Con:

```
int realDaysPerIdealDay = 4;
const int WORK_DAYS_PER_WEEK = 5;
int sum = 0;
for (int j = 0; j < NUMBER_OF_TASKS; j++) {
  int realTaskDays = taskEstimate[j] * realDaysPerIdealDay;
  int realTaskWeeks = (realdays / WORK_DAYS_PER_WEEK);
  sum += realTaskWeeks;
}
```

Notate che `sum`, qui sopra, non è un nome particolarmente utile, almeno è ricercabile. Il codice "parlante" produce una funzione più lunga, ma considerate quanto è più facile trovare `WORK_DAYS_PER_WEEK` rispetto a tutte le situazioni in cui è stato usato il numero `5`, ed eliminare dalla lista tutte le istanze che hanno un altro significato.

Evitate le codifiche

Abbiamo già abbastanza codifiche nella nostra vita senza aggiungerne di nuove. Il tipo di codifica e le informazioni sulla visibilità nei nomi aggiungono la necessità di decodificarne il significato. Non sembra molto ragionevole obbligare ogni nuovo assunto a studiare l'ennesimo "linguaggio" di codifica oltre a dover studiare il codice (che di per sé non è cosa da poco) sul quale dovrà lavorare. Si tratta di un carico mentale inutile

quando lo scopo è quello di tentare di risolvere un problema. Inoltre i nomi codificati sono difficili da pronunciare e facili da sbagliare.

Notazione ungherese

Nei tempi antichi, quando potevamo contare su linguaggi legati alla lunghezza dei nomi, abbiamo violato questa regola per necessità e con un certo senso di colpa. Il Fortran imponeva le codifiche, dato che la prima lettera era un codice che stabiliva il tipo. Le prime versioni di BASIC consentivano l'uso solo di una lettera più una cifra. La notazione ungherese ha consentito di superare questo limite.

La notazione ungherese era considerata piuttosto importante per l'API C di Windows, quando tutto era un intero o un puntatore `long` o un puntatore `void` o una delle tante implementazioni di una "stringa" (con utilizzi e attributi differenti). A quei tempi il compilatore non controllava i tipi, e pertanto i programmatori avevano bisogno di "qualcosa" per ricordarli.

Nei nuovi linguaggi abbiamo a disposizione molti più tipi e i compilatori ricordano e applicano i tipi. Ma in più vi è anche la tendenza a realizzare classi più piccole e funzioni più brevi, e così in genere è più facile vedere il punto in cui è dichiarata ogni variabile impiegata.

I programmatori Java non hanno bisogno di codificare i tipi. Gli oggetti sono fortemente tipizzati e gli ambienti di editing sono migliorati al punto da individuare un errore di tipo molto prima della compilazione! Pertanto, al giorno d'oggi, la notazione ungherese e altre forme di codifica dei tipi sono semplici impedimenti. Complicano ogni modifica del nome o del tipo di una variabile, funzione o classe. Complicano la lettura del codice. E introducono la possibilità che il sistema di codifica confonda chi legge.

```
PhoneNumber phoneString;
// il nome non cambia anche se cambia il tipo!
```

Prefissi per i membri

Non è più necessario nemmeno usare il prefisso `m_` per le variabili membro. Le classi e le funzioni dovrebbero essere così compatte da rendere inutile questa abitudine. E in più dovreste impiegare un ambiente di editing in grado di evidenziare o colorare appositamente i membri.

```
public class Part {
  private String m_dsc; // La descrizione testuale
  void setName(String name) {
    m dsc = name;
  }
}
-----------------------------------------
public class Part {
  String description;
  void setDescription(String description) {
    this.description = description;
  }
}
```

Fra l'altro, la gente tende a ignorare il prefisso (o suffisso), per vedere solo la parte significativa del nome. Più leggiamo il codice, meno vediamo i prefissi. Alla fine i prefissi divengono elementi superflui, utili solo a distinguere il vecchio codice dal nuovo.

Interfacce e implementazioni

Talvolta queste sono particolari tipi di codifiche. Per esempio, immaginate di realizzare una ABSTRACT FACTORY per la creazione di forme. Questa factory sarà un'interfaccia e sarà implementata da una classe concreta. Come bisognerebbe chiamarle? `IShapeFactory` e `ShapeFactory`? Preferisco lasciare le interfacce senza aggiunte. Il prefisso `I`, così comune al giorno d'oggi, è come minimo una distrazione e anche un eccesso di informazioni. Non voglio che i miei utenti sappiano che uso un'interfaccia. Voglio solo che sappiano che è una `ShapeFactory`. Pertanto, se devo codificare il nome dell'interfaccia o dell'implementazione, scelgo quello dell'implementazione. Chiamarla `ShapeFactoryImp`, o anche con il terribile `CShapeFactory`, è sempre meglio che codificare il nome dell'interfaccia.

Evitate le mappe mentali

I lettori non devono essere costretti a tradurre mentalmente i nomi in altri nomi che già conoscono. Questo problema generalmente deriva dalla scelta di usare termini che non appartengono né al dominio del problema né al dominio della soluzione.

Questo è un problema con i nomi di variabili mono-lettera. Certamente il contatore di un ciclo può chiamarsi `i` o `j` o `k` (ma mai `l`!) se il suo campo di visibilità è molto piccolo e non vi sono altri nomi in conflitto. Questo perché l'uso di nomi mono-lettera per i contatori dei cicli è ormai una tradizione. Tuttavia, nella maggior pare degli altri contesti, un nome mono-lettera non è una buona scelta; questo perché costringe chi legge a richiamare alla mente il vero concetto. Non c'è peggior motivazione per scegliere il nome `c` perché `a` e `b` erano già occupati.

In generale i programmatori sono persone intelligenti. Talvolta, pertanto, amano mostrare le loro conoscenze con giochi di parole. Dopotutto, chi riesce a ricordarsi (ora e nel tempo) che `r` è la versione in lettere minuscole dell'url senza host e schema, deve essere davvero intelligente.

Una differenza fra un programmatore intelligente e un programmatore professionale è che il secondo sa che *prima di tutto viene la chiarezza*. I professionisti usano le proprie capacità in modo positivo e scrivono codice che gli altri siano in grado di comprendere.

Nomi di classi

Le classi e gli oggetti dovrebbero avere nomi parlanti, come `Customer`, `WikiPage`, `Account` o `AddressParser`. Evitate parole come `Manager`, `Processor`, `Data` o `Info` nel nome di una classe. Il nome di una classe non deve essere un verbo.

Nomi di metodi

I nomi di metodi dovrebbero contenere un verbo o una frase verbale, come in `postPayment`, `deletePage` o `save`. I metodi di accesso, mutatori e predicati dovrebbero avere un nome che dipende dal valore, con il prefisso `get`, `set` e `is` secondo lo standard javabean (`http://java.sun.com/products/javabeans/docs/spec.html`).

```
string name = employee.getName();
customer.setName("mike");
if (paycheck.isPosted())...
```

Quando i costruttori vengono sottoposti a overload, usate nomi di metodi che descrivono gli argomenti. Per esempio,

```
Complex fulcrumPoint = Complex.FromRealNumber(23.0);
```

è generalmente meglio di:

```
Complex fulcrumPoint = new Complex(23.0);
```

Considerate anche l'idea di richiederne l'uso rendendo `private` i costruttori corrispondenti.

Non fate i "simpatici"

Se i nomi sono troppo particolari, se li ricorderanno solo coloro che hanno lo stesso *sense of humor* dell'autore e solo per un tempo limitato. Dopo un po' chi si ricorderà del significato della funzione `HolyHandGrenade`? Oh, è un nome simpatico, ma forse in questo caso sarebbe molto meglio `DeleteItems`. Preferite la chiarezza a un nome divertente.
Nel codice la "simpatia" spesso affiora sotto forma di termini colloquiali o gergali. Per esempio, non usate il nome `whack()` per intendere `kill()`. Non usate battute di spirito (tra l'altro comprensibili in una sola lingua) come `eatMyShorts()` per intendere `abort()`.
In poche parole, siate descrittivi.

Una parola, un concetto

Scegliete una parola per un determinato concetto astratto e poi continuate a usarla. Per esempio, non ha senso usare `fetch`, `retrieve` e `get` per metodi equivalenti di classi differenti. Poi come farete a ricordare quale nome di metodo avete usato con quale classe? Purtroppo, spesso occorre risalire all'azienda, al gruppo o al singolo programmatore che ha scritto la libreria o la classe per ricordare quale termine può essere stato usato. In caso contrario, dovrete dedicare parecchio tempo a scorrere gli header e gli altri esempi di codice. I nuovi ambienti di editing come Eclipse e IntelliJ forniscono indizi contestuali, come la lista dei metodi che potete richiamare su un determinato oggetto. Ma notate che la lista normalmente non fornisce i commenti che avete scritto su nomi di funzione e elenchi di parametri. Se siete fortunati vi fornirà il *nome* dei parametri presenti nelle dichiarazioni delle funzioni. I nomi di funzione devono essere comprensibili e devono anche essere coerenti, in modo da poter selezionare il metodo corretto senza ulteriori esplorazioni.
Analogamente, è fonte di confusione avere un `controller` e un `manager` e un `driver` tutti nella stessa base di codice. Qual è la differenza sostanziale fra un `DeviceManager` e un `ProtocolController`? Perché non sono entrambi `controller` o entrambi `manager`? E se in realtà fossero `driver`? Il nome fa pensare a due oggetti di tipo molto differente oltre che appartenenti a classi differenti.
Un lessico coerente sarà molto apprezzato dai programmatori che dovranno usare il vostro codice.

Non siate fuorvianti

Evitate di usare la stessa parola per due scopi. Usare lo stesso termine per due idee differenti significa confondere le cose.
Se seguite la regola "una parola un concetto", potreste avere, per esempio, molte delle classi con un metodo `add`. Fintantoché gli elenchi di parametri e i valori restituiti dei vari metodi `add` sono semanticamente equivalenti, tutto andrà bene.
Tuttavia qualcuno potrebbe decidere di usare la parola `add` per pura "coerenza", anche dove non si deve "aggiungere" nulla. Supponiamo che abbiate molte classi nelle quali `add` crea un nuovo valore aggiungendo o concatenando due valori. Ora immaginiamo che stiamo scrivendo una nuova classe che ha un metodo che inserisce il suo unico parametro in una collezione di dati. Tale metodo dovrà chiamarsi `add`? Potrebbe sembrare coerente, perché abbiamo già altri metodi `add`, ma in questo caso, le semantiche sono differenti, e pertanto piuttosto dovremmo usare un nome come `insert` o `append`. Chiamare il nuovo metodo `add` sarebbe quindi fuorviante.
Il nostro obiettivo, come autori, è quello di rendere il nostro codice il più possibile comprensibile. Vogliamo che il significato del nostro codice risalti subito, non richieda uno studio intensivo. Vogliamo adottare un approccio divulgativo, nel quale l'autore ha la responsabilità di essere comprensibile e l'approccio accademico, nel quale è compito dell'allievo comprendere il significato di quanto legge.

Usate nomi tratti dal dominio della soluzione

Ricordate che coloro che leggono il vostro codice saranno dei programmatori. Pertanto sentitevi liberi di usare termini informatici, nomi di algoritmi, nomi di pattern, termini matematici e così via. Non è saggio trarre tutti i nomi dal dominio del problema, perché non vogliamo che i nostri collaboratori debbano continuamente rivolgersi al cliente per chiedergli il significato di ogni nome, quando invece conoscono lo stesso concetto sotto un altro nome.

Il nome `AccountVisitor` ha un grande significato per un programmatore che conosce il pattern VISITOR. Cosa può significare, invece, un nome come `JobQueue`? I programmatori devono svolgere parecchie operazioni molto tecniche. Scegliere nomi tecnici per tali operazioni sembra essere la scelta più sensata.

Usate nomi tratti dal dominio del problema

Quando non esiste un termine in "computer-ese" per quello che state facendo, usate un nome tratto dal dominio del problema. Quanto meno il programmatore che esegue la manutenzione del vostro codice potrà domandarne il significato a un esperto del dominio. Mantenere distinti i concetti fra dominio della soluzione e dominio del problema è compito di un buon progettista e programmatore. Il codice che ha più a che fare con concetti appartenenti al dominio del problema dovrà impiegare più nomi tratti dal dominio del problema.

Aggiungete un contesto significativo

Vi sono alcuni nomi che sono significativi così come sono, ma non sono molti. Piuttosto, occorre collocare i nomi in un contesto appropriato per chi leggerà il codice, racchiudendoli in classi, funzioni o namespace con un nome adatto. Se ciò non fosse possibile, l'ultima risorsa consiste nell'adottare dei prefissi per il nome.

Immaginate di avere delle variabili chiamate `firstName`, `lastName`, `street`, `houseNumber`, `city`, `state` e `zipcode`. Prese tutte insieme è abbastanza evidente che esse costituiscano un indirizzo. Ma cosa accade se vedete solo la variabile `state` in un metodo? Pensereste automaticamente che si tratti di una parte di un indirizzo?

Potete aggiungere un contesto tramite l'uso di prefissi: `addrFirstName`, `addrLastName`, `addrState` e così via. Quanto meno coloro che leggono il codice capiranno che queste variabili fanno parte di una struttura più ampia. Naturalmente, una soluzione migliore consisterebbe nel creare una classe chiamata `Address`. In tal modo, anche il compilatore saprà che le variabili rientrano in un concetto più grande.

Considerate il metodo presentato nel Listato 2.1. Pensate che le variabili avrebbero bisogno di un contesto più significativo? Il nome della funzione fornisce solo una parte del contesto; la parte rimanente è fornita dall'algoritmo. Leggendo la funzione, si vede che le tre variabili, `number`, `verb` e `pluralModifier`, fanno semplicemente parte del messaggio "`GuessStatistics`". Sfortunatamente, il contesto deve essere dedotto. La prima volta che si osserva il metodo, il significato delle variabili non è chiaro.

Listato 2.1 Variabili con contesto non chiaro.

```
private void printGuessStatistics(char candidate, int count) {
  String number;
  String verb;
  String pluralModifier;
  if (count == 0) {
    number = "no";
    verb = "are";
    pluralModifier = "s";
  } else if (count == 1) {
    number = "1";
    verb = "is";
    pluralModifier = "";
  } else {
    number = Integer.toString(count);
    verb = "are";
    pluralModifier = "s";
  }
  String guessMessage = String.format(
    "There %s %s %s%s", verb, number, candidate, pluralModifier
  );
  print(guessMessage);
}
```

La funzione è un po' troppo lunga e le variabili sono usate un po' ovunque. Per suddividere la funzione in frammenti più piccoli dobbiamo creare una classe `GuessStatisticsMessage` e trasformare le tre variabili in campi di tale classe. Ciò fornisce un contesto chiaro per le tre variabili. Ora fanno *assolutamente* parte di `GuessStatisticsMessage`. Il miglioramento in termini di contesto consente anche di raffinare l'algoritmo suddividendolo in più funzioni, più piccole (Listato 2.2).

Listato 2.2 Ora le variabili hanno un contesto.

```
public class GuessStatisticsMessage {
  private String number;
  private String verb;
  private String pluralModifier;

  public String make(char candidate, int count) {
    createPluralDependentMessageParts(count);
    return String.format(
      "There %s %s %s%s",
      verb, number, candidate, pluralModifier );
  }

  private void createPluralDependentMessageParts(int count) {
    if (count == 0) {
      thereAreNoLetters();
    } else if (count == 1) {
      thereIsOneLetter();
```

```
    } else {
      thereAreManyLetters(count);
    }
  }

  private void thereAreManyLetters(int count) {
      number = Integer.toString(count);
      verb = "are";
      pluralModifier = "s";
  }

  private void thereIsOneLetter() {
    number = "1";
    verb = "is";
    pluralModifier = "";
  }

  private void thereAreNoLetters() {
    number = "no";
    verb = "are";
    pluralModifier = "s";
  }
}
```

Non aggiungete contesti inesistenti

In un'immaginaria applicazione chiamata "Gas Station Deluxe", non è una buona idea usare come prefisso per ogni classe `GSD`. Francamente, opereresti contro i vostri strumenti. Digitate `G` e premete il tasto di completamento e ottenete un lungo elenco di ogni classe del sistema. È davvero utile? Perché mai complicare la vita all'IDE?

Analogamente, immaginate di aver inventato una classe `MailingAddress` nel modulo di contabilità di `GSD` e di averla chiamata `GSDAccountAddress`. Successivamente, avete bisogno di un indirizzo postale per l'applicazione di gestione dei clienti. Usate `GSDAccountAddress`? Vi sembra il nome corretto? Dieci caratteri su 17 sono ridondanti o irrilevanti.

I nomi brevi generalmente sono migliori dei nomi lunghi, sempre che il loro significato sia chiaro. Non aggiungete a un nome più contestualità del necessario.

I nomi `accountAddress` e `customerAddress` sono bei nomi per le istanze della classe `Address`, mentre sono cattivi nomi per le classi. `Address` è un buon nome per una classe. Se occorre distinguere fra indirizzi postali, MAC e web, si può usare `PostalAddress`, `MAC` e `URI`. I nomi risultanti sono più precisi, che poi è lo scopo di ogni scelta di nomi.

Conclusioni

L'aspetto più complesso nella scelta di un buon nome è che richiede buone capacità descrittive e un background culturale condiviso. Questo è un problema più culturale che tecnico, operativo o gestionale. Di conseguenza molte persone che operano in questo campo non imparano a farlo molto bene.

A volte si ha anche il timore di rinominare le cose, aspettandosi le obiezioni degli altri sviluppatori. Non condividiamo tali timori e troviamo che sia davvero positivo che i nomi possano cambiare (in meglio, naturalmente). La maggior parte delle volte non memorizziamo, davvero, il nome di classi e metodi. Usiamo gli strumenti disponibili per gestire questo genere di dettagli in modo da poterci concentrare sul fatto che il codice riesca a leggersi quasi come se si trattasse di frasi (pur in inglese) o quanto meno in termini di tabelle e di struttura dei dati (non sempre una frase è il modo migliore per visualizzare i dati). Probabilmente finirete per sorprendere qualcuno con le operazioni di ridenominazione, come accade con ogni altra attività di miglioramento del codice. Ma non arrendetevi.

Seguite alcune di queste regole e cercate di migliorare la leggibilità del vostro codice. Se dovete eseguire la manutenzione di codice altrui, usate gli strumenti di refactoring per tentare di risolvere questi problemi. Questi sforzi saranno ripagati nel breve periodo e continueranno a rendere "dividendi" anche nel lungo periodo.