

ROBERT C. MARTIN

Clean Code



**Guida per diventare
bravi artigiani nello sviluppo
agile di software**

APOGEO

ROBERT C. MARTIN

Clean Code



**Guida per diventare
bravi artigiani nello sviluppo
agile di software**

APOGEO

CLEAN CODE
GUIDA PER DIVENTARE BRAVI ARTIGIANI NELLO SVILUPPO AGILE DI
SOFTWARE

Robert C. Martin

APOGEO

© Apogeo - IF - Idee editoriali Feltrinelli s.r.l.
Socio Unico Giangiacomo Feltrinelli Editore s.r.l.

ISBN edizione cartacea: 9788850334384

Authorized translation from the English language edition, entitled CLEAN CODE: A HANDBOOK OF AGILE SOFTWARE CRAFTSMANSHIP, 1st Edition by ROBERT MARTIN, published by Pearson Education, Inc, publishing as Prentice Hall, Copyright (c) 2009 Pearson Education, Inc. All rights reserved.

Il presente file può essere usato esclusivamente per finalità di carattere personale. Tutti i contenuti sono protetti dalla Legge sul diritto d'autore.

Nomi e marchi citati nel testo sono generalmente depositati o registrati dalle rispettive case produttrici.

L'edizione cartacea è in vendita nelle migliori librerie.

~

Sito web: www.apogeonline.com

Scopri le novità di Apogeo su [Facebook](#)

Seguici su Twitter [@apogeonline](#)

Collegati con noi su [LinkedIn](#)

Rimani aggiornato iscrivendoti alla nostra [newsletter](#)

Ad Ann Marie: l'infinito amore della mia vita.

Prefazione

Una delle caramelle che preferiamo, qui in Danimarca, si chiama Ga-Jol, i cui intensi effluvi di liquirizia sono un toccasana per il nostro clima, così freddo e umido. Parte del nostro interesse per le caramelle Ga-Jol deriva dalla frase, fra il saggio e il faceto, stampata sulla linguetta di ogni scatolina. Questa mattina ho acquistato una confezione da due pacchetti di queste caramelle e una recava questo vecchio detto danese:

Ærlighed i små ting er ikke nogen lille ting.

“L’onestà nelle cose da poco, non è una cosa da poco.” è un’ottima introduzione a quello che già intendeva dire in queste righe. Le piccole cose contano. Questo è un libro sui piccoli problemi il cui peso è tutt’altro che piccolo.

“Dio si ritrova nei dettagli”, disse l’architetto Ludwig Mies van der Rohe. Questa citazione richiama le discussioni contemporanee sul ruolo dell’architettura nello sviluppo di software, e in particolare nel mondo Agile. Bob e io talvolta ci siamo scontrati proprio su questo argomento. E sì, Mies van der Rohe era attento all’utilità e all’eternità delle forme degli edifici basati su una grande architettura. D’altra parte, egli selezionò personalmente ogni singola maniglia di ogni singolo edificio che aveva costruito. Perché? Perché le piccole cose contano.

Nel “dibattito” in corso sullo sviluppo TDD (*Test-Driven Development*), Bob e io abbiamo scoperto di concordare sul fatto che l’architettura del software ha una collocazione importante nello

sviluppo, sebbene abbiamo, probabilmente, idee differenti sull'esatto significato di questa affermazione. Tali cavilli non sono però particolarmente importanti, perché possiamo dare per scontato che i veri professionisti dedichino del tempo a riflettere e a pianificare il progetto fin dall'inizio. I concetti di fine anni Novanta sulla progettazione guidata unicamente dai test e dal codice sono ormai tramontati. Purtuttavia, l'attenzione al dettaglio è un elemento ancora più caratteristico della professionalità di qualsiasi grande visione. Innanzitutto, è proprio dalla cura delle piccole cose che i professionisti traggono competenza e fiducia per operare su quelle grandi. In secondo luogo, il piccolo difetto costruttivo, la portiera che non si chiude perfettamente o la mattonella leggermente disassata nel pavimento, o anche la scrivania in disordine, dissipano in un lampo l'immagine dell'intero insieme. Per questo parliamo di *codice pulito*.

In ogni caso, l'architettura è solo una delle metafore dello sviluppo di software e, in particolare, di quella parte del software che deve occuparsi di fornire il prodotto iniziale, nello stesso modo in cui un architetto fornisce un edificio nuovo di zecca. In questi tempi di Scrum e Agile, il focus è sulla rapidità dell'uscita sul mercato. Vogliamo che la fabbrica del software possa operare a pieno regime. Queste sono le fabbriche umane: riflettere, considerare il fatto che essere programmati significa operare su un prodotto preesistente o sulle richieste dell'utente, per creare un nuovo prodotto. La metafora della produzione incombe sempre più fortemente in questo modo di pensare. Gli aspetti produttivi dei costruttori d'auto giapponesi, della catena di montaggio, hanno ispirato molto Scrum.

Tuttavia, anche nel settore automobilistico, la parte principale del lavoro non sta tanto nella produzione, quanto nella manutenzione (o nell'evitarla il più possibile). Nel software, l'80 percento (se non di più) di quello che facciamo rientra nella categoria "manutenzione":

attività di riparazione. Invece di adottare il tipico atteggiamento occidentale di concentrarci sulla realizzazione di buon software, dovremmo cominciare a ragionare più nei panni del “manutentore” di un’abitazione o del meccanico di auto. Che cosa dice il management giapponese a questo proposito?

All’incirca nel 1951, sulla scena giapponese nacque un approccio chiamato TPM, *Total Productive Maintenance*. Tale approccio si concentrava più sulla manutenzione che sulla produzione. Uno dei grandi pilastri dell’approccio TPM erano i cosiddetti Principi delle 5 “S”. Si tratta di un insieme di *discipline*, un termine, quest’ultimo, scelto non a caso. Questi Principi delle 5 “S” sono alla base della progettazione “snella” (*lean*), un altro termine molto in voga in Occidente, e sempre più diffuso anche negli ambienti di sviluppo software. Questi principi sono imprescindibili. La buona pratica dello sviluppo software richiede proprio tale disciplina: attenzione, concentrazione e riflessione. Non si tratta solo del “fare”, *solo del fare in modo* che la macchina produttiva possa operare a pieno regime. La filosofia delle 5 “S” riguarda i seguenti concetti.

- *Seiri*, organizzazione. È fondamentale sapere dove si trovano le cose (usando approcci come una corretta denominazione). Pensate che la denominazione degli identificatori non sia poi così importante? Leggete i prossimi capitoli.
- *Seiton*, ordine. Un vecchio proverbio recita: “Un posto per ogni cosa e ogni cosa al suo posto”. Un frammento di codice deve trovarsi esattamente dove è previsto che debba trovarsi; se non si trova lì, fate in modo che lo sia.
- *Seiso*, pulizia. Tenete l’ambiente di lavoro libero da fili, olio, oggetti e materiali di scarto. Che cos’hanno da dire gli autori del libro sul codice commentato che ha il solo scopo di ricordare

passaggi precedenti o desideri per il futuro? Che dovete sbarazzarvene.

- *Seiketsu*, standardizzazione. Tutti devono concordare sul modo in cui mantenere pulito l’ambiente di lavoro. Pensate che questo libro abbia qualcosa da dire sulla coerenza dello stile di programmazione e sulla scelta di pratiche comuni nel gruppo? Da dove vengono tali standard? Continuate a leggere.
- *Shutsuke*, disciplina. Significa adottare una precisa auto-disciplina, riflettere frequentemente sul proprio lavoro ed essere pronti a cambiare.

Se accettate la sfida (sì, perché di una vera sfida si tratta) di leggere e poi mettere in pratica questo libro, finirete sicuramente per comprendere e apprezzare quest’ultimo punto. Qui giungiamo, infine, alle radici della professionalità, in un’attività che dovrebbe considerare l’intero ciclo di vita di un prodotto. Se gestiamo le automobili (e i meccanismi in genere) con un approccio TPM, la manutenzione per guasti (ovvero dopo che i bug sono ormai emersi) diventa l’eccezione. Saliamo di un livello: ispezioniamo le macchine quotidianamente e interveniamo sulle parti usurate prima che si rompano, o eseguiamo l’equivalente del classico “cambio dell’olio” ogni 10.000 km per prevenire le usure. Nel codice, dedichiamoci senza pietà al refactoring. Si può salire di un ulteriore livello, in quanto l’approccio TPM si è rinnovato oltre 50 anni fa: costruendo macchine che siano, già in partenza, di facile manutenzione. Far sì che il codice risulti leggibile è un passo importante per renderlo anche eseguibile. Le nuove tecniche, introdotte in ambiente TPM negli anni Sessanta, prevedono l’introduzione di una macchina interamente nuova o la sostituzione di quelle vecchie. Come ci esorta Fred Brooks, probabilmente dovremmo riscrivere da zero parti importanti del software all’incirca ogni sette anni, per eliminare la “sporcizia”. Forse addirittura dovremmo ridurre

questa costante temporale di Brooks, contandola invece in settimane, giorni o ore invece che in anni. È lì che si trovano i dettagli.

I dettagli recano dentro di loro una grande potenza, e vi è qualcosa di umile e profondo in questo approccio alla vita, come possiamo, come da stereotipo, attenderci da ogni approccio di origine orientale. Ma questo non è un atteggiamento solo orientale; anche i proverbii occidentali sono ricchi di ammonimenti in questo senso. La citazione che abbiamo appena scritto relativa al *Seiton* è stata usata da un pastore dell'Ohio, che considerava letteralmente l'ordine “come un rimedio per ogni tipo di malvagità”. È che dire del *Seiso*? La pulizia è l'anticamera della divinità. Per quanto possa essere bella una casa, una tavola in disordine le toglie ogni splendore. E del *Shutsuke* a proposito delle piccole cose? Chi è onesto nelle piccole cose, lo è anche in quelle grandi. Questo significa anticipare: rifattorizzare oggi, con il giusto anticipo, prendendosi tempo per le successive “grandi” decisioni, invece di procrastinare. Un attimo di tempo ora ne fa risparmiare nove domani. Il mattino ha l'oro in bocca. Non rimandare a domani ciò che puoi fare oggi (quello era il senso originale della frase “con il giusto anticipo” nell'approccio “snello” prima che tutto cadesse sotto le sgrinfie dei consulenti software). E che dire del curare le cose piccole, l'impegno sul dettaglio in vista del risultato? Da piccoli semi nascono grandi alberi. E dell'integrare del lavoro preventivo nella routine quotidiana? Meglio prevenire che curare. Una mela al giorno toglie il medico di torno. L'approccio “clean code” rende onore alle radici più profonde della nostra cultura, o almeno alla nostra cultura così come era una volta, o come forse dovrebbe essere, e il tutto basato solo sull'attenzione al dettaglio.

Anche nella grande letteratura che parla di architettura troviamo righe a supporto di questi cosiddetti dettagli. Pensate alle maniglie di Mies van der Rohe. Questo è *Seiri*. Si tratta dell'attenzione al nome di

ogni singola variable. Dovreste denominare ogni variabile con la stessa cura che dedichereste a un figlio.

Come sa chiunque abbia una casa, tali cure e tali raffinamenti non finiscono mai. L'architetto Christopher Alexander, padre del concetto di *pattern* e dei linguaggi dei *pattern*) considera ogni atto di progettazione come un piccolo atto di riparazione locale. E considera la realizzazione dei dettagli più fini della struttura l'unica competenza dell'architetto; le forme più grandi possono essere demandate ai pattern e la loro applicazione agli abitanti. La progettazione è una continua evoluzione, non solo quando aggiungiamo una nuova stanza alla casa, ma quando curiamo la tinteggiatura, quando sostituiamo i tappeti consumati o quando cambiamo il lavandino della cucina. La maggior parte delle attività reca in sé sentimenti analoghi. Nella nostra ricerca di qualcuno cui ascrivere l'idea che Dio si trovi in dettagli, ci troviamo in buona compagnia, con lo scrittore francese dell'Ottocento Gustav Flaubert. Il poeta francese Paul Valery ci confessa che una poesia non è mai terminata e richiede un lavoro continuo; smettere di lavorarvi sarebbe una resa. Tale cura per i dettagli è comune in tutte le attività che puntano all'eccellenza. Pertanto, forse qui non diciamo nulla di nuovo, ma nella lettura di questo libro verrete esortati ad adottare alcune buone pratiche alle quali potete aver rinunciato per stanchezza o per privilegiare la spontaneità, e quindi a “rispondere ai cambiamenti”.

Sfortunatamente, in genere non consideriamo tali riflessioni come elementi fondanti dell'arte della programmazione. Abbandoniamo il codice a se stesso troppo presto, non perché sia terminato, ma perché le nostre priorità riguardano più l'aspetto esteriore che la sostanza del prodotto che forniamo.

Questa superficialità ha un costo finale: alla fine il problema salta sempre all'occhio. La ricerca, in ambito professionale e accademico, si umilia limitandosi a esortare alla pulizia del codice. Ai tempi in cui

lavoravo presso i Bell Labs Software Production Research (dunque in *produzione*) avevamo alcune norme condivise, una delle quali suggeriva che uno stile di indentazione coerente era uno degli indicatori statisticamente più significativi della ridotta densità di bug. Si supponeva che l'architettura o il linguaggio di programmazione o qualche altro concetto di alto livello fosse da solo responsabile della qualità. In quanto persone che dovevano la loro professionalità alla capacità di impiegare gli strumenti disponibili e a ottimi metodi di progettazione, ci sentivamo insultati dal valore che queste macchine da produzione, i codificatori, aggiungevano tramite la mera applicazione di uno stile di indentazione coerente. Per citare il mio blocco di appunti di 17 anni fa, tale stile distingue l'eccellenza dalla mera competenza. La visione giapponese comprende il valore fondamentale del lavoratore e, ancora di più, dei sistemi di sviluppo che si fondono sulle semplici azioni quotidiane di tali lavoratori. La qualità è il risultato di milioni di singoli e disinteressati atti di cura, non solo di un qualche metodo di sviluppo piovuto dal cielo. Il fatto che questi atti siano semplici non significa che siano banali, e poi resta da dimostrare che siano davvero tanto semplici. Sono piccoli, ma ciononostante sono responsabili della grandezza, ma di più, della bellezza, di ogni attività umana. Ignorarli significa non essere pienamente umani.

Naturalmente, sono sempre convinto che si debba pensare su una scala più ampia, e in particolare sono convinto del valore degli approcci all'architettura basati su una profonda conoscenza del dominio e sull'usabilità del software. Il libro non tratta questo argomento o, almeno, non lo tratta direttamente. Questo libro ha in sé un messaggio più sottile, la cui profondità non dovrebbe essere sottovalutata. Rientra nel filone di alcuni personaggi chiave della progettazione del software, come Peter Sommerlad, Kevlin Henney e Giovanni Asproni. I loro mantra sono “The code is the design” e “Simple code”. Anche se

dobbiamo sempre ricordare che l’interfaccia è il programma, e che le sue strutture dicono molto della struttura del nostro programma, è fondamentale adottare continuamente la semplice affermazione che la struttura vive dentro il codice. E mentre la rielaborazione, secondo la metafora della produzione, introduce nuovi costi, la rielaborazione della struttura aumenta il valore. Dovremmo considerare il nostro codice come una bellissima espressione articolata di un nobile impegno di progettazione: progettazione intesa come un processo, non come un punto terminale statico. È nel codice che emergono le metriche d’architettura dell’accoppiamento e della coesione. Se sentite Larry Constantine parlarvi di accoppiamento e coesione, lo sentirete parlare di codice, non di concetti astratti in stile UML. Richard Gabriel ci dice nel suo saggio *Abstraction Descant* che l’astrazione è il male. Il codice è contro il male e il *codice pulito* è semidivino.

Tornando alla mia scatolina di Ga-Jol, penso che sia importante notare che il piccolo proverbio danese ci esorta non solo a fare attenzione alle piccole cose, ma anche a essere onesti nelle cose di poco valore. Significa essere onesti con il codice, onesti coi colleghi sullo stato del nostro codice e, soprattutto, essere onesti con noi stessi a proposito del nostro codice. Abbiamo fatto del nostro meglio per “lasciare il luogo più pulito di come l’abbiamo trovato”? Abbiamo eseguito il refactoring del nostro codice prima di fornirlo? Questi non sono dettagli ininfluenti, ma concetti che si situano proprio al centro dei valori Agile. Si tratta di una pratica consigliata in Scrum che il refactoring entri nel concetto stesso di “Finito!”. Né l’architettura né il codice pulito si basano sulla perfezione, ma solo sull’onestà e sul fatto di dare il proprio meglio. Errare è umano; perdonare è divino. In Scrum, rendiamo tutto visibile. Siamo onesti sullo stato del nostro codice perché il codice non è mai perfetto. Diventiamo così più

pienamente esseri umani, più meritevoli del divino e più vicini alla grandezza dei dettagli.

Nella nostra professione, abbiamo disperatamente bisogno di tutto l'aiuto possibile. Se un pavimento pulito riduce gli incidenti e una buona organizzazione degli attrezzi incrementa la produttività, sono assolutamente favorevole. Questo libro è la migliore espressione pragmatica dei principi “snelli” allo sviluppo di software che io abbia mai visto. Non mi aspettavo niente di meno da questo piccolo gruppo di personaggi, che per anni si sono sforzati, insieme, non solo a essere migliori, ma anche di divulgare le proprie competenze, nel loro ambito, tramite opere come quella che tenete in mano. Ora che l'editore mi ha inviato questo manoscritto sento che il mondo è un posto un po' migliore.

Ora che ho terminato questo esercizio, scritto con i miei migliori auspici, è tempo di mettere in ordine la mia scrivania.

*James O. Coplien
Mørdrup, Danimarca*

Introduzione

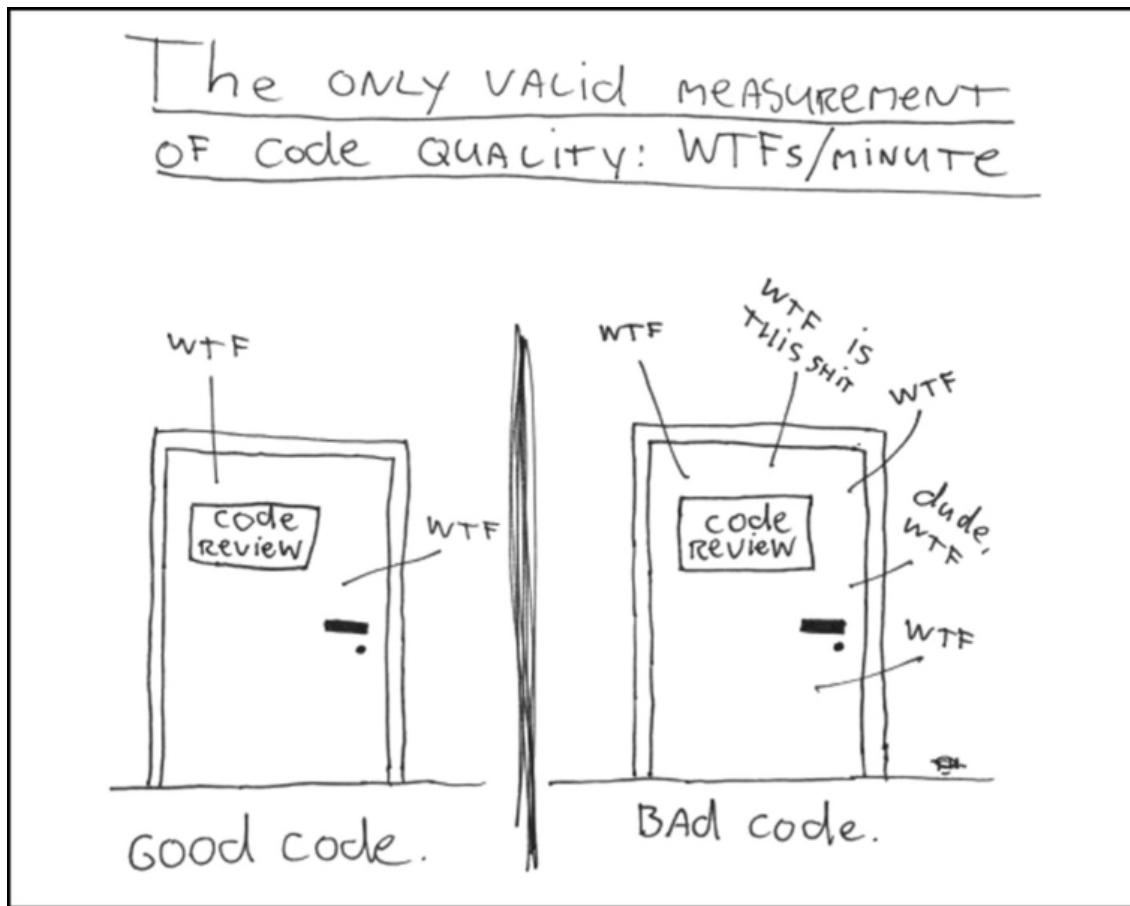


Figura I.1 Riprodotta per gentile concessione di Thom Holwerda:

http://www.osnews.com/story/19266/WTFs_m.

Quale porta rappresenta il vostro codice? Quale porta rappresenta il vostro team o la vostra azienda? Perché siamo chiusi dentro quella stanza? È solo una normale revisione di codice o abbiamo trovato un'intera sequenza di terribili problemi poco dopo l'uscita? Stiamo

eseguendo il debugging nel panico, vagando nel codice sul quale abbiamo lavorato così duramente? I clienti se ne stanno andando a frotte e i manager ci stanno col fiato sul collo? Come possiamo assicurarci di trovarci dietro la porta giusta quando il gioco si fa duro? La risposta è: l'esperienza.

Sono due i componenti che permettono di acquisire esperienza: la conoscenza e il lavoro. Occorre la conoscenza dei principi, dei pattern, delle tecniche e delle euristiche “artigianali”; e poi occorre anche mettere in pratica tale conoscenza, lavorando duramente e provando.

Potrei insegnarvi tutta la fisica su cui si basa l'andare in bicicletta. In effetti, gli aspetti matematici sono relativamente semplici. La gravità, l'attrito, il momento angolare, il centro di massa e così via, sono tutti aspetti illustrabili in una paginetta di equazioni. Sulla base di queste formule potrei dimostrarvi che andare in bicicletta è possibile e fornirvi tutta la conoscenza necessaria per farlo. E inevitabilmente, la prima volta che provaste a farlo, cadreste.

Anche programmare non è differente. Potremmo scrivere tutti i “migliori” principi di programmazione (*clean code*) e forse pensereste di poter procedere (in altre parole, vi faremmo solo cadere dalla bicicletta), ma allora che razza di insegnanti saremmo, e quale utilità avrebbe tutto ciò per voi studenti?

No. Questa non è la strada seguita da questo libro.

Imparare a scrivere codice “pulito”, *clean code*, non è facile. Richiede qualcosa di più della semplice conoscenza di principi e tecniche. Ci vuole anche il “sudore”. Dovete fare pratica, e anche andare incontro a insuccessi. Dovete osservare gli altri mentre provano e falliscono. Dovete vederli inciampare e tornare sui loro passi. Dovete vederli soffrire per le loro decisioni e vedere il prezzo che pagano per aver preso decisioni sbagliate.

Preparatevi a lavorare duramente mentre leggerete questo libro. Questo non è un “libro facile” che potete leggere in aereo e terminare prima dell’atterraggio. Questo libro vi farà lavorare, ma lavorare duramente. Quale tipo di lavoro dovrete svolgere? Leggerete del codice, grandi quantità di codice. E vi verrà chiesto di riflettere su quello che c’è di giusto e quello che c’è di sbagliato in tale codice. Vi verrà chiesto di seguire i passaggi mentre vengono estratti dei moduli, che poi vengono ricombinati in modo differente. Questo richiederà tempo e impegno; ma pensiamo che ne valga la pena.

Abbiamo diviso questo libro in tre parti. I primi capitoli descrivono i principi, i pattern e le tecniche del codice pulito. Vi troverete una grande quantità di codice, e riuscire a comprenderlo non sarà sempre facile. Questa prima parte vi prepara alla seconda. E se dovreste arrendersi dopo aver letto i primi capitoli... buona fortuna a voi!

La seconda parte del libro è più solida. È costituita da numerosi casi di studio di complessità crescente. Ogni caso di studio è un esercizio di “pulizia” di un frammento di codice o di trasformazione di codice problematico in codice meno problematico. L’attenzione al dettaglio sarà notevole. Vi troverete a scorrere, avanti e indietro, fra descrizione e i listati. Dovrete analizzare e comprendere il codice sul quale stiamo lavorando e seguire i ragionamenti proposti e le motivazioni che hanno spinto a effettuare ogni singola modifica. Dedicategli del tempo, perché l’impegno vi richiederà giorni.

La terza parte di questo libro è il lascito finale. Si tratta di un unico capitolo contenente un elenco di euristiche e intuizioni raccolte nel corso della creazione dei casi di studio. Mentre elaboravamo e raffinavamo il codice dei casi di studio, abbiamo documentato ogni motivazione delle nostre azioni sotto forma di euristiche e avvertenze. Abbiamo tentato di studiare le nostre stesse reazioni al codice che stavamo leggendo e modificando, e abbiamo cercato di catturare il

perché abbiamo ragionato in un certo modo e abbiamo agito di conseguenza. Il risultato è una base di conoscenze che descrive il modo in cui ragioniamo quando scriviamo, leggiamo e raffiniamo il codice.

Questa base di conoscenze non avrà particolare valore se non avrete svolto il lavoro di leggere con cura i casi di studio presentati nella seconda parte di questo libro, nei quali abbiamo annotato con cura ogni modifica apportata, con riferimenti in avanti alle euristiche. Questi riferimenti in avanti sono specificati fra parentesi quadre nel seguente modo: [H22]. Ciò vi permette di vedere il contesto in cui tali euristiche sono state applicate e scritte! Le euristiche, da sole, non sono sufficientemente utili: a contare davvero è la relazione fra tali euristiche e le singole decisioni che abbiamo preso mentre raffinavamo il codice dei casi di studio.

Per agevolarvi ulteriormente a proposito di queste relazioni, abbiamo previsto a fine libro un indice di questi riferimenti. Potete quindi usarlo per trovare agevolmente la posizione in cui è stata applicata una determinata euristica.

Se leggete solo la prima e la terza sezione, saltando del tutto i casi di studio, vi ritrovereste a leggere l'ennesimo libro "leggero" sulla scrittura del buon software. Se invece dedicherete un tempo adeguato ai casi di studio, seguendo ogni singolo passo, ogni più piccola decisione, se vi metterete nei nostri panni e vi costringerete a seguire i nostri stessi percorsi mentali, acquisirete una comprensione molto più ricca di questi principi, pattern, tecniche ed euristiche. Non si tratterà più di una conoscenza "leggera". Vi sarà entrata nelle dita, negli occhi e nella mente. Diventerà un automatismo, così come lo è andare in bicicletta.

Ringraziamenti

Grazie alle mie due illustratrici, Jeniffer Kohnke e Angela Brooks. Jennifer è autrice delle simpatiche e creative immagini che aprono ogni

capitolo e anche dei ritratti di Kent Beck, Ward Cunningham, Bjarne Stroustrup, Ron Jeffries, Grady Booch, Dave Thomas, Michael Feathers e mio.

Angela, invece, è autrice delle illustrazioni interne dei capitoli. Aveva già preparato per me varie immagini nel corso degli anni, fra le quali molte immagini presenti nel libro *Agile Software Development: Principles, Patterns, and Practices*. È anche la mia primogenita, della quale sono molto orgoglioso.

Un ringraziamento particolare va ai miei revisori, Bob Bogetti, George Bullock, Jeffrey Overbey e in particolare Matt Heusser. Sono stati brutali, crudeli, implacabili. Grazie a ciò mi hanno permesso di apportare i miglioramenti necessari.

Ringrazio il mio editor, Chris Guzikowski, per il suo sostegno, incoraggiamento e per la sua giovialità. Grazie anche al personale editoriale di Pearson, fra cui Raina Chrobak, per avermi permesso di essere tranquillo e puntuale.

Grazie a Micah Martin e a tutto il gruppo di 8th Light (<http://www.8thlight.com>) per le loro indicazioni e il loro incoraggiamento.

Grazie agli *Object Mentors*, di ieri, di oggi e di domani, fra i quali: Bob Koss, Michael Feathers, Michael Hill, Erik Meade, Jeff Langr, Pascal Roy, David Farber, Brett Schuchert, Dean Wampler, Tim Ottlinger, Dave Thomas, James Grenning, Brian Button, Ron Jeffries, Lowell Lindstrom, Angelique Martin, Cindy Sprague, Libby Ottlinger, Joleen Craig, Janice Brown, Susan Rosso e tanti altri.

Grazie a Jim Newkirk, mio amico e partner professionale, che mi ha insegnato molto di più di quanto possa immaginare. Grazie a Kent Beck, Martin Fowler, Ward Cunningham, Bjarne Stroustrup, Grady Booch e a tutti gli altri miei mentori, compatrioti e colleghi. Grazie a John Vlissides per esserci stato nei momenti che contano. Grazie ai ragazzi

di Zebra per aver sopportato i miei vaneggiamenti sulla lunghezza
“giusta” di una funzione.

E, infine, grazie a voi per aver letto tutti questi ringraziamenti.

Capitolo 1

Codice pulito



State leggendo questo libro per due motivi. Innanzitutto, siete programmatori. In secondo luogo, volette diventare buoni programmatori. Bene. Abbiamo bisogno di buoni programmatori.

Questo è un libro sulla buona programmazione. Per questo motivo è ricco di codice. Esamineremo il codice da ogni angolazione possibile. Lo osserveremo dall'alto, lo esamineremo dal basso e gli guarderemo anche dentro. Entro la fine del libro vi ritroverete a conoscere molto di più sul codice. Ma in più, vi aiuteremo a distinguere il buon codice dal cattivo codice. Imparerete così le tecniche e i dettagli da considerare per scrivere buon codice. E imparerete anche a trasformare il cattivo codice in buon codice.

Che il codice sia!

Si potrebbe ritenere che un libro sul codice sia un po' fuori tempo, ovvero che il codice non rappresenti più un problema; che piuttosto occorrerebbe riflettere su modelli e requisiti. Addirittura, qualcuno ha suggerito che saremmo prossimi alla fine del codice. Che presto tutto il codice verrà generato e non più scritto. Che i programmatore, semplicemente, non saranno più necessari, perché saranno i dirigenti a generare i programmi a partire dalle specifiche.

Qualcuno si dice convinto... Non ci libereremo mai del codice, perché il codice rappresenta i dettagli dei requisiti.

Oltre un certo livello, tali dettagli non possono essere ignorati o astratti; devono essere specificati. E specificare i requisiti a un livello di dettaglio tale che una macchina possa eseguirli secondo un *programma*. Tale specifica è *il codice*.

Mi aspetto che il livello di astrazione dei nostri linguaggi continuerà ad aumentare. Mi aspetto anche che il numero di linguaggi specifici di un dominio continuerà a crescere. Questa sarà una buona cosa. Ma essi non elimineranno il codice. In effetti, tutte le specifiche scritte in questi linguaggi di alto livello e specifici di un dominio *saranno...* codice! E dovrà comunque essere rigoroso, accurato e sufficientemente formale e dettagliato affinché una macchina possa comprenderlo ed eseguirlo.

Coloro che pensano che il codice un giorno sparirà sono come i matematici che sperano che un giorno verrà scoperta una matematica che non deve essere formale. Sperano che un giorno scopriremo un modo per creare macchine che possano fare ciò che vogliamo invece di ciò che diciamo loro. Queste macchine dovranno essere in grado di comprenderci così bene da poter tradurre esigenze specificate in modo vago in programmi perfettamente funzionanti che rispondano esattamente a tali esigenze.

Ciò non accadrà mai. Nemmeno gli esseri umani, con tutto il loro intuito e la loro creatività, sono stati in grado di creare sistemi di successo dalle vaghe ipotesi dei loro clienti. In effetti, se la disciplina della specifica dei requisiti ci ha insegnato qualcosa, è che i requisiti ben specificati sono altrettanto formali del codice e possono fungere da test eseguibili per tale codice!

Ricordate che il codice in realtà è il linguaggio in cui, alla fine, esprimiamo i requisiti. Potremmo creare linguaggi che siano più vicini ai requisiti. Potremmo creare strumenti che ci aiutano ad analizzare e assemblare tali requisiti in strutture formali. Ma non elimineremo mai la precisione necessaria e quindi il codice esisterà sempre.

Cattivo codice



Recentemente stavo leggendo la prefazione del libro di Kent Beck, *Implementation Patterns* [Beck07]. Dice: “... questo libro si basa su una premessa piuttosto fragile: che sia importante usare buon codice...”. Una premessa *fragile*? Non sono d'accordo! Penso che tale premessa sia una delle più solide, supportate e ricche di significato di tutte le premesse possibili nel nostro campo (e penso che Kent lo sappia). Sappiamo che il buon codice è importante perché così tante volte ne abbiamo sentito la mancanza.

So di un’azienda che, verso la fine degli anni Ottanta, scrisse un’applicazione *eccezionale*. Era molto popolare, e molti professionisti la acquistarono e utilizzarono. Ma poi i cicli delle release iniziarono ad allungarsi. I bug non venivano risolti da una release alla successiva. I tempi di caricamento si allungavano e aumentavano anche i crash. Mi ricordo il giorno in cui ho chiuso il prodotto per l’ultima volta al culmine della frustrazione: non l’ho mai più usato. Poco dopo l’azienda chiuse i battenti.

Due decenni più tardi ho incontrato uno dei primi dipendenti di tale azienda e gli ho chiesto cosa fosse accaduto. La sua risposta confermò i miei timori. Avevano messo il prodotto sul mercato in fretta e furia e avevano fatto un bel... ehm... “groviglio” nel codice. Aggiungendogli poi sempre più funzionalità, il codice peggiorò sempre più, fino al punto che era diventato ingestibile. È stato il cattivo codice a far chiudere l’azienda.

Siete mai stati significativamente bloccati a causa di codice mal realizzato? Se siete programmatori di una certa esperienza probabilmente lo avete sperimentato molte volte. In effetti, esiste anche un termine per descrivere quella sensazione. In inglese è “wading”, che in italiano suona “guado” [ma fa anche pensare al “guano”..., NdT]. Ci troviamo a “guadare” nel cattivo codice. Attraversiamo come un acquitrino pieno di rovi e di trappole nascoste. Tentiamo disperatamente di procedere, sperando in un po’ di fortuna, in qualche indizio, che ci schiarisca il cammino; ma tutto ciò che vediamo intorno è sempre e solo codice senza capo né coda.

Naturalmente vi sarete sentiti bloccati dal cattivo codice. Ma allora... perché mai l’avete scritto così?

Cercavate di essere più veloci? Avevate fretta? Probabilmente è così. Forse pensavate di non avere il tempo di fare un buon lavoro; che il capo si sarebbe arrabbiato con voi se vi foste “persi” a raffinare il

vostro codice. Forse eravate semplicemente stanchi di lavorare su quel programma e non vedevate l'ora di finirlo. O magari avete dato un'occhiata ai lavori arretrati che si accumulavano e vi siete resi conto che dovevate mettere insieme in qualche modo questo modulo per poter passare al successivo. L'abbiamo fatto tutti.

Tutti abbiamo guardato disgustati all'obbrobrio che avevamo appena messo insieme e poi abbiamo deciso di rimandare il problema a un altro momento. Tutti abbiamo sentito il sollievo di vedere il nostro diavolo di programma funzionare e abbiamo deciso che un "qualsiasi" funzionante è sempre meglio di nulla. Tutti abbiamo detto "Torno a lavorarci più tardi". Naturalmente, a quei tempi non conoscevamo la legge di LeBlanc: "Più tardi significa mai".

Il costo totale di possedere un vero groviglio di codice

Se siete programmati da almeno due o tre anni, probabilmente vi sarete ritrovati significativamente rallentati nel vostro lavoro da codice pasticcato scritto da altri. Se poi siete programmati da ben più di due o tre anni, probabilmente vi sarà capitato di aver scritto codice pasticcato che ha rallentato il lavoro vostro o altrui. Il grado di questo rallentamento può essere significativo. Nell'arco di un anno o due, i team che stavano procedendo molto rapidamente all'inizio di un progetto possono trovarsi a procedere a passo di lumaca. Ogni modifica apportata al codice pregiudica il funzionamento di altre due o tre parti del codice. Nemmeno la più piccola modifica risulta facile. Ogni aggiunta o modifica al sistema richiede di risalire ai grovigli, alle pieghe e ai nodi attualmente presenti, per poter aggiungere ulteriori grovigli, pieghe e nodi. Nel corso del tempo la "matassa" diviene così

grossa e così intricata, che è davvero impossibile trovarne il verbale bando. Assolutamente meglio lasciar perdere.

Mentre il groviglio cresce, la produttività del team continua a decrescere, approssimandosi asintoticamente allo zero. Mentre la produttività decresce, il management fa l'unica cosa possibile; aumenta lo staff incaricato del progetto nella speranza di incrementare la produttività. Ma quel nuovo staff non conosce la struttura del sistema. Non è in grado di distinguere fra una modifica che rientra nella struttura del sistema e una modifica che la sovverte completamente. Di più: il nuovo staff, e tutto il team in generale, sono sottoposti a tremende pressioni che spingono ad aumentare la produttività. Di conseguenza combinano ancora più grovigli, riducendo la produttività sempre più in prossimità dello zero (Figura 1.1).

La grande riprogettazione cosmica

Alla fine i membri del team si ribellano. Informano il management di non poter più continuare a sviluppare con questa inestricabile base di codice. Richiedono una riprogettazione. Da capo. Il management non intende dedicare risorse a una vera ri-costruzione del progetto, ma nessuno può negare che la produttività è a livelli infimi. Alla fine si piegano alle richieste degli sviluppatori e autorizzano la grande riprogettazione cosmica.

Viene selezionato un nuovo “tiger” team. Tutti vogliono far parte di questo team, perché si tratta di un nuovo progetto. Non vedono l'ora di iniziare e di creare qualcosa di davvero eccezionale. Ma solo i migliori e i più competenti vengono scelti per entrare a far parte del nuovo team. Gli altri devono continuare a svolgere la manutenzione del sistema attuale.

Ora abbiamo due team in competizione. Il tiger team deve realizzare un nuovo sistema che faccia tutto ciò che faceva il vecchio sistema. Ma

non solo, il nuovo sistema deve tenere il passo con le aggiunte applicate al vecchio sistema. Il management non sostituirà il vecchio sistema finché il nuovo sistema non potrà fare tutto ciò che faceva il vecchio.

Questa competizione può durare moltissimo tempo. Si è trattato di dieci anni. E al termine della competizione, i membri iniziali del “tiger” team se ne erano andati già da parecchio, e gli attuali membri chiedono che il nuovo sistema venga riprogettato perché è un tale groviglio...

Se vi è capitata anche una piccola parte della storia che ho appena raccontato, allora sapete già che dedicare del tempo a curare la pulizia del codice non è solo una questione economica; è una questione di sopravvivenza professionale.

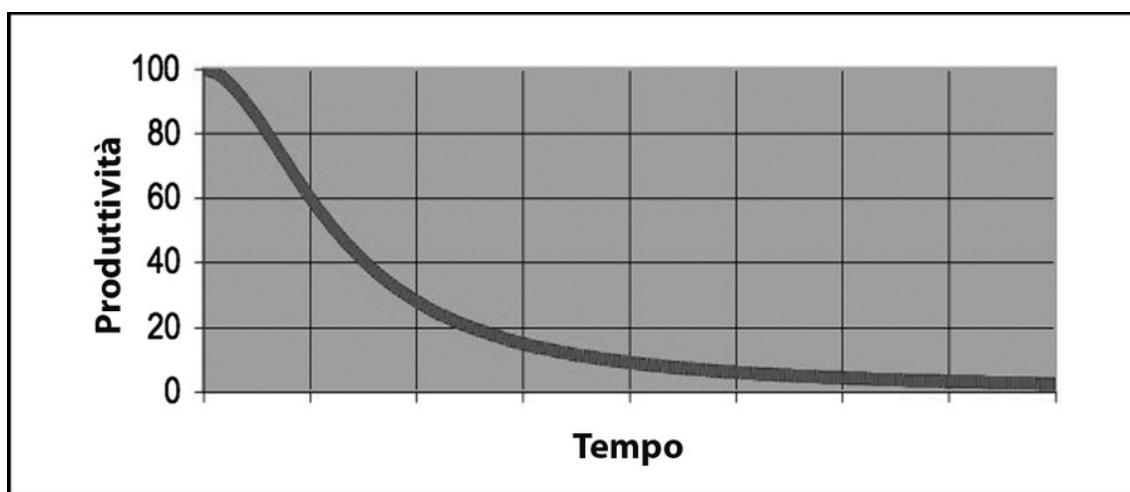


Figura 1.1 Produttività vs. tempo.

Una questione di atteggiamento

Vi è mai capitato di dover affrontare un groviglio di tale complessità che un intervento da poche ore finisce per richiedere settimane? Vi è mai capitato di dover intervenire su una singola riga di codice, e di essere poi costretti a modificare centinaia di moduli? Questi sintomi sono fin troppo comuni.

Perché accade tutto questo, al codice? Perché il buon codice si trasforma così rapidamente in cattivo codice? Le spiegazioni sono molte. Ci lamentiamo del fatto che i requisiti sono cambiati al punto tale da sovvertire completamente la struttura iniziale. Protestiamo perché i tempi sono troppo stretti per poter fare le cose nel modo giusto.

Chiacchieriamo degli stupidi manager e dei clienti insopportabili, di iniziative marketing inutili e dei filtri per le telefonate. Ma il problema, cari miei, non sta tutto attorno, ma in noi stessi. Non siamo professionali.

Sì, è una “pillola amara”, lo so. Come è possibile che quel groviglio sia solo colpa *nostra*? Hai dimenticato i requisiti? E che dire dei tempi? Lo sappiamo tutti che i nostri manager non capiscono niente e che il marketing punta sempre nelle direzioni sbagliate! Possibile che non abbiano nessuna colpa?

No. I manager e il marketing guardano a noi per le informazioni di cui hanno bisogno per promettere tempi e risultati; e anche quando non ci considerano, non dovremmo essere titubanti nel dire loro ciò che pensiamo. Gli utenti guardano a noi per convalidare il modo in cui i requisiti andranno a trasformarsi in un sistema. I project manager guardano a noi per cercare di stabilire le scadenze. Siamo profondamente complici nella pianificazione del progetto e condividiamo gran parte delle responsabilità in tutti i fallimenti; in particolare se tali fallimenti sono dovuti a cattivo codice!

“Ma... aspetta un attimo!”, direte. “Se non faccio quello che mi dice il capo, vengo licenziato.” Probabilmente no. La maggior parte dei manager vuole conoscere la verità, anche se sembrerà non tenerne conto. La maggior parte dei manager vuole ottenere buon codice, anche se si comporta in modo ossessivo sui tempi. Possono difendere con passione le scadenze e i requisiti; ma questo è il loro lavoro. Il *vostro*, invece, è quello di difendere il codice con uguale passione.

Per esemplificare, immaginate di essere un chirurgo e che il vostro paziente vi dica di non perdere tutto quel tempo a lavare e disinfeccare tutti i ferri e i materiali (quando nel 1847, per la prima volta, Ignaz Semmelweis raccomandò il lavaggio delle mani, fu respinto sulla base del fatto che i medici erano troppo occupati e non avrebbero avuto il tempo di lavarsi le mani tra una visita e l'altra). Chiaramente il paziente è il “capo”; e tuttavia il chirurgo deve assolutamente rifiutarsi di accettare. Perché? Perché il chirurgo conosce molto più del paziente quali sono i rischi di infezione. Sarebbe ben poco professionale (per non dire criminale) che il chirurgo accogliesse le richieste del paziente.

È altrettanto poco professionale che i programmati accettino le richieste dei manager che non comprendono i rischi del creare grovigli.

Il dilemma di base

I programmati affrontano un dilemma fra priorità. Tutti gli sviluppatori con qualche anno di esperienza sanno che i grovigli di software sui quali si sono trovati a lavorare li hanno rallentati parecchio. E tuttavia tutti gli sviluppatori sentono la pressione a creare nuovi grovigli per stare nelle scadenze. In breve, non si prendono il tempo necessario per procedere speditamente!

I veri professionisti sanno che la seconda parte del dilemma è sbagliata. *Non* si rientra nelle scadenze creando un groviglio. In effetti, il groviglio produce i suoi effetti di rallentamento quasi istantaneamente, e vi costringerà a non rispettare le scadenze. L'unico modo per rientrare nelle scadenze, l'unico modo per essere veloci, consiste nel mantenere, costantemente, il codice il più pulito possibile.

L'arte della pulizia del codice

Supponiamo che siate convinti che il codice mal realizzato sia un significativo impedimento. Supponiamo che accettiate il fatto che l'unico modo per procedere rapidamente consista nel mantenere “pulito”, *clean*, il codice. Allora dovete domandarvi: “Come si scrive del codice pulito?”. Non ha senso tentare di scrivere codice pulito se non si sa esattamente cosa significhi scrivere codice pulito!

Il problema è che scrivere codice pulito è un po’ come dipingere. Molti di noi sanno distinguere un quadro ben o mal realizzato. Ma la capacità di distinguere una bella opera da una brutta non fa di noi dei pittori. Analogamente, la capacità di distinguere il codice pulito dal codice mal realizzato non ci insegna a scrivere codice pulito!

Scrivere codice pulito richiede un impiego disciplinato di una miriade di piccole tecniche applicate tramite un senso, faticosamente appreso, di “pulizia”. Proprio questo “sesto senso” è la chiave. Alcuni di noi lo hanno, innato. Altri devono faticare per acquisirlo. Questo senso non solo ci permette di vedere se il codice è “buono” o “cattivo”, ma ci mostra anche la strategia per applicare questa nostra disciplina alla trasformazione di codice “sporco” in codice pulito.

Un programmatore senza questo “sesto senso” può anche vedere un modulo mal realizzato e riconoscere il groviglio, ma non saprà da che parte iniziare per risolvere il problema. Un programmatore *dotato* di questo “sesto senso” vedrà un modulo mal realizzato e ne trarrà opzioni e varianti. Il “sesto senso” aiuterà tale programmatore a scegliere la variante migliore e lo guiderà a tracciare una sequenza di comportamenti atti a curare le trasformazioni necessarie per partire da lì e arrivare dove è necessario.

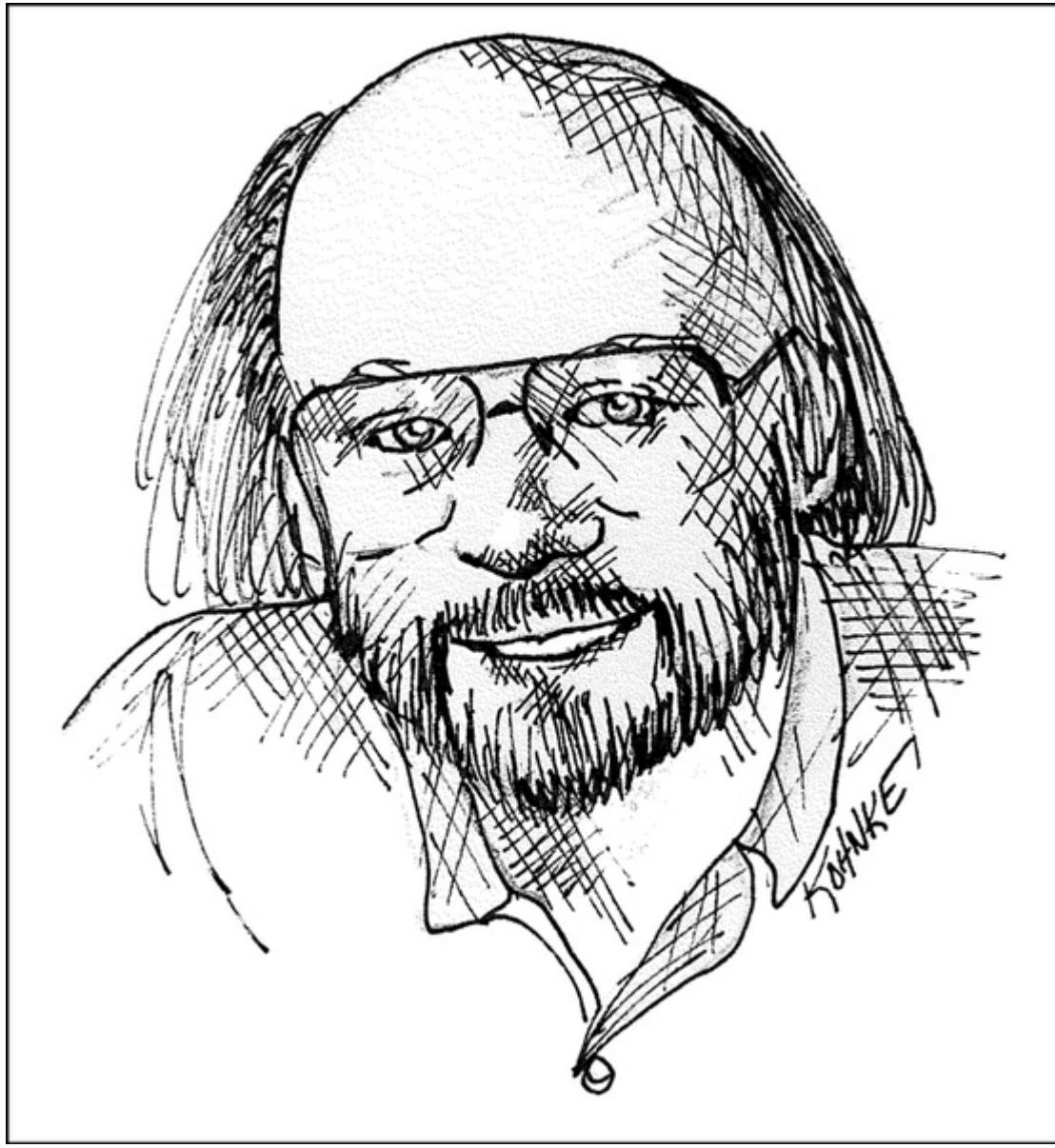
In breve, un programmatore che scrive codice in modo pulito è come un artista che parte da una tela bianca e le applica una serie di trasformazioni vino a ottenere un sistema programmato in modo elegante.

Che cosa si intende con “codice pulito”?

Probabilmente esistono tante definizioni quanti sono i programmatori.
Pertanto ho chiesto cosa ne pensavano ad alcuni programmatori di
chiara fama ed esperienza.

Bjarne Stroustrup

Inventore di C++ e autore di C++: *linguaggio, libreria standard,
principi di programmazione.*



Voglio che il mio codice sia elegante ed efficiente. La sua logica deve essere evidente, in modo da ridurre al minimo la possibilità che nasconda dei bug, le sue dipendenze minime, per facilitare la manutenzione, la sua gestione degli errori completa, secondo una strategia articolata, e le sue prestazioni il più possibile ottimali, per evitare che altri siano tentati di pregiudicare la correttezza del codice con ottimizzazioni scriteriate. Il codice pulito fa una cosa sola e la fa bene.

Bjarne usa la parola “elegante”. Non è una parola da poco! Il dizionario ne dà varie definizioni: *gradevole e in stile nell’aspetto o*

nei modi; piacevolmente ingegnoso e semplice. È bello l'uso della parola “piacevole”.

Apparentemente Bjarne pensa che il codice pulito sia *piacevole* da leggere. Si dovrebbe avere la stessa sensazione che si ha quando si osserva un bel juke-box o una bella automobile.

Bjarne menziona anche efficienza. Forse questo non dovrebbe sorprenderci proveniendo dall'inventore del C++; ma penso che dietro ci sia qualcosa di più della pura ricerca della velocità.

I cicli a vuoto sono ineleganti, per niente piacevoli. E notate anche la parola che Bjarne usa per descrivere le conseguenze di tale ineleganza. Usa il verbo “tentare”. Qui si cela una profonda verità. Il cattivo codice *induce* la tentazione di far crescere il groviglio! Quando qualcuno si trova a dover modificare del codice mal realizzato, tende a peggiorarlo ulteriormente.

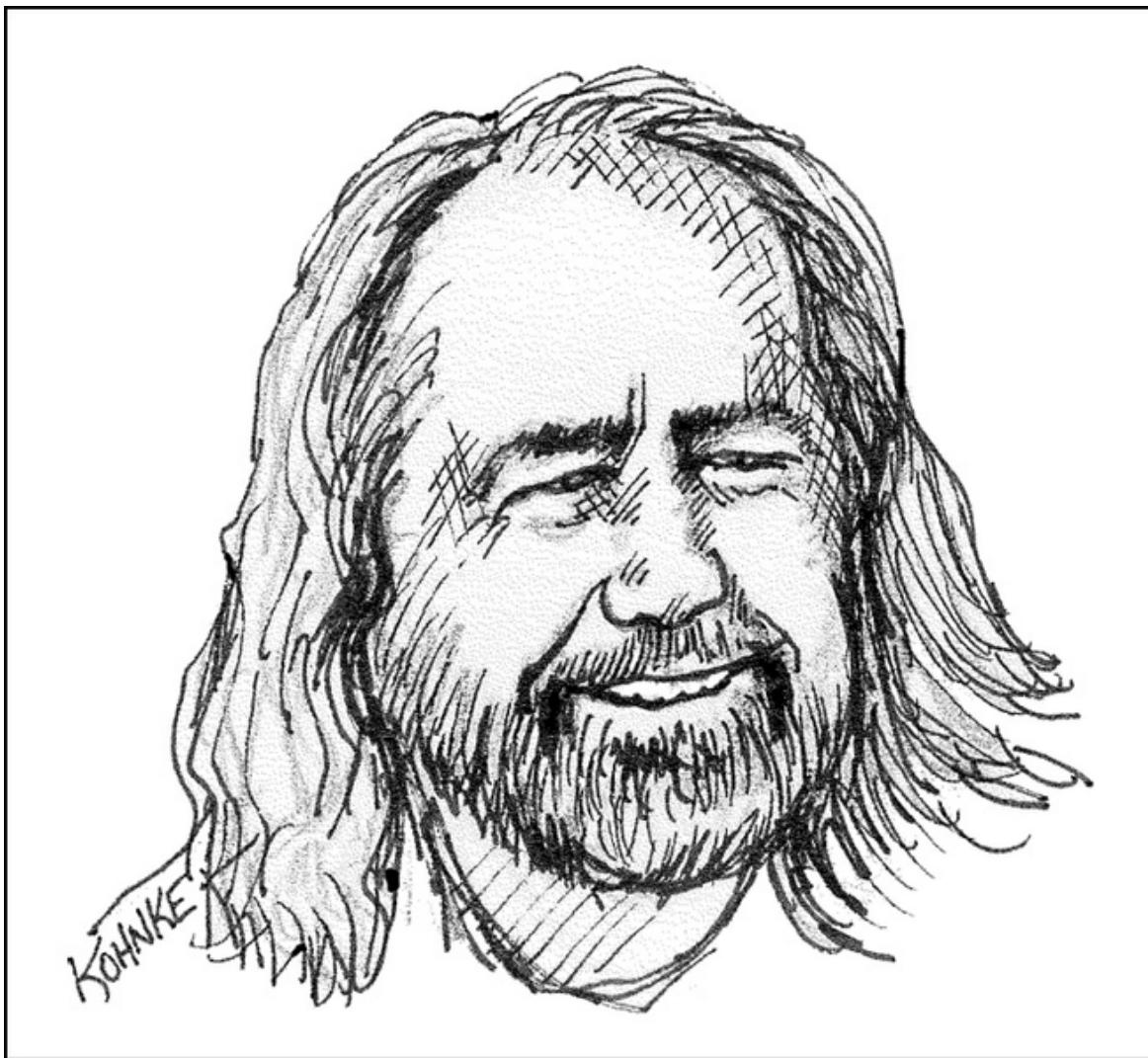
Dave Thomas e Andy Hunt, più pragmatici, ne hanno parlato in un altro modo. Hanno usato la metafora delle finestre rotte. Un edificio con delle finestre rotte sembra abbandonato. Così anche gli altri smettono di curarsene. Così iniziano a rompersi sempre più finestre. Alla fine cominciano a rompere attivamente anche quelle superstite. Qualcuno inizia a deturparne la facciata con graffiti e anche l'immondizia inizia ad accumularsi nei paraggi. Tutto il processo di degrado è però partito da un'unica finestra rossa.

Bjarne dice anche che la gestione degli errori deve essere completa. Questo rientra in una disciplina di attenzione ai dettagli. Una gestione degli errori approssimativa è solo uno dei modi in cui i programmati trascurano i dettagli. Un altro sono gli sprechi di memoria, le competizioni per le risorse sono un'altra. Un'altra ancora? Convenzioni incoerenti di denominazione. In pratica per ottenere codice pulito occorre una grande attenzione ai dettagli.

Bjarne conclude affermando che il codice pulito deve fare una cosa e farla bene. Non è un caso che così tanti principi di progettazione del software possano essere ridotti a questa semplice ammonizione. Tanti programmatore hanno tentato di comunicare questo concetto. Il cattivo codice tenta di fare troppe cose: il suo scopo è confuso e ambiguo. Il codice pulito è *specifico*. Ogni funzione, ogni classe, ogni modulo espone un unico intento che rimane del tutto distinta, senza “inquinamenti”, rispetto ai dettagli circostanti.

Grady Booch

Autore del libro *Object Oriented Analysis and Design with Applications*.



Il codice pulito è semplice e diretto. Il codice pulito si legge come una prosa ben scritta. Il codice pulito non cela mai l'intento dello sviluppatore, ma piuttosto è pieno di immediate astrazioni e di un flusso di controllo chiaro.

Grady ribadisce alcuni dei punti espressi da Bjarne, ma adotta la prospettiva della *leggibilità*. Mi piace in particolare quando dice che il codice pulito dovrebbe leggersi “come una prosa ben scritta”.

Ripensate a un bel libro che avete letto. Ricordate come le parole sparivano per lasciare il posto alle immagini? Non era un po’ come assistere a un film? Forse meglio ancora! Vedevate i personaggi, udivate i suoni, percepivate la tensione e l’humor.

Leggere del codice pulito non sarà mai come leggere *Il signore degli anelli*. Ciononostante, la metafora letteraria non è affatto fuori luogo. Come un bel racconto, il codice pulito dovrebbe esporre chiaramente le tensioni insite nel problema da risolvere. Dovrebbe spingere le tensioni fino a un apice e poi dare al lettore quell’“Ahaaa! Naturalmente!”, dove i problemi e le tensioni si risolvono grazie a una soluzione ovvia.

Poi trovo che l'accostamento di Grady “immediate astrazioni” sia un affascinante ossimoro! Sembra che le due parole non possano essere messe in combinazione fra loro, ma così facendo convogliano un messaggio potente. Il nostro codice deve essere immediato e non speculativo. Deve contenere solo il necessario. I nostri lettori devono percepire le nostre scelte come decisive.

“Big” Dave Thomas

Fondatore di OTI, padrino della strategia Eclipse.



Il codice pulito può essere letto ed esteso anche da uno sviluppatore che non sia l'autore. È dotato di unit test e test di accettabilità. Usa nomi significativi. Fornisce un unico modo per svolgere un'unica operazione. Presenta dipendenze minime, definite esplicitamente, e fornisce un'API chiara e minima. Il codice dovrà essere autodocumentante, in quanto, a seconda del linguaggio, non sempre tutte le informazioni necessarie possono essere espresse chiaramente nel solo codice.

Big Dave condivide il desiderio di Grady: la leggibilità, ma con una variante importante. Dave sostiene che il codice pulito facilita le estensioni da parte di *altri* sviluppatori. Questo può sembrare ovvio,

ma è sempre meglio considerarlo. Dopotutto c'è una certa differenza fra il codice facile da leggere e il codice facile da modificare.

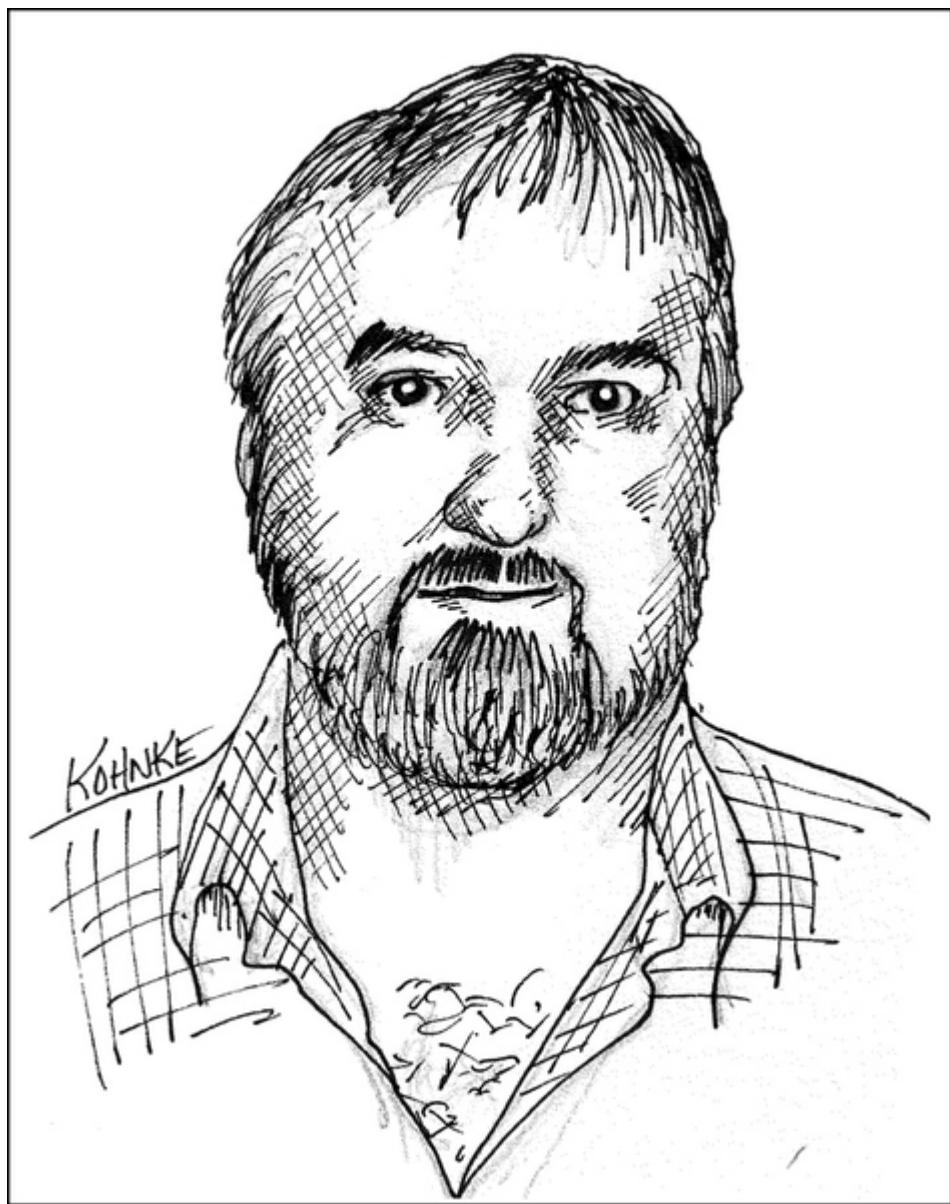
Dave lega la pulizia ai test! Dieci anni fa questa affermazione avrebbe fatto alzare più di un sopracciglio. Ma la disciplina Test-Driven Development ha avuto un impatto profondo nel settore e ormai è preminente. Dave ha ragione. Il codice, senza test, non è del tutto “pulito”. Per quanto possa essere elegante, per quanto possa essere leggibile e accessibile, senza test, è ancora da raffinare.

Dave usa due volte l'aggettivo “minimo”. Apparentemente gli piace il codice compatto, più del codice troppo ampio. In effetti, questo è stato un tipico *refrain* di tutta la letteratura dedicata al software, fin dagli inizi. *Smaller is better.*

Dave dice anche che il codice deve essere *autodocumentante*. Questo è un riferimento all'idea di Knuth [Knuth92], che il codice debba essere scritto in una forma tale da poter essere leggibile dagli esseri umani.

Michael Feathers

Autore del libro *Working Effectively with Legacy Code*.



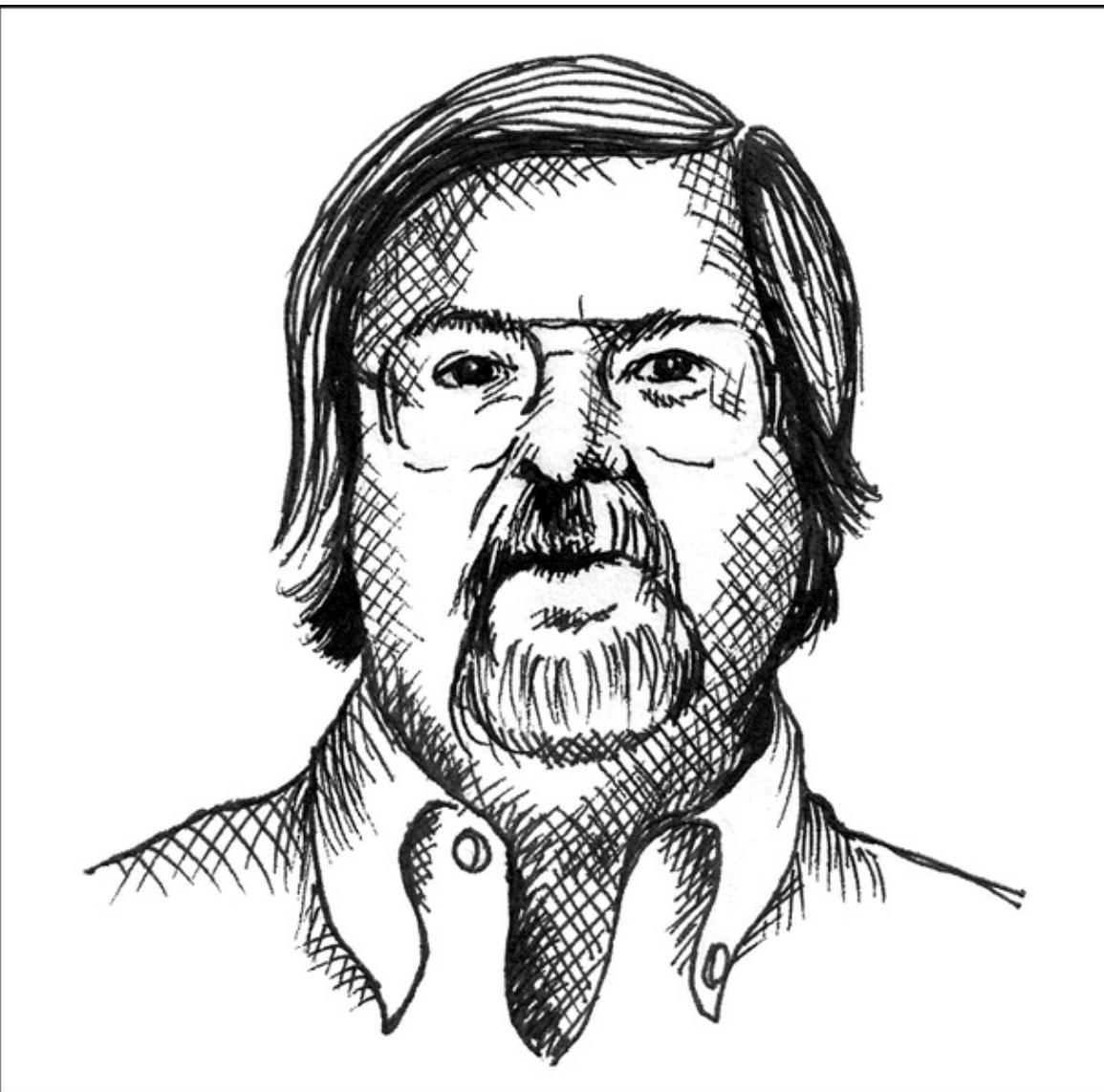
Potrei elencare tutte le qualità del codice pulito, ma ve n'è una che sovrasta tutte le altre. Il codice pulito sembra sempre essere scritto da qualcuno che ha lavorato con cura. Non ha nulla di ovvio che permetta di migliorarlo. Tutto, nel codice, è già stato considerato dall'autore, e se si prova a immaginare dei miglioramenti, poi si ritorna al punto di partenza, apprezzando ancora di più il codice che è stato scritto, codice realizzato da qualcuno che ha lavorato con cura.

Sì, “con cura”. Proprio questo è il vero argomento di questo libro.
Forse un sottotitolo adatto sarebbe: “Come aver cura del codice”.

Michael va dritto al punto. Il codice pulito è codice realizzato con cura. Qualcuno gli ha dedicato del tempo perché fosse semplice e ordinato. Qualcuno ha dedicato attenzione ai dettagli. Qualcuno se n'è curato.

Ron Jeffries

Autore dei libri *Extreme Programming Installed* ed *Extreme Programming Adventures in C#*.



Ron ha iniziato la sua carriera di programmatore in Fortran presso lo Strategic Air Command e ha scritto codice in quasi ogni linguaggio e per quasi ogni macchina. Vale la pena di considerare con attenzione le sue parole.

Negli ultimi anni ho iniziato, e quasi terminato, le regole di Beck per la semplificazione del codice. In ordine di priorità, il codice semplice:

- supera tutti i test;
- non contiene duplicazioni;
- esprime tutte le idee progettuali presenti nel sistema;
- minimizza il numero di entità (classi, metodi, funzioni e così via).

Fra questi punti, mi interessa principalmente la duplicazione. Quando viene ripetuta continuativamente la stessa operazione, è segno che un'idea che avete in mente non è ben rappresentata nel codice. Occorre scoprire tale idea. Poi tentare di esprimere tale idea più chiaramente.

L'espressività, per me, comprende l'uso di nomi significativi, e spesso mi capita di cambiare il nome delle cose più volte, prima di esserne pienamente convinto. Coi nuovi strumenti di programmazione come Eclipse, ogni cambio di nome è davvero "economico", e quindi cambiare nomi non è più un problema. Ma l'espressività va ben oltre i nomi. Mi sincero anche del fatto che un oggetto o un metodo non faccia più di una cosa. Se si tratta di un oggetto, probabilmente dovrà essere suddiviso in due o più oggetti. Se si tratta di un metodo, occorrerà usare Extract Method per eseguirne il refactoring, ottenendo così un metodo che dice più chiaramente ciò che fa, più alcuni sottometodi che dicono come questo deve essere fatto.

La duplicazione e l'espressività già mi portano piuttosto avanti verso quello che considero essere codice pulito; inoltre, anche intervenire in questi due sensi su codice mal realizzato può fare una grande differenza. C'è però un'altra operazione che mi premuro di fare, e che sarà un po' più complicata da spiegare.

Dopo anni di questo lavoro, mi sembra che tutti i programmi siano costituiti da elementi molto simili. Un esempio è "trova delle cose in una raccolta". Che si tratti di un database dei record dei dipendenti, o di una mappa hash di chiavi e valori, o di un array di elementi di qualche genere, spesso ci troviamo a volere un determinato elemento tratto da tale collezione. Quando accade, spesso includo la specifica implementazione in un metodo o in una classe più astratti. Questo mi fornisce un paio di interessanti vantaggi.

Ora posso implementare la funzionalità con qualcosa di più semplice, per esempio una mappa hash, ma poiché ora tutti i riferimenti a tale ricerca rientrano nella mia piccola astrazione, posso cambiare l'implementazione ogni volta che voglio. Posso procedere rapidamente conservando nel contempo la possibilità di cambiare, in un secondo tempo.

Inoltre, l'astrazione della collezione spesso richiama la mia attenzione su quello che "davvero" sta accadendo, e mi impedisce di scegliere la strada dell'implementazione di un

comportamento arbitrario della collezione quando ciò di cui ho davvero bisogno è solo un modo piuttosto semplice per ottenere ciò di cui ho bisogno.

Bassa duplicazione, alta espressività, e realizzazione anticipata di semplici astrazioni. Ecco, per me, cosa significa che il codice è pulito.

Ecco: in pochi paragrafi, Ron ha riassunto il contenuto di questo libro. Nessuna duplicazione, una sola cosa, espressività, piccole astrazioni. C'è proprio tutto.

Ward Cunningham

Inventore di Wiki, inventore di Fit, coinventore della eXtreme Programming. Forza motrice di Design Patterns. Leader morale di Smalltalk e OO. Padrino di tutti coloro che si prendono cura del codice.



Vi rendete conto che state lavorando su codice pulito quando ogni routine che leggete fa esattamente ciò che ci si aspetta. Diventa poi bellissimo quando il codice fa sembrare che il linguaggio sia stato fatto proprio per risolvere quel problema.

Affermazioni come questa sono caratteristiche di Ward. Le leggete, annuite, e passate all'argomento successivo. È tutto così ragionevole, così ovvio, che ci si accorge appena che è qualcosa di davvero profondo. Potreste pensare che è più o meno quello che vi aspettavate. Ma esaminiamo la frase più da vicino.

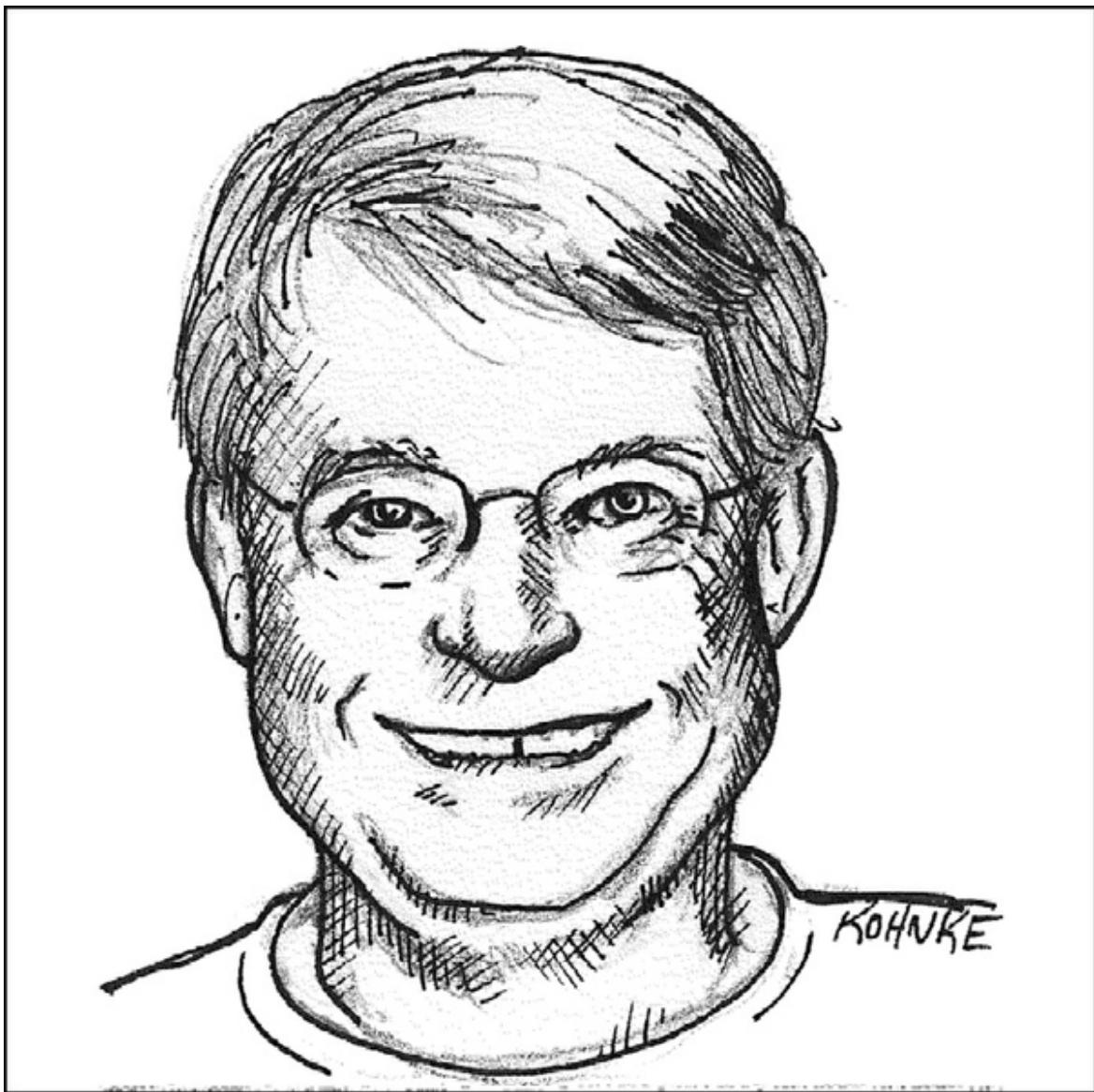
“... esattamente ciò che ci si aspetta”: quando è stata l’ultima volta che avete visto un modulo che fosse esattamente quello che vi aspettavate? Molto più spesso i moduli che ci capitano sono aggrovigliati, complicati. Non è forse questa la “norma”? Non vi trovate, spesso, a cercare di trovare e a tenere in mano i fili del ragionamento che spuntano dall’intero sistema e a tentare di seguirli all’interno del modulo che state leggendo? Quando è stata l’ultima volta che avete letto del codice e avete annuito come forse avete fatto leggendo l’affermazione di Ward?

Ward si aspetta che quando si legge del codice pulito non si presenti alcuna sorpresa. In effetti, non ci dovrebbe volere nemmeno molto impegno. Lo leggete, e vi accorgete che fa esattamente quello che vi aspettavate. Sarà tutto ovvio, semplice, e avvincente. Ogni modulo prepara la scena per il successivo. Ogni modulo fa capire come dovrà essere scritto il successivo. I programmi di *questa* pulizia sono così profondamente ben scritti che quasi non li si nota. Il programmatore li ha resi così meravigliosamente semplici come tutti i progetti ben realizzati.

E che dire del concetto di bellezza di Ward? Ci siamo sempre lamentati del fatto che i nostri linguaggi non erano adatti ai nostri problemi. Ma l’affermazione di Ward rimette a noi l’onere. Dice che il bel codice *fa sembrare che il linguaggio sia fatto proprio per quel problema!* Pertanto è nostra la responsabilità di far sembrare semplice il linguaggio! Attenti a voi, “ortodossi” del linguaggio! Non è il linguaggio che fa sembrare semplici i programmi. È il programmatore che fa sembrare semplice il linguaggio!

Scuole di pensiero

Che dire di me, “Uncle Bob”?



Che cosa penso che sia il codice pulito? Questo libro vi dirà, fin nei minimi dettagli, ciò che io e i miei colleghi consideriamo codice pulito. Vi diremo ciò che pensiamo renda pulito il nome di una variabile, una funzione, una classe e così via. Presenteremo queste opinioni come affermazioni assolute, senza scusarci per la durezza. Per noi, a questo punto della nostra carriera, sono valori assoluti. Sono la *nostra scuola di pensiero* sulla pulizia del codice.

In genere gli appassionati di arti marziali non concordano su quale sia la migliore delle arti marziali, o su quale sia la migliore tecnica

nell’ambito di un’arte marziale. Spesso i maestri di arti marziali formano le proprie scuole di pensiero e raccolgono gli allievi per insegnargliele. È così che troviamo il *Gracie Jiu Jitsu*, fondato e insegnato dalla famiglia Gracie in Brasile. Troviamo il *Hakkoryu Jiu Jitsu*, fondato e insegnato da Okuyama Ryuho a Tokyo. Troviamo il *Jeet Kune Do*, fondato e insegnato da Bruce Lee negli Stati Uniti.

Gli allievi di questi approcci si immergono negli insegnamenti del fondatore. Dedicano se stessi all’apprendimento degli insegnamenti del loro maestro, spesso escludendo del tutto quelli di altri maestri. Più tardi, quando gli allievi progrediscono nella loro arte marziale, possono andare a studiare da un altro maestro, per ampliare le proprie conoscenze e le proprie tecniche. Alcuni proseguono nel perfezionamento delle loro abilità, scoprendo nuove tecniche e fondando una loro scuola.

Nessuna di tutte queste scuole ha la verità *assoluta*. Tuttavia all’interno di una determinata scuola ci si comporta come se gli insegnamenti e le tecniche fossero quelle *giuste*. Dopotutto, esiste un modo giusto per praticare l’Hakkoryu Jiu Jitsu o il Jeet Kune Do. Ma questa correttezza nell’ambito di una scuola non rende meno validi gli insegnamenti di un’altra scuola.

Considerate questo libro come una descrizione di un’ipotetica *Object Mentor School of Clean Code*. Le tecniche e gli insegnamenti che vi forniamo sono il modo in cui praticare la nostra “arte marziale”. Vogliamo farvi comprendere che seguendo questi insegnamenti, godrete i vantaggi dei quali noi stessi abbiamo goduto, e imparerete a scrivere codice pulito e professionale. Ma non commettete l’errore di pensare che quanto vi diciamo sia “giusto” in senso assoluto. Esistono altre scuole e altri maestri altrettanto professionali. Dovreste pensare di imparare anche da loro.

In effetti, molte delle raccomandazioni presentate in questo libro possono essere controverse. Magari non concorderete con tutte. Magari sarete fortemente contrari ad alcune di esse. Non c'è problema. Non siamo certo un'autorità suprema. D'altra parte, le raccomandazioni presentate in questo libro sono il frutto di riflessioni lunghe e profonde. Le abbiamo apprese in decenni di esperienze positive e, soprattutto, negative, dagli errori. Pertanto, che concordiate o meno con quanto suggerito, sarebbe un peccato che non consideriate e rispettiate quello che è il nostro punto di vista.

Gli autori

Il campo `@author` di un Javadoc ci dice chi siamo. Siamo gli autori. E una caratteristica degli autori è che hanno dei lettori. In effetti, è *responsabilità* degli autori riuscire a comunicare bene coi loro lettori. La prossima volta che scriverete una riga di codice, ricordatevi che voi ne siete gli autori, e che scrivete a dei lettori che vi giudicheranno per quello che avrete scritto.

Potreste chiedervi: “Ma quanto viene letto, davvero, il codice? Il grosso dell’impegno non sta forse nello scriverlo?”.

Avete mai provato a riprodurre una sessione di editing? Negli anni Ottanta e Novanta avevamo editor come Emacs, che registravano ogni azione alla tastiera. Potevate lavorare per un’ora e poi riprodurre l’intera sessione di editing come nel Fast Forward di un film. Quando capitava di farlo, i risultati erano affascinanti.

La maggior parte delle attività era di scrolling e di accesso ad altri moduli!

Bob entra nel modulo. Con uno scroll raggiunge la funzione da modificare. Si ferma, considerando le opzioni. Raggiunge la cima del modulo per controllare l’inizializzazione di una variabile. Scroll verso il basso e poi inizia a scrivere. Ooops, sta cancellando quello che ha scritto! Scrive ancora. Cancella ancora! Scrive metà di qualcosa, ma poi la cancella! Raggiunge un’altra funzione che richiama la funzione che sta modificando, per

vedere come viene richiamata.Torna su e digita proprio quel codice che aveva appena cancellato.Si ferma.Cancella di nuovo il codice!Apre un'altra finestra e controlla una sottoclasse. Tale funzione ha un override?...

Insomma... avete capito. In effetti, il rapporto fra il tempo speso a leggere e quello impiegato a scrivere è ben oltre il 10:1. Siamo *costantemente* impegnati a leggere il vecchio codice mentre cerchiamo di scrivere quello nuovo.

Poiché questo rapporto è così elevato, vogliamo che la lettura del codice sia semplice, anche se scriverlo sarà più complicato. Naturalmente non vi è modo per scrivere del codice senza leggerlo, e quindi *scrivere codice facile da leggere ci renderà anche più facile... scrivere il codice.*

Non si scappa da questa regola. Non è possibile scrivere del codice senza poter comprendere il codice circostante. Il codice che state tentando di scrivere oggi sarà difficile o facile da scrivere nella misura in cui il codice circostante è difficile o facile da leggere. Pertanto, se volete procedere rapidamente, se volete terminare rapidamente, se volete che il vostro codice sia facile da scrivere, rendetelo facile da leggere.

La regola dei boy-scout

Ma scrivere bene il codice non basta. Il codice deve anche essere mantenuto pulito. Tutti abbiamo visto il codice degradarsi nel corso del tempo. Pertanto dobbiamo anche assumerci un ruolo attivo nel prevenire questo degrado.

I boy-scout hanno una semplice regola, perfetta per la nostra professione (adattato dal messaggio di addio di Robert Stephenson Smyth Baden-Powell agli scout).

Lascia l'area in cui ti sei fermato più pulita di come l'hai trovata.

Se lasciamo il nostro codice un po' più pulito di come l'abbiamo trovato, il codice, semplicemente, non si deteriora. Questa attività di pulizia non deve necessariamente essere fatta in grande stile. Può trattarsi di un intervento sul nome di una variabile, della suddivisione di una funzione che è cresciuta troppo, dell'eliminazione di una duplicazione, del raffinamento di un'istruzione `if` composita.

Immaginate cosa significhi lavorare su un progetto nel quale il codice *migliora* a ogni passata... Pensate che qualsiasi altra scelta sarebbe professionale? In effetti, un continuo miglioramento non è forse un elemento fondamentale della professionalità?

Il “prequel” e i principi

Per molti versi questo libro è un “prequel” di un libro che ho scritto nel 2002 intitolato *Agile Software Development: Principles, Patterns, and Practices* (PPP). Questo libro PPP tratta i principi della progettazione a oggetti (*object-oriented*, OO) e molte delle tecniche usate dagli sviluppatori professionisti. Se non avete ancora letto il libro PPP, scoprirete che continua la storia raccontata da questo libro. E se lo avete già letto, qui vi ritroverete molte delle suggestioni, applicate al livello del codice.

In questo libro troverete sporadici riferimenti a vari principi di progettazione, fra i quali: *Single Responsibility Principle* (SRP), *Open Closed Principle* (OCP) e *Dependency Inversion Principle* (DIP). Questi principi sono descritti approfonditamente nel libro PPP.

Conclusioni

I libri sull'arte non promettono di trasformarvi in artisti. Tutto ciò che possono fare è fornirvi gli strumenti, le tecniche e le riflessioni di altri artisti. Anche questo libro non può promettervi di trasformarvi in

buoni programmatore. Non può promettervi di infondervi questo “sesto senso”. Tutto ciò che può fare è mostrarvi le riflessioni di alcuni buoni programmatore e i trucchi, le tecniche e gli strumenti che essi usano.

Esattamente come un libro sull’arte, anche questo è ricco di dettagli. Vi troverete grandi quantità di codice. Vedrete esempi di buon codice e di cattivo codice. Vedrete anche il cattivo codice mentre si trasforma in buon codice. Vedrete liste di euristiche, discipline e tecniche. Troverete numerosi esempi. Tutto il resto sta a voi.

C’è una vecchia battuta che riguarda il violinista che si è perso e non riesce a trovare la strada per raggiungere il teatro? Ferma nella via un vecchio passante e gli domanda: “Come si arriva alla Scala?”. Il vecchio osserva il violinista e il violino che porta sotto il braccio, e risponde: “Pratica, figliolo, ci vuole pratica!”

Bibliografia

- [Beck07]: Kent Beck, *Implementation Patterns*, Addison-Wesley, Boston 2007.
- [Knuth92]: Donald E. Knuth, *Literate Programming*, Center for the Study of Language and Information, Leland Stanford Junior University, 1992.

Capitolo 2

Nomi significativi *di Tim Ottinger*



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 modificatevi 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 programmatore 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".

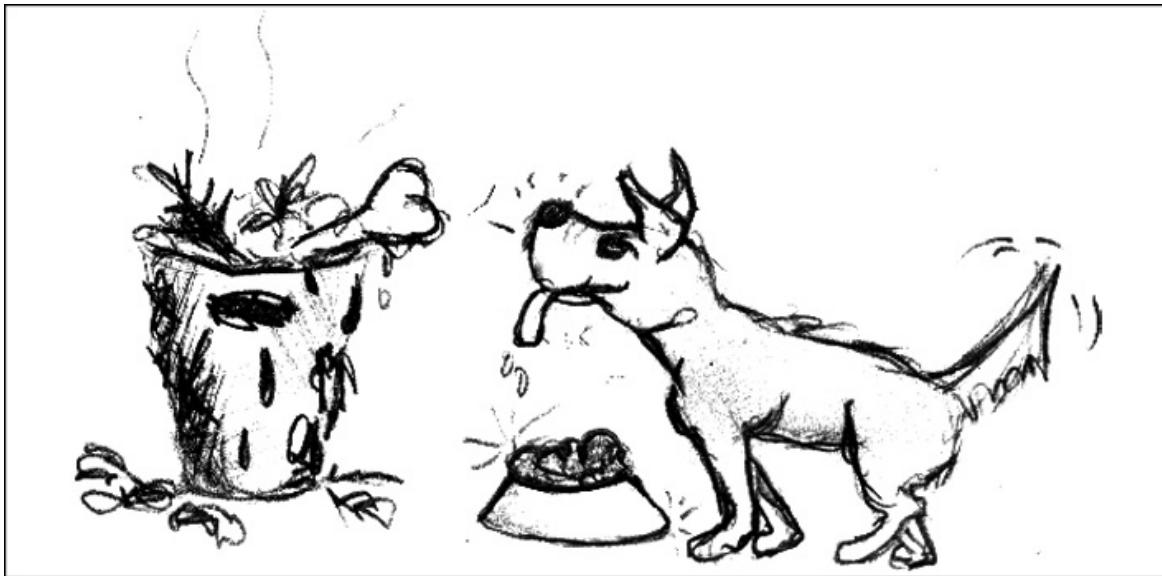
Scrivere in modo simile dei concetti simili è un'*informazione*. Un uso incoerente dei nomi è *disinformazione*. Negli ambienti Java di oggi possiamo contare sul completamento automatico del codice. Scriviamo alcuni caratteri di un nome e premiamo una certa combinazione di tasti e otteniamo un elenco 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 = 1;
if ( O == 1 )
    a = 01;
else
    l = 01;
```

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 cominciato 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 programmatore 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 parte 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 programmatore 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 predicatori 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 state 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 risulti 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 programmati. 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 programmati 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",  
            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, operereste 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.

Capitolo 3

Funzioni



Vi fu un tempo in cui componevamo i nostri sistemi con routine e subroutine. Poi, nell'era dei linguaggi Fortran e PL/1, componevamo i nostri sistemi con programmi, sottoprogrammi e funzioni. Al giorno d'oggi solo la funzione è sopravvissuta. Le funzioni sono il primo elemento organizzativo di ogni programma. Scriverle bene è l'argomento di questo capitolo.

Considerate il codice presentato nel Listato 3.1. È difficile trovare una funzione lunga in FitNesse, (uno strumento di test open source, <http://www.fitnesse.org>) ma dopo un po' di ricerca ho trovato la seguente. Non solo è lunga, ma contiene codice duplicato, molte stringhe ostiche e molti strani e arcani tipi di dati e API. Provate a dedicarle tre minuti per capire a che cosa serve.

Listato 3.1 HtmlUtil.java (FitNesse 20070619).

```
public static String testableHtml(
    PageData pageData,
    boolean includeSuiteSetup
) throws Exception {
    WikiPage wikiPage = pageData.getWikiPage();
    StringBuffer buffer = new StringBuffer();
    if (pageData.hasAttribute("Test")) {
        if (includeSuiteSetup) {
            WikiPage suiteSetup =
                PageCrawlerImpl.getInheritedPage(
                    SuiteResponder.SUITE_SETUP_NAME, wikiPage
                );
            if (suiteSetup != null) {
                WikiPagePath pagePath =
                    suiteSetup.getPageCrawler().getFullPath(suiteSetup);
                String pagePathName = PathParser.render(pagePath);
                buffer.append("!include -setup .")
                    .append(pagePathName)
                    .append("\n");
            }
            WikiPage setup =
                PageCrawlerImpl.getInheritedPage("SetUp", wikiPage);
            if (setup != null) {
                WikiPagePath setupPath =
                    wikiPage.getPageCrawler().getFullPath(setup);
                String setupPathName = PathParser.render(setupPath);
                buffer.append("!include -setup .")
                    .append(setupPathName)
                    .append("\n");
            }
        }
        buffer.append(pageData.getContent());
        if (pageData.hasAttribute("Test")) {
            WikiPage teardown =
                PageCrawlerImpl.getInheritedPage("TearDown", wikiPage);
            if (teardown != null) {
                WikiPagePath tearDownPath =
                    wikiPage.getPageCrawler().getFullPath(teardown);
                String tearDownPathName = PathParser.render(tearDownPath);
                buffer.append("\n")
                    .append("!include -teardown .")
                    .append(tearDownPathName)
                    .append("\n");
            }
            if (includeSuiteSetup) {
                WikiPage suiteTeardown =

```

```

        PageCrawlerImpl.getInheritedPage(
            SuiteResponder.SUITE_TEARDOWN_NAME,
            wikiPage
        );
        if (suiteTeardown != null) {
            WikiPagePath pagePath =
                suiteTeardown.getPageCrawler().getFullPath(suiteTeardown);
            String pagePathName = PathParser.render(pagePath);
            buffer.append("!include -teardown .")
                .append(pagePathName)
                .append("\n");
        }
    }
}
pageData.setContent(buffer.toString());
return pageData.getHtml();
}

```

Siete riusciti a capire che cosa fa la funzione in questi tre minuti? Probabilmente no. Accadono troppe cose e a troppi livelli di astrazione. Vi sono strane stringhe e chiamate a una funzione, insieme a istruzioni `if` a doppio annidamento controllate da flag.

Tuttavia, semplicemente con alcune semplici estrazioni di metodi, un po' di ridenominazione e un pizzico di ristrutturazione, è possibile catturare lo scopo della funzione nelle nove righe del Listato 3.2. Provate ora a scoprirlne il significato, sempre nei soliti 3 minuti.

Listato 3.2 HtmlUtil.java (dopo il refactoring).

```

public static String renderPageWithSetupsAndTear downs(
    PageData pageData, boolean isSuite
) throws Exception {
    boolean isTestPage = pageData.hasAttribute("Test");
    if (isTestPage) {
        WikiPage testPage = pageData.getWikiPage();
        StringBuffer newPageContent = new StringBuffer();
        includeSetupPages(testPage, newPageContent, isSuite);
        newPageContent.append(pageData.getContent());
        includeTeardownPages(testPage, newPageContent, isSuite);
        pageData.setContent(newPageContent.toString());
    }
    return pageData.getHtml();
}

```

Se non conoscete FitNesse, probabilmente non riuscirete a cogliere tutti i dettagli. Ciononostante, probabilmente avrete capito che questa funzione effettua l'inclusione di alcune pagine di configurazione e chiusura in una pagina di test e poi esegue il rendering di tale pagina in

HTML. Se conoscete JUnit (uno strumento open source di unit test per Java, <http://www.junit.org>), probabilmente vi sarete resi conto che questa funzione appartiene a un framework di test web. E, naturalmente, è così. Scoprire tali informazioni dal Listato 3.2 è abbastanza facile, mentre il significato risulta fumoso nel Listato 3.1.

Che cos' è che rende una funzione come quella del Listato 3.2 più facile da leggere e da comprendere? Come possiamo fare in modo che una funzione comunichi il proprio scopo? Quali attributi possiamo dare alle nostre funzioni che consentano a un lettore casuale di intuire il tipo di programma all'interno del quale operano?

Che sia piccola!

La prima regola delle funzioni è che siano piccole. La seconda è che *siano ancora più piccole*. Questa non è un'affermazione che posso dimostrare. Non posso fornire riferimenti a ricerche che dimostrino che le funzioni molto piccole siano migliori. Quello che vi dico è che per circa quattro decenni ho scritto funzioni di tutte le dimensioni. Ho scritto veri abomini da tremila righe. Ho scritto una buona quantità di funzioni da cento/trecento righe. E ho scritto funzioni da venti o trenta righe. Ebbene, l'esperienza mi ha insegnato, fra tentativi ed errori, che le funzioni è meglio che siano molto piccole.

Negli anni Ottanta eravamo soliti dire che una funzione non deve essere più grande di una schermata. Naturalmente parliamo di un'epoca in cui i terminali VT100 avevano 24 righe per 80 colonne e gli editor occupavano 4 righe per le opzioni e i comandi. Al giorno d'oggi, con un font appropriato e un bel monitor di grandi dimensioni, potete far stare 150 caratteri su una riga e 100 o più righe in una schermata. Le righe non saranno lunghe 150 caratteri. Le funzioni non dovranno essere lunghe 100 righe. In generale non dovrebbero superare le 20 righe.

Quanto deve essere corta una funzione? Nel 1999 sono andato a trovare Kent Beck a casa sua in Oregon. Ci siamo seduti e ci siamo messi a programmare. A un certo punto mi ha mostrato un piccolo programma Java/Swing che aveva chiamato *Sparkle*. Generava sullo schermo un effetto molto simile alla bacchetta magica della fata madrina di Cenerentola. Muovendo il mouse, dal puntatore si sprigionavano tante scintille, che poi cadevano sul fondo della finestra simulando la forza di gravità. Quando Kent mi ha mostrato il codice, sono rimasto impressionato dalla compattezza di tutte le funzioni. Ero abituato alle funzioni dei programmi Swing, che occupavano molto più spazio, in verticale. Ogni funzione di *questo* programma era di due o tre o quattro righe. Tutto era così meravigliosamente chiaro. Ogni funzione raccontava una sola cosa. E ognuna portava alla successiva in un ordine assoluto. Ecco *come* dovrebbero essere scritte le funzioni! (Ho chiesto a Kent se avesse ancora una copia del programma, ma non è riuscito a trovarla. Ho cercato anche in tutti i miei vecchi computer, ma senza risultato. Tutto ciò che mi è rimasto è il ricordo di quel programma).

Quanto devono essere corte le funzioni? Bene, dovrebbero essere più corte di quella del Listato 3.2! In effetti, il Listato 3.2 può essere accorciato fino a ottenere il Listato 3.3.

Listato 3.3 HtmlUtil.java (dopo un ulteriore refactoring).

```
public static String renderPageWithSetupsAndTear downs (
    PageData pageData, boolean isSuite) throws Exception {
    if (isTestPage(pageData))
        includeSetupAndTeardownPages(pageData, isSuite);
    return pageData.getHtml();
}
```

Blocchi e indentazione

Questo implica che i blocchi delle istruzioni `if`, `else`, `while` e così via debbano essere di una sola riga. Probabilmente tale riga dovrà

contenere una chiamata a funzione. Non solo questo garantirà la compattezza della funzione esterna, ma aggiunge anche del valore documentario, perché la funzione richiamata nel blocco può avere un bel nome descrittivo.

Questo implica anche che le funzioni non dovrebbero essere grandi al punto da contenere strutture annidate. Pertanto, il livello di indentazione di una funzione non dovrebbe essere superiore a uno o due. Questo, naturalmente, fa sì che le funzioni siano più facili da leggere e da comprendere.

Che faccia una cosa sola



Dovrebbe risultare molto chiaro che la funzione del Listato 3.1 fa molto più di una cosa sola. Fra le altre cose, crea dei buffer, legge delle

pagine, cerca delle pagine ereditate, genera percorsi, aggiunge strane stringhe e genera l'HTML. Il Listato 3.1 è molto impegnato a svolgere diverse cose. Al contrario, il Listato 3.3 fa un'unica cosa. Include gli attacchi e le chiusure nelle pagine di test.

La seguente indicazione risale in una forma o nell'altra a trenta o più anni fa.

LE FUNZIONI DEVONO FARE UNA COSA SOLA. DEVONO FARLA BENE.NON DEVONO FARE ALTRO.

Il problema di questa indicazione è che è difficile sapere che cosa significa “una cosa”. Il Listato 3.3 fa una cosa? È facile rendersi conto che in realtà fa tre cose.

1. Determina se la pagina è una pagina di test.
2. In caso affermativo, include attacchi e chiusure.
3. Esegue il rendering della pagina in HTML.

Come stanno le cose? La funzione fa una cosa o tre cose? Notate che i tre passi della funzione si trovano a un livello di astrazione inferiore rispetto al nome scelto per la funzione. Possiamo descrivere la funzione con un unico paragrafo “`to`”, ovvero “per...” Il linguaggio Logo utilizzava la parola chiave `to` nello stesso modo in cui Ruby e Python usano `def`. Quindi ogni funzione iniziava con la parola chiave `to`. Ciò aveva un effetto interessante sul modo in cui le funzioni venivano progettate:

`to (per) RenderPageWithSetupsAndTear downs`, controlliamo se la pagina è una pagina di test e in caso affermativo, includiamo gli attacchi e le chiusure. In ogni caso eseguiamo il rendering della pagina in HTML.

Se una funzione fa solo dei passi che si trovano a un solo livello sotto quello che dice il nome scelto per la funzione, allora la funzione sta facendo una sola cosa. Dopotutto, il motivo per il quale scriviamo delle funzioni è proprio di decomporre un grosso concetto (in altre

parole, il nome della funzione) in un insieme di passi situati al livello di astrazione successivo.

È evidente che il Listato 3.1 contiene dei passi che si trovano a vari livelli di astrazione. Pertanto, chiaramente sta facendo più di una cosa. Anche il Listato 3.2 ha due livelli di astrazione, come dimostrato dalla nostra capacità di ridurla ulteriormente. Ma sarebbe molto difficile ridurre il Listato 3.3 mantenendone il senso. Potremmo estrarre l’istruzione `if` in una funzione chiamata `includeSetupsAndTear downsIfTestPage`, ma ciò riaffermerebbe lo stesso codice senza cambiarne il livello di astrazione.

Pertanto, un altro modo per capire che una funzione sta facendo più di “una cosa” è scoprire se riuscite a estrarre da essa un’altra funzione con un nome che non è semplicemente una riaffermazione della sua implementazione [G34].

Sezioni all’interno delle funzioni

Osservate il Listato 4.7 nel prossimo capitolo. Notate che la funzione `generatePrimes` è suddivisa in sezioni: *dichiarazioni*, *inizializzazioni* e *setaccio*. Questo è un segno evidente che sta facendo più di una cosa. Le funzioni che fanno una sola cosa non possono essere suddivise ragionevolmente in sezioni.

Un livello di astrazione per funzione

Per assicurarsi che le nostre funzioni stiano facendo “una cosa”, dobbiamo assicurarci che le istruzioni della nostra funzione siano tutte allo stesso livello di astrazione. È facile vedere come il Listato 3.1 violi questa regola. Vi si trovano concetti che sono a un elevato livello di astrazione, come `getHtml()`; altri che sono a un livello di astrazione

intermedio, come `String pagePathName = PathParser.render(pagePath);` e altri ancora che sono a un livello piuttosto basso, come `.append("\n")`.

Mescolare i livelli di astrazione all'interno di una funzione è sempre fonte di confusione. Chi legge il codice potrebbe non riuscire a capire se una determinata espressione è essenziale o un dettaglio. Peggio ancora: come le finestre rotte, una volta che si uniscono dei dettagli ai concetti essenziali, nella funzione tendono ad accumularsi sempre più dettagli.

Leggete il codice da cima a fondo: la regola dei passi

Vogliamo che il codice possa leggersi come una sorta di racconto ([KP78], p. 37). Vogliamo che ogni funzione sia seguita da quelle al livello di astrazione successivo, in modo da poter leggere il programma, scendendo un livello di astrazione alla volta mentre scorriamo la lista delle funzioni. La chiamo *La regola dei passi*.

In altre parole, vogliamo essere in grado di leggere il programma come se fosse un insieme di paragrafi “Per...”, ognuno dei quali descrive il livello di astrazione corrente e fa riferimento ai successivi paragrafi “Per...” posti al livello successivo.

Per includere gli attacchi e le chiusure, includiamo gli attacchi, poi includiamo il contenuto della pagina di test e poi includiamo le chiusure.

Per includere gli attacchi, includiamo l'attacco della suite se questa è una suite, poi includiamo l'attacco standard.

Per includere l'attacco della suite, cerchiamo la pagina “SuiteSetUp” nella gerarchia superiore e aggiungiamo un'istruzione include con il percorso di tale pagina.

Per ricercare nella gerarchia superiore...

Ne consegue che è molto difficile per i programmatore imparare a seguire questa regola e scrivere funzioni che rimangano a un unico livello di astrazione. Ma imparare questo trucco è anche molto importante. È la chiave che permette di mantenere brevi le funzioni e

per assicurarsi che facciano “una cosa”. Creare codice che possa essere letto come una serie di paragrafi “Per...” è una tecnica efficace per garantire la coerenza del livello di astrazione.

Osservate il Listato 3.7 alla fine di questo capitolo. Mostra l’intera funzione `testableHtml` ri-fattorizzata secondo i principi appena descritti. Notate come ogni funzione introduce la successiva e ogni funzione rimane a un livello di astrazione coerente.

Istruzioni switch

È difficile compattare un’istruzione `switch` (qui, naturalmente, includo anche le catene di `if/else`). Anche un’istruzione `switch` con due soli `case` è più grande di quanto dovrebbe essere un blocco o una funzione. Inoltre è difficile creare un’istruzione `switch` che faccia una cosa sola. Per loro natura, le istruzioni `switch` fanno sempre n cose. Sfortunatamente le istruzioni `switch` non possono essere evitate, ma *possiamo* assicurarci che ogni istruzione `switch` si trovi “sepolta” in una classe di basso livello e non venga mai ripetuta. Per farlo, naturalmente, usiamo il *polimorfismo*.

Considerate il Listato 3.4. Mostra solo una delle operazioni che possono dipendere dal tipo di dipendente.

Listato 3.4 Payroll.java.

```
public Money calculatePay(Employee e)
throws InvalidEmployeeType {
    switch (e.type) {
        case COMMISSIONED:
            return calculateCommissionedPay(e);
        case HOURLY:
            return calculateHourlyPay(e);
        case SALARIED:
            return calculateSalariedPay(e);
        default:
            throw new InvalidEmployeeType(e.type);
    }
}
```

Vi sono parecchi problemi in questa funzione. Innanzitutto, è grande e crescerà ogni volta che dovrà essere aggiunto un nuovo tipo di dipendente. In secondo luogo, è evidente che fa più di una cosa. Terzo: viola il principio SRP (*Single Responsibility Principle* –

http://en.wikipedia.org/wiki/Single_responsibility_principle,

<http://www.objectmentor.com/resources/articles/srp.pdf>) perché c'è più di un motivo per doverla cambiare. Quarto: viola il principio OCP (*Open Closed Principle* – http://en.wikipedia.org/wiki/Open/closed_principle,

<http://www.objectmentor.com/resources/articles/ocp.pdf>) perché deve cambiare ogni volta che vengono aggiunti nuovi tipi. Ma forse il problema peggiore di questa funzione è che vi è un numero illimitato di altre funzioni realizzate con la stessa struttura. Per esempio potremmo avere

```
isPayday(Employee e, Date date)
```

o

```
deliverPay(Employee e, Money pay)
```

e anche molte altre. E tutte avrebbero la stessa, problematica, struttura.

La soluzione di questo problema (Listato 3.5) consiste nel seppellire l'istruzione `switch` dentro un'ABSTRACT FACTORY [GOF] e non farla mai vedere a nessuno. La factory userà l'istruzione `switch` per creare appropriate istanze delle derivate di `Employee` e le varie funzioni, come `calculatePay`, `isPayday` e `deliverPay`, verranno inviate in modo polimorfico tramite l'interfaccia di `Employee`.

La mia regola generale per le istruzioni `switch` è che possono essere tollerate se compaiono una sola volta, se vengono usate per creare oggetti polimorfici e se sono nascoste dietro una relazione di ereditarietà in modo che il resto del sistema non possa vederle [G23]. Naturalmente ogni circostanza è unica e vi sono casi in cui occorre violare una o più parti di tale regola.

Listato 3.5 Employee e Factory.

```
public abstract class Employee {  
    public abstract boolean isPayday();  
    public abstract Money calculatePay();  
    public abstract void deliverPay(Money pay);  
}  
-----  
public interface EmployeeFactory {  
    public Employee makeEmployee(EmployeeRecord r) throws InvalidEmployeeType;  
}  
-----  
public class EmployeeFactoryImpl implements EmployeeFactory {  
    public Employee makeEmployee(EmployeeRecord r) throws InvalidEmployeeType {  
        switch (r.type) {  
            case COMMISSIONED:  
                return new CommissionedEmployee(r);  
            case HOURLY:  
                return new HourlyEmployee(r);  
            case SALARIED:  
                return new SalariedEmployee(r);  
            default:  
                throw new InvalidEmployeeType(r.type);  
        }  
    }  
}
```

Usate nomi descrittivi

Nel Listato 3.7 ho cambiato il nome della nostra funzione d'esempio da `testableHtml a SetupTeardownIncluder.render`. Questo nome è molto migliore, perché descrive meglio l'attività svolta dalla funzione. Ho anche dato a ciascuno dei metodi privati un nome altrettanto descrittivo come `isTestable` o `includeSetupAndTeardownPages`. È difficile sopravvalutare il valore di un buon nome. Ricordo il Principio di Ward: “Sapete che state lavorando con codice pulito quando ogni routine fa esattamente quanto ci si aspetta”. Metà della fatica necessaria per aderire a questo principio consiste nello scegliere un buon nome per piccole funzioni che facciano una cosa sola. Più piccola e mirata è una funzione, più facile sarà scegliere un nome descrittivo.

Non temete di creare nomi lunghi. Un nome lungo e descrittivo è sempre meglio di un nome breve ed enigmatico. Un nome lungo e

descrittivo è sempre meglio di un commento lungo e descrittivo. Adottate una convenzione di denominazione che consenta di leggere con facilità più parole nei nomi di funzione e poi sfruttate tali parole per dare alla funzione un nome che comunichi il suo scopo.

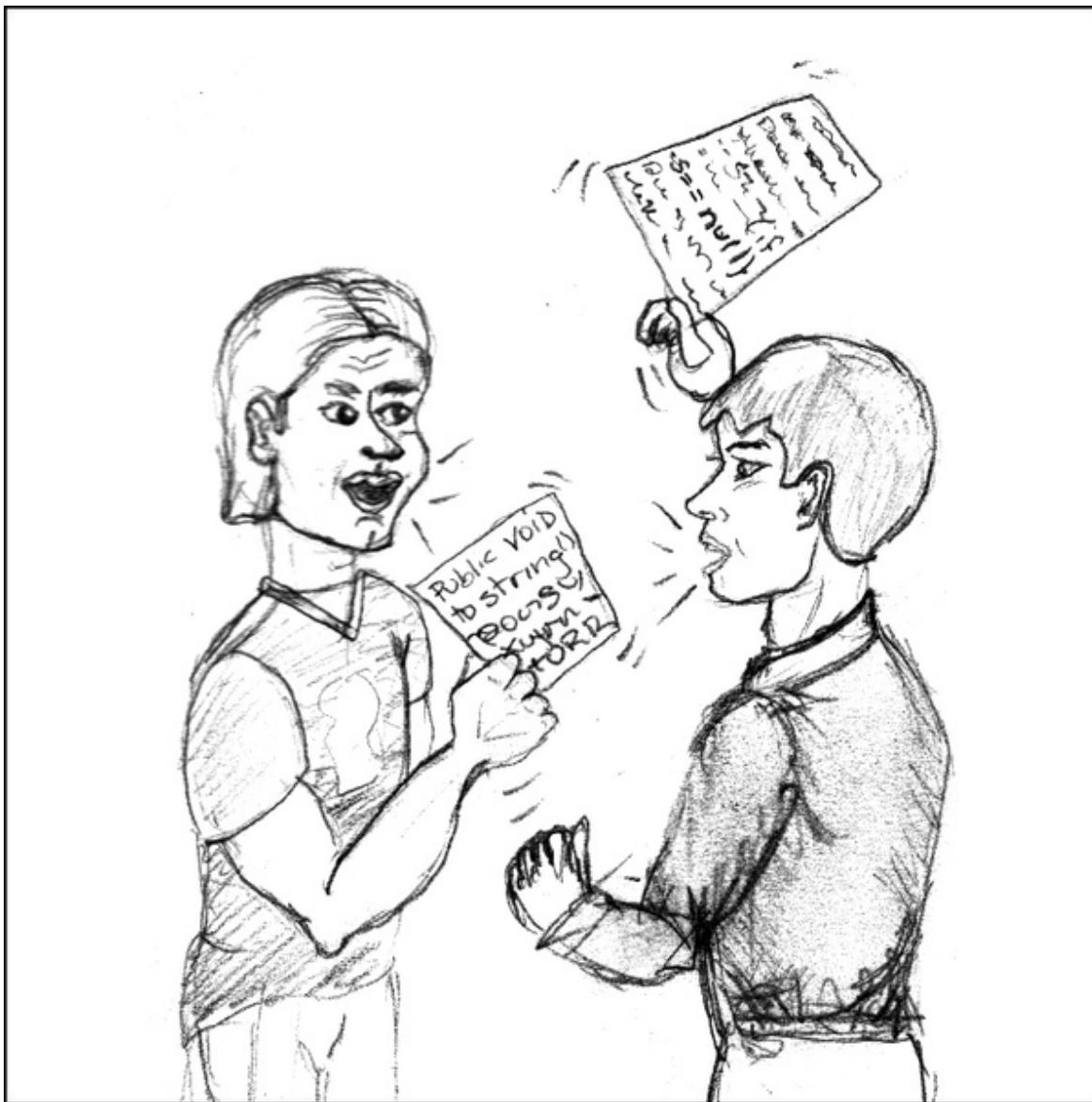
Non temete di dedicare del tempo alla scelta un nome. In effetti, dovreste provare a usare più nomi e a leggere il codice ovunque venga usato tale identificatore. I nuovi ambienti IDE, come Eclipse o IntelliJ, facilitano molto l'operazione di cambiare i nomi. Usate uno di questi IDE e sperimentate l'uso di nomi differenti, fino a trovarne uno davvero efficace.

La scelta di nomi descrittivi vi chiarirà la struttura del modulo e vi aiuterà a migliorarla. Non è insolito che l'individuazione di un buon nome abbia come risultato una ristrutturazione migliorativa del codice.

Siate coerenti con i vostri nomi. Usate le stesse frasi e gli stessi nomi e verbi nei nomi di funzione che scegliete per i vostri moduli.

Considerate, per esempio, i nomi `includeSetupAndTeardownPages`, `includeSetupPages`, `includeSuiteSetupPage` e `includeSetupPage`. Il fatto di adottare la stessa fraseologia in tali nomi fa in modo che la sequenza racconti una storia. In effetti, se vi avessi mostrato solo la sequenza precedente, vi sareste chiesti: “Che cosa è accaduto a `includeTeardownPages`, `includeSuiteTeardownPage` e `includeTeardownPage`?” Com’è che era? “... fa esattamente quanto ci si aspetta.”

Argomenti di funzione



Il numero ideale di argomenti per una funzione è zero (*niladica*). Poi uno (*monadica*), seguito a breve distanza da due (*diadica*). I tre argomenti (*triadica*) dovrebbero essere evitati, ove possibile. Per prevedere più di tre argomenti (*poliadica*) occorre una giustificazione davvero particolare (e, in ogni caso, evitate comunque di farlo).

Gli argomenti sono un problema. Richiedono molta energia concettuale. Questo è il motivo per cui li ho eliminati quasi tutti dall'esempio. Considerate, per esempio, lo `StringBuffer` dell'esempio.

Avremmo potuto passarlo come un argomento invece di renderlo una variabile di istanza, ma poi chi avrebbe letto il codice avrebbe dovuto interpretarlo ogni volta che lo vedeva. Quando leggete la storia raccontata dal modulo, `includeSetupPage()` è più facile da comprendere rispetto a `includeSetupPageInto(newPageContent)`. L'argomento si trova in un altro livello di astrazione rispetto al nome della funzione e vi costringe a conoscere un dettaglio (in altre parole, `StringBuffer`) che non è particolarmente importante in tale posizione.

Gli argomenti sono ancora di più un problema dal punto di vista dei test. Immaginate la difficoltà di scrivere tutti i casi di test per garantire che tutte le varie combinazioni di argomenti funzionino correttamente. Se non vi sono argomenti, la cosa diventa banale. Se vi è un solo argomento, non è troppo difficile. Con due argomenti il problema diviene un po' più complesso. Con più di due argomenti, i test di ogni combinazione dei valori appropriati può diventare un incubo.

Gli argomenti di output sono più difficili da comprendere rispetto agli argomenti di input. Quando leggiamo una funzione, siamo abituati all'idea che le informazioni arrivino alla funzione tramite gli argomenti e ne escano tramite il valore restituito. Normalmente non ci aspettiamo che le informazioni escano dalla funzione tramite gli argomenti. Questo è il motivo per cui gli argomenti di output spesso causano dei dubbi.

Un argomento di input è la prima alternativa migliore rispetto a una funzione senza argomenti. `SetupTeardownIncluder.render(pageData)` è abbastanza facile da comprendere. Chiaramente vogliamo eseguire il rendering dei dati dell'oggetto `pageData`.

Forme monadiche comuni

Vi sono due motivi molto comuni per passare un unico argomento a una funzione. Potreste avere una domanda relativa a tale argomento,

come in `boolean fileExists("MyFile")`. Oppure potreste voler operare su tale argomento, trasformandolo in qualche modo e restituendolo. Per esempio, `InputStream fileOpen("MyFile")` trasforma la `String` del nome di un file in un valore di tipo `InputStream`. Questi due usi sono quelli che coloro che leggono il codice si aspettano quando vedono una funzione. Dovreste scegliere dei nomi che chiariscano questa distinzione e usare sempre le due forme in un contesto di coerenza (vedi “*Separate i comandi dalle richieste*”, più avanti).

Una forma meno comune, ma comunque molto utile, per una funzione a un solo argomento è un *evento*. In questa forma vi è un argomento di input ma nessun argomento di output. Il programma deve interpretare la chiamata a funzione come un evento e usare l’argomento per modificare lo stato del sistema, per esempio, `void passwordAttemptFailedNtimes(int attempts)`. Usate questa forma con cautela. Deve essere molto chiaro a chi legge il codice che questo è un evento. Scegliete i nomi e i contesti con attenzione.

Cercate di evitare ogni funzione monadica che esuli da queste forme, per esempio, `void includeSetupPageInto(StringBuffer pageText)`. Usare per una trasformazione un argomento di output invece del valore restituito è fonte di confusione. Se una funzione deve trasformare il suo argomento di input, la trasformazione dovrà avvenire sfruttando il valore restituito. In effetti, `StringBuffer transform(StringBuffer in)` è sempre meglio di `void transform(StringBuffer out)`, anche se l’implementazione del primo caso restituisce semplicemente l’argomento di input. Quanto meno si tratta di una forma di trasformazione.

Flag usati come argomenti

I flag usati come argomenti sono brutti. Passare un valore booleano a una funzione è davvero una pessima abitudine. Complica

immediatamente la *signature* del metodo, proclamando a gran voce che questa funzione fa più di una cosa. Fa una cosa se il flag è `true` e un'altra se il flag è `false`!

Nel Listato 3.7 non avevamo scelta, perché i chiamanti passavano già tale flag e volevo limitare l'estensione del refactoring alla funzione e ai livelli sottostanti. Ciononostante, la chiamata a metodo `render(true)` è solo fonte di confusione per chi si troverà a leggere il codice. Passare col mouse sopra la chiamata e vedere `render(boolean isSuite)` è d'aiuto, ma non basta. Dovremmo suddividere la funzione in due parti:

```
renderForSuite() e renderForSingleTest().
```

Funzioni diadiche

Una funzione con due argomenti è più difficile da comprendere rispetto a una funzione monadica. Per esempio, `writeField(name)` è più facile da comprendere rispetto a `writeField(output-stream, name)`. (Ho appena terminato il refactoring di un modulo che utilizzava la forma diadica. Ho reso `OutputStream` un campo della classe e ho convertito tutte le chiamate `writeField` in forma monadica. Il risultato è stato molto più pulito.) Sebbene il significato di entrambe sia chiaro, sulla prima lo sguardo scorre, afferrando immediatamente il significato. La seconda richiede una breve pausa, il tempo di ignorare il primo parametro. E *questo*, naturalmente, alla lunga causa problemi, perché non dovremmo mai ignorare alcuna parte del codice. Le parti che decidiamo di ignorare sono quelle in cui si annidano i bug.

Esistono situazioni, naturalmente, nelle quali è il caso di utilizzare due argomenti. Per esempio, `Point p = new Point(0,0);` è perfettamente ragionevole. È normale che i punti cartesiani prevedano due argomenti. In effetti saremmo sorpresi del contrario: `new Point(0)`. Ma in questo caso i due argomenti sono *componenti ordinati che formano un unico*

valore! Al contrario, `output-stream` e `name` non hanno né una coesione, né un ordinamento naturale.

Anche le funzioni diadiche più ovvie, come `assertEquals(expected, actual)` sono problematiche. Quante volte vi capiterà di invertire `actual` con `expected`? I due argomenti non hanno alcun ordinamento naturale. L'ordinamento `expected, actual` è una convenzione in più da imparare.

Le diadi non sono poi così malvage e certamente vi capiterà di doverle scrivere. Tuttavia, consideratene sempre i costi e sfruttate eventuali meccanismi utili per convertirle in monadi. Per esempio, potreste rendere il metodo `writeField` un membro di `OutputStream`, in modo da poter scrivere `outputStream.writeField(name)`. Oppure potreste rendere `output Stream` una variable membro della classe corrente, in modo da non doverla passare. Oppure potreste estrarre una nuova classe come `FieldWriter` che accetti l'`outputStream` nel suo costruttore e che abbia un metodo `write`.

Triadi

Le funzioni che accettano tre argomenti sono significativamente più difficili da comprendere rispetto alle diadi. I problemi di ordinamento, di tempo necessario per comprenderne il significato e di possibilità di ignorarli sono più che raddoppiati. Vi suggerisco di riflettere bene prima di creare una triade.

Per esempio, considerate il tipico *overload* di `assertEquals` che prende tre argomenti: `assertEquals(message, expected, actual)`. Quante volte avete letto il messaggio (`message`) e avete pensato che fosse quello previsto (`expected`)? Mi è capitato parecchie volte di essermi fermato a riflettere sul possibile significato di questa triade. In effetti, ogni volta che lo vedo, prima lo riguardo e poi decido di ignorare il messaggio.

D'altra parte, ecco una triade che non è poi così insidiosa:

```
assertEquals(1.0, amount, .001). Anche se dà da pensare, vale la pena di usarla. È sempre bene ricordare che l'uguaglianza dei valori in virgola mobile è qualcosa di relativo.
```

Oggetti usati come argomenti

Quando una funzione sembra aver bisogno di più di due o tre argomenti, è probabile che alcuni di questi argomenti debbano essere inseriti in una loro classe. Considerate, per esempio, la differenza fra le due dichiarazioni seguenti:

```
Circle makeCircle(double x, double y, double radius);  
Circle makeCircle(Point center, double radius);
```

La riduzione del numero di argomenti tramite la creazione di oggetti può sembrare poco utile, ma non è così. Quando vengono passati dei gruppi di variabili, come `x` e `y` nell'esempio precedente, probabilmente fanno parte parte di un concetto che merita un nome a sé.

Liste di argomenti

Talvolta vogliamo passare a una funzione un numero variabile di argomenti. Considerate, per esempio, il metodo `String.format`:

```
String.format("%s worked %.2f hours.", name, hours);
```

Se gli argomenti variabili vengono tutti trattati allo stesso modo, come nell'esempio precedente, sono equivalenti a un unico argomento di tipo `List`. In base a questo ragionamento, `String.format` in realtà è diadica. In effetti, la dichiarazione seguente di `String.format` è chiaramente diadica.

```
public String format(String format, Object... args)
```

Pertanto valgono le stesse regole. Le funzioni che accettano un numero variabile di argomenti possono essere monadi, diadi o anche

triadi. Ma sarebbe un errore dare loro più argomenti.

```
void monad(Integer... args);  
void dyad(String name, Integer... args);  
  
void triad(String name, int count, Integer... args);
```

Verbi e parole chiave

La scelta di un buon nome per una funzione può essere fondamentale per chiarire lo scopo della funzione e l'ordine e lo scopo degli argomenti. Nel caso di una monade, la funzione e l'argomento dovrebbero formare a una buona coppia verbo/nome. Per esempio, `write(name)` è molto evocativo. Qualsiasi cosa sia quel `name`, verrà sicuramente scritta. Un nome anche migliore potrebbe essere `writeField(name)`, che ci dice anche che `name` è un campo.

Quest'ultimo è un esempio di come il nome di una funzione possa diventare una sorta di parola chiave. Usando questa forma codifichiamo i nomi degli argomenti nel nome della funzione. Per esempio, `assertEquals` potrebbe avere un nome migliore, come `assertExpectedEqualsActual(expected, actual)`. Questo riduce notevolmente il problema di dover ricordare l'ordine degli argomenti.

Niente effetti collaterali

Gli effetti collaterali sono come “bugie”. La vostra funzione promette di fare una cosa, ma fa anche altre cose di *nascosto*. Talvolta si tratta di modifiche inattese delle variabili della sua classe. Talvolta interviene sui parametri passati alla funzione o ai valori globali del sistema. In ogni caso si tratta di interventi non evidenti e potenzialmente distruttivi che spesso producono strane associazioni temporali e dipendenze.

Considerate, per esempio, l'apparentemente innocua funzione presentata nel Listato 3.6. Questa funzione usa un algoritmo standard

per confrontare uno `userName` e una password. Restituisce `true` se è tutto ok e `false` in caso contrario. Ma ha anche un effetto collaterale. Riuscite a individuarlo?

Listato 3.6 UserValidator.java.

```
public class UserValidator {  
    private Cryptographer cryptographer;  
    public boolean checkPassword(String userName, String password) {  
        User user = UserGateway.findByName(userName);  
        if (user != User.NULL) {  
            String codedPhrase = user.getPhraseEncodedByPassword();  
            String phrase = cryptographer.decrypt(codedPhrase, password);  
            if ("Valid Password".equals(phrase)) {  
                Session.initialize();  
                return true;  
            }  
        }  
        return false;  
    }  
}
```

L'effetto collaterale, naturalmente, è la chiamata a `Session.initialize()`. La funzione `checkPassword`, dovrebbe, così dice il nome, controllare la password. Il nome non dice che inizializza la sessione. Pertanto un chiamante che creda al nome della funzione corre il rischio di cancellare i dati della sessione corrente nel caso in cui decida di controllare la validità dell'utente.

Questo effetto collaterale crea un accoppiamento temporale. In pratica, `checkPassword` può essere richiamata solo in determinati momenti (in altre parole, quando è sicuro di inizializzare la sessione). Se viene richiamata in un altro momento, i dati della sessione potrebbero inavvertitamente perdere. Gli accoppiamenti temporali generano confusione, in particolare quando vengono nascosti in un effetto collaterale. Se vi deve essere un accoppiamento temporale, ciò deve risultare nel nome della funzione. In questo caso potremmo rinominare la funzione `checkPasswordAndInitializeSession`, sebbene questo certamente violi la prescrizione di “Fare una sola cosa”.

Argomenti di output

Gli argomenti vengono naturalmente interpretati come *input* di una funzione. Se programmate da qualche anno, sono sicuro che vi sarà capitato di imbattervi in un argomento che in realtà era un *output* invece di un *input*. Per esempio:

```
appendFooter(s);
```

Questa funzione aggiunge `s` dopo qualcosa? Oppure aggiunge qualcosa dopo `s`? In pratica, `s` è un *input* o un *output*? Non ci vuole molto per vedere la signature della funzione e scoprire che:

```
public void appendFooter(StringBuffer report)
```

Questo chiarisce il dubbio, ma solo dopo aver controllato la dichiarazione della funzione. Tutto ciò che costringe a controllare la signature della funzione obbliga a rallentare il lavoro. Si tratta di un'interruzione cognitiva e deve essere evitata.

Prima della programmazione a oggetti talvolta era necessario avere degli argomenti di *output*. Tuttavia, questa necessità è sostanzialmente sparita nei linguaggi OO, perché deve *trattarsi* di un argomento di *output*. In altre parole, sarebbe meglio che `appendFooter` venisse richiamata come:

```
report.appendFooter();
```

In generale, gli argomenti di *output* dovrebbero essere evitati. Se la vostra funzione deve modificare lo stato di qualcosa, lo deve fare per l'oggetto cui appartiene.

Separate i comandi dalle richieste

Le funzioni devono *fare* qualcosa oppure *rispondere* qualcosa, non entrambe le cose. La vostra funzione dovrebbe modificare lo stato di un oggetto oppure restituire qualche informazione su tale oggetto. Fare

entrambe le cose spesso induce in confusione. Considerate, per esempio, la seguente funzione:

```
public boolean set(String attribute, String value);
```

Questa funzione imposta il valore di un attributo con nome e restituisce `true` se ha successo e `false` se tale attributo non esiste. Questo ci porta a realizzare pessime istruzioni come la seguente:

```
if (set("username", "unclebob")) ...
```

Immaginate questa istruzione dal punto di vista di chi si trovasse a leggere il codice. Che cosa significa? Chiede se l'attributo `username` era uguale a `unclebob`? O sta chiedendo se all'attributo `username` è stato assegnato con successo il valore `unclebob`? È difficile comprendere il significato dalla chiamata, perché non è chiaro se la parola “`set`” è da intendersi come un verbo o un aggettivo.

L'autore intendeva usare “`set`” come un verbo, ma nel contesto dell'istruzione `if` *sembra* quasi un aggettivo. Pertanto l'istruzione si legge “Se l'attributo `username` è stato precedentemente impostato a `unclebob`” e non come “Imposta l'attributo `username` al valore `unclebob` e, se l'operazione funziona, allora...”. Potremmo tentare di risolvere questo dilemma rinominando la funzione `set` come `setAndCheckIfExists`, ma questo non sarebbe di grande aiuto per migliorare la leggibilità dell'istruzione `if`. La vera soluzione consiste nel separare il comando dalla richiesta, in modo da eliminare l'ambiguità.

```
if (attributeExists("username")) {  
    setAttribute("username", "unclebob");  
    ...  
}
```

Scegliete le eccezioni invece di restituire codici di errore

Il fatto di restituire codici d'errore dalle funzioni di comando è una subdola violazione della separazione dei comandi. Fa sì che i comandi vengano usati come espressioni nei predicati delle istruzioni `if`.

```
if (deletePage(page) == E_OK)
```

Questo esempio non soffre della confusione verbo/aggettivo, ma produce strutture fortemente annidate. Quando restituite un codice d'errore, obbligate il chiamante a gestire immediatamente l'errore.

```
if (deletePage(page) == E_OK) {
    if (registry.deleteReference(page.name) == E_OK) {
        if (configKeys.deleteKey(page.name.makeKey()) == E_OK) {
            logger.log("page deleted");
        } else {
            logger.log("configKey not deleted");
        }
    } else {
        logger.log("deleteReference from registry failed");
    }
} else {
    logger.log("delete failed");
    return E_ERROR;
}
```

D'altra parte, se usate le eccezioni invece di restituire dei codici d'errore, il codice di elaborazione degli errori può essere separato dal flusso principale del codice e semplificato:

```
try {
    deletePage(page);
    registry.deleteReference(page.name);
    configKeys.deleteKey(page.name.makeKey());
}
catch (Exception e) {
    logger.log(e.getMessage());
}
```

Estraete i blocchi try/catch

I blocchi `try/catch` sono brutti da vedere. Confondono la struttura del codice e mescolano l'elaborazione degli errori con la normale elaborazione. Pertanto è meglio estrarre il corpo dei blocchi `try` e `catch`, inserendoli in funzioni a sé stanti.

```

public void delete(Page page) {
    try {
        deletePageAndAllReferences(page);
    }
    catch (Exception e) {
        logError(e);
    }
}
private void deletePageAndAllReferences(Page page) throws Exception {
    deletePage(page);
    registry.deleteReference(page.name);
    configKeys.deleteKey(page.name.makeKey());
}
private void logError(Exception e) {
    logger.log(e.getMessage());
}

```

In questo esempio, la funzione `delete` si occupa solo dell'elaborazione degli errori. È facile da comprendere e anche da ignorare. La funzione `deletePageAndAllReferences` si occupa solo dei processi di cancellazione di una pagina. La gestione degli errori può essere ignorata. Ciò fornisce una separazione netta che rende il codice più facile da comprendere e modificare.

La gestione degli errori è “una cosa”

Le funzioni dovrebbero fare una sola cosa. La gestione degli errori, per l'appunto, è “una cosa”. Pertanto, una funzione che gestisce gli errori dovrebbe fare solo quello. Questo implica (come nell'esempio precedente) che se in una funzione si trova la parola chiave `try`, dovrebbe essere la primissima parola della funzione e non vi dovrebbe essere nulla dopo i blocchi `catch/finally`.

Il magnete per dipendenze Error.java

La restituzione di codici d'errore, solitamente implica che vi è una qualche classe o enumerazione nella quale sono definiti tutti i codici d'errore.

```

public enum Error {
    OK,

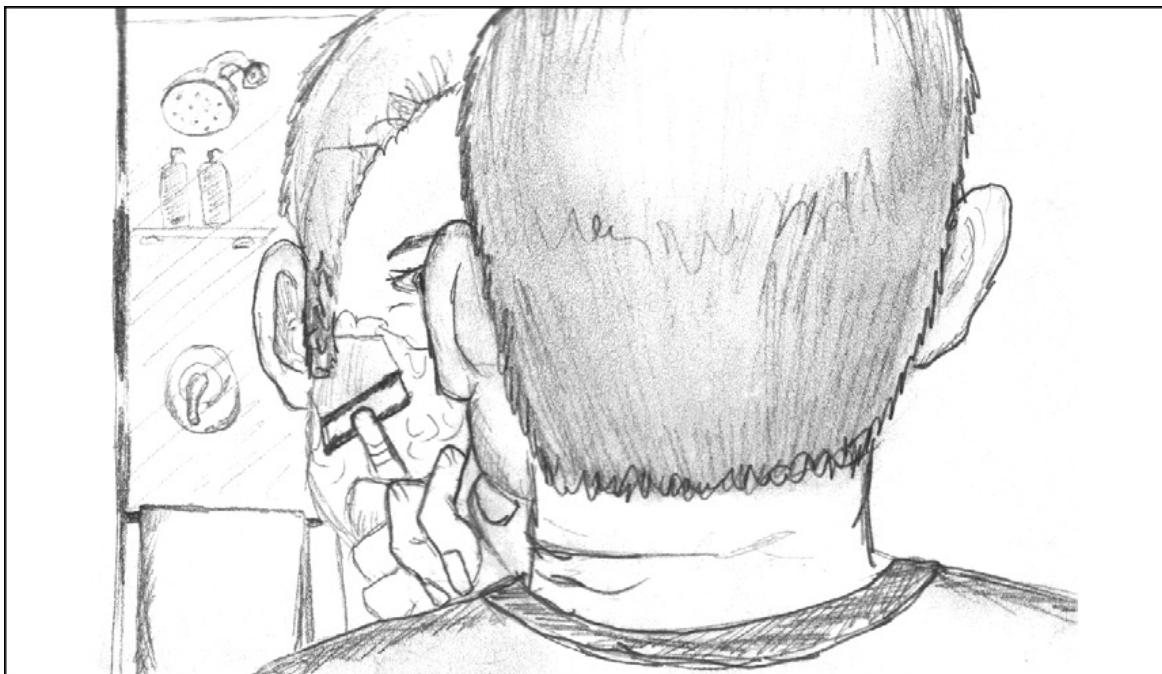
```

```
INVALID,  
NO_SUCH,  
LOCKED,  
OUT_OF_RESOURCES,  
WAITING FOR EVENT;  
}
```

Le classi come questa attirano le *dipendenze come un magnete*; molte altre classi dovranno importarla e usarla. Pertanto, quando cambierà l'enumerazione `Error`, dovranno essere modificate e ricompilate tutte queste classi (coloro che sentivano di poter andare via senza ricompilazione e successiva deploy sono stati scovati e affrontati). Questo pone un carico eccessivo sulla classe `Error`. I programmatore non vorranno aggiungere nuovi errori, perché saranno costretti a rivedere tutto. Pertanto riutilizzeranno i vecchi codici d'errore invece di aggiungerne di nuovi.

Quando usate le eccezioni invece dei codici d'errore, le nuove eccezioni sono *derivate* della classe delle eccezioni. Possono essere aggiunte senza costringere a ricompilazioni (questo è un esempio di OCP (*Open Closed Principle*) [PPP02]).

Non ripetetevi (il principio DRY)



Osservate con attenzione il Listato 3.1 e noterete che vi è un algoritmo ripetuto per ben quattro volte: per `setUp`, `SuitesetUp`, `tearDown` e `SuitetearDown`. Non è facile individuare questa duplicazione, perché le quattro istanze si trovano mescolate con altro codice e la duplicazione non è uniforme. Ciononostante, la duplicazione è un problema, perché appesantisce il codice e richiede quattro modifiche nel caso in cui l'algoritmo dovesse cambiare. Per non parlare delle quattro opportunità di commettere errori o dimenticanze. Questa duplicazione è stata risolta con il metodo `include` presentato nel Listato 3.7. Osservate ancora tale codice e notate come riducendo tale duplicazione la leggibilità dell'intero modulo sia molto migliorata.

La duplicazione è all'origine di tutti i mali del software. Molti principi e tecniche sono stati creati proprio con lo scopo di controllarla o eliminarla (principio DRY, *Don't Repeat Yourself* [PRAG]). Considerate, per esempio, che tutte le forme normali del database di Codd servono proprio a eliminare le duplicazioni nei dati. Considerate anche come la programmazione a oggetti serva a concentrare nelle

classi base il codice che altrimenti sarebbe ridondante. La programmazione strutturata, la programmazione Aspect Oriented, la programmazione Component Oriented, sono tutte, almeno in parte, strategie volte a eliminare la duplicazione. Sembra che fin dall'invenzione della subroutine, le innovazioni nello sviluppo del software siano state un continuo tentativo di eliminare la duplicazione dal nostro codice sorgente.

Programmazione strutturata

Alcuni programmatore seguono le regole della programmazione strutturata di Edsger Dijkstra [SP72]. Dijkstra disse che ogni funzione e ogni blocco all'interno di una funzione, dovrebbe avere un solo ingresso e una sola uscita. Seguire queste regole significa far sì che in una funzione esiste una sola istruzione `return`, che in un ciclo non ci dovrebbe essere nessuna istruzione `break` o `continue` e che non vi dovrebbe essere assolutamente *mai* un'istruzione `goto`.

Anche se concordiamo con gli obiettivi e la disciplina della programmazione strutturata, tali regole non presentano molti vantaggi se le funzioni sono molto piccole. È solo nelle grandi funzioni che tali regole forniscono vantaggi significativi.

Pertanto se mantenete piccole le vostre funzioni, quell'occasionale presenza di più istruzioni `return`, `break` o `continue` non farà danni e talvolta può perfino rivelarsi più espressiva di una rigida regola “un solo ingresso, una sola uscita”. Quanto ai `goto`, hanno qualche senso solo nelle grosse funzioni, e pertanto dovrebbero essere evitati.

Come scrivere le funzioni in questo modo?

Scrivere software è un po' come scrivere un qualsiasi altro testo. Quando scrivete una relazione o un articolo, prima dovete scrivere le vostre riflessioni, e poi dovete raffinarle per renderle più scorrevoli. La prima bozza sembrerà ostica e disorganizzata, e pertanto la rielaborerete e riorganizzerete e la correggerete fino a perfezionarla in termini di leggibilità e comprensibilità.

Quando scrivo delle funzioni, mi vengono lunghe e complesse. Uso un sacco di indentazione e di cicli annidati. Hanno lunghi elenci di argomenti. I nomi sono arbitrari e vi è anche del codice duplicato. Ma ho anche una serie di test specifici che considerano ognuna di queste involute righe del codice.

Pertanto correggo e raffino tale codice, suddivido le funzioni, modifico i nomi, elimino le duplicazioni. Faccio dimagrire i metodi e li risistemo. Talvolta spezzo intere classi, e sempre sottopongo il tutto a test.

Alla fine, mi ritrovo funzioni che seguono le regole che ho presentato in questo capitolo. Non le scrivo “bene” fin da subito. Penso che nessuno ne sia capace.

Conclusioni

Ogni sistema si basa su un linguaggio specifico di un dominio, progettato da dei programmatore per descrivere tale sistema. Le funzioni sono i verbi di tale linguaggio e le classi sono i nomi. Questo non è un ritorno nostalgico al concetto ormai stantio in base al quale i nomi e i verbi del documento dei requisiti sono il primo indizio per sviluppare le classi e le funzioni di un sistema. Piuttosto, questa è una verità molto più antica. L'arte della programmazione è ed è sempre stata, l'arte della progettazione del linguaggio.

I grandi programmatore considerano i sistemi come se fossero storie da raccontare, più che come programmi da scrivere. Usano le

funzionalità del linguaggio di programmazione scelto per costruire un linguaggio più ricco ed espressivo che possa essere usato per raccontare tale storia. Parte di questo linguaggio specifico del dominio è proprio costituito dalla gerarchia delle funzioni che descrivono tutte le azioni che si svolgono all'interno di tale sistema. In un artistico atto di ricorsione, tali azioni vengono scritte in modo da generare un linguaggio molto specifico di un dominio, per raccontare la loro piccola parte della storia.

Questo capitolo ha spiegato come scrivere bene le funzioni. Se seguirete le regole presentate, le vostre funzioni saranno brevi, avranno nomi “parlanti” e saranno ben organizzate. Ma non dimenticate mai che il vero obiettivo è quello di raccontare la storia del sistema, e che le funzioni che scrivete devono contribuire, ognuna, a creare un linguaggio chiaro e preciso utile per questo racconto.

SetupTeardownIncluder

Listato 3.7 SetupTeardownIncluder.java.

```
package fitnesse.html;
import fitnesse.responders.run.SuiteResponder;
import fitnesse.wiki.*;
public class SetupTeardownIncluder {
    private PageData pageData;
    private boolean isSuite;
    private WikiPage testPage;
    private StringBuffer newPageContent;
    private PageCrawler pageCrawler;
    public static String render(PageData pageData) throws Exception {
        return render(pageData, false);
    }
    public static String render(PageData pageData, boolean isSuite)
        throws Exception {
        return new SetupTeardownIncluder(pageData).render(isSuite);
    }
    private SetupTeardownIncluder(PageData pageData) {
        this.pageData = pageData;
        testPage = pageData.getWikiPage();
        pageCrawler = testPage.getPageCrawler();
        newPageContent = new StringBuffer();
    }
    private String render(boolean isSuite) throws Exception {
        this.isSuite = isSuite;
        if (isTestPage())
```

```

        includeSetupAndTeardownPages();
        return pageData.getHtml();
    }
    private boolean isTestPage() throws Exception {
        return pageData.hasAttribute("Test");
    }
    private void includeSetupAndTeardownPages() throws Exception {
        includeSetupPages();
        includePageContent();
        includeTeardownPages();
        updatePageContent();
    }
    private void includeSetupPages() throws Exception {
        if (isSuite)
            includeSuiteSetupPage();
        includeSetupPage();
    }
    private void includeSuiteSetupPage() throws Exception {
        include(SuiteResponder.SUITE SETUP NAME, "-setup");
    }
    private void includeSetupPage() throws Exception {
        include("SetUp", "-setup");
    }
    private void includePageContent() throws Exception {
        newPageContent.append(pageData.getContent());
    }
    private void includeTeardownPages() throws Exception {
        includeTeardownPage();
        if (isSuite)
            includeSuiteTeardownPage();
    }
    private void includeTeardownPage() throws Exception {
        include("TearDown", "-teardown");
    }
    private void includeSuiteTeardownPage() throws Exception {
        include(SuiteResponder.SUITE TEARDOWN NAME, "-teardown");
    }
    private void updatePageContent() throws Exception {
        pageData.setContent(newPageContent.toString());
    }
    private void include(String pageName, String arg) throws Exception {
        WikiPage inheritedPage = findInheritedPage(pageName);
        if (inheritedPage != null) {
            String pagePathName = getPathNameForPage(inheritedPage);
            buildIncludeDirective(pagePathName, arg);
        }
    }
    private WikiPage findInheritedPage(String pageName) throws Exception {
        return PageCrawlerImpl.getInheritedPage(pageName, testPage);
    }
    private String getPathNameForPage(WikiPage page) throws Exception {
        WikiPagePath pagePath = pageCrawler.getFullPath(page);
        return PathParser.render(pagePath);
    }
    private void buildIncludeDirective(String pagePathName, String arg) {
        newPageContent
            .append("\n!include ")
            .append(arg)
            .append(" .")
            .append(pagePathName)
    }
}

```

```
        .append("\n");
    }
}
```

Bibliografia

- [GOF]: Gamma et al., Design Patterns: Elements of Reusable Object Oriented Software, Addison-Wesley, Boston 1996.
- [KP78]: Kernighan and Plaugher, The Elements of Programming Style, 2d. ed., McGraw- Hill, New York 1978.
- [PPP02]: Robert C. Martin, Agile Software Development: Principles, Patterns, and Practices, Prentice Hall, Upper Saddle River, New Jersey 2002.
- [PRAG]: Andrew Hunt, Dave Thomas, The Pragmatic Programmer, Addison-Wesley, Boston 2000. Edizione italiana, *Il Pragmatic Programmer*, Apogeo, Milano 2018.
- [SP72]: O.-J. Dahl, E. W. Dijkstra, C. A. R. Hoare, Structured Programming, Academic Press, Londra 1972.

Capitolo 4

Commenti *di Brian W. Kernighan e P. J. Plaugher*

Non commentate il cattivo codice; piuttosto, riscrivetelo. ([KP78], p. 144)



Nulla può essere più utile di un buon commento ben posizionato. Nulla può rendere più disturbante un modulo di un ammasso di commenti frivoli e inutili. Nulla può essere più pericoloso di un vecchio commento che propaga falsità ed errate informazioni.

I commenti non sono come la “Lista di Schindler”. Non sono “pura bontà”. In effetti, i commenti sono, al massimo, un male necessario. Se i nostri linguaggi di programmazione fossero sufficientemente espressivi

o se avessimo il talento necessario per impiegare tali linguaggi per esprimere il nostro intento, non dovremmo usare molti commenti (magari proprio nessun commento).

L'uso corretto dei commenti ha lo scopo di compensare la nostra incapacità di esprimerci con il solo codice. Notate che ho usato la parola “incapacità”. Lo ammetto: i commenti sono sempre dei fallimenti. Dobbiamo scriverli, perché senza di essi non sempre è possibile esprimere quanto dobbiamo dire, ma il loro uso non è certo una gioia.

Pertanto quando vi capiterà di dover scrivere un commento, pensateci bene e cercate di scoprire se non esiste un modo per esprimere la stessa cosa nel codice. Ogni volta che riuscite a esprimervi con il solo codice, datevi una “pacca sulla spalla”. Ogni volta che scrivete un commento, fate in modo di provare il senso di incapacità di esprimersi appieno con il solo codice.

Perché sono così ostile ai commenti? Perché mentono. Non sempre e non intenzionalmente, ma troppo spesso. Più è vecchio un commento e più si allontana dal codice che descrive, e quindi più è probabile che sia, semplicemente, sbagliato. Il motivo è semplice. I programmatore non possono, realisticamente, eseguirne la manutenzione.

Il codice cambia e subisce una sua evoluzione. Parti di esso si spostano da un punto a un altro. Tali parti si scindono e si riproducono e poi si rimettono insieme come chimere. Sfortunatamente i commenti non sempre seguono le loro evoluzioni, *sarebbe impossibile*. E fin troppo spesso i commenti finiscono per allontanarsi dal codice che descrivono e divengono come orfani, sempre meno accurati. Per esempio, osservate che cosa è accaduto a questo commento e alla riga che intendeva descrivere:

```
MockRequest request;
private final String HTTP_DATE REGEXP =
  "[SMTWF] [a-z]{2} \\, \\s[0-9]{2} \\s[JFMASOND] [a-z]{2} \\s" +
  "[0-9]{4} \\s[0-9]{2} \\:[0-9]{2} \\:[0-9]{2} \\sGMT";
private Response response;
```

```
private FitNesseContext context;
private FileResponder responder;
private Locale saveLocale;

// Esempio: "Tue, 02 Apr 2003 22:18:49 GMT"
```

Le variabili di istanza probabilmente sono state aggiunte successivamente fra la costante `HTTP_DATE_REGEXP` e quello che doveva essere il commento descrittivo.

È anche possibile ribadire che i programmatori dovrebbero essere disciplinati al punto da aggiornare i commenti, in termini di rilevanza e accuratezza. Sono d'accordo, dovrebbero. Ma penso che sia meglio dedicare le energie a rendere il codice talmente chiaro ed espressivo da non aver bisogno di commenti.

I commenti imprecisi sono molto peggiori dell'assenza di commenti: sono deludenti e fuorvianti. Stabiliscono aspettative che non verranno mai esaudite. Stabiliscono regole che non vigono più o addirittura che andrebbero evitate.

La verità si trova in un solo luogo: il codice. Solo il codice dice davvero quello che fa. È l'unica fonte di informazioni davvero accurate. Pertanto, sebbene i commenti possano talvolta essere necessari, cerchiamo in tutti i modi di ridurli al minimo.

I commenti non bastano a migliorare il codice cattivo

Una delle motivazioni che più spingono a scrivere commenti è la problematicità del codice. Scriviamo un modulo e sappiamo che è confuso e disorganizzato. Sappiamo che è un bel groviglio. Allora ci diciamo: “Ummm... sarà meglio che ci metta un commento qui!”. No! Sarà meglio che lo sistemiate!

Poter contare su codice chiaro ed espressivo con pochi commenti è molto meglio che avere codice complesso e intricato ma ricco di commenti. Invece di spendere il vostro tempo a scrivere commenti che

spiegano il groviglio che avete fatto, dedicatelo a dipanare quel groviglio.

Spiegatevi nel codice

Vi sono certamente situazioni in cui il codice non riesce a spiegare se stesso. Sfortunatamente, molti programmati lo addotano come scusa, sostenendo che il codice solo raramente, forse mai, può spiegare se stesso. Questo è palesemente falso. Che cosa preferireste trovare?

Questo:

```
// Controlla se employee ha diritto a tutti i benefit
if ((employee.flags & HOURLY_FLAG) &&
    (employee.age > 65))
```

O questo?

```
if (employee.isEligibleForFullBenefits())
```

Pochi secondi sono sufficienti per comprendere l'intento del codice. In molti casi si tratta semplicemente di creare una funzione che dice la stessa cosa del commento che pensate di scrivere.

Buoni commenti

Alcuni commenti sono necessari o utili. Ne vedremo ora alcuni che considero meritevoli del tempo impiegato per scriverli. Tenete però sempre in considerazione che l'unico commento buono è quel commento che trovate il modo di non scrivere.

Commenti legali

Talvolta gli standard aziendali nell'ambito della programmazione ci obbligano a scrivere determinati commenti per motivi legali. Per esempio, le note di copyright e l'autore sono informazioni necessarie e ragionevoli per un commento posto all'inizio di ogni file di codice

sorgente. Ecco, per esempio, il commento standard che poniamo all'inizio di ogni file di codice sorgente in FitNesse. Sono felice che il nostro IDE nasconde automaticamente questo commento.

```
// Copyright (C) 2003,2004,2005 by Object Mentor, Inc. All rights reserved.  
// Released under the terms of the GNU General Public License version 2 or later.
```

Commenti come questo non dovrebbero essere contratti o pagine e pagine di “legalese”. Ove possibile, fate riferimento a una licenza standard o a un altro documento esterno invece di inserire nel commento tutti i termini e le condizioni.

Commenti informativi

Talvolta è utile fornire alcune informazioni tramite un commento. Per esempio, considerate questo commento che spiega il valore restituito da un metodo astratto:

```
// Restituisce un'istanza del Responder sottoposto a test.  
protected abstract Responder responderInstance();
```

Un commento come questo talvolta può essere utile, ma ove possibile è meglio usare il nome della funzione per suggerire tale informazione. Per esempio, in questo caso il commento potrebbe essere reso ridondante rinominando la funzione: `responderBeingTested`.

Ecco un caso un po' migliore:

```
// formato richiesto: kk:mm:ss EEE, MMM dd, yyyy  
Pattern timeMatcher = Pattern.compile(  
"\\"d*:\\d*:\\d* \\\w*, \\\w* \\\d*, \\\d*");
```

In questo caso il commento ci fa sapere che l'espressione regolare deve individuare un'ora e una data formattate con la funzione

`SimpleDateFormat.format` usando la stringa di formattazione specificata.

Ciononostante, tutto sarebbe più chiaro e pulito, se questo codice fosse stato spostato in un'apposita classe speciale che convertisse il formato delle date e delle ore. A questo punto, probabilmente il commento sarebbe diventato superfluo.

Descrizione dell'intento

Talvolta un commento va ben oltre la descrizione dell'implementazione e fornisce la motivazione di una decisione. Nel seguente caso vediamo un'interessante decisione documentata da un commento. Per confrontare due oggetti, l'autore ha deciso di voler ordinare gli oggetti della sua classe prima degli oggetti di ogni altra.

```
public int compareTo(Object o)
{
    if(o instanceof WikiPagePath)
    {
        WikiPagePath p = (WikiPagePath) o;
        String compressedName = StringUtil.join(names, "");
        String compressedArgumentName = StringUtil.join(p.names, "");
        return compressedName.compareTo(compressedArgumentName);
    }
    return 1; // siamo maggiori perché siamo del tipo corretto.
}
```

Ecco un esempio anche migliore. Potreste non concordare con la soluzione del problema scelta dal programmatore, ma almeno ora sapete che cosa tentava di fare.

```
public void testConcurrentAddWidgets() throws Exception {
    WidgetBuilder widgetBuilder =
        new WidgetBuilder(new Class[]{BoldWidget.class});
    String text = "'''bold text'''";
    ParentWidget parent =
        new BoldWidget(new MockWidgetRoot(), "'''bold text'''");
    AtomicBoolean failFlag = new AtomicBoolean();
    failFlag.set(false);
    //Questo è il nostro miglior tentativo di ottenere una competizione
    //creando un gran numero di thread.
    for (int i = 0; i < 25000; i++) {
        WidgetBuilderThread widgetBuilderThread =
            new WidgetBuilderThread(widgetBuilder, text, parent, failFlag);
        Thread thread = new Thread(widgetBuilderThread);
        thread.start();
    }
    assertEquals(false, failFlag.get());
}
```

Chiarimenti

Talvolta sembra opportuno tradurre il significato di uno strano argomento o valore restituito in qualcosa di leggibile. In generale è

meglio trovare un modo per chiarire il significato di tale argomento o valore restituito; ma quando esso fa parte della libreria standard o di codice che non è possibile modificare, un commento chiarificatore può essere utile.

```
public void testCompareTo() throws Exception {
    WikiPagePath a = PathParser.parse("PageA");
    WikiPagePath ab = PathParser.parse("PageA.PageB");
    WikiPagePath b = PathParser.parse("PageB");
    WikiPagePath aa = PathParser.parse("PageA.PageA");
    WikiPagePath bb = PathParser.parse("PageB.PageB");
    WikiPagePath ba = PathParser.parse("PageB.PageA");
    assertTrue(a.compareTo(a) == 0);           // a == a
    assertTrue(a.compareTo(b) != 0);           // a != b
    assertTrue(ab.compareTo(ab) == 0);         // ab == ab
    assertTrue(a.compareTo(b) == -1);          // a < b
    assertTrue(aa.compareTo(ab) == -1);        // aa < ab
    assertTrue(ba.compareTo(bb) == -1);        // ba < bb
    assertTrue(b.compareTo(a) == 1);           // b > a
    assertTrue(ab.compareTo(aa) == 1);         // ab > aa
    assertTrue(bb.compareTo(ba) == 1);         // bb > ba
}
```

Si corre però il rischio, naturalmente, che il commento sia impreciso. Scorrete l'esempio precedente e vedrete che è difficile verificare che tutto sia corretto. Questo spiega sia perché il chiarimento è necessario, sia perché è pericoloso. Pertanto, prima di scrivere dei commenti come questo, verificate che non vi sia un modo migliore di procedere e poi verificate assolutamente che i commenti siano precisi.

Avvertenze



Talvolta è utile mettere in guardia gli altri programmatori su determinate conseguenze. Per esempio, ecco un commento che spiega perché un determinato caso di test è stato disattivato:

```
// Non eseguite, a meno che
// abbiate del tempo da perdere.
public void _testWithReallyBigFile()
{
    writeLinesToFile(10000000);
    response.setBody(testFile);
    response.readyToSend(this);
    String responseString = output.toString();
    assertEquals("Content-Length: 1000000000", responseString);
    assertTrue(bytesSent > 1000000000);
}
```

Al giorno d'oggi, naturalmente, avremmo disattivato il caso di test usando l'attributo `@Ignore` con una stringa descrittiva appropriata:

`@Ignore("Takes too long to run")`. Ma prima di JUnit 4, si era soliti porre un *underscore* davanti al nome del metodo. Il commento, pur vago, chiarisce la cosa.

Ecco un altro esempio, più pregnante:

```
public static SimpleDateFormat makeStandardHttpDateFormat()
{
    //SimpleDateFormat non è thread safe,
    //pertanto dobbiamo creare ciascuna istanza in modo indipendente.
    SimpleDateFormat df = new SimpleDateFormat("EEE, dd MMM yyyy HH:mm:ss z");
    df.setTimeZone(TimeZone.getTimeZone("GMT"));
    return df;
}
```

Potreste obiettare che vi sono modi migliori per risolvere questo problema. Io potrei concordare con voi. Ma il commento, così come è, è perfettamente ragionevole. Ha lo scopo di impedire a qualche programmatore di usare un inizializzatore statico nel nome dell'efficienza.

Commenti TODO

Talvolta è ragionevole lasciare degli appunti `//TODO` per le parti da finire. Nel caso seguente, il commento TODO spiega perché la funzione ha questa implementazione inconsistente e come dovrebbe essere la versione finale della funzione.

```
// TODO-MdM cose non necessarie
// Ci aspettiamo che sparisca nel modello finale
protected VersionInfo makeVersion() throws Exception {
    return null;
}
```

I “TODO” sono operazioni che il programmatore pensa che dovrebbero essere fatte, ma che, per qualche motivo, al momento sono in sospeso. Può trattarsi di una nota per cancellare una funzionalità indesiderata o della richiesta a qualcun altro affinché consideri un problema. Potrebbe essere una richiesta affinché qualcun altro pensi a un nome migliore o a una nota che ricordi di fare una modifica che

dipende da un evento già pianificato. Qualsiasi sia il significato del “todo”, *non è* una scusa per lasciare del cattivo codice nel sistema.

Al giorno d’oggi, la maggior parte degli IDE offre particolari simboli e funzionalità per individuare tutti i commenti TODO, pertanto è più difficile perderseli. Ciononostante, evitate che il vostro codice sia pieno di “TODO”. Controllateli regolarmente ed eliminate quelli che potete.

Amplificazione

Un commento può essere usato per amplificare l’importanza di qualcosa che altrimenti potrebbe sembrare illogico.

```
String listItemContent = match.group(3).trim();
// Il trim qui è importante. Elimina gli spazi
// iniziali che potrebbero fare in modo che l'item
// venga riconosciuto come un'altra lista.
new ListItemWidget(this, listItemContent, this.level + 1);
return buildList(text.substring(match.end()));
```

Javadoc nelle API pubbliche

Non vi è nulla di altrettanto utile e gradevole di un’API pubblica ben documentata. I javadoc della libreria standard Java ne sono un esempio. Sarebbe difficile, a dire poco, scrivere programmi Java senza di essi. Se state scrivendo un’API pubblica, certamente dovrete anche scrivere dei buoni javadoc per essa. Ma tenete in considerazione anche l’altro consiglio di questo capitolo. I javadoc possono essere altrettanto fuorvianti, fuori posto e disonesti di ogni altro genere di commenti.

Cattivi commenti

La maggior parte dei commenti rientra in questa categoria. Solitamente sono solo pretesti per programmare malamente o

giustificazioni di decisioni scadenti, un po' come se il programmatore parlasse a se stesso.

Pensieri

Scrivere un commento solo perché sentite di doverlo fare o perché il processo lo richiede, è del tutto inutile. Se decidete di scrivere un commento, dedicategli il tempo necessario per assicurarvi che sia il miglior commento che possiate scrivere.

Qui, per esempio, riporto un caso che ho trovato in FitNesse, dove un commento poteva anche essere utile. Ma l'autore aveva fretta o forse non vi ha dedicato particolare attenzione. I suoi pensieri generano un enigma:

```
public void loadProperties()
{
    try
    {
        String propertiesPath = propertiesLocation + "/" + PROPERTIES_FILE;
        FileInputStream propertiesStream = new FileInputStream(propertiesPath);
        loadedProperties.load(propertiesStream);
    }
    catch(IOException e)
    {
        // Nessun file di proprietà? Carica i default
    }
}
```

Che cosa significa quel commento nel blocco `catch`? Chiaramente significava qualcosa per l'autore, ma il significato è piuttosto oscuro. Apparentemente, se otteniamo una `IOException`, significa che non vi era alcun file delle proprietà; e in tal caso verranno caricati tutti i default. Ma chi carica tutti i default? Sono già stati caricati prima della chiamata a `loadProperties.load`? O `loadProperties.load` raccoglie l'eccezione, carica i default e poi passa avanti l'eccezione, che dobbiamo ignorare? O `loadProperties.load` carica tutti i default prima di tentare di caricare il file? L'autore voleva solo rassicurarsi sul fatto che lasciava vuoto il blocco `catch`? O (e anche questa è una terribile possibilità) l'autore

tentava di dirsi di tornare a lavorare su questa parte per scrivere il codice che doveva caricare i default? Saremo quindi costretti a esaminare il codice in altre parti del sistema per scoprire che cosa accade.

Ogni commento che vi costringe a cercare in un altro modulo il significato di quel commento fallisce nel suo tentativo di comunicare con voi e non vale nemmeno i bit che consuma.

Commenti ridondanti

Il Listato 4.1 presenta una semplice funzione con un commento iniziale completamente ridondante. Probabilmente ci vuole più tempo a leggere il commento che a leggere e comprendere il codice.

Listato 4.1 waitForClose.

```
// Metodo di servizio che termina se this.closed è true.  
// Lancia un'eccezione se viene raggiunto il timeout.  
public synchronized void waitForClose(final long timeoutMillis) throws Exception {  
    if(!closed)  
    {  
        wait(timeoutMillis);  
        if(!closed)  
            throw new Exception("MockResponseSender could not be closed");  
    }  
}
```

Quale scopo ha questo commento? Certamente non è più informativo rispetto al codice. Non spiega il codice, né fornisce l'intento o la motivazione. Non è più facile da leggere rispetto al codice. In effetti, è anche meno preciso rispetto al codice e spinge chi legge ad accettare tale mancanza di precisione senza comprendere il vero significato del codice. È un po' come se un affabile venditore di auto usate vi garantisse che non avete alcun bisogno di guardare sotto il cofano motore.

Ora considerate la gran quantità di javadoc inutili e ridondanti presentati nel Listato 4.2, tratto da Tomcat. Questi commenti servono solo a congestionare e nascondere il codice. Non hanno alcuno scopo

documentale. A peggiorare le cose, vi ho mostrato solo i primi... ma ve ne sono parecchi in questo modulo.

Listato 4.2 ContainerBase.java (Tomcat).

```
public abstract class ContainerBase
    implements Container, Lifecycle, Pipeline,
    MBeanRegistration, Serializable {
    /**
     * The processor delay for this component.
     */
    protected int backgroundProcessorDelay = -1;
    /**
     * The lifecycle event support for this component.
     */
    protected LifecycleSupport lifecycle = new LifecycleSupport(this);
    /**
     * The container event listeners for this Container.
     */
    protected ArrayList listeners = new ArrayList();
    /**
     * The Loader implementation with which this Container is
     * associated.
     */
    protected Loader loader = null;
    /**
     * The Logger implementation with which this Container is
     * associated.
     */
    protected Log logger = null;
    /**
     * Associated logger name.
     */
    protected String logName = null;
    /**
     * The Manager implementation with which this Container is
     * associated.
     */
    protected Manager manager = null;
    /**
     * The cluster with which this Container is associated.
     */
    protected Cluster cluster = null;
    /**
     * The human-readable name of this Container.
     */
    protected String name = null;
    /**
     * The parent Container to which this Container is a child.
     */
    protected Container parent = null;
    /**
     * The parent class loader to be configured when we install a
     * Loader.
     */
    protected ClassLoader parentClassLoader = null;
    /**
     * The Pipeline object with which this Container is
     * associated.
     */
```

```

 */
protected Pipeline pipeline = new StandardPipeline(this);
/**
 * The Realm with which this Container is associated.
 */
protected Realm realm = null;
/**
 * The resources DirContext object with which this Container
 * is associated.
 */
protected DirContext resources = null;

```

Commenti fuorvianti

Talvolta, magari con tutte le migliori intenzioni, un programmatore fa un'affermazione nei propri commenti che è tutt'altro che accurata. Provate a tornare al ridondante, ma anche fuorviante, commento che abbiamo visto nel Listato 4.1.

In che senso il commento è fuorviante? Il metodo non esce *quando* `this.closed` diviene `true`. Esce *se* `this.closed` è `true`; in caso contrario, attende un certo time-out e poi lancia un'eccezione se `this.closed` non è ancora `true`.

Questa subdola imprecisione, inserita in un commento che è più difficile da leggere del corpo del codice, può far sì che un altro programmatore richiami inavvertitamente questa funzione aspettandosi che termini non appena `this.closed` diviene `true`. Il povero programmatore dovrà così svolgere una sessione di debug nel tentativo di scoprire perché il suo codice è così lento.

Commenti obbligati

Non è molto intelligente imporre una regola che dice che ogni funzione deve avere un javadoc o ogni variable deve avere un commento. Commenti come questo non fanno altro che congestionare il codice, propagare cose non vere e introdurre confusione e disorganizzazione.

Per esempio, l'imposizione di un javadoc per ogni funzione genera abomini come quello rappresentato nel Listato 4.3. Questa “roba” non aggiunge nulla e serve solo a offuscare il codice e a creare le condizioni per l'insorgere di falsità.

Listato 4.3

```
/**  
 *  
 * @param title Il titolo del CD  
 * @param author L'autore del CD  
 * @param tracks Il numero di tracce del CD  
 * @param durationInMinutes La durata del CD in minuti  
 */  
public void addCD(String title, String author,  
                   int tracks, int durationInMinutes) {  
    CD cd = new CD();  
    cd.title = title;  
    cd.author = author;  
    cd.tracks = tracks;  
    cd.duration = duration;  
    cdList.add(cd);  
}
```

Commenti a “log”

Talvolta i programmatori aggiungono un commento all'inizio di un modulo ogni volta che lo modificano. Questi commenti si accumulano come in un log, e registrano ogni modifica svolta. Mi è capitato di vedere moduli con decine di pagine di queste righe di log.

```
* Modifiche (dal 11-Ott-2001)  
* -----  
* 11-Ott-2001 : Riorganizzato la classe e spostata in un nuovo package  
   com.jrefinery.date (DG);  
* 05-Nov-2001 : Aggiunto un metodo getDescription() ed eliminata  
   la classe NotableDate;  
* 12-Nov-2001 : IBD richiede il metodo setDescription(), ora che la classe  
   NotableDate non c'è più (DG); Modificate getPreviousDayOfWeek(),  
   getFollowingDayOfWeek() e getNearestDayOfWeek() per correggere  
   dei bug (DG);  
* 05-Dic-2001 : Corretto bug nella classe SpreadsheetDate (DG);  
* 29-Mag-2002 : Spostate le costanti dei mesi in un'interfaccia a parte  
   (MonthConstants) (DG);  
* 27-Ago-2002 : Corretto bug nel metodo addMonths(), grazie a N???levka Petr (DG);  
* 03-Ott-2002 : Corretti errori rilevati da Checkstyle (DG);  
* 13-Mar-2003 : Implementato Serializable (DG);  
* 29-Mag-2003 : Corretto bug nel metodo addMonths (DG);  
* 04-Set-2003 : Implementato Comparable. Aggiornati i javadoc isInRange (DG);
```

```
* 05-Gen-2005 : Corretto bug nel metodo addYears() (1096282) (DG);
```

Molto tempo fa vi era un buon motivo per creare e gestire questi log all'inizio di ogni modulo. Non avevamo i sistemi di controllo del codice sorgente di oggi. Al giorno d'oggi, tuttavia, questi log non fanno altro che offuscare il modulo e quindi dovrebbero essere completamente rimossi.

Puro “rumore”

Talvolta si trovano commenti che non servono proprio a nulla. Ribadiscono l'ovvio e non forniscono alcuna informazione.

```
/**  
 * Costruttore di default.  
 */  
  
protected AnnualDateRule() {  
  
    Ma no?!?! Davvero?  
    E che dire di questo:  
  
    /** Il giorno del mese. */  
  
    private int dayOfMonth;
```

E poi vi è anche la ridondanza dell'inutilità:

```
/**  
 * Restituisce il giorno del mese.  
 *  
 * @return il giorno del mese.  
 */  
public int getDayOfMonth() {  
    return dayOfMonth;  
}
```

Questi commenti sono talmente inutili che presto impariamo a ignorarli. Mentre leggiamo il codice, i nostri occhi semplicemente li saltano. E poi i commenti iniziano a mentire quando il codice attorno a loro cambia.

Il primo commento del Listato 4.4 sembra appropriato (l'attuale tendenza degli IDE di controllare l'ortografia nei commenti sarà un vero balsamo per tutti coloro che leggono molto codice). Spiega perché

il blocco `catch` è stato ignorato. Ma il secondo commento è puro rumore. Apparentemente il programmatore era talmente frustrato dai blocchi `try/catch` di questa funzione che aveva bisogno di un po' d'aria.

Listato 4.4 startSending.

```
private void startSending()
{
    try
    {
        doSending();
    }
    catch(SocketException e)
    {
        // normale. qualcuno ha bloccato la richiesta.
    }
    catch(Exception e)
    {
        try
        {
            response.add(ErrorResponder.makeExceptionString(e));
            response.closeAll();
        }
        catch(Exception e1)
        {
            //Basta, vi prego!
        }
    }
}
```

Invece di scrivere un inutile e liberatorio commento, il programmatore avrebbe dovuto riconoscere che la sua frustrazione si sarebbe potuta risolvere migliorando la struttura del suo codice. Avrebbe dovuto indirizzare le sue energie nell'estrazione dell'ultimo blocco `try/catch` in una funzione distinta, come si può vedere nel Listato 4.5.

Listato 4.5 startSending (dopo un refactoring).

```
private void startSending()
{
    try
    {
        doSending();
    }
    catch(SocketException e)
    {
        // normale. qualcuno ha bloccato la richiesta.
    }
    catch(Exception e)
    {
```

```

        addExceptionAndCloseResponse (e) ;
    }
}
private void addExceptionAndCloseResponse (Exception e)
{
try
{
    response.add(ErrorResponder.makeExceptionString (e)) ;
    response.closeAll () ;
}
catch (Exception e1)
{
}
}
}

```

Resistite alla tentazione di creare puro “rumore” con la determinazione nel ripulire il vostro codice. Diventerete programmatori migliori e anche più felici.

Assoluto “rumore”

Anche i javadoc possono essere fastidiosamente “rumorosi”. Quale scopo hanno i seguenti javadoc (tratti da una nota libreria open-source)? Risposta: nessuno. Sono solo commenti ridondanti e rumorosi scritti con lo scopo di fornire chissà quale documentazione.

```

/** Il nome. */
private String name;
/** La versione. */
private String version;
/** Il licenceName. */
private String licenceName;
/** La versione. */

private String info;

```

Leggete questi commenti con attenzione. Lo vedete l’errore copia-incolla? Se gli autori non dedicano la necessaria attenzione ai commenti mentre li scrivono (o incollano), perché mai coloro che leggono il codice dovrebbero considerarli?

Non usate un commento al posto di una funzione o una variabile

Considerate il seguente frammento di codice:

```
// il modulo della lista globale <mod> dipende  
// dal sottosistema?  
  
if (smodule.getDependSubsystems().contains(subSysMod.getSubSystem()) )
```

Si potrebbe riscrivere senza commento come:

```
ArrayList moduleDependees = smodule.getDependSubsystems();  
String ourSubSystem = subSysMod.getSubSystem();  
  
if (moduleDependees.contains(ourSubSystem))
```

L'autore del codice potrebbe aver scritto prima il commento (improbabile) e poi aver scritto il codice in base al commento. Tuttavia, l'autore avrebbe dovuto eseguire il refactoring del codice, nel modo descritto, così da rimuovere il commento.

Contrassegni di posizione

Talvolta i programmatori amano contrassegnare una determinata posizione in un file di codice sorgente. Per esempio, recentemente ho trovato questa cosa in un programma che esaminavo:

```
// Actions /////////////////////////////////
```

Vi sono rare situazioni in cui è sensato raccogliere determinate funzioni sotto un banner come questo. Ma in generale queste cose dovrebbero essere eliminate, in particolare la lunga sequenza di slash alla fine.

Pensatela in questo modo. Un banner è evidente e ovvio solo se non si ripresenta troppo spesso. Pertanto usateli con molta parsimonia e solo quando il vantaggio è significativo. Se eccederete con l'uso, finiranno nel rumore di fondo e verranno ignorati.

Commenti per le parentesi graffe chiuse

Talvolta i programmatori inseriscono dei commenti alle parentesi graffe chiuse, come nel Listato 4.6. Questo può avere senso nelle funzioni molto lunghe con strutture profondamente annidate, ma non

servono a nulla nelle piccole funzioni incapsulate che prediligiamo. Pertanto, se pensate di dover commentare le parentesi graffe chiuse, cercate piuttosto di accorciare le vostre funzioni.

Listato 4.6 wc.java.

```
public class wc {
    public static void main(String[] args) {
        BufferedReader in = new BufferedReader(new InputStreamReader(System.in));
        String line;
        int lineCount = 0;
        int charCount = 0;
        int wordCount = 0;
        try {
            while ((line = in.readLine()) != null) {
                lineCount++;
                charCount += line.length();
                String words[] = line.split("\\W");
                wordCount += words.length;
            } //while
            System.out.println("wordCount = " + wordCount);
            System.out.println("lineCount = " + lineCount);
            System.out.println("charCount = " + charCount);
        } // try
        catch (IOException e) {
            System.err.println("Error:" + e.getMessage());
        } //catch
    } //main
}
```

Attribuzioni

/* Aggiunto da Rick */

I sistemi di controllo del codice sorgente sono precisissimi nel ricordare chi ha aggiunto cosa e quando. Non vi è alcuna necessità di inquinare il codice con queste cose. Potreste pensare che tali commenti possano essere utili per aiutare gli altri a sapere a chi chiedere informazioni sul codice. Ma la realtà è che questi commenti tendono a rimanere in giro per anni e anni, divenendo sempre meno accurati e rilevanti.

Di nuovo, il sistema di controllo del codice sorgente è un luogo migliore per questo genere di informazioni.

Codice commentato

Poche abitudini sono altrettanto fastidiose del porre il codice in commenti. Non fatelo!

```
InputStreamResponse response = new InputStreamResponse();
response.setBody(formatter.getResultStream(), formatter.getByteCount());
// InputStream resultsStream = formatter.getResultStream();
// StreamReader reader = new StreamReader(resultsStream);

// response.setContent(reader.read(formatter.getByteCount()));
```

Coloro che vedranno quel codice commentato non avranno il coraggio di cancellarlo. Penseranno che sia lì per un motivo e che sia troppo importante per cancellarlo. Pertanto il codice commentato finisce per accumularsi come il fondo in una pessima bottiglia di vino.

Considerate il seguente esempio tratto da apache commons:

```
this.bytePos = writeBytes(pngIdBytes, 0);
//hdrPos = bytePos;
writeHeader();
writeResolution();
//dataPos = bytePos;
if (writeImageData()) {
    writeEnd();
    this.pngBytes = resizeByteArray(this.pngBytes, this.maxPos);
} else {
    this.pngBytes = null;
}

return this.pngBytes;
```

Perché queste due righe di codice sono commentate? Sono importanti? Sono state lasciate lì come annotazione per una modifica imminente? O sono solo residui che qualcuno ha commentato anni fa e, semplicemente, non si è preoccupato di eliminare?

Un tempo, parliamo degli anni Sessanta, il fatto di trasformare in commenti il codice poteva essere utile. Ma oggi contiamo su ottimi sistemi di controllo del codice sorgente. Tali sistemi possono ricordarci il codice impiegato. Non abbiamo più bisogno di usare questi tipi di commenti. Cancellate il codice. Non lo perderete. Fidatevi.

Commenti HTML

La presenza di codice HTML nei commenti al codice sorgente è un abominio, come potete sincerarvi leggendo il codice sottostante. Complica la leggibilità dei commenti nel posto in cui dovrebbero essere più facili da leggere: l'editor/IDE. Se i commenti dovranno essere estratti da qualche strumento (come Javadoc) per comparire in una pagina web, allora l'aggiunta del codice HTML rientra nelle responsabilità di tale strumento, non del programmatore.

```
/**
 * Task per i test di correttezza.
 * Questo task esegue dei test di Fitnesse e pubblica i risultati.
 * <p/>
 * <pre>
 * Uso:
 * <taskdef name="execute-fitness-tests"
 *          classname="fitnesse.ant.ExecuteFitnessTestsTask";
 *          classpathref="classpath"; />;
 * OPPURE
 * <taskdef classpathref="classpath";
 *          resource="tasks.properties"; />;
 * <p/>
 * <execute-fitness-tests
 *          suitepage="FitNesse.SuiteAcceptanceTests";
 *          fitnesseport="8082";
 *          resultsdir="${results.dir}";
 *          resultshtmlpage="fit-results.html";
 *          classpathref="classpath"; />;
 * </pre>
 */
```

Informazioni fuori posizione

Se dovete scrivere un commento, allora assicuratevi che descriva il codice circostante. Non offrite informazioni di sistema nel contesto di un commento locale. Considerate, per esempio, il commento javadoc sottostante. A parte il fatto che è del tutto ridondante, offre anche informazioni sulla porta di default. E comunque la funzione non ha assolutamente alcun controllo su tale default. Il commento non sta descrivendo la funzione, ma una qualche altra parte del sistema, molto molto lontana. Naturalmente non vi è alcuna garanzia che questo

commento verrà modificato quando cambierà il codice che contiene il default.

```
/**  
 * Porta utilizzata da fitnesse. Default = <b>8082</b>.   
 *  
 * @param fitnessePort  
 */  
public void setFitnessePort(int fitnessePort)  
{  
    this.fitnessePort = fitnessePort;  
}
```

Eccesso di informazioni

Non infilate nei commenti i dettagli di interessanti discussioni storiche o di descrizioni irrilevanti. Il commento sottostante è stato estratto da un modulo progettato per verificare che una funzione potesse eseguire la codifica e decodifica base64. A parte il numero di RFC, chi leggerà questo codice non ha alcun bisogno delle esotiche informazioni contenute nel commento.

```
/*  
RFC 2045 - Multipurpose Internet Mail Extensions (MIME)  
Parte 1: Formato degli Internet Message Bodies  
sezione 6.8. Base64 Content-Transfer-Encoding  
Il processo di codifica rappresenta i gruppi di bit di input (24 bit) come  
stringhe di output di quattro caratteri codificati. Procedendo da sinistra a  
destra, un gruppo di input da 24 bit viene formato concatenando tre gruppi di  
input a 8 bit.  
Questi 24 bit vengono poi trattati come quattro gruppi concatenati da 6 bit,  
ognuno dei quali viene tradotto in un'unica cifra nell'alfabeto base64.  
Nella codifica base64 di uno stream di bit, tale stream si presume ordinato  
a partire dal bit più significativo.  
In pratica, il primo bit dello stream sarà nel bit di ordine più elevato nel  
primo byte di 8 bit e l'ottavo bit sarà nel bit di ordine inferiore del primo  
byte di 8 bit e così via.  
*/
```

Scarso legame con il codice

La connessione fra il commento e il codice che descrive dovrebbe essere evidente. Se vi date il disturbo di scrivere un commento, quanto

vorreste che chi legge sia in grado di vedere il commento e il codice e di capire il motivo del commento.

Considerate, per esempio, questo commento tratto da apache commons:

```
/*
 * iniziate con un array di dimensioni sufficienti per contenere tutti i pixel
 * (più i byte del filtro) e altri 200 byte per l'header
 */
this.pngBytes = new byte[((this.width + 1) * this.height * 3) + 200];
```

Che cos’è un *byte del filtro*? Ha a che fare con il `+1`? O con il `* 3`? O con entrambi? Un pixel è di un byte? Perché `200`? Lo scopo di un commento è quello di spiegare il codice che non si spiega da solo. È un peccato quando un commento richiede... spiegazioni.

Intestazioni di funzioni

Le funzioni brevi non hanno bisogno di troppe descrizioni. Un nome “parlante” per una piccola funzione che faccia una cosa sola è solitamente meglio di un commento di intestazione.

Javadoc nel codice non pubblico

Per quanto possano essere utili i javadoc per le API pubbliche, essi rappresentano un anatema per il codice che non è rivolto all’uso pubblico. Produrre pagine javadoc per classi e funzioni interne di un sistema non è utile a nessuno e la maggiore formalità dei commenti javadoc è di valore sostanzialmente nullo.

Esempio

Ho scritto il modulo presentato nel Listato 4.7 per il primo *XP Immersion*. Aveva lo scopo di rappresentare un cattivo esempio di programmazione e scelta dei commenti. Kent Beck ha poi eseguito il

refactoring di questo codice in una forma molto più sensata davanti ad alcune decine di allievi entusiasti. Successivamente ho adattato l'esempio per il mio libro *Agile Software Development, Principles, Patterns, and Practices* e il primo dei miei articoli *Craftsman* pubblicati sul “Software Development Magazine”.

Ciò che trovo affascinante di questo modulo è che vi fu un tempo in cui molti di noi l'avrebbero considerato “ben documentato”. Oggi lo consideriamo negativamente. Provate a vedere quanti problemi riuscite a individuarvi.

Listato 4.7 GeneratePrimes.java.

```
/*
 * Questa classe genera dei numeri primi fino a un massimo specificato
 * dall'utente. L'algoritmo usato è il Setaccio di Eratostene.
 * <p>
 * Eratosthenes di Cirene, a. c. 276 AC, Cirene, Libia --
 * d. c. 194, Alessandria. Il primo uomo a calcolare la
 * circonferenza della terra. Noto anche per aver lavorato sui
 * calendari con anni bisestili e aver gestito la libreria di Alessandria.
 * <p>
 * L'algoritmo è semplice. Dato un array di interi
 * che partono da 2. Elimina tutti i multipli di 2. Trova il
 * prossimo intero disponibile ed elimina tutti i suoi multipli.
 * Ripeti fino ad aver superato la radice quadrata del valore
 * massimo.
 *
 * @author Alphonse
 * @version 13 Feb 2002 atp
 */
import java.util.*;
public class GeneratePrimes
{
    /**
     * @param maxValue è il limite della generazione.
     */
    public static int[] generatePrimes(int maxValue)
    {
        if (maxValue >= 2) // l'unico caso valido
        {
            // dichiarazioni
            int s = maxValue + 1; // dimensioni array
            boolean[] f = new boolean[s];
            int i;
            // inizializza l'array a true.
            for (i = 0; i < s; i++)
                f[i] = true;
            // elimina i non-primi
            f[0] = f[1] = false;
            // setaccio
            int j;
            for (i = 2; i < Math.sqrt(s) + 1; i++)
```

```

{
    if (f[i]) // se i è primo, elimina i suoi multipli.
    {
        for (j = 2 * i; j < s; j += i)
            f[j] = false; // un multiplo non può essere primo
    }
}
// quanti primi ci sono?
int count = 0;
for (i = 0; i < s; i++)
{
    if (f[i]) count++; // conteggio.
}
int[] primes = new int[count];
// sposta i primi nel risultato
for (i = 0, j = 0; i < s; i++)
{
    if (f[i]) // se è primo
        primes[j++] = i;
}
return primes; // restituisci i primi
}
else // maxValue < 2
    return new int[0]; // restituisce un array nullo se l'input è errato.
}
}

```

Nel Listato 4.8 trovate una versione dello stesso modulo dopo un refactoring. Notate che l'uso dei commenti è molto limitato. Vi sono solo due commenti nell'intero modulo. Entrambi sono descrittivi.

Listato 4.8 PrimeGenerator.java (dopo un refactoring).

```

/**
 * Questa classe Genera dei numeri primi fino a un massimo specificato
 * dall'utente. L'algoritmo usato è il Setaccio di Eratostene.
 * Dato un array di interi a partire da 2:
 * trova il primo intero non eliminato ed elimina tutti i suoi
 * multipli. Ripeti finché non vi sono più multipli
 * nell'array.
 */
public class PrimeGenerator {
    private static boolean[] crossedOut;
    private static int[] result;
    public static int[] generatePrimes(int maxValue)
    {
        if (maxValue < 2)
            return new int[0];
        else
        {
            uncrossIntegersUpTo(maxValue);
            crossOutMultiples();
            putUncrossedIntegersIntoResult();
            return result;
        }
    }
    private static void uncrossIntegersUpTo(int maxValue)
    {

```

```

crossedOut = new boolean[maxValue + 1];
for (int i = 2; i < crossedOut.length; i++)
    crossedOut[i] = false;
}
private static void crossOutMultiples()
{
    int limit = determineIterationLimit();
    for (int i = 2; i <= limit; i++)
        if (notCrossed(i))
            crossOutMultiplesOf(i);
}
private static int determineIterationLimit()
{
    // Ogni multiplo nell'array ha un fattore primo che è minore
    // o uguale alla radice delle dimensioni dell'array,
    // così che possiamo evitare di eliminare i multipli
    // dei numeri maggiori di tale radice.
    double iterationLimit = Math.sqrt(crossedOut.length);
    return (int) iterationLimit;
}
private static void crossOutMultiplesOf(int i)
{
    for (int multiple = 2*i;
        multiple < crossedOut.length;
        multiple += i)
        crossedOut[multiple] = true;
}
private static boolean notCrossed(int i)
{
    return crossedOut[i] == false;
}
private static void putUncrossedIntegersIntoResult()
{
    result = new int[numberOfUncrossedIntegers()];
    for (int j = 0, i = 2; i < crossedOut.length; i++)
        if (notCrossed(i))
            result[j++] = i;
}
private static int numberOfUncrossedIntegers()
{
    int count = 0;
    for (int i = 2; i < crossedOut.length; i++)
        if (notCrossed(i))
            count++;
    return count;
}
}

```

È facile notare che il primo commento è ridondante, perché è un po' come quello della funzione `generatePrimes`. Ciononostante, penso che il commento sia utile per facilitare chi legge l'algoritmo, quindi ho preferito lasciarlo.

Il secondo argomento è quasi certamente necessario. Spiega l'uso della radice quadrata come limite del ciclo. Non sono riuscito a trovare

alcun nome di variabile o struttura del programma che chiarisse a sufficienza le cose. D'altra parte, l'uso della radice quadrata pone dei dubbi. Risparmio davvero del tempo limitando l'iterazione alla radice quadrata? Non è che il calcolo della radice quadrata richiede più tempo di quanto ne faccia risparmiare?

Vale la pena di rifletterci. L'uso della radice quadrata come limite dell'iterazione soddisfa il vecchio programmatore C e Assembly che abita in me, ma non sono sicuro che valga il tempo e l'impegno che chiedo agli altri per comprenderne la motivazione.

Bibliografia

- [KP78]: Kernighan and Plaugher, *The Elements of Programming Style*, 2d. ed., McGraw-Hill, New York 1978.

Capitolo 5

Formattazione



Quando qualcuno guarderà “sotto il cofano” del nostro codice, vogliamo che sia impressionato dalla pulizia, dalla coerenza e dall’attenzione al dettaglio che si percepisce. Vogliamo che ne ammirino l’ordine. Vogliamo vedere le loro sopracciglia allargarsi

mentre ne scorrono i moduli. Vogliamo che percepiscano che è opera di professionisti. Se invece vi trovano un ammasso di codice che sembra scritto da una compagnia di marinai ubriachi, ne trarranno l'impressione che la stessa “attenzione” sia stata rivolta a ogni altro aspetto del progetto.

Assicuratevi che il vostro codice sia ben formattato. Dovreste scegliere un insieme di regole semplici in base alle quali formattare il vostro codice e poi dovreste applicare coerentemente tali regole. Se lavorate in un team, tutto il team dovrebbe accordarsi su un unico insieme di regole di formattazione, alle quali si atterranno tutti. Sarà utile anche impiegare uno strumento automatico in grado di applicare tali regole di formattazione per voi.

Lo scopo della formattazione

Prima di tutto, è importante essere chiari: la formattazione del codice è importante. È troppo importante per ignorarla e troppo importante per trattarla religiosamente. La formattazione del codice è comunicazione e la comunicazione è la massima priorità dello sviluppatore professionale.

Forse pensavate che “far funzionare il codice” fosse la vostra massima priorità. Spero però che, giunti a questo punto, questo libro vi abbia dissuaso da tale idea. La funzionalità che create oggi ha buone probabilità di cambiare nella prossima release, ma la leggibilità del vostro codice avrà un effetto profondo su tutte le modifiche che verranno eseguite. Lo stile di programmazione e leggibilità definiscono dei precedenti che continueranno a influenzare manutenibilità ed estendibilità anche molto tempo dopo che il codice è stato modificato. Il vostro stile e la vostra disciplina sopravviveranno, anche al di là del vostro codice.

Pertanto quali sono gli elementi di formattazione che ci aiutano a comunicare meglio?

Formattazione verticale

Iniziamo con la formattazione verticale. Quanto dovrebbe essere grande un file di codice sorgente? In Java, le dimensioni del file sono strettamente legate alle dimensioni della classe. Parleremo di dimensioni di una classe quando parleremo delle classi. Per il momento consideriamo solo le dimensioni di un file.

Quanto sono lunghi, in genere, i file di codice sorgente Java? Naturalmente le dimensioni (e anche lo stile di scrittura) variano molto. La Figura 5.1 presenta alcune di queste differenze.

Sono rappresentati sette progetti differenti: Junit, FitNesse, testNG, Time and Money, JDepend, Ant e Tomcat. Le linee che attraversano i rettangoli considerano la lunghezza minima e massima dei file in ciascun progetto. Il rettangolo rappresenta approssimativamente un terzo, una deviazione standard, dei file. (Il rettangolo si estende per $\sigma/2$ sopra e sotto la media. Sì, lo so che la distribuzione della lunghezza dei file non è normale e che quindi la deviazione standard non è matematicamente precisa. Ma qui non siamo alla ricerca della precisione. Stiamo solo cercando di farci un'idea.) dei file. Il centro del rettangolo è la media. Pertanto le dimensioni medie di un file del progetto FitNesse è di circa 65 righe e circa un terzo dei file si colloca fra le 40 e le 100+ righe. Il file più grande di FitNesse è di circa 400 righe e il più piccolo è di 6 righe. Notate che questa è una scala logaritmica, e pertanto anche una piccola differenza in senso verticale implica una grande differenza di dimensioni in termini assoluti.

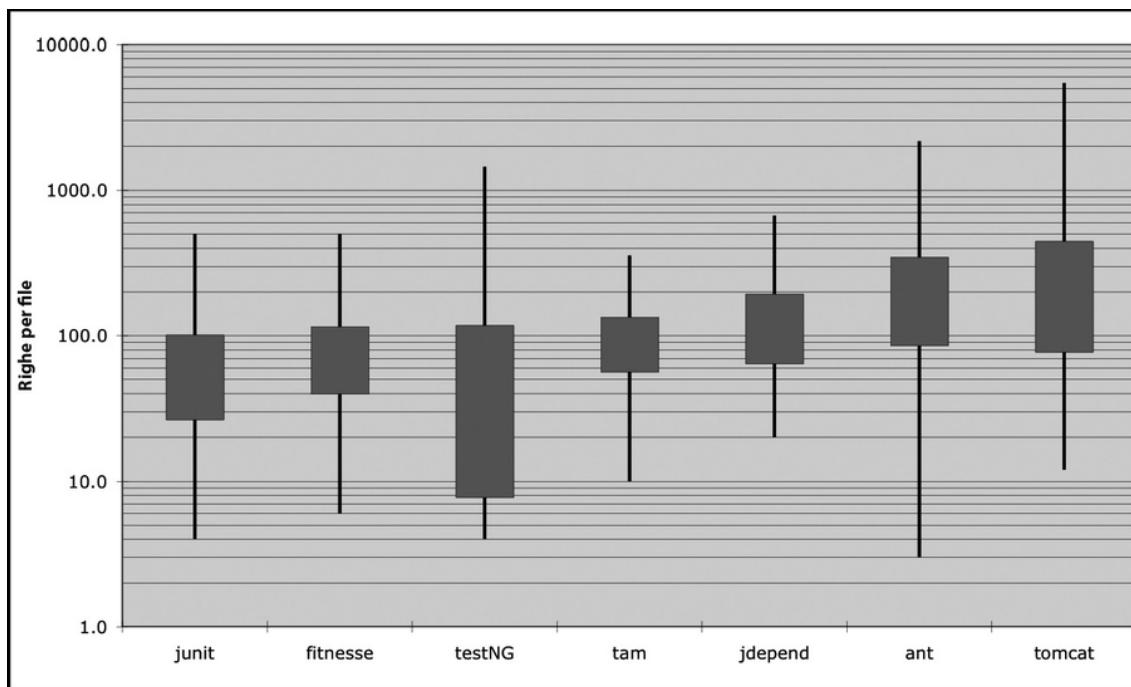


Figura 5.1 Distribuzione della lunghezza dei file in scala logaritmica (altezza rettangolo = sigma).

Junit, FitNesse e Time and Money sono costituiti da file relativamente piccoli. Nessuno supera le 500 righe e la maggior parte di questi file si estende per meno di 200 righe. Tomcat e Ant, d'altra parte, hanno alcuni file lunghi alcune migliaia di righe e circa la metà di essi ha più di 200 righe.

Che cosa significa tutto questo? Sembra che sia possibile realizzare sistemi di tutto rispetto (FitNesse si avvicina alle 50.000 righe) impiegando file di circa 200 righe, al massimo 500 righe. Anche se questa non è una regola universale, dovrebbe essere considerata una caratteristica molto positiva. I piccoli file, solitamente, sono più facili da comprendere di quelli lunghi.

La metafora della rivista

Pensate a un articolo ben scritto. Lo leggete verticalmente. In cima vi aspettate un titolo che vi comunichi l'argomento e vi consenta di decidere se si tratta di qualcosa di interesse. Il primo paragrafo vi presenterà l'intera storia, evitando tutti i dettagli e fornendovi solo i concetti principali. Procedendo verso il basso, aumenteranno i dettagli, fino a raggiungere tutte le date, i nomi, le citazioni, le affermazioni e così via.

Un file di codice sorgente dovrebbe essere un po' come un articolo. Il nome dovrebbe essere semplice ma descrittivo. Quanto al nome dovrebbe essere sufficiente per farci capire se ci troviamo nel modulo corretto. La parte superiore del file di codice sorgente dovrebbe fornire i concetti e algoritmi di alto livello. I dettagli dovrebbero aumentare procedendo lungo il file, finché, alla fine del file di codice sorgente, troviamo le funzioni di livello più basso e i dettagli.

Una rivista è costituita da molti articoli; la maggior parte dei quali molto piccoli. Alcuni sono un po' più grandi. Pochissimi riempiono un'intera pagina. È questo che rende fruibile una rivista. Se la rivista fosse costituita da un'unica lunga storia, contenente un agglomerato disorganizzato di fatti, date e nomi, semplicemente non la leggeremmo.

Spaziatura verticale fra i concetti

Quasi tutto il codice si legge da sinistra a destra e dall'alto in basso. Ogni riga rappresenta un'espressione o una clausola e ogni gruppo di righe rappresenta un ragionamento. Questi ragionamenti dovrebbero essere separati dagli altri da alcune righe vuote.

Considerate, per esempio, il Listato 5.1. Vi sono alcune righe vuote che separano la dichiarazione del package, le importazioni ed entrambe le funzioni. Questa regola, estremamente semplice, ha un effetto profondo sul layout visuale del codice. Ogni riga vuota è un indizio visuale che identifica un nuovo concetto, distinto. Mentre scorrete il

listato, il vostro sguardo verrà attratto dalla prima riga che segue una riga vuota.

Listato 5.1 BoldWidget.java.

```
package fitnesse.wikitext.widgets;
import java.util.regex.*;
public class BoldWidget extends ParentWidget {
    public static final String REGEXP = "'''.+?'''";
    private static final Pattern pattern = Pattern.compile("'''(.+?)'''",
        Pattern.MULTILINE + Pattern.DOTALL
    );
    public BoldWidget(ParentWidget parent, String text) throws Exception {
        super(parent);
        Matcher match = pattern.matcher(text);
        match.find();
        addChildWidgets(match.group(1));
    }
    public String render() throws Exception {
        StringBuffer html = new StringBuffer("<b>");
        html.append(childHtml()).append("</b>");
        return html.toString();
    }
}
```

Anche solo eliminando queste righe, come nel Listato 5.2, la leggibilità del codice cala drasticamente.

Listato 5.2 BoldWidget.java.

```
package fitnesse.wikitext.widgets;
import java.util.regex.*;
public class BoldWidget extends ParentWidget {
    public static final String REGEXP = "'''.+?'''";
    private static final Pattern pattern = Pattern.compile("'''(.+?)'''",
        Pattern.MULTILINE + Pattern.DOTALL);
    public BoldWidget(ParentWidget parent, String text) throws Exception {
        super(parent);
        Matcher match = pattern.matcher(text);
        match.find();
        addChildWidgets(match.group(1));
    }
    public String render() throws Exception {
        StringBuffer html = new StringBuffer("<b>");
        html.append(childHtml()).append("</b>");
        return html.toString();
    }
}
```

Questo effetto è ancora più pronunciato se allontanate lo sguardo e poi vi ritornate. Nel primo esempio i gruppi di righe saltano all'occhio, mentre nel secondo esempio tutto si “impasta”. La differenza fra questi due listati sta solo nella spaziatura verticale.

Densità verticale

Se un’apertura separa i concetti, la densità verticale implica una forte associazione. Pertanto le righe di codice strettamente correlate dovrebbero presentarsi verticalmente “dense”. Notate come gli inutili commenti presentati nel Listato 5.3 spezzino la stretta associazione fra le due variabili di istanza.

Listato 5.3

```
public class ReporterConfig {  
    /**  
     * Il nome della classe del reporter listener  
     */  
    private String m_className;  
    /**  
     * Le proprietà del reporter listener  
     */  
    private List<Property> m_properties = new ArrayList<Property>();  
    public void addProperty(Property property) {  
        m_properties.add(property);  
    }  
}
```

Il Listato 5.4 è molto più facile da leggere e rientra in una sola “occhiata”. Basta osservarlo per capire che si tratta di una classe con due variabili e un metodo, senza dover vagare troppo con lo sguardo. Il listato precedente obbliga a un maggiore impegno, con lo sguardo, per ottenere lo stesso livello di comprensione.

Listato 5.4

```
public class ReporterConfig {  
    private String m_className;  
    private List<Property> m_properties = new ArrayList<Property>();  
    public void addProperty(Property property) {  
        m_properties.add(property);  
    }  
}
```

Distanza verticale

Vi è mai capitato di vagare all’interno di una classe, saltando da una funzione alla successiva, scorrendo in lungo e in largo il file di codice

sorgente, nel tentativo di indovinare le relazioni fra le funzioni, solo per perdervi in una sorta di labirinto? Vi è mai capitato di ricercare affannosamente la catena di ereditarietà per la definizione di una variabile o di una funzione? È frustrante, perché cercate disperatamente di comprendere che *cosa* fa il sistema, ma vi trovate a sprecare tempo ed energie nel tentativo di individuare, e poi ricordare, *come* ricomporre i pezzi.

I concetti strettamente correlati dovrebbero trovarsi vicini, verticalmente [G10]. Chiaramente questa regola non si applica a concetti che si trovano in file distinti. Ma d'altra parte i concetti strettamente correlati non dovrebbero trovarsi separati in file distinti, se non per un ottimo motivo. In effetti, questo è uno dei motivi per evitare l'uso di variabili protette.

Per quei concetti che sono così strettamente correlati da appartenere allo stesso file di codice sorgente, la separazione verticale dovrebbe essere un'indicazione dell'importanza dell'uno per la comprensibilità dell'altro. Vogliamo evitare di costringere coloro che leggono il codice a saltare qua e là nel file di codice sorgente e nelle classi.

Dichiarazioni di variabili

Le variabili dovrebbero essere dichiarate il più possibile vicine al luogo in cui vengono usate. Poiché le nostre funzioni sono molto corte, le variabili locali dovrebbero trovarsi in cima a ogni funzione, come in questa funzione tratta da Junit 4.3.1.

```
private static void readPreferences() {
    InputStream is= null;
    try {
        is= new FileInputStream(getPreferencesFile());
        setPreferences(new Properties(getPreferences()));
        getPreferences().load(is);
    } catch (IOException e) {
        try {
            if (is != null) is.close();
        } catch (IOException el) {
        }
    }
}
```

```
}
```

Le variabili di controllo per i cicli solitamente dovrebbero essere dichiarate nell'istruzione del ciclo, come in questa piccola funzione, tratta sempre dalla stessa fonte.

```
public int countTestCases() {  
    int count= 0;  
    for (Test each : tests)  
        count += each.countTestCases();  
    return count;  
}
```

In alcuni rari casi una variabile può dover essere dichiarata in cima a un blocco o appena prima di un ciclo in una funzione non breve. Potete vedere una variabile di questo tipo nel frammento tratto dal bel mezzo di una lunga funzione di TestNG.

```
...  
for (XmlTest test : m_suite.getTests()) {  
    TestRunner tr = m_runnerFactory.newTestRunner(this, test);  
    tr.addListener(m_textReporter);  
    m_testRunners.add(tr);  
    invoker = tr.getInvoker();  
    for (ITestNGMethod m : tr.getBeforeSuiteMethods()) {  
        beforeSuiteMethods.put(m.getMethod(), m);  
    }  
    for (ITestNGMethod m : tr.getAfterSuiteMethods()) {  
        afterSuiteMethods.put(m.getMethod(), m);  
    }  
}  
...
```

Variabili di istanza

Le *variabili di istanza*, d'altra parte, dovrebbero essere dichiarate in cima alla classe. Questo non dovrebbe aumentare la distanza verticale di queste variabili, perché in una classe ben progettata, esse vengono usate da molti dei metodi della classe, se non tutti.

Esistono molte discussioni sulla corretta collocazione delle variabili di istanza. In C++ adottavamo comunemente la cosiddetta “regola delle forbici”, che richiedeva che tutte le variabili di istanza si trovassero in fondo. La convenzione comune in Java, al contrario, dice di collocarle tutte in cima alla classe. Non vedo alcun buon motivo per seguire

convenzioni differenti. L'importante è che le variabili di istanza vengano dichiarate in una posizione ben precisa. Così tutti sapranno dove andare a cercare le dichiarazioni.

Considerate, per esempio, lo strano caso della classe `TestSuite` di JUnit 4.3.1. Ho molto “ritagliato” questa classe per arrivare al punto. Se osservate circa a metà del listato, vedrete le dichiarazioni di due variabili di istanza. Sarebbe difficile trovare un luogo migliore per *nasconderle*. Chi si troverà a leggere questo codice inciamperà nelle dichiarazioni forse solo per caso (come è capitato a me).

```
public class TestSuite implements Test {  
    static public Test createTest(Class<? extends TestCase> theClass, String name) {  
        ...  
    }  
    public static Constructor<? extends TestCase>  
    getTestConstructor(Class<? extends TestCase> theClass)  
    throws NoSuchMethodException {  
        ...  
    }  
  
    public static Test warning(final String message) {  
        ...  
    }  
  
    private static String exceptionToString(Throwable t) {  
        ...  
    }  
  
    private String fName;  
  
private Vector<Test> fTests= new Vector<Test>(10);  
  
    public TestSuite() {  
    }  
  
    public TestSuite(final Class<? extends TestCase> theClass) {  
        ...  
    }  
  
    public TestSuite(Class<? extends TestCase> theClass, String name) {  
        ...  
    }  
    ...  
}
```

Funzioni dipendenti

Se una funzione ne richiama un'altra, esse dovrebbero trovarsi verticalmente vicine e la chiamante dovrebbe essere sopra quella richiamata, se possibile. Ciò dona al programma un flusso naturale. Se questa convenzione viene seguita coerentemente, coloro che leggeranno il codice si renderanno conto che le definizioni di funzioni si trovano poco dopo il loro uso. Considerate, per esempio, il frammento tratto da FitNesse presentato nel Listato 5.5. Notate come la funzione superiore richiami quelle sottostanti e come quelle, a loro volta, richiamino quelle successive. Così è facile trovare le funzioni richiamate, a tutto vantaggio della leggibilità dell'intero modulo.

Listato 5.5 WikiPageResponder.java.

```
public class WikiPageResponder implements SecureResponder {
    protected WikiPage page;
    protected PageData pageData;
    protected String pageTitle;
    protected Request request;
    protected PageCrawler crawler;
    public Response makeResponse(FitNesseContext context, Request request)
        throws Exception {
        String pageName = getPageNameOrDefault(request, "FrontPage");
        loadPage(pageName, context);
        if (page == null)
            return notFoundResponse(context, request);
        else
            return makePageResponse(context);
    }
    private String getPageNameOrDefault(Request request, String defaultPageName)
    {
        String pageName = request.getResource();
        if (StringUtil.isBlank(pageName))
            pageName = defaultPageName;
        return pageName;
    }
    protected void loadPage(String resource, FitNesseContext context)
        throws Exception {
        WikiPagePath path = PathParser.parse(resource);
        crawler = context.root.getPageCrawler();
        crawler.setDeadEndStrategy(new VirtualEnabledPageCrawler());
        page = crawler.getPage(context.root, path);
        if (page != null)
            pageData = page.getData();
    }
    private Response notFoundResponse(FitNesseContext context, Request request)
        throws Exception {
        return new NotFoundResponder().makeResponse(context, request);
    }
    private SimpleResponse makePageResponse(FitNesseContext context)
        throws Exception {
        pageTitle = PathParser.render(crawler.getFullPath(page));
        return new SimpleResponse("Page", pageTitle);
    }
}
```

```
        String html = makeHtml(context);
        SimpleResponse response = new SimpleResponse();
        response.setMaxAge(0);
        response.setContent(html);
        return response;
    }

    ...
}
```

A proposito, questo frammento di codice fornisce un esempio di come le costanti debbano essere mantenute al livello appropriato [G35]. La costante "FrontPage" avrebbe potuto essere seppellita nella funzione `getPageNameOrDefault`, ma ciò avrebbe nascosto in una funzione di livello eccessivamente basso una costante conosciuta e prevista. È stato meglio passare tale costante dalla posizione in cui è sensato sapere che verrà usata.

Affinità concettuale



Alcuni frammenti di codice “vogliono stare vicini”. Hanno una certa affinità concettuale. Più forte è tale affinità, minore distanza verticale dovrebbe separarli.

Come abbiamo visto, questa affinità può basarsi su una dipendenza diretta, come quando una funzione ne richiama un’altra o una funzione usa una variabile. Ma vi sono altre possibili cause di affinità. Può essere dovuta al fatto che un gruppo di funzioni svolge operazioni simili. Considerate questo frammento di codice tratto da Junit 4.3.1:

```

public class Assert {
    static public void assertTrue(String message, boolean condition) {
        if (!condition) fail(message);
    }
    static public void assertTrue(boolean condition) {
        assertTrue(null, condition);
    }
    static public void assertFalse(String message, boolean condition) {
        assertTrue(message, !condition);
    }
    static public void assertFalse(boolean condition) {
        assertFalse(null, condition);
    }
}
...

```

Queste funzioni hanno una forte affinità concettuale, perché condividono lo stesso schema di denominazione e svolgono varianti della stessa operazione di base. Il fatto che si richiamano fra loro è secondario. Anche se non lo facessero, dovrebbero comunque stare vicine fra loro.

Ordinamento verticale

In generale vogliamo che le dipendenze nelle chiamate a funzione procedano verso il basso. In pratica, una funzione che viene richiamata dovrebbe trovarsi sotto una funzione che esegue la chiamata (l'esatto opposto di linguaggi come il Pascal, il C e il C++, che impongono che le funzioni siano definite, o almeno dichiarate, prima dell'uso). Questo crea un flusso armonioso lungo il modulo di codice sorgente, dai livelli superiori ai livelli inferiori.

Come in un articolo, ci aspettiamo che i concetti più importanti vengano per primi e che siano espressi senza troppo “inquinamento” da parte dei dettagli. Ci aspettiamo che i dettagli di basso livello vengano per ultimi. Questo ci consente di sfogliare il file di codice sorgente, di farci un’idea del significato delle prime funzioni, il tutto senza immergerci nei dettagli. Il Listato 5.5 è organizzato in questo modo. Esempi ancora migliori si trovano nel Listato 15.5 e nel Listato 3.7.

Formattazione orizzontale

Quanto dovrebbe essere larga una riga? Per rispondere a questa domanda, proviamo a osservare la larghezza delle righe in un tipico programma. Di nuovo, esaminiamo i sette progetti che abbiamo considerato per la formattazione verticale. La Figura 5.2 presenta la distribuzione della lunghezza delle righe dei sette progetti. La regolarità è impressionante, in particolare attorno ai 45 caratteri. In effetti, ogni singola dimensione dai 20 ai 60 caratteri ammonta all'1 percento del numero totale di righe. In totale si tratta del 40 percento! Forse un altro 30 percento si trova tra le righe più corte di 10 caratteri. Ricordate che questa è una scala logaritmica, e quindi l'aspetto lineare della discesa oltre gli 80 caratteri in realtà è molto significativa. È evidente che i programmatore prediligono le righe brevi.

Questo suggerisce di preferire l'impiego di righe brevi. Il vecchio limite di Hollerith di 80 caratteri è ormai un po' arbitrario e nulla più vieta che le righe siano lunghe 100 o anche 120 caratteri. Ma probabilmente non è mai utile andare oltre.

Ero solito seguire la regola di non dover mai eseguire scorimenti verso destra. Ma al giorno d'oggi i monitor sono larghi e i giovani programmatore possono stringere il font al punto da riuscire a far stare fino a 200 caratteri in una riga dello schermo. Non fatelo.

Personalmente imposto il mio limite a 120.

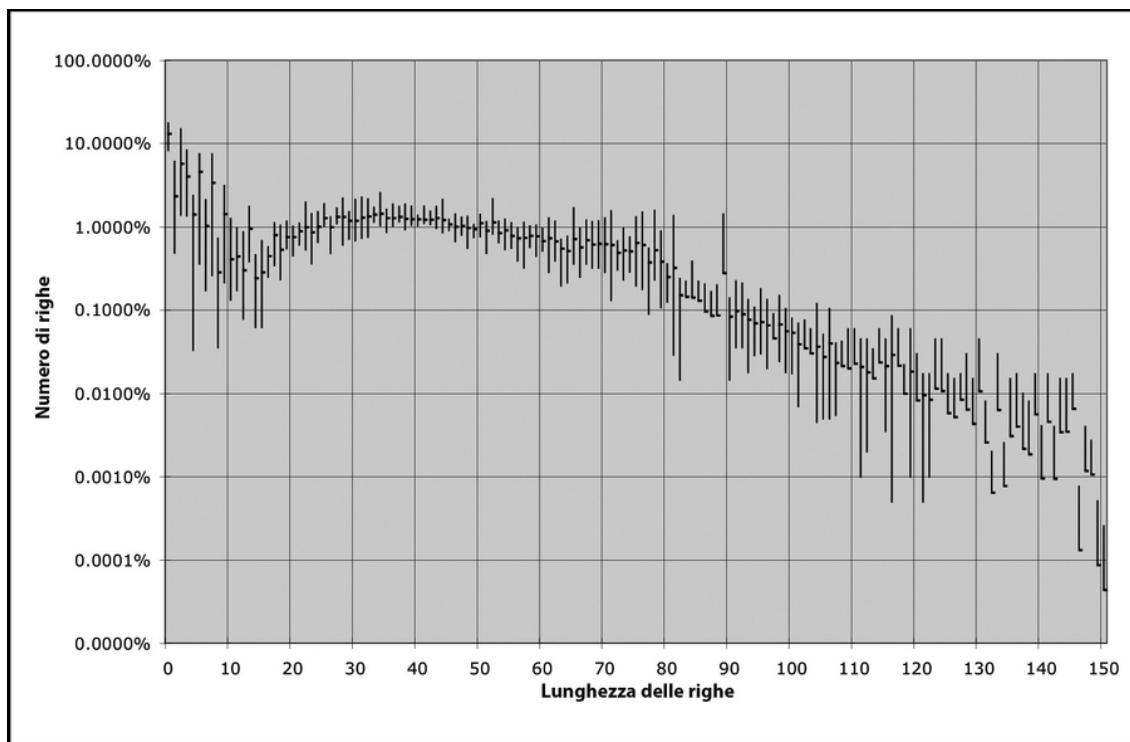


Figura 5.2 Distribuzione della lunghezza delle righe in Java.

Spaziatura e densità orizzontale

Usiamo la spaziatura orizzontale per associare quegli elementi che sono fortemente correlati e per disassociare quegli elementi che sono più debolmente correlati. Considerate la seguente funzione:

```
private void measureLine(String line) {
    lineCount++;
    int lineSize = line.length();
    totalChars += lineSize;
    lineWidthHistogram.addLine(lineSize, lineCount);
    recordWidestLine(lineSize);
}
```

Ho circondato gli operatori di assegnamento con uno spazio per evidenziarli. Le istruzioni di assegnamento hanno principalmente due elementi distinti: il lato sinistro e il lato destro. I due spazi evidenziano tale separazione.

Al contrario, non ho messo spazi fra i nomi di funzione e le parentesi aperte. Questo perché la funzione e i suoi argomenti sono strettamente correlati. Separandoli li renderemmo disgiunti e non è quello che vogliamo. Ho separato gli argomenti nelle parentesi della chiamata a funzione per evidenziare la virgola e mostrare che gli argomenti sono separati.

Un altro uso dello spazio permette di evidenziare la precedenza degli operatori.

```
public class Quadratic {  
    public static double root1(double a, double b, double c) {  
        double determinant = determinant(a, b, c);  
        return (-b + Math.sqrt(determinant)) / (2*a);  
    }  
    public static double root2(int a, int b, int c) {  
        double determinant = determinant(a, b, c);  
        return (-b - Math.sqrt(determinant)) / (2*a);  
    }  
    private static double determinant(double a, double b, double c) {  
        return b*b - 4*a*c;  
    }  
}
```

Notate come si leggono bene le equazioni. Non vi sono spazi fra i fattori perché hanno una precedenza elevata. I termini sono separati da uno spazio perché la somma e la sottrazione hanno una precedenza inferiore.

Sfortunatamente, la maggior parte degli strumenti di riformattazione del codice non considera la precedenza degli operatori e impone sempre la stessa spaziatura. Pertanto questi accorgimenti tendono a perdere dopo una riformattazione del codice.

Allineamento orizzontale

Quando programmavo in Assembly (Ma chi voglio prendere in giro? Io sono ancora un programmatore Assembly. Il lupo perde il pelo... ma non il vizio!), usavo l'allineamento orizzontale per evidenziare determinate strutture. Quando ho iniziato a programmare in C, C++ e,

alla fine, Java, ho continuato a tentare di allineare tutti i nomi di variabili di un insieme di dichiarazioni o tutti gli rvalue di un insieme di istruzioni di assegnamento. Il mio codice poteva avere il seguente aspetto:

```
public class FitNesseExpediter implements ResponseSender
{
    private   Socket      socket;
    private   InputStream  input;
    private   OutputStream output;
    private   Request     request;
    private   Response    response;
    private   FitNesseContext context;
    protected long    requestParsingTimeLimit;
    private   long     requestProgress;
    private   long     requestParsingDeadline;
    private   boolean   hasError;
    public FitNesseExpediter(Socket s, FitNesseContext context) throws Exception
    {
        this.context = context;
        socket = s;
        input = s.getInputStream();
        output = s.getOutputStream();
        requestParsingTimeLimit = 10000;
    }
}
```

Tuttavia ho scoperto che questo tipo di allineamento non è utile. L'allineamento sembra evidenziare le cose sbagliate e allontana lo sguardo dal vero intento. Per esempio, nella lista di dichiarazioni precedente sareste tentati di scorrere la lista dei nomi di variabili senza considerarne il tipo. Analogamente, nella lista di istruzioni di assegnamento sareste tentati di scorrere la lista di rvalue senza nemmeno considerare l'operatore di assegnamento. A peggiorare le cose, gli strumenti automatici di riformattazione solitamente eliminano questo tipo di allineamento.

Pertanto, alla fine, non faccio più questo genere di cose. Al giorno d'oggi preferisco le dichiarazioni e gli assegnamenti non allineati, come potete vedere di seguito, perché evidenziano un importante difetto: se ho lunghi elenchi che devono essere allineati, *il problema sta nella lunghezza dell'elenco*, non nella mancanza di allineamento. La

lunghezza dell’elenco di dichiarazioni in FitNesseExpediter (vedi sotto) suggerisce che questa classe debba essere suddivisa.

```
public class FitNesseExpediter implements ResponseSender
{
    private Socket socket;
    private InputStream input;
    private OutputStream output;
    private Request request;
    private Response response;
    private FitNesseContext context;
    protected long requestParsingTimeLimit;
    private long requestProgress;
    private long requestParsingDeadline;
    private boolean hasError;
    public FitNesseExpediter(Socket s, FitNesseContext context) throws Exception
    {
        this.context = context;
        socket = s;
        input = s.getInputStream();
        output = s.getOutputStream();
        requestParsingTimeLimit = 10000;
    }
}
```

Indentazione

Un file di codice sorgente ha una struttura gerarchica. Vi sono informazioni relative all’intero file, alle singole classi del file, ai metodi delle classi, ai blocchi dei metodi e poi ricorsivamente ai sottoblocchi dei blocchi. Ogni livello di questa gerarchia è anche un livello di visibilità (scope) all’interno del quale possono essere dichiarati dei nomi e nel quale vengono interpretate le dichiarazioni e le istruzioni eseguibili.

Per rendere più visibile questa gerarchia di livelli, indentiamo le righe di codice sorgente in proporzione alla loro posizione nella gerarchia. Le istruzioni a livello del file, come quelle della maggior parte delle classi, non hanno alcun livello di indentazione. I metodi all’interno di una classe sono indentati di un livello a destra rispetto alla classe. Le implementazioni di tali metodi sono indentate di un livello a destra rispetto alla dichiarazione del metodo. Le

implementazioni dei blocchi sono indentate di un livello a destra rispetto al contenitore e così via.

I programmati contano moltissimo su questo schema di indentazione. Allineano visivamente le righe per chiarirne la posizione nella gerarchia. Questo consente loro di individuare rapidamente i livelli, come le implementazioni delle istruzioni `if` o `while` inadatte. Scorrono il margine sinistro alla ricerca delle dichiarazioni di nuovi metodi, nuove variabili e anche nuove classi. Senza indentazione, i programmi sarebbero praticamente illeggibili a un occhio umano.

Considerate i seguenti programmi, che sono sintatticamente e semanticamente identici:

```
public class FitNesseServer implements SocketServer { private FitNesseContext context; public FitNesseServer(FitNesseContext context) { this.context = context; } public void serve(Socket s) { serve(s, 10000); } public void serve(Socket s, long requestTimeout) { try { FitNesseExpediter sender = new FitNesseExpediter(s, context); sender.setRequestParsingTimeLimit(requestTimeout); sender.start(); } catch(Exception e) { e.printStackTrace(); } } }
```

```
-----
```

```
public class FitNesseServer implements SocketServer { private FitNesseContext context; public FitNesseServer(FitNesseContext context) { this.context = context; } public void serve(Socket s) { serve(s, 10000); } public void serve(Socket s, long requestTimeout) { try { FitNesseExpediter sender = new FitNesseExpediter(s, context); sender.setRequestParsingTimeLimit(requestTimeout); sender.start(); } catch (Exception e) { e.printStackTrace(); } } }
```

Lo sguardo individua immediatamente la struttura del file indentato. Si trovano subito le variabili, i costruttori, i metodi di accesso e gli altri metodi. Bastano pochi secondi per rendersi conto che questo è un semplice front-end per un socket, con un time-out. La versione non

indentata, al contrario, è del tutto incomprensibile, senza studiarne i dettagli.

Infrangere l'indentazione

Talvolta si sarebbe tentati di infrangere la regola di indentazione per i costrutti brevi: istruzioni `if`, cicli `while` o funzioni. Ogni volta che ho ceduto a questa tentazione, quasi sempre sono tornato sui miei passi e ho rimesso l'indentazione. Pertanto in generale evito di far collassare i livelli nel seguente modo:

```
public class CommentWidget extends TextWidget
{
    public static final String REGEXP = "^[^\\r\\n]*(?:^(?:\\r\\n) | \\n|\\r)?";
    public CommentWidget(ParentWidget parent, String text) {super(parent, text);}
    public String render() throws Exception {return "";}
}
```

Preferisco espandere e indentare i livelli nel seguente modo:

```
public class CommentWidget extends TextWidget {
    public static final String REGEXP = "^[^\\r\\n]*(?:^(?:\\r\\n) | \\n|\\r)?";
    public CommentWidget(ParentWidget parent, String text) {
        super(parent, text);
    }
    public String render() throws Exception {
        return "";
    }
}
```

Livelli fintizi

Talvolta il corpo di un'istruzione `while` o `for` è fintizio, come potete vedere di seguito. Non mi piace ricorrere a questo tipo di strutture e tento di evitarle. Quando non posso evitarle, mi assicuro che il corpo, pur fintizio, sia correttamente indentato e circondato da parentesi graffe.

Non so dirvi quante volte sono stato inganato da un punto e virgola posto alla fine di un ciclo `while` sulla stessa riga. Se non rendete visibile tale punto e virgola indentandolo su una sua riga, risulterà invisibile.

```
while (dis.read(buf, 0, readBufferSize) != -1)
;
```

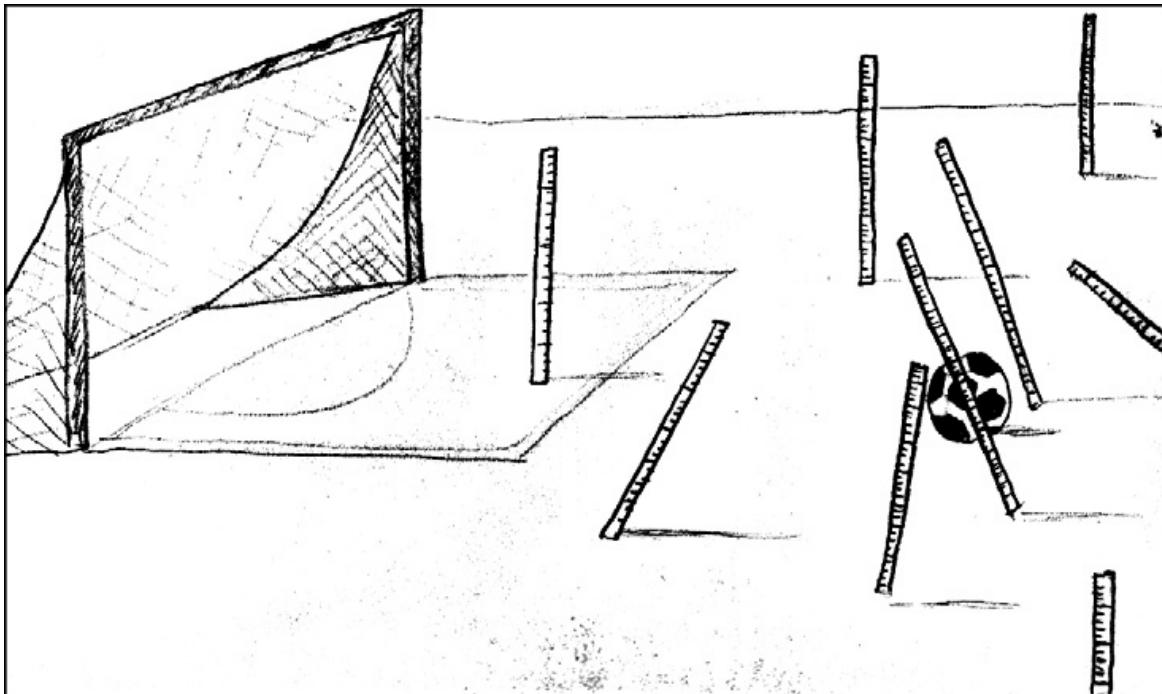
Le regole del team

Ogni programmatore ha le sue regole di formattazione preferite, ma se lavora in un team, allora prevale il team.

Un team di sviluppatori potrebbe accordarsi su un determinato stile di formattazione e ogni membro di tale team dovrà adottare tale stile. Vogliamo che il software abbia uno stile coerente. Non vogliamo che sembri scritto da un ammasso di individui litigiosi.

Quando ho iniziato il progetto FitNesse, nel 2002, mi sono seduto con il team per elaborare uno stile di programmazione. Il tutto ha richiesto dieci minuti. Abbiamo deciso dove andavano le parentesi graffe, di quanti caratteri eseguire l'indentazione, come denominare le classi, le variabili e i metodi e così via. Poi abbiamo codificato tali regole nel formattatore di codice del nostro IDE e abbiamo proseguito così. Queste non erano le regole che io preferivo; erano le regole decise dal team. Come membro di tale team le ho seguite nella realizzazione del codice di tutto il progetto FitNesse.

Ricordate: un buon sistema software è costituito da un insieme di documenti che si leggono senza problemi. Devono avere uno stile coerente e coeso. Chi legge deve essere sicuro che la formattazione che troverà in un file di codice sorgente la ritroverà negli altri. L'ultima cosa che vogliamo fare è complicare il codice sorgente scrivendolo ognuno con il proprio stile.



Le regole di formattazione di Uncle Bob

Le regole che uso personalmente sono molto semplici e sono illustrate dal codice presentato nel Listato 5.6. Consideratelo un esempio di codice perfettamente standardizzato.

Listato 5.6 CodeAnalyzer.java.

```
public class CodeAnalyzer implements JavaFileAnalysis {
    private int lineCount;
    private int maxLineWidth;
    private int widestLineNumber;
    private LineWidthHistogram lineWidthHistogram;
    private int totalChars;
    public CodeAnalyzer() {
        lineWidthHistogram = new LineWidthHistogram();
    }
    public static List<File> findJavaFiles(File parentDirectory) {
        List<File> files = new ArrayList<File>();
        findJavaFiles(parentDirectory, files);
        return files;
    }
    private static void findJavaFiles(File parentDirectory, List<File> files) {
        for (File file : parentDirectory.listFiles()) {
            if (file.getName().endsWith(".java"))
                files.add(file);
            else if (file.isDirectory())
                findJavaFiles(file, files);
        }
    }
}
```

```

        }
    }
    public void analyzeFile(File javaFile) throws Exception {
        BufferedReader br = new BufferedReader(new FileReader(javaFile));
        String line;
        while ((line = br.readLine()) != null)
            measureLine(line);
    }
    private void measureLine(String line) {
        lineCount++;
        int lineSize = line.length();
        totalChars += lineSize;
        lineWidthHistogram.addLine(lineSize, lineCount);
        recordWidestLine(lineSize);
    }
    private void recordWidestLine(int lineSize) {
        if (lineSize > maxLineWidth) {
            maxLineWidth = lineSize;
            widestLineNumber = lineCount;
        }
    }
    public int getLineCount() {
        return lineCount;
    }
    public int getMaxLineWidth() {
        return maxLineWidth;
    }
    public int getWidestLineNumber() {
        return widestLineNumber;
    }
    public LineWidthHistogram getLineWidthHistogram() {
        return lineWidthHistogram;
    }
    public double getMeanLineWidth() {
        return (double)totalChars/lineCount;
    }
    public int getMedianLineWidth() {
        Integer[] sortedWidths = getSortedWidths();
        int cumulativeLineCount = 0;
        for (int width : sortedWidths) {
            cumulativeLineCount += lineCountForWidth(width);
            if (cumulativeLineCount > lineCount/2)
                return width;
        }
        throw new Error("Cannot get here");
    }
    private int lineCountForWidth(int width) {
        return lineWidthHistogram.getLinesforWidth(width).size();
    }
    private Integer[] getSortedWidths() {
        Set<Integer> widths = lineWidthHistogram.getWidths();
        Integer[] sortedWidths = (widths.toArray(new Integer[0]));
        Arrays.sort(sortedWidths);
        return sortedWidths;
    }
}

```

Capitolo 6

Oggetti e strutture



Vi è un motivo per cui manteniamo private le variabili. Non vogliamo che qualcun altro dipenda da loro. Vogliamo mantenere la libertà di cambiare il loro tipo o la loro implementazione a seconda delle esigenze. Perché, allora, così tanti programmatore aggiungo

automaticamente dei *getter* e *setter* ai loro oggetti, esponendo così le loro variabili private come se fossero pubbliche?

Astrazione dei dati

Considerate la differenza fra il Listato 6.1 e il Listato 6.2. Entrambi rappresentano i dati di un punto sul piano cartesiano. Ma uno espone la propria implementazione e l'altro la nasconde completamente.

Listato 6.1 Punto concreto.

```
public class Point {  
    public double x;  
    public double y;  
}
```

Listato 6.2 Punto astratto.

```
public interface Point {  
    double getX();  
    double getY();  
    void setCartesian(double x, double y);  
    double getR();  
    double getTheta();  
    void setPolar(double r, double theta);  
}
```

L'aspetto più interessante del Listato 6.2 è che non vi è alcun modo di sapere se l'implementazione è in coordinate rettangolari o polari (o magari nessuna delle due!). E tuttavia l'interfaccia rappresenta inequivocabilmente una struttura.

Ma essa rappresenta qualcosa di più di una semplice struttura. I metodi stabiliscono una politica degli accessi. Potete leggere le singole coordinate in modo indipendente, ma dovete impostare le coordinate insieme con un'operazione atomica.

Il Listato 6.1, al contrario, è con ogni evidenza implementato in coordinate rettangolari e ci costringe a manipolare tali coordinate in modo indipendente. Questo espone l'implementazione. In effetti,

esporrebbe l'implementazione anche se le variabili fossero private e se usassimo i getter e setter mono-variabile.

Non nascondiamo l'implementazione solo per il gusto di mettere un livello di funzioni fra le variabili. Il fatto di nascondere l'implementazione consente di introdurre un'astrazione! Una classe non si limita a gestire le sue variabili tramite getter e setter. Piuttosto espone delle interfacce astratte che consentono agli utenti di manipolare l'*essenza* dei dati, senza doverne conoscere l'implementazione.

Considerate il Listato 6.3 e il Listato 6.4. Il primo usa termini concreti per comunicare il livello di carburante di un veicolo, mentre il secondo lo fa applicando l'astrazione della percentuale. Nel caso concreto si può essere ragionevolmente sicuri che questi sono solo metodi di accesso delle variabili. Nel caso astratto non si ha alcun indizio sulla forma dei dati.

Listato 6.3 Veicolo concreto.

```
public interface Vehicle {  
    double getFuelTankCapacityInGallons();  
    double getGallonsOfGasoline();  
}
```

Listato 6.4 Veicolo astratto.

```
public interface Vehicle {  
    double getPercentFuelRemaining();  
}
```

Dei due casi, la seconda opzione è preferibile. Non vogliamo esporre i dettagli dei nostri dati. Piuttosto vogliamo esprimere i nostri dati in termini astratti. Questo non si ottiene semplicemente usando interfacce e/o getter e setter. Occorre riflettere bene sul modo migliore per rappresentare i dati contenuti in un oggetto. L'opzione peggiore consiste nell'aggiungere allegramente dei getter e setter.

Asimmetria dei dati/oggetti

Questi due esempi mostrano la differenza esistente fra oggetti e strutture. Gli oggetti nascondono i loro dati dietro astrazioni ed espongono le funzioni che operano su tali dati. Le strutture espongono i loro dati e non hanno alcuna funzione significativa. Rileggete questa frase. Notate la natura complementare delle due definizioni. Sono praticamente opposte. Questa differenza può sembrare banale, ma ha implicazioni notevoli.

Considerate, per esempio, l'esempio delle forme procedurali presentato nel Listato 6.5. La classe `Geometry` opera sulle classi delle tre forme. Le classi delle forme sono semplici strutture senza alcun comportamento. Tutti i comportamenti si trovano nella classe `Geometry`.

Listato 6.5 Forme procedurali.

```
public class Square {
    public Point topLeft;
    public double side;
}
public class Rectangle {
    public Point topLeft;
    public double height;
    public double width;
}
public class Circle {
    public Point center;
    public double radius;
}
public class Geometry {
    public final double PI = 3.141592653589793;
    public double area(Object shape) throws NoSuchShapeException
    {
        if (shape instanceof Square) {
            Square s = (Square)shape;
            return s.side * s.side;
        }
        else if (shape instanceof Rectangle) {
            Rectangle r = (Rectangle)shape;
            return r.height * r.width;
        }
        else if (shape instanceof Circle) {
            Circle c = (Circle)shape;
            return PI * c.radius * c.radius;
        }
        throw new NoSuchShapeException();
    }
}
```

I programmati OO (object-oriented) potrebbero storcere il naso e dire che questo è un approccio procedurale – e avrebbero ragione. Ma considerate che cosa accadrebbe se a `Geometry` venisse aggiunta una funzione `perimeter()`. Le classi delle forme non cambierebbero! Non cambierebbero neppure le classi che dipendessero da quelle forme! D'altra parte, se aggiungessi una nuova forma, dovrei modificare tutte le funzioni di `Geometry` per gestirla. Di nuovo, rileggete la frase. Notate che le due condizioni sono diametricamente opposte?

Ora considerate la soluzione object-oriented presentata nel Listato 6.6. Qui il metodo `area()` è polimorfico. Non è più necessaria alcuna classe `Geometry`. Pertanto se aggiungo una nuova forma, non influenzano nessuna delle *funzioni* esistenti, ma se aggiungo una nuova funzione devo modificare tutte le *forme*! (Esistono altri modi per farlo, ben noti ai progettisti OO: per esempio le tecniche VISITOR o dual-dispatch. Ma queste tecniche comportano dei costi e in genere producono la struttura simile a quella di un programma procedurale.)

Listato 6.6 Forme polimorfiche.

```
public class Square implements Shape {  
    private Point topLeft;  
    private double side;  
    public double area() {  
        return side*side;  
    }  
}  
public class Rectangle implements Shape {  
    private Point topLeft;  
    private double height;  
    private double width;  
    public double area() {  
        return height * width;  
    }  
}  
public class Circle implements Shape {  
    private Point center;  
    private double radius;  
    public final double PI = 3.141592653589793;  
    public double area() {  
        return PI * radius * radius;  
    }  
}
```

Di nuovo, notiamo la natura complementare di queste due definizioni; sono praticamente l'una l'opposto dell'altra! Questo chiarisce la dicotomia di fondo esistente fra oggetti e strutture:

Il codice procedurale (che impiega strutture) facilita l'aggiunta di nuove funzioni senza modificare l'assetto esistente. Il codice OO, al contrario, facilita l'aggiunta di nuove classi senza modificare funzioni esistenti.

Vale anche il contrario:

Il codice procedurale complica l'aggiunta di nuove strutture, perché tutte le funzioni devono cambiare. Il codice OO complica l'aggiunta di nuove funzioni, perché tutte le classi devono cambiare.

Pertanto, ciò che è difficile per la programmazione OO è facile per le procedure e viceversa!

In ogni sistema complesso vi sono situazioni in cui vogliamo aggiungere nuovi tipi di dati invece che nuove funzioni. Per questi casi, gli oggetti e la programmazione OO sono più appropriati. D'altra parte, vi sono anche situazioni in cui abbiamo bisogno di aggiungere nuove funzioni e non nuovi tipi di dati. In tal caso il codice procedurale e le strutture risulteranno più appropriate.

I programmatore più esperti sanno che l'idea che tutto possa diventare un oggetto è un mito. Talvolta è davvero meglio usare delle semplici strutture e delle procedure.

La Legge di Demetra

Vi è un'euristica molto nota, chiamata *Legge di Demetra* (http://en.wikipedia.org/wiki/Law_of_Demeter), che dice che un modulo non dovrebbe conoscere i dettagli degli oggetti che manipola. Come abbiamo visto nei paragrafi precedenti, gli oggetti nascondono i loro dati ed espongono le operazioni. Significa che un oggetto non dovrebbe mostrare la propria struttura interna tramite i metodi di accesso, perché

ciò vorrebbe dire esporre, e non nascondere, la propria struttura interna.

Più precisamente, la Legge di Demetra dice che un metodo f di una classe C dovrebbe richiamare solo i metodi di:

- C ;
- un oggetto creato da f ;
- un oggetto passato come argomento a f ;
- un oggetto contenuto in una variabile di istanza di C .

Il metodo *non* dovrebbe richiamare metodi su oggetti restituiti da una delle funzioni consentite. In altre parole, “parlate con chi conoscete, non con gli sconosciuti”.

Il seguente codice (trovato da qualche parte nel framework di apache) viola la Legge di Demetra (fra le altre cose) perché richiama la funzione `getScratchDir()` sul valore restituito da `getOptions()` e poi richiama `getAbsolutePath()` sul valore restituito da `getScratchDir()`.

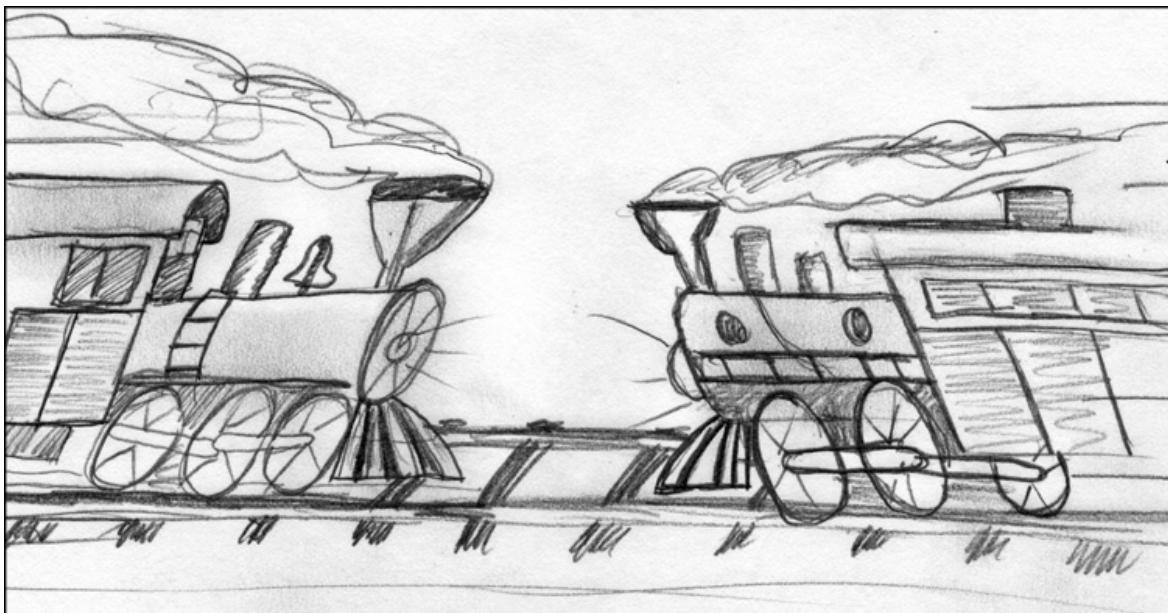
```
final String outputDir = ctxt.getOptions().getScratchDir().getAbsolutePath();
```

Relitti ferroviari

Questo genere di codice spesso viene chiamato relitto ferroviario (*train wreck*) perché ha l’aspetto di un ammasso di coppie di vagoni. Queste catene di chiamate sono generalmente considerate cattivi esempi di stile e dovrebbero essere evitate [G36]. Solitamente è meglio suddividerle come segue:

```
Options opts = ctxt.getOptions();
File scratchDir = opts.getScratchDir();

final String outputDir = scratchDir.getAbsolutePath();
```



Questi due frammenti di codice violano la Legge di Demetra?

Certamente il modulo che li contiene sa che l'oggetto `ctxt` contiene delle opzioni, che contengono una directory `scratch`, la quale ha un percorso assoluto. Si tratta di parecchie nozioni per una funzione. La funzione chiamante deve sapere come gestire parecchi tipi di oggetti.

Che questa sia una violazione della Legge di Demetra dipende dal fatto che `ctxt`, `Options` e `ScratchDir` siano oggetti o strutture. Se sono oggetti, allora la loro struttura interna dovrebbe essere nascosta e non esposta e pertanto la conoscenza del loro funzionamento rappresenta una chiara violazione della Legge di Demetra. Se invece `ctxt`, `Options` e `ScratchDir` sono solo strutture senza alcun comportamento, è naturale che espongano la loro struttura interna e pertanto non vi è alcuna violazione.

L'uso delle funzioni di accesso confonde la situazione. Se il codice fosse stato scritto come segue, probabilmente non ci saremmo preoccupati delle violazioni alla Legge di Demetra.

```
final String outputDir = ctxt.options.scratchDir.getAbsolutePath;
```

Tutto sarebbe meno confuso se le strutture avessero solo variabili pubbliche e nessuna funzione, e gli oggetti avessero variabili private e

funzioni pubbliche. Tuttavia, vi sono framework e standard (per esempio “beans”) che richiedono che anche le semplici strutture abbiano funzioni di accesso e mutatori.

Ibridi

Questa confusione talvolta genera esempi poco fortunati di strutture ibride: per metà oggetti e per metà strutture. Hanno funzioni che fanno qualcosa di significativo e hanno anche variabili pubbliche o metodi accessori e mutatori pubblici che, di fatto, rendono pubbliche le variabili private, inducendo anche altre funzioni esterne a usare tali variabili così come un programma procedurale userebbe una struttura [Refactoring].

Tali ibridi complicano l’aggiunta di nuove funzioni, ma anche l’aggiunta di nuove strutture. In pratica rappresentano il peggio dei due mondi. Evitate quindi di crearli. Sono chiari indicatori di un sistema raffazzonato, i cui autori non erano certi (o non sapevano) di doversi proteggere dalle funzioni o dai tipi.

Nascondere la struttura

E se `ctxt`, `options` e `scratchDir` fossero oggetti con un loro comportamento? Allora, dal momento che gli oggetti dovrebbero nascondere la propria struttura interna, non dovremmo potervi entrare. Come otterremmo, quindi, il percorso assoluto della directory scratch?

```
ctxt.getAbsolutePathOfScratchDirectoryOption();
```

o

```
ctx.getScratchDirectoryOption().getAbsolutePath()
```

La prima opzione potrebbe portare a un florilegio di metodi sull’oggetto `ctxt`. La seconda presume che `getScratchDirectoryOption()`

restituisca una struttura, non un oggetto. Nessuna delle due è una buona opzione.

Se `ctxt` è un oggetto, dovremmo potergli chiedere di *fare qualcosa*; non dovremmo chiedergli i suoi dettagli interni. Pertanto perché vogliamo il percorso assoluto della directory scratch? Che cosa pensiamo di farne? Considerate il seguente codice tratto dallo stesso modulo (un po' di righe più in basso):

```
String outFile = outputDir + "/" + className.replace('.', '/') + ".class";
FileOutputStream fout = new FileOutputStream(outFile);

BufferedOutputStream bos = new BufferedOutputStream(fout);
```

La miscela di svariati livelli di dettaglio [G34][G6] è problematica. Punti, barre, estensioni di file e oggetti `file` non dovrebbero trovarsi così mescolati, tanto meno con il codice che li racchiude. Ma anche ignorando questo fatto, vediamo che lo scopo di ottenere il percorso assoluto della directory scratch è quello di creare un file scratch con un determinato nome.

E se chiedessimo all'oggetto `ctxt` di farlo?

```
BufferedOutputStream bos = ctxt.createScratchFileStream(classFileName);
```

Questo sembra del tutto ragionevole per un oggetto! Questo consente a `ctxt` di nascondere il proprio funzionamento interno ed evita che la funzione corrente violi la Legge di Demetra operando all'interno di oggetti dei quali non dovrebbe sapere nulla.

Data Transfer Object

Sostanzialmente, una struttura è una classe con variabili pubbliche e senza funzioni, un cosiddetto Data Transfer Object o DTO. I DTO sono strutture molto utili, in particolare per comunicare con i database o per eseguire il parsing di messaggi dai socket e altro ancora. Spesso rappresentano la prima di una serie di passaggi di traduzione che

convertono i dati grezzi di un database in oggetti nel codice dell'applicazione.

Un po' più comune è la forma a “bean” presentata nel Listato 6.7. I bean hanno delle variabili private manipolate da metodi getter e setter. La quasi-incapsulazione dei bean sembra tranquillizzare i puristi OO, ma in genere non fornisce particolari vantaggi.

Listato 6.7 address.java.

```
public class Address {  
    private String street;  
    private String streetExtra;  
    private String city;  
    private String state;  
    private String zip;  
    public Address(String street, String streetExtra,  
        String city, String state, String zip) {  
        this.street = street;  
        this.streetExtra = streetExtra;  
        this.city = city;  
        this.state = state;  
        this.zip = zip;  
    }  
    public String getStreet() {  
        return street;  
    }  
    public String getStreetExtra() {  
        return streetExtra;  
    }  
    public String getCity() {  
        return city;  
    }  
    public String getState() {  
        return state;  
    }  
    public String getZip() {  
        return zip;  
    }  
}
```

Active Record

Gli Active Record rappresentano una forma particolare di DTO. Sono strutture con variabili pubbliche (ad accesso tramite bean); ma in genere sono dotate di metodi di navigazione come `save` e `find`. In genere

questi Active Record sono traduzioni dirette delle tabelle di un database o di altre fonti di dati.

Sfortunatamente spesso notiamo che gli sviluppatori tentano di trattare queste strutture come se fossero oggetti, inserendovi dei metodi di manipolazione. Questo però crea un ibrido fra una struttura e un oggetto.

La soluzione, naturalmente, consiste nel trattare l'Active Record come una struttura e nel creare oggetti distinti che contengono le regole operative e che nascondono i propri dati interni (che probabilmente sono solo istanze dell'Active Record).

Conclusioni

Gli oggetti espongono i comportamenti e nascondono i dati. Questo facilita l'aggiunta di nuovi tipi di oggetti senza costringere a modificare i comportamenti esistenti. Inoltre complica l'aggiunta di nuovi comportamenti agli oggetti esistenti. Le strutture espongono i dati e non hanno alcun comportamento significativo. Questo facilita l'aggiunta di nuovi comportamenti alle strutture esistenti, ma complica l'aggiunta di nuove strutture alle funzioni esistenti.

In ogni sistema talvolta ricerchiamo la flessibilità data dalla possibilità di aggiungere nuovi tipi di dati e così preferiamo usare degli oggetti per quella parte del sistema. Altre volte ricerchiamo la flessibilità data dalla possibilità di aggiungere nuovi comportamenti e così in tale parte del sistema preferiamo i tipi di dati e le procedure. I buoni sviluppatori di software comprendono questi dilemmi senza pregiudizi e scelgono l'approccio più adatto al compito che stanno svolgendo.

Bibliografia

- [Refactoring]: Martin Fowler et al., Refactoring: Improving the Design di Existing Code, Addison-Wesley, Boston 1999.

Gestione degli errori di Michael Feathers



Potrebbe sembrare strano trovare un capitolo dedicato alla gestione degli errori in un libro che parla della pulizia del codice. Ma la gestione degli errori è una delle tante cose con cui abbiamo a che fare quando programmiamo. Un input potrebbe essere malformato e i device possono guastarsi. In breve, le cose possono andare storte e quando lo fanno, come programmatore siamo responsabili del fatto che il nostro codice sappia cosa fare.

La connessione con la pulizia del codice, tuttavia, dovrebbe essere evidente. Molte basi di codice sono completamente dominate dalla gestione degli errori. Dico dominate, non perché si occupino solo della

gestione degli errori, ma perché è quasi impossibile scoprire che cosa fa il codice, a causa della quantità di parti dedicate alla gestione degli errori. La gestione degli errori è importante, *ma se offusca la logica*, è sbagliata.

In questo capitolo tratterò alcune tecniche e farò alcune considerazioni che potrete usare per scrivere codice che sia al contempo pulito e solido: codice che gestisce gli errori con grazia e stile.

Usate eccezioni al posto dei codici di return

Tanto tempo fa, molti linguaggi non avevano le eccezioni. In tali linguaggi le tecniche di gestione e report degli errori erano limitate. Potevate impostare un flag d'errore o restituire un codice d'errore che il chiamante avrebbe poi controllato. Il codice presentato nel Listato 7.1 illustra questi approcci.

Listato 7.1 DeviceController.java.

```
public class DeviceController {
    ...
    public void sendShutDown() {
        DeviceHandle handle = getHandle(DEV1);
        // Controlla lo stato del device
        if (handle != DeviceHandle.INVALID) {
            // Salva lo stato del device nel campo del record
            retrieveDeviceRecord(handle);
            // Se non è sospeso, chiudi
            if (record.getStatus() != DEVICE_SUSPENDED) {
                pauseDevice(handle);
                clearDeviceWorkQueue(handle);
                closeDevice(handle);
            } else {
                logger.log("Device suspended. Unable to shut down");
            }
        } else {
            logger.log("Invalid handle for: " + DEV1.toString());
        }
    }
    ...
}
```

Il problema di questi approcci è che congestionano il chiamante. Il chiamante deve occuparsi degli errori immediatamente dopo la chiamata. Sfortunatamente, è facile dimenticarsene. Per questo motivo è meglio lanciare un'eccezione quando si incontra un errore. Il codice chiamante è più pulito. La sua logica non è offuscata dalla gestione degli errori.

Il Listato 7.2 presenta il codice dopo aver scelto di lanciare delle eccezioni nei metodi che possono rilevare degli errori.

Listato 7.2 DeviceController.java (con le eccezioni).

```
public class DeviceController {  
    ...  
    public void sendShutDown() {  
        try {  
            tryToShutDown();  
        } catch (DeviceShutDownError e) {  
            logger.log(e);  
        }  
    }  
    private void tryToShutDown() throws DeviceShutDownError {  
        DeviceHandle handle = getHandle(DEV1);  
        DeviceRecord record = retrieveDeviceRecord(handle);  
        pauseDevice(handle);  
        clearDeviceWorkQueue(handle);  
        closeDevice(handle);  
    }  
    private DeviceHandle getHandle(DeviceID id) {  
        ...  
        throw new DeviceShutDownError("Invalid handle for: " + id.toString());  
        ...  
    }  
    ...  
}
```

Notate come tutto è molto più pulito. Non si tratta solo di estetica: il codice è migliore perché due concetti che prima erano mischiati, l'algoritmo di chiusura del device e la gestione degli errori, ora sono separati. Ora è possibile osservarli e comprenderli in modo indipendente.

Scrivere prima l'istruzione try-catch-finally

Uno degli aspetti più interessanti delle eccezioni è che esse definiscono un loro livello di visibilità (scope) all'interno del programma. Quando eseguite il codice contenuto nella porzione `try` di un'istruzione `try-catch-finally`, affermate che l'esecuzione può fermarsi in qualche punto e poi riprendere nel `catch`.

In un certo senso, i blocchi `try` sono come transazioni. Il `catch` deve lasciare il programma in uno stato coerente, qualsiasi cosa accada nel `try`. Per questo motivo è una buona abitudine iniziare con un'istruzione `try-catch-finally` quando si scrive del codice che potrebbe lanciare eccezioni. Questo aiuta a definire ciò che l'utente di tale codice dovrà aspettarsi, qualsiasi cosa vada storta nel codice che viene eseguito nel `try`.

Osserviamo un esempio. Dobbiamo scrivere del codice che acceda a un file e legga alcuni oggetti serializzati.

Iniziamo con uno unit test, il quale mostra che se il file non esiste otterremo un'eccezione:

```
@Test(expected = StorageException.class)
public void retrieveSectionShouldThrowOnInvalidFileName() {
    sectionStore.retrieveSection("invalid - file");
}
```

Il test ci porta a creare questo “moncone”:

```
public List<RecordedGrip> retrieveSection(String sectionName) {
    // return fittizio finché non avremo una vera implementazione
    return new ArrayList<RecordedGrip>();
}
```

Il nostro test fallisce perché non lancia un'eccezione. Poi, modifichiamo la nostra implementazione in modo che tenti di accedere a un file non valido. Questa operazione lancia un'eccezione:

```
public List<RecordedGrip> retrieveSection(String sectionName) {
    try {
        FileInputStream stream = new FileInputStream(sectionName)
    } catch (Exception e) {
        throw new StorageException("retrieval error", e);
    }
    return new ArrayList<RecordedGrip>();
}
```

Il nostro test ora passa perché abbiamo raccolto l’eccezione con `catch`. A questo punto, possiamo eseguire il refactoring. Possiamo restringere il tipo dell’eccezione che raccogliamo perché coincida con il tipo che viene lanciato dal costruttore di `FileInputStream`:

```
FileNotFoundException:  
  
public List<RecordedGrip> retrieveSection(String sectionName) {  
    try {  
        FileInputStream stream = new FileInputStream(sectionName);  
        stream.close();  
    } catch (FileNotFoundException e) {  
        throw new StorageException("retrieval error", e);  
    }  
    return new ArrayList<RecordedGrip>();  
}
```

Ora che abbiamo definito il livello di visibilità (scope) con una struttura `try-catch`, possiamo usare il *Test-Driven Development* (TDD) per realizzare il resto della logica, la quale verrà aggiunta fra la creazione del `FileInputStream` e la chiusura e potrà fingere che sia tutto a posto.

Provate a scrivere dei test che forzino le eccezioni e poi aggiungete dei comportamenti all’handler per soddisfare i vostri test. Questo vi condurrà a realizzare prima il livello di transazione del blocco `try` e vi aiuterà a mantenere la natura transazionale di tale livello di visibilità (scope).

Usate eccezioni non controllate

Il dibattito è ormai chiuso. Per anni i programmati Java hanno discusso sui vantaggi e svantaggi delle eccezioni controllate. Quando furono introdotte nella prima versione di Java, le eccezioni controllate sembrarono un’ottima idea. La signature di ogni metodo avrebbe elencato tutte le eccezioni che poteva passare al suo chiamante. Inoltre, queste eccezioni facevano parte del tipo del metodo. Il vostro codice

non poteva passare la compilazione se la signature non corrispondeva a quanto stabilito.

A quel tempo, pensavamo che le eccezioni controllate fossero un’ottima idea; e sì, in effetti avevano *qualche* vantaggio. Tuttavia, oggi è chiaro che non sono necessarie per produrre software solido. Il linguaggio C# non ha le eccezioni controllate e, nonostante validi tentativi, non le ha neanche il C++. E neanche Python o Ruby. Tuttavia è possibile scrivere ottimo software in tutti questi linguaggi. Per questo motivo, dobbiamo decidere, ma davvero, se vale la pena di pagare il “prezzo” delle eccezioni controllate.

Quale prezzo? Il prezzo delle eccezioni controllate è una violazione del Principio Open/Closed [Martin]. Se lanciate un’eccezione controllata da un metodo del vostro codice e la `catch` si trova tre livelli sopra, *dovrete dichiarare tale eccezione nella signature di ogni metodo situato fra qui e la catch*. Ciò significa che una modifica a un livello inferiore del software può costringere a modificare varie signature a più livelli soprastanti. I moduli modificati dovranno essere ricompilati e sottoposti a deploy, anche se in realtà nulla è cambiato.

Considerate la gerarchia delle chiamate di un sistema di grandi dimensioni. Le funzioni superiori richiamano le funzioni sottostanti, che a loro volta richiamano le funzioni sotto di esse, all’infinito. Ora immaginiamo che una delle funzioni di livello più basso venga modificata in modo da dover lanciare un’eccezione. Se tale eccezione è controllata, allora la signature della funzione dovrà aggiungere una clausola `throws`. Ma questo significa che ogni funzione che richiama la nostra funzione modificata, dovrà essere a sua volta modificata in modo che raccolga la nuova eccezione o per aggiungere alla sua signature la clausola `throws` appropriata. E questo all’infinito. Il risultato è una cascata di modifiche che devono risalire dai livelli più bassi del software a quelli più elevati! L’incapsulazione così si dissolve, perché

tutte le funzioni lungo il percorso di una `throw` devono conoscere i dettagli di questa eccezione di basso livello. Dato che lo scopo delle eccezioni è quello di consentire di gestire gli errori in un altro punto, è un vero peccato che le eccezioni controllate infrangano l'incapsulazione in questo modo.

Le eccezioni controllate possono, talvolta, essere utili quando si deve scrivere una libreria critica. Ma nel normale processo di sviluppo di un'applicazione i costi della dipendenza superano di gran lunga i vantaggi.

Fornite un contesto con le eccezioni

Ogni eccezione lanciata deve fornire un contesto sufficiente per determinare l'origine e la posizione di un errore. In Java, potete ottenere uno *stack trace* da qualsiasi eccezione; tuttavia, uno stack trace non vi dirà lo scopo dell'operazione fallita.

Create dei messaggi d'errore informativi da passare insieme alle eccezioni. Indicate l'operazione fallita e il tipo di fallimento. Se l'applicazione esegue un logging, passategli sufficienti informazioni per poter individuare l'errore.

Definite le classi per le eccezioni in termini di esigenze del chiamante

Vi sono molti modi per classificare gli errori. Possiamo classificarli in base all'origine: da quale componente provengono? O in base al tipo: problemi a un device, alla rete o errori di programmazione? Tuttavia, quando definiamo le classi per le eccezioni di un'applicazione, la nostra più grande preoccupazione dovrebbe essere *il modo in cui vengono raccolte*.

Osserviamo un cattivo esempio di classi per la classificazione delle eccezioni. Ecco un’istruzione `try-catch-finally` per una chiamata a una libreria esterna. Comprende tutte le eccezioni che le chiamate possono lanciare:

```
ACMEPort port = new ACMEPort(12);
try {
    port.open();
} catch (DeviceResponseException e) {
    reportPortError(e);
    logger.log("Device response exception", e);
} catch (ATM1212UnlockedException e) {
    reportPortError(e);
    logger.log("Unlock exception", e);
} catch (GMXError e) {
    reportPortError(e);
    logger.log("Device response exception");
} finally {
...
}
```

Tale istruzione contiene molta duplicazione e questo non è particolarmente sorprendente. Nella maggior parte delle situazioni di gestione delle eccezioni, il lavoro che facciamo è relativamente standard, indipendentemente dalla causa. Dobbiamo registrare l’errore e assicurarsi di poter procedere.

In questo caso, poiché sappiamo che il lavoro che stiamo svolgendo è praticamente lo stesso indipendentemente dall’eccezione, possiamo semplificare notevolmente il nostro codice incorporando l’API che richiamiamo e assicurandoci che restituisca un tipo di eccezione comune:

```
LocalPort port = new LocalPort(12);
try {
    port.open();
} catch (PortDeviceFailure e) {
    reportError(e);
    logger.log(e.getMessage(), e);
} finally {
...
}
```

La nostra classe `LocalPort` è solo un wrapper che raccoglie e traduce le eccezioni lanciate dalla classe `ACMEPort`:

```

public class LocalPort {
    private ACMEPort innerPort;
    public LocalPort(int portNumber) {
        innerPort = new ACMEPort(portNumber);
    }
    public void open() {
        try {
            innerPort.open();
        } catch (DeviceResponseException e) {
            throw new PortDeviceFailure(e);
        } catch (ATM1212UnlockedException e) {
            throw new PortDeviceFailure(e);
        } catch (GMXError e) {
            throw new PortDeviceFailure(e);
        }
    }
}

```

I wrapper come quello che abbiamo definito per `ACMEPort` possono essere molto utili. Infatti, il wrapping di API esterne è una delle migliori tecniche di programmazione. Quando si incorpora un'API esterna, si minimizzano le dipendenze da essa: in futuro potrete decidere di spostarla in un'altra libreria senza troppi problemi. Il wrapping facilita anche la simulazione di chiamate all'esterno in fase di test del codice.

Un ultimo vantaggio del wrapping è il fatto che non lega alle scelte progettuali delle API di un determinato produttore. Potete definire un'API con la quale vi trovate bene. Nell'esempio precedente, abbiamo definito un unico tipo di eccezione per un problema a una porta del device e abbiamo scoperto di poter scrivere codice molto più pulito.

Spesso un'unica classe per eccezioni è sufficiente per una determinata area del codice. Le informazioni inviate con l'eccezione permettono di distinguere gli errori. Usate classi differenti solo se esistono situazioni in cui vogliamo raccogliere una sola eccezione e consentire a un'altra di procedere.

Definite il flusso “normale”

Se seguite il consiglio presentato in precedenza, otterrete una buona separazione fra la logica operativa e la gestione degli errori. La parte principale del vostro codice inizierà a somigliare a un normale algoritmo.



Tuttavia, questo processo sospingerà il rilevamento degli errori ai margini del programma. Incorporerete le API esterne in modo da poter lanciare le vostre eccezioni e definirete un handler sopra il vostro codice, in modo da poter gestire ogni condizione di uscita. La maggior parte delle volte questo è un ottimo approccio, ma vi sono situazioni in cui non si vuole uscire dal programma.

Vediamo un esempio. Ecco del codice che somma le spese in un'applicazione di fatturazione:

```
try {
    MealExpenses expenses = expenseReportDAO.getMeals(employee.getID());
    m_total += expenses.getTotal();
} catch(MealExpensesNotFound e) {
    m_total += getMealPerDiem();
}
```

In questo caso, se il pasto viene consumato, diviene parte del totale. In caso contrario, il dipendente riceve una somma per tale giorno. L'eccezione offusca la logica. Non sarebbe meglio se non avessimo a che fare con il caso speciale? In tal caso, il nostro codice sarebbe molto più semplice:

```
MealExpenses expenses = expenseReportDAO.getMeals(employee.getID());  
m_total += expenses.getTotal();
```

Possiamo davvero renderlo così semplice? Certo che possiamo. Possiamo modificare `ExpenseReportDAO` in modo che restituiscia sempre un oggetto `MealExpense`. Se non vi sono spese per i pasti, restituisce un oggetto `MealExpense` che restituisce il totale *per giorno*:

```
public class PerDiemMealExpenses implements MealExpenses {  
    public int getTotal() {  
        // restituisci il totale per giorno  
    }  
}
```

Questo è chiamato SPECIAL CASE PATTERN [Fowler]. Create una classe o configurate un oggetto in modo che gestisca un caso speciale. In tal modo, il codice client non dovrà più occuparsi del comportamento che rappresenta solo un'eccezione. Tale comportamento viene incapsulato in un oggetto dedicato al caso speciale.

Non restituite null

Penso che qualsiasi discussione sulla gestione degli errori debba parlare anche di quelle cose che favoriscono l'insorgere di errori. Il primo della lista è il fatto di restituire `null`. Non so dire quante applicazioni ho incontrato in cui una riga sì e una no c'era un controllo di un `null`. Ecco un esempio:

```
public void registerItem(Item item) {  
    if (item != null) {  
        ItemRegistry registry = persistentStore.getItemRegistry();  
        if (registry != null) {  
            Item existing = registry.getItem(item.getID());  
            if (existing.getBillingPeriod().hasRetailOwner()) {
```

```
        existing.register(item);
    }
}
}
```

Se vi trovate a lavorare su un progetto contenente codice di questo tipo, la cosa potrebbe non sembrare così allarmante, ma invece questo è un bel problema! Quando restituiamo `null`, ci stiamo aumentando il carico di lavoro e stiamo creando problemi ai chiamanti. Basta dimenticarsi un controllo di un `null` per mandare fuori controllo un'applicazione.

Avete notato il fatto che manca il controllo del `null` nella seconda riga dell’istruzione `if` annidata? Che cosa sarebbe accaduto, a runtime, se `persistentStore` fosse `null`? Avremmo avuto una `NullPointerException` a runtime e forse (speriamo) qualcuno avrebbe raccolto la `NullPointerException` al livello superiore. Non va bene. Che cosa si dovrebbe fare, esattamente, in risposta a una `NullPointerException` lanciata dalle profondità dell’applicazione?

È facile dire che il problema del codice precedente è che manca il controllo di un `null`, ma in realtà il problema è che ce ne sono troppi. Se siete tentati di far restituire `null` a un metodo, piuttosto lanciate un'eccezione o restituite un oggetto Special Case. Se richiamate un metodo che restituisce `null` da un'API esterna, considerate l'idea di eseguire il wrapping di tale metodo in un metodo che lanci un'eccezione o restituisca un oggetto Special Case.

In molti casi, gli oggetti Special Case sono la soluzione più semplice. Immaginate di avere il seguente codice:

```
List<Employee> employees = getEmployees();
if (employees != null) {
    for(Employee e : employees) {
        totalPay += e.getPay();
    }
}
```

Al momento, `getEmployees` può restituire `null`, ma è obbligata a farlo? Se modifichiamo `getEmployee` in modo che restituisca una lista vuota, possiamo ripulire il codice:

```
List<Employee> employees = getEmployees();  
for(Employee e : employees) {  
    totalPay += e.getPay();  
}
```

Fortunatamente, Java ha un `Collections.emptyList()` che restituisce una lista immutabile che possiamo usare per questo scopo:

```
public List<Employee> getEmployees() {  
    if(... non vi sono dipendenti ...)  
        return Collections.emptyList();  
}
```

Se programmate in questo modo, ridurrete le probabilità che si verifichino `NullPointerExceptions` e il vostro codice sarà più pulito.

Non passate null

Far restituire `null` ai metodi è “male”, ma passare `null` ai metodi è anche peggio. A meno che utilizziate un’API che si aspetta che passiate un `null`, dovreste sempre evitare di passare un `null` nel vostro codice.

Osserviamo un esempio per capire il perché. Ecco un semplice metodo che calcola una metrica per due punti:

```
public class MetricsCalculator  
{  
    public double xProjection(Point p1, Point p2) {  
        return (p2.x - p1.x) * 1.5;  
    }  
    ...  
}
```

Che cosa accadrebbe se qualcuno dovesse passare `null` come argomento?

```
calculator.xProjection(null, new Point(12, 13));
```

Naturalmente otterremmo una `NullPointerException`.

Come rimediare? Potremmo creare un nuovo tipo di eccezione e lanciarla:

```
public class MetricsCalculator
{
    public double xProjection(Point p1, Point p2) {
        if (p1 == null || p2 == null) {
            throw new InvalidArgumentException(
                "Invalid argument for MetricsCalculator.xProjection");
        }
        return (p2.x - p1.x) * 1.5;
    }
}
```

Così va meglio? Sarà forse un po' meglio di un'eccezione per puntatore null, ma ricordate, dobbiamo definire un handler per `InvalidArgumentException`. E come dovrebbe essere tale handler? Esiste una soluzione migliore?

Vi è un'altra alternativa. Potremmo usare un insieme di `assert`:

```
public class MetricsCalculator
{
    public double xProjection(Point p1, Point p2) {
        assert p1 != null : "p1 should not be null";
        assert p2 != null : "p2 should not be null";
        return (p2.x - p1.x) * 1.5;
    }
}
```

Va bene come documentazione, ma non risolve il problema. Se qualcuno passa un `null`, avremo sempre un errore a runtime.

Nella maggior parte dei linguaggi di programmazione non vi è alcun buon modo per gestire un `null` passato accidentalmente da un chiamante. L'idea è che è sempre meglio non passare un `null`. Se lo fate, dovete anche specificare chiaramente che un `null` in un elenco di argomenti indica la presenza di un problema.

Conclusioni

Il codice pulito è facilmente comprensibile, ma deve anche essere solido. Questi obiettivi non sono in conflitto. Possiamo scrivere codice

pulito e solido, se consideriamo la gestione degli errori come un ambito distinto, qualcosa che si possa osservare in modo indipendente rispetto al flusso principale del programma. Se saremo in grado di farlo, potremo operare sui due ambiti in modo indipendente e migliorare notevolmente la manutenibilità del codice.

Bibliografia

- [Martin]: Robert C. Martin, Agile Software Development: Principles, Patterns, and Practices, Prentice Hall, Upper Saddle River, New Jersey 2002.

Capitolo 8

Delimitazioni *di James Grenning*



Raramente abbiamo il controllo di tutto il software dei nostri sistemi. Talvolta acquistiamo package esterni o usiamo del codice open-source. Altre volte dipendiamo dai team interni, che realizzano componenti o

subsistemi per noi. In qualche modo dobbiamo integrare con pulizia questo codice esterno con il nostro. In questo capitolo parleremo delle tecniche che consentono di garantire il rispetto delle delimitazioni del nostro software.

Usare codice esterno

Esiste una tensione naturale fra il fornitore e l'utilizzatore di un'interfaccia. I fornitori di package e framework puntano a un'ampia applicabilità, perché possa operare in molti ambienti ed essere adatta a un'ampia utenza. Gli utilizzatori, d'altra parte, vogliono un'interfaccia che sia calibrata sulle loro specifiche esigenze. Questa tensione può provocare problemi alla linea di delimitazione dei nostri sistemi.

Osserviamo per esempio `java.util.Map`. Come potete vedere esaminando la Figura 8.1, le `Map` hanno un'interfaccia molto ampia, ricca di funzionalità. Certamente questa potenza e flessibilità è utile, ma può anche essere un vincolo. Per esempio, la nostra applicazione potrebbe costruire e utilizzare una `Map`. La nostra intenzione potrebbe essere che nessuno di coloro che riceveranno la nostra `Map` possa cancellare alcunché dalla mappa. Ma proprio in cima all'elenco troviamo il metodo `clear()`. Chiunque utilizzi una `Map` ha anche la possibilità di cancellarla. O magari la convenzione del nostro sistema è che solo determinati tipi di oggetti possano essere memorizzati nella `Map`, ma le `Map` non limitano con la precisione di cui abbiamo bisogno i tipi di oggetti che possono contenere. Un utilizzatore può inserire in una `Map` oggetti di qualsiasi tipo.

- `clear() void - Map`
- `containsKey(Object key) boolean - Map`
- `containsValue(Object value) boolean - Map`
- `entrySet() Set - Map`
- `equals(Object o) boolean - Map`
- `get(Object key) Object - Map`
- `getClass() Class<? extends Object> - Object`
- `hashCode() int - Map`
- `isEmpty() boolean - Map`
- `keySet() Set - Map`
- `notify() void - Object`
- `notifyAll() void - Object`
- `put(Object key, Object value) Object - Map`
- `putAll(Map t) void - Map`
- `remove(Object key) Object - Map`
- `size() int - Map`
- `toString() String - Object`
- `values() Collection - Map`
- `wait() void - Object`
- `wait(long timeout) void - Object`
- `wait(long timeout, int nanos) void - Object`

Figura 8.1 I metodi di `Map`.

Se la nostra applicazione deve creare una `Map` di `Sensor`, potreste creare i sensori nel seguente modo:

```
Map sensors = new HashMap();
```

Poi, quando una qualche altra parte del codice dovrà accedere a un sensore, troverete questo codice:

```
Sensor s = (Sensor)sensors.get(sensorId);
```

Non lo troveremo una volta sola, ma più e più volte in tutto il codice. Il client di questo codice avrà la responsabilità di estrarre un `Object` dalla `Map` e poi di convertirlo nel tipo appropriato. Il tutto funziona, ma il codice è tutt'altro che “pulito”. Inoltre, questo codice non racconta bene la sua storia, anche se potrebbe. La leggibilità di questo codice può essere notevolmente migliorata usando i *generici*, come potete vedere qui di seguito:

```
Map<Sensor> sensors = new HashMap<Sensor>();
...
Sensor s = sensors.get(sensorId);
```

Tuttavia, questo non risolve il problema: `Map<Sensor>` fornisce più funzionalità di quelle che dobbiamo o vogliamo usare.

Usare tranquillamente un'istanza di `Map<Sensor>` in tutto il sistema significherebbe dover intervenire in moltissimi punti qualora l'interfaccia di `Map` dovesse cambiare. Potreste pensare che tale cambiamento sia improbabile, ma mi ricordo che è cambiata quando in Java 5 è stato aggiunto il supporto dei generici. In effetti, abbiamo visto che i sistemi non potevano usare i generici data l'enorme quantità di modifiche necessarie per poter utilizzare appieno `Map`.

Un modo più pulito per utilizzare `Map` potrebbe essere il seguente. Nessun utilizzatore di `sensors` avrebbe alcun problema derivante dall'uso o meno dei generici. Tale scelta è diventata (e sempre dovrebbe essere) un dettaglio implementativo.

```
public class Sensors {
    private Map sensors = new HashMap();
    public Sensor getById(String id) {
        return (Sensor) sensors.get(id);
    }
    //codice
}
```

L'interfaccia nella linea di delimitazione (`Map`) è nascosta. Potrà così evolvere, avendo un impatto minimo sul resto dell'applicazione. L'uso dei generici non è più un problema perché la conversione e la gestione dei tipi vengono svolte nella classe `Sensors`.

Questa interfaccia è anche calibrata e vincolata in modo da rispondere alle esigenze dell'applicazione. In tal modo il codice è più facile da comprendere e più difficile da sovvertire. La classe `Sensors` potrà così applicare le sue regole progettuali e operative.

Non vi sto dicendo che ogni uso di `Map` debba essere encapsulato in questa forma. Piuttosto, vi suggerisco di non passare delle `Map` (o

qualsiasi altra interfaccia operante in una delimitazione) qua e là per il sistema. Se usate un’interfaccia con delimitazione come `Map`, mantenetela all’interno della classe o in prossimità della famiglia di classi in cui viene usata. Evitate di farla restituire da (o di accettarla come argomento per) API pubbliche.

Esplorazione delle delimitazioni

Il codice di provenienza esterna ci aiuta a ottenere più funzionalità in minor tempo. Dove iniziamo quando vogliamo utilizzare un pacchetto esterno? Non è compito nostro sottoporre a test il codice esterno, ma è sicuramente nel nostro interesse scrivere dei test per il codice esterno che usiamo.

Supponete che non sia perfettamente chiaro come usare la libreria esterna che ci interessa. Potremmo dedicare un giorno o due (o più) alla lettura della documentazione e a decidere come intendiamo usarla. Poi potremmo scrivere il nostro codice in modo che richiami il codice esterno e scoprire se fa quello che pensiamo. Molto probabilmente ci troveremo impegnati in lunghe sessioni di debugging nel tentativo di scoprire se i bug che si verificano sono attribuibili al nostro codice o al loro.

Studiare il codice esterno è difficile. Anche integrare il codice esterno è difficile. Fare entrambe le cose contemporaneamente è doppiamente difficile. E se scegliessimo un altro approccio? Invece di sperimentare e provare le nuove cose nel nostro codice di produzione, potremmo scrivere dei test per esplorare la nostra conoscenza del codice esterno. Jim Newkirk chiama tali test *learning test* ([BeckTDD], pp. 136–137).

Nei learning test richiamiamo l’API esterna, come ci aspettiamo di usarla nella nostra applicazione. Sostanzialmente svolgiamo degli

esperimenti controllati per verificare la nostra conoscenza di tale API. I test si concentrano su ciò che vogliamo usare dell'API.

Imparare a usare log4j

Immaginiamo di voler usare il package apache `log4j` invece del nostro logger. Lo scarichiamo e apriamo la pagina introduttiva della documentazione. Senza leggere troppo, scriviamo il nostro primo caso di test, aspettandoci di ottenere un “hello” sulla console.

```
@Test  
public void testLogCreate() {  
    Logger logger = Logger.getLogger("MyLogger");  
    logger.info("hello");  
}
```

All'esecuzione, il logger produce un errore che dice che abbiamo bisogno di un “qualcosa” chiamato `Appender`. Leggiamo un altro po' e scopriamo che vi è un `ConsoleAppender`. Pertanto creiamo un `ConsoleAppender` e cerchiamo di capire se abbiamo individuato i segreti del logging su console.

```
@Test  
public void testLogAddAppender() {  
    Logger logger = Logger.getLogger("MyLogger");  
    ConsoleAppender appender = new ConsoleAppender();  
    logger.addAppender(appender);  
    logger.info("hello");  
}
```

Questa volta scopriamo che l'`Appender` non ha uno stream di output. Che strano: sembra solo logico che debba averne uno. Dopo un piccolo aiuto da parte di Google, proviamo il codice seguente:

```
@Test  
public void testLogAddAppender() {  
    Logger logger = Logger.getLogger("MyLogger");  
    logger.removeAllAppenders();  
    logger.addAppender(new ConsoleAppender(  
        new PatternLayout("%p %t %m%n"),  
        ConsoleAppender.SYSTEM_OUT));  
    logger.info("hello");  
}
```

Funziona; alla console compare un messaggio log contenente il nostro “hello”! Però ci sembra strano il fatto di dover dire al `ConsoleAppender` che deve scrivere sulla console.

Interessante: se eliminiamo l’argomento `ConsoleAppender.SystemOut`, vediamo che il nostro “hello” viene comunque inviato. Ma se togliamo il `PatternLayout`, ancora una volta il codice si lamenta della mancanza di uno stream di output. Un comportamento davvero strano.

Leggendo con più attenzione la documentazione, vediamo che il costruttore di default di `ConsoleAppender` è “non configurato”, il che non sembra né logico né utile. Sembra quasi un bug o quanto meno un’incoerenza di log4j.

Un altro po’ di Google, di letture e di test e alla fine otteniamo il Listato 8.1. Abbiamo scoperto molto meglio come funziona log4j e abbiamo distillato questa nostra conoscenza in un insieme di semplici unit test.

Listato 8.1 LogTest.java.

```
public class LogTest {
    private Logger logger;
    @Before
    public void initialize() {
        logger = Logger.getLogger("logger");
        logger.removeAllAppenders();
        Logger.getRootLogger().removeAllAppenders();
    }
    @Test
    public void basicLogger() {
        BasicConfigurator.configure();
        logger.info("basicLogger");
    }
    @Test
    public void addAppenderWithStream() {
        logger.addAppender(new ConsoleAppender(
            new PatternLayout("%p %t %m%n"),
            ConsoleAppender.SYSTEM_OUT));
        logger.info("addAppenderWithStream");
    }
    @Test
    public void addAppenderWithoutStream() {
        logger.addAppender(new ConsoleAppender(
            new PatternLayout("%p %t %m%n")));
        logger.info("addAppenderWithoutStream");
    }
}
```

Ora sappiamo come ottenere un semplice *console logger* ben inizializzato e possiamo incapsulare tale conoscenza nella nostra classe logger, in modo che il resto della nostra applicazione sia isolata dall’interfaccia di delimitazione di `log4j`.

I learning test sono più che gratis

I learning test non costano nulla. Comunque dovremmo imparare a usare l’API e scrivere questi test è un modo facile e isolato per acquisire tale conoscenza. I learning test sono esperimenti precisi, utili per espandere la nostra conoscenza.

Non solo i learning test sono gratis, ma restituiscono l’investimento di tempo con gli interessi. Quando vi sono nuove release di un pacchetto esterno, eseguo i learning test per scoprire se vi sono differenze di comportamento.

I learning test verificano che i package esterni che usiamo funzionino come ci aspettiamo. Una volta integrati, non vi sono garanzie che il codice esterno rimanga compatibile con le nostre esigenze. Gli autori potrebbero essere stati indotti a modificare il loro codice per rispondere a determinate esigenze. Possono aver corretto dei bug e aggiunto nuove funzionalità. A ogni release arrivano anche nuovi rischi. Se il package esterno cambia in qualche modo incompatibile con i nostri test, lo scopriremo subito.

Che abbiate bisogno o meno delle conoscenze offerte dai learning test, una delimitazione netta dovrebbe essere supportata da un insieme di test rivolti all’esterno, che mettano alla prova l’interfaccia come farà il codice di produzione. Senza questi test della delimitazione, utili per facilitare la migrazione, potremmo essere tentati di rimanere con la vecchia versione più a lungo di quanto dovremmo.

Usare codice che non esiste ancora

Vi è anche un altro tipo di delimitazione: quella che separa il noto dall'ignoto. Spesso vi sono punti nel codice in cui la nostra conoscenza sembra piombare in un baratro. Talvolta ciò che si trova dall'altro lato della delimitazione è inconoscibile (quanto meno al momento). Talvolta scegliamo di non guardare al di là della delimitazione.

Alcuni anni fa facevo parte di un team di sviluppo di un software per un sistema di comunicazione via radio. Vi era un sottosistema, il “Transmitter”, del quale conoscevamo molto poco e i responsabili del sottosistema non erano ancora arrivati al punto di definire la loro interfaccia. Non volevamo rimanere bloccati, e così abbiamo cominciato a lavorare mantenendoci ben lontani dalle parti di codice ancora ignote.

Avevamo un'idea piuttosto chiara di dove finiva il nostro mondo e dove cominciava il nuovo. Mentre lavoravamo, talvolta ci siamo scontrati contro questa delimitazione. Sebbene la nebbia offuscasse il nostro sguardo oltre la delimitazione, il nostro lavoro ci indicava come avremmo voluto l'interfaccia di tale delimitazione. Volevamo dire al trasmettitore qualcosa nel seguente modo:

Regola il trasmettitore sulla frequenza indicata ed emetti una rappresentazione analogica dei dati in arrivo da questo stream.

Non avevamo alcuna idea di come sarebbe stato realizzato, perché l'API non era ancora stata progettata. Pertanto abbiamo deciso di rimandare i dettagli a un secondo tempo.

Per evitare di bloccarci, abbiamo definito una nostra interfaccia. Giustamente l'abbiamo chiamata `Transmitter`. Le abbiamo dato un metodo chiamato `transmit` che accettava una frequenza e un flusso di dati. Questa era l'interfaccia che volevamo avere.

Un vantaggio dello scrivere l’interfaccia come si desidera è che è sotto il nostro controllo. Questo aiuta a mantenere leggibile il codice client e a concentrarsi su ciò che si sta tentando di ottenere.

Nella Figura 8.2, potete vedere che abbiamo isolato le classi `CommunicationsController` rispetto all’API del trasmettitore (che era al di là del nostro controllo e ancora indefinita). Usando l’interfaccia specifica della nostra applicazione, abbiamo curato la pulizia e l’espressività del codice di `CommunicationsController`. Una volta che è stata definita l’API del trasmettitore, abbiamo scritto il `TransmitterAdapter` per colmare il gap. L’ADAPTER (vedere il pattern Adapter in [GOF]) incapsulava l’interazione con l’API e forniva un unico punto sul quale intervenire per seguire l’evoluzione dell’API.

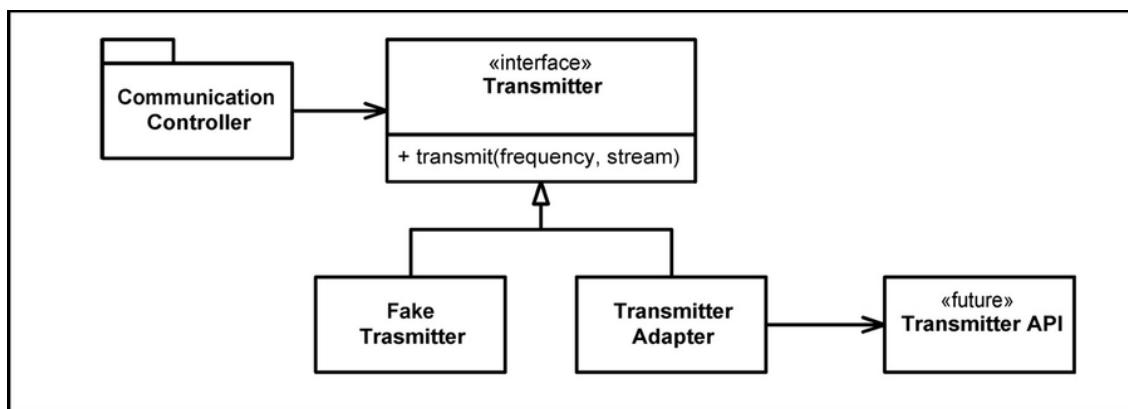


Figura 8.2 Predizione del trasmettitore.

Questa struttura del codice ci ha anche dato un comodo banco di test (maggiori informazioni si trovano in [WELC]). Utilizzando un apposito `FakeTransmitter`, potevamo mettere alla prova le classi `CommunicationsController`. Una volta pronta la `TransmitterAPI` potevamo anche creare dei test della delimitazione, per assicurarci che funzionasse correttamente.

Delimitazioni chiare

Sulla linea di delimitazione accadono cose interessanti. I cambiamenti sono una di queste cose. Un buon progetto software si adatta ai cambiamenti senza grossi investimenti e rielaborazioni. Quando usiamo del codice che è al di là del nostro controllo, occorre applicare particolare cura per proteggere il nostro investimento e assicurarci che i successivi interventi non siano troppo costosi.

Il codice in prossimità della delimitazione ha bisogno di una separazione netta e di test che definiscano le aspettative. Dovremmo evitare di far conoscere al nostro codice troppi dettagli del software esterno. È meglio dipendere da qualcosa che è sotto il nostro controllo che da qualcosa che non possiamo controllare.

Gestiamo le delimitazioni con il software esterno facendo in modo di avere poche posizioni nel codice che vi fanno riferimento. Possiamo incorporarle come abbiamo fatto con `Map` o possiamo usare un ADAPTER per eseguire la conversione dalla nostra interfaccia a quella che ci verrà fornita. In ogni caso il nostro codice comunicherà molto meglio, promuovendo un utilizzo coerente attraverso la delimitazione e avrà meno punti di manutenzione quando il codice esterno dovesse cambiare.

Bibliografia

- [BeckTDD]: Kent Beck, Test Driven Development, Addison-Wesley, Boston 2003.
- [GOF]: Gamma et al., Design Patterns: Elements di Reusable Object Oriented Software, Addison-Wesley, Boston 1996.
- [WELC]: Working Effectively with Legacy Code, Addison-Wesley, Boston 2004.

Capitolo 9

Unit test



La nostra professione ha fatto passi da gigante negli ultimi dieci anni. Nel 1997 nessuno aveva sentito parlare di sviluppo TDD (Test-Driven Development). Per la maggior parte di noi, gli unit test erano brevi frammenti di codice usa-e-getta che scrivevamo per assicurarci che i nostri programmi “funzionavano”. Scrivevamo faticosamente le nostre classi e i nostri metodi e poi escogitavamo del codice ad hoc per

sottoporli a test. In genere questo prevedeva l'uso di qualche semplice programma driver che ci consentisse di interagire manualmente con il programma che avevamo scritto.

Mi ricordo di aver scritto un programma C++ per un sistema embedded real-time a metà degli anni Novanta. Il programma era un semplice timer con la seguente signature:

```
void Timer::ScheduleCommand(Command* theCommand, int milliseconds)
```

L'idea era semplice; il metodo `execute` di `Command` sarebbe stato eseguito in un nuovo thread dopo il numero specificato di millisecondi. Il problema consisteva nel sottoporlo a test.

Ho messo insieme un semplice programma driver che restava in ascolto della tastiera. Ogni volta che veniva premuto un tasto, attivava un comando che avrebbe digitato il relativo carattere cinque secondi dopo. Pigiavo i tasti al ritmo di una melodia e attendevo che venissero riprodotti sullo schermo cinque secondi dopo.

I ... want-a-girl ... just ... like-the-girl-who-marr ... ied ... dear ... old ... dad.

Canticchiaivo e nel frattempo pigiavo il tasto “.” e poi continuavo a canticchiare mentre i punti comparivano sullo schermo.

Quello era il mio test! Una volta che avevo visto che funzionava e dopo averlo dimostrato ai colleghi, buttavo via il test.

Come ho detto, la nostra professione ha fatto passi da gigante. Al giorno d'oggi scriverei un test che mi assicuri il corretto funzionamento di ogni singolo dettaglio di tale codice. Isolerei il mio codice dal sistema operativo invece di richiamare semplicemente le funzioni standard di temporizzazione. Inserirei tali funzioni di temporizzazione in una simulazione in modo da poter avere il controllo assoluto sul tempo. Programmerei dei comandi per attivare dei flag booleani e poi lascerei andare il tempo, osservando tali flag e assicurandomi che vadano da `false` a `true` mentre cambio il tempo.

Una volta passati questi test, mi assicurerrei che tali test siano comodi da usare per chiunque abbia bisogno di lavorare sul codice. Mi assicurerrei che i test e il codice vengano registrati insieme nello stesso package di codice sorgente.

Sì, abbiamo fatto molta strada; ma altrettanta ne dobbiamo fare. I movimenti Agile e TDD (Test-Driven Development) hanno incoraggiato molti programmatore a scrivere unit test automatici e ogni giorno fanno nuovi proseliti. Ma nella folle corsa ad aggiungere nuovi test alla nostra disciplina, molti programmatore hanno perso per strada alcune delle motivazioni più specifiche e importanti della scrittura di buoni test.

Le tre leggi dello sviluppo TDD (Test-Driven Development)

Al giorno d'oggi tutti sanno che lo sviluppo TDD ci chiede di scrivere per primi gli unit test, prima ancora di scrivere il codice effettivo del software. Ma tale regola è solo la punta dell'iceberg. Considerate le tre leggi seguenti (Robert C. Martin, *Professionalism and Test-Driven Development*, Object Mentor, IEEE Software, May/June 2007 (Vol. 24,

No. 3) pp. 32–36, <http://doi.ieee.org/10.1109/MS.2007.85>).

- *Prima legge* – Non scriverai il codice di produzione finché non avrai scritto uno unit test di stress.
- *Seconda legge* – Non procederai nello unit test più di quanto sia sufficiente per ottenere un fallimento. E la mancata compilazione è un fallimento.
- *Terza legge* – Non scriverai più codice di produzione di quello che è sufficiente a passare l'attuale test di stress.

La lettura di queste tre leggi vi indurrà pensieri che vi terranno impegnati forse una trentina di secondi. I test e il codice di produzione vanno scritti *insieme*, dove i test vanno scritti appena prima del codice di produzione.

Se lavoriamo in questo modo, ci troveremo a scrivere decine di test ogni giorno, centinaia di test ogni mese e migliaia di test ogni anno. Se lavoriamo in questo modo, tali test copriranno praticamente tutto il codice di produzione. La pura massa di test, che può benissimo rivaleggiare con le dimensioni del codice di produzione, può rappresentare un notevole problema gestionale.

Curate la pulizia dei test

Alcuni anni fa mi è stato chiesto di occuparmi di un team: avevano deciso esplicitamente che il loro codice di test *non doveva* essere sottoposto agli stessi standard di qualità del loro codice di produzione. Si sono dati l'un l'altro la licenza di infrangere le regole nei loro unit test. Il loro motto era: “Quick & Dirty”. Le loro variabili *non dovevano* ricevere necessariamente un buon nome, le loro funzioni di test non dovevano necessariamente essere brevi e descrittive. Il loro codice di test non doveva necessariamente essere ben progettato e accuratamente partizionato. Bastava solo che il codice di test funzionasse e che si occupasse del codice di produzione.

Alcuni di voi potrebbero anche simpatizzare con tale decisione. Forse, magari molto tempo fa, anche voi avete scritto dei test come quello che ho scritto io per la classe `Timer`. È un bel passo quello che separa quel tipo di test usa-e-getta da un intero pacchetto di unit test automatici. Pertanto, come il team del quale mi sono occupato, potreste decidere che dei test raffazzonati siano sempre meglio dell'assenza totale di test.

Ma questo team non si è reso conto del fatto che avere dei test scadenti è equivalente a non avere test, o anche peggio. Il problema è che i test devono cambiare insieme al codice di produzione. Più scadenti sono i test, più sarà difficile modificarli. Più è ingarbugliato il codice di test, più vi ritroverete a dedicare più tempo a realizzare i nuovi test che a scrivere il nuovo codice di produzione. Modificando il codice di produzione, i vecchi test inizieranno a non funzionare più e il groviglio del codice di test complicherà ogni tentativo di fargli superare i test. Pertanto i test inizieranno a essere visti, sempre di più, come un peso.

Di release in release il costo di manutenzione del pacchetto dei test del mio team cresceva. Alla fine divenne il principale motivo di lamentele fra gli sviluppatori. Quando i dirigenti domandarono perché le loro stime stavano lievitando così tanto, gli sviluppatori incolparono i test. Alla fine furono costretti a sbarazzarsi del tutto del pacchetto dei test.

Ma senza un pacchetto dei test perdettero la capacità di assicurarsi che le modifiche alla loro base di codice funzionassero come previsto. Senza un pacchetto dei test non potevano garantire che le modifiche a una parte del loro sistema non pregiudicasse il funzionamento di altre parti. Pertanto il loro tasso di difettosità iniziò a crescere. Mentre cresceva il numero di difetti inattesi, iniziarono a temere ogni singola modifica. Smisero così di raffinare il codice di produzione, perché temevano che le modifiche facessero più male che bene. Il loro codice di produzione iniziò a sgretolarsi. Alla fine non avevano più alcun test, avevano in mano del codice di produzione ingarbugliato e zeppo di bug, i loro clienti erano frustrati e con la sensazione che l'impegno rappresentato dai test li aveva “uccisi”.

In un certo senso avevano ragione. Il loro impegno con i test li *aveva uccisi*, ma la colpa non era dei test, bensì della loro decisione di

impiegare test malformati. Se avessero curato la pulizia dei loro test, il loro impegno con i test sarebbe stato utile, anziché dannoso. Mi sento di dirlo con una certa sicurezza, in quanto ho partecipato e curato le attività di molti team che hanno usato con profitto unit test puliti.

La morale della storia è semplice: *il codice di test è importante tanto quanto il codice di produzione*. Non è un cittadino di Serie B. Richiede riflessioni, cura nella progettazione e attenzione. Deve essere mantenuto altrettanto pulito del codice di produzione.

I test garantiscono le “...bilità”

Se non curerete la pulizia dei vostri test, li perderete. E con loro, perderete anche lo strumento che garantisce la flessibilità del codice di produzione. Proprio così: sono gli *unit test* che garantiscono la flessibilità, manutenibilità e riutilizzabilità del nostro codice. Il motivo è semplice: se avete i test, non avrete mai paura di modificare il codice! Senza i test ogni modifica potrebbe introdurre un bug. Per quanto sia flessibile la vostra architettura, per quanto sia ben partizionata la sua struttura, senza test sarete riluttanti ogni volta che dovete apportare una modifica, per il timore di introdurre bug che non verranno rilevati.

Con i test tale paura in gran parte svanisce. Maggiore è la copertura dei test, minore è la paura. Potete apportare ogni modifica anche se il codice ha un’architettura perfettibile e una struttura intricata e fumosa. In realtà, *potete sfruttare* i test anche *per migliorare* tale architettura e tale struttura senza alcun timore!

Pertanto un pacchetto di unit test che riguardi l’intero codice di produzione è la chiave per avere una struttura e un’architettura che siano il più possibile pulite. I test garantiscono tutte le “...bilità”, perché garantiscono ogni possibile intervento.

Pertanto se i vostri test sono scadenti, ciò minerà la vostra capacità di modificare il codice e inizierete a perdere la capacità di migliorare la sua struttura. Peggiori sono i vostri test, peggiore diverrà il vostro codice. Alla fine perderete i test e vedrete sgretolarsi anche il codice.

Test “puliti”

Che cosa rende “pulito” un test? Tre aspetti. La leggibilità, la leggibilità e anche la leggibilità. La leggibilità è forse ancora più importante negli unit test che nel codice di produzione. Che cosa rende leggibili i test? Le stesse caratteristiche che rendono leggibile tutto il codice: la chiarezza, la semplicità e l'espressività. In un test volete dire il più possibile con il minor numero possibile di espressioni.

Considerate il codice tratto da FitNesse presentato nel Listato 9.1. Questi tre test sono difficili da comprendere e certamente possono essere migliorati. Innanzitutto, vi è una grande quantità di codice duplicato [G5] nelle continue chiamate ad `addPage` e `assertSubString`. Ancora più importante: questo codice è ricco di dettagli che interferiscono con l'espressività del test.

Listato 9.1 SerializedPageResponderTest.java.

```
public void testGetPageHierachyAsXml() throws Exception
{
    crawler.addPage(root, PathParser.parse("PageOne"));
    crawler.addPage(root, PathParser.parse("PageOne.ChildOne"));
    crawler.addPage(root, PathParser.parse("PageTwo"));
    request.setResource("root");
    request.addInput("type", "pages");
    Responder responder = new SerializedPageResponder();
    SimpleResponse response =
        (SimpleResponse) responder.makeResponse(
            new FitNesseContext(root), request);
    String xml = response.getContent();
    assertEquals("text/xml", response.getContentType());
    assertSubString("<name>PageOne</name>", xml);
    assertSubString("<name>PageTwo</name>", xml);
    assertSubString("<name>ChildOne</name>", xml);
}
public void testGetPageHierachyAsXmlDoesntContainSymbolicLinks()
    throws Exception
{
```

```

WikiPage pageOne = crawler.addPage(root, PathParser.parse("PageOne"));
crawler.addPage(root, PathParser.parse("PageOne.ChildOne"));
crawler.addPage(root, PathParser.parse("PageTwo"));
PageData data = pageOne.getData();
WikiPageProperties properties = data.getProperties();
WikiPageProperty symLinks = properties.set(SymbolicPage.PROPERTY_NAME);
symLinks.set("SymPage", "PageTwo");
pageOne.commit(data);
request.setResource("root");
request.addInput("type", "pages");
Responder responder = new SerializedPageResponder();
SimpleResponse response = (SimpleResponse) responder.makeResponse(
    new FitNesseContext(root), request);
String xml = response.getContent();
assertEquals("text/xml", response.getContentType());
assertSubString("<name>PageOne</name>", xml);
assertSubString("<name>PageTwo</name>", xml);
assertSubString("<name>ChildOne</name>", xml);
assertNotSubString("SymPage", xml);
}
public void testGetDataAsHtml() throws Exception
{
    crawler.addPage(root, PathParser.parse("TestPageOne"), "test page");
    request.setResource("TestPageOne");
    request.addInput("type", "data");
    Responder responder = new SerializedPageResponder();
    SimpleResponse response = (SimpleResponse) responder.makeResponse(
        new FitNesseContext(root), request);
    String xml = response.getContent();
    assertEquals("text/xml", response.getContentType());
    assertSubString("test page", xml);
    assertSubString("<Test", xml);
}

```

Per esempio, osservate le chiamate a `PathParser`. Trasformano delle stringhe in istanze di `PagePath` usate da `crawler`. Questa trasformazione è completamente irrilevante per il test e serve solo a offuscare lo scopo. Anche i dettagli che circondano la creazione del `responder` e la raccolta e conversione della risposta sono solo “rumore”. Poi c’è anche il modo grossolano di costruire l’URL della richiesta da una risorsa e un argomento (ho contribuito io stesso a scrivere questo codice, pertanto mi sento libero di critirarlo senza scrupoli).

In buona sostanza, questo codice non è stato realizzato avendo in mente la leggibilità. Chi sarà costretto a leggerlo si ritroverà inondato da un mare di dettagli che devono essere compresi prima di capire il significato dei test.

Ora considerate i nuovi test presentati nel Listato 9.2. Fanno esattamente la stessa cosa, ma sono stati sottoposti a refactoring, ottenendo una forma molto più pulita e descrittiva.

Listato 9.2 SerializedPageResponderTest.java (dopo un refactoring).

```
public void testGetPageHierarchyAsXml() throws Exception {
    makePages("PageOne", "PageOne.ChildOne", "PageTwo");
    submitRequest("root", "type:pages");
    assertResponseIsXML();
    assertResponseContains(
        "<name>PageOne</name>", "<name>PageTwo</name>", "<name>ChildOne</name>");
}
public void testSymbolicLinksAreNotInXmlPageHierarchy() throws Exception {
    WikiPage page = makePage("PageOne");
    makePages("PageOne.ChildOne", "PageTwo");
    addLinkTo(page, "PageTwo", "SymPage");
    submitRequest("root", "type:pages");
    assertResponseIsXML();
    assertResponseContains(
        "<name>PageOne</name>", "<name>PageTwo</name>", "<name>ChildOne</name>");
    assertResponseDoesNotContain("SymPage");
}
public void testDataAsXml() throws Exception {
    makePageWithContent("TestPageOne", "test page");
    submitRequest("TestPageOne", "type:data");
    assertResponseIsXML();
    assertResponseContains("test page", "<Test");
}
```

Lo schema COSTRUISCI-UTILIZZA-CONTROLLA

(<http://fitness.org/FitNesse.AcceptanceTestPatterns>) risulta evidente dalla struttura di questi test. Ognuno dei test è chiaramente suddiviso in tre parti. La prima parte prepara i dati del test, la seconda parte opera sui dati del test e la terza parte controlla che l'operazione fornisca i risultati previsti.

Notate che la maggior parte dei dettagli inutili è stata eliminata. I test vanno dritti al punto e usano solo i tipi di dati e le funzioni di cui hanno effettivamente bisogno. Chiunque legga questi test dovrebbe essere in grado di capire molto rapidamente che cosa fanno, senza distrazioni e senza essere sommerso dai dettagli.

Un linguaggio per test che sia specifico del dominio

I test presentati nel Listato 9.2 mostrano la tecnica che consiste nel costruire un linguaggio specifico di un dominio per i vostri test. Invece di usare le API impiegate dai programmatore per manipolare il sistema, realizziamo un insieme di funzioni e strumenti di servizio che fanno uso di tali API e che rendono i test più facili da scrivere e da leggere.

Queste funzioni e questi strumenti di servizio divengono quindi un'API specializzata impiegata dai test. Formano un *linguaggio* di test che i programmatore possono usare per facilitare la scrittura dei test e per aiutare coloro che, successivamente, si troveranno a leggere tali test.

Questa API di test non è realizzata una volta per tutte; piuttosto evolve a seguito del continuo refactoring del codice di test che inizialmente era troppo contaminato da dettagli inutili. Così come mi avete visto eseguire il refactoring del Listato 9.1 per produrre il Listato 9.2, così anche ogni sviluppatore assennato dovrebbe eseguire il refactoring del proprio codice di test in forme più compatte ed espessive.

Un doppio standard

In un certo senso il team di cui ho parlato all'inizio di questo capitolo aveva qualche ragione. Il codice dell'API di test *deve* sottostare a un insieme differente di standard progettuali rispetto al codice di produzione. Deve essere semplice, succinto ed espressivo, ma non deve essere altrettanto efficiente del codice di produzione. Dopotutto, opera in un ambiente di test, non in un ambiente di produzione e questi due ambienti hanno esigenze molto differenti.

Considerate il test presentato nel Listato 9.3. Ho scritto questo test nell'ambito di un sistema di controllo ambientale in fase di prototipo.

Senza entrare nei dettagli, si capisce che questo test controlla che l'allarme per bassa temperatura, il riscaldamento e la ventilazione siano ON quando la temperatura è troppo fredda.

Listato 9.3 EnvironmentControllerTest.java.

```
@Test
public void turnOnLoTempAlarmAtThreshold() throws Exception {
    hw.setTemp(WAY_TOO_COLD);
    controller.tic();
    assertTrue(hw.heaterState());
    assertTrue(hw.blowerState());
    assertFalse(hw.coolerState());
    assertFalse(hw.hiTempAlarm());
    assertTrue(hw.loTempAlarm());
}
```

Vi sono, naturalmente, molti dettagli. Per esempio, che cos'è quella funzione `tic`? Ma ignorate questo dettaglio mentre leggete questo test. Piuttosto chiedetevi se vi sembra che lo stato finale del sistema sia coerente col fatto che la temperatura sia troppo fredda.

Notate, mentre leggete il test, che il vostro sguardo deve continuare a rimbalzare fra il nome dello stato controllato e il suo *senso*. Prima vedete `heaterState` e poi i vostri occhi scorrono a sinistra sull'`assertTrue`. Prima vedete `coolerState` e poi i vostri occhi devono tornare a sinistra sull'`assertFalse`. Questo è noioso e inaffidabile. Complica la lettura del test.

Ho migliorato la leggibilità di questo test trasformandolo nel Listato 9.4.

Listato 9.4 EnvironmentControllerTest.java (dopo un refactoring).

```
@Test
public void turnOnLoTempAlarmAtThreshold() throws Exception {
    wayTooCold();
    assertEquals("HBchL", hw.getState());
}
```

Naturalmente ho nascosto il dettaglio della funzione `tic` creando una funzione `wayTooCold`. Ma l'aspetto da notare è la strana stringa in `assertEquals`. Una lettera maiuscola significa "on", una minuscola

significa “off” e le lettere sono sempre nel seguente ordine: {heater, blower, cooler, hi-temp-alarm, lo-temp-alarm}.

Anche se questo si avvicina a una violazione della regola delle mappe mentali (“Evitate le mappe mentali”, nel Capitolo 2), sembra appropriato in questo caso. Notate che, ora che conoscete il significato, i vostri occhi scorrono su tale stringa e potete interpretare rapidamente i risultati. La lettura del test diviene quasi piacevole. Date un’occhiata al Listato 9.5 e considerate quanto è facile comprendere questi test.

Listato 9.5 EnvironmentControllerTest.java (selezione più ampia).

```
@Test
public void turnOnCoolerAndBlowerIfTooHot() throws Exception {
    tooHot();
    assertEquals("hBChl", hw.getState());
}
@Test
public void turnOnHeaterAndBlowerIfTooCold() throws Exception {
    tooCold();
    assertEquals("HBchl", hw.getState());
}
@Test
public void turnOnHiTempAlarmAtThreshold() throws Exception {
    wayTooHot();
    assertEquals("hBCHl", hw.getState());
}
@Test
public void turnOnLoTempAlarmAtThreshold() throws Exception {
    wayTooCold();
    assertEquals("HBchL", hw.getState());
}
```

La funzione `getState` si trova nel Listato 9.6. Notate che il suo codice non è molto efficiente. Per renderlo efficiente, probabilmente avrei dovuto usare uno `StringBuffer`.

Listato 9.6 MockControlHardware.java.

```
public String getState() {
    String state = "";
    state += heater ? "H" : "h";
    state += blower ? "B" : "b";
    state += cooler ? "C" : "c";
    state += hiTempAlarm ? "H" : "h";
    state += loTempAlarm ? "L" : "l";
    return state;
}
```

Gli `StringBuffer` sono “bruttini”. Anche nel codice di produzione cerco di evitarli se il costo è accettabile; e potreste ritener che il costo del codice presentato nel Listato 9.6 è molto piccolo. Tuttavia, questa applicazione è chiaramente per un sistema embedded real-time ed è probabile che le risorse disponibili (computer e memoria) siano molto limitate. L’ambiente di *test*, al contrario, non ha particolari vincoli.

Questa è la natura del doppio standard. Vi sono cose che non potreste mai fare in un ambiente di produzione e che sono perfettamente fattibili in un ambiente di test. Solitamente i problemi riguardano la memoria o l’efficienza della CPU. Ma non riguardano mai un problema di pulizia del codice.

Una sola richiesta per test

Vi è una scuola di pensiero (si veda il blog di Dave Astel:

<http://www.artima.com/weblogs/viewpost.jsp?thread=35578>) che dice che ogni funzione di test di una JUnit dovrebbe avere una e una sola istruzione `assert`. Questa regola può sembrare draconiana, ma il vantaggio si può vedere nel Listato 9.5. Tali test giungono a un’unica conclusione di rapida e facile comprensione. E cosa dire del Listato 9.2? Sembra impossibile poter, in qualche modo, unire l’affermazione che l’output sia XML e che contiene determinate sottostringhe. Tuttavia, possiamo suddividere il test in due test distinti, ognuno dei quali ha la sua specifica affermazione (Listato 9.7).

Listato 9.7 SerializedPageResponderTest.java (un’unica affermazione).

```
public void testGetPageHierarchyAsXml() throws Exception {
    givenPages("PageOne", "PageOne.ChildOne", "PageTwo");
    whenRequestIsIssued("root", "type:pages");
    thenResponseShouldBeXML();
}
public void testGetPageHierarchyHasRightTags() throws Exception {
    givenPages("PageOne", "PageOne.ChildOne", "PageTwo");
    whenRequestIsIssued("root", "type:pages");
    thenResponseShouldContain(
        "<name>PageOne</name>", "<name>PageTwo</name>", "<name>ChildOne</name>");
```

}

Notate che ho cambiato il nome delle funzioni in modo da usare la convenzione *given-when-then* (*dato... - quando... - allora...*) [RSpec]. Questo semplifica ulteriormente la leggibilità dei test. Sfortunatamente, questa suddivisione dei test ha prodotto una duplicazione del codice. Possiamo eliminare la duplicazione usando il pattern TEMPLATE METHOD [GOF] e ponendo le parti *given* e *when* nella classe base e le parti *then* in classi derivate differenti. O potremmo creare una classe di test completamente distinta e porre le parti *given* e *when* nella funzione `@Before` e le parti *then* in varie funzioni `@Test`. Ma questo sembra un meccanismo troppo complesso per un problema così piccolo. Alla fine, preferisco gli `assert` multipli presentati nel Listato 9.2.

Penso che la regola dell'unica richiesta sia una buona indicazione. Solitamente tento di creare un linguaggio specifico del dominio per i test, come nel Listato 9.5. Ma non disdegno la possibilità di inserire più di una richiesta in un test. Penso che quello che dobbiamo cercare di fare sia di ridurre al minimo il numero di richieste presenti in un test.

Un unico concetto per ogni test

Forse una regola migliore suggerirebbe di sottoporre a test un unico concetto per ogni funzione di test (“Attenetevi al codice!”). Non vogliamo realizzare lunghe funzioni che applichino tanti test disparati uno dopo l'altro.

Il Listato 9.8 è un esempio di questo tipo. Questo test dovrebbe essere suddiviso in tre test indipendenti, perché sottopone a test tre cose indipendenti. Unire il tutto in un'unica funzione costringe chi legge a capire il perché di ogni sezione e che cosa controlla ognuna delle sezioni.

Listato 9.8

```

/**
 * Test vari per il metodo addMonths().
 */
public void testAddMonths() {
    SerialDate d1 = SerialDate.createInstance(31, 5, 2004);
    SerialDate d2 = SerialDate.addMonths(1, d1);
    assertEquals(30, d2.getDayOfMonth());
    assertEquals(6, d2.getMonth());
    assertEquals(2004, d2.getYYYY());
    SerialDate d3 = SerialDate.addMonths(2, d1);
    assertEquals(31, d3.getDayOfMonth());
    assertEquals(7, d3.getMonth());
    assertEquals(2004, d3.getYYYY());
    SerialDate d4 = SerialDate.addMonths(1, SerialDate.addMonths(1, d1));
    assertEquals(30, d4.getDayOfMonth());
    assertEquals(7, d4.getMonth());
    assertEquals(2004, d4.getYYYY());
}

```

Le tre funzioni di test probabilmente dovrebbero essere disposte nel seguente modo.

- Dato (*given*) l'ultimo giorno di un mese di 31 giorni (come maggio):
 1. quando (*when*) si aggiunge un mese a tale data, tale che l'ultimo giorno di tale mese è il 30 (come giugno), allora (*then*) la data dovrebbe essere il 30 di tale mese, non il 31;
 2. quando (*when*) si aggiungono due mesi a tale data, l'ultimo giorno di tale mese è il 31, allora (*then*) la data dovrebbe essere il 31.
- Dato (*given*) l'ultimo giorno di un mese di 30 giorni (come giugno):
 1. Quando (*when*) si aggiunge un mese a tale data, tale che l'ultimo giorno di tale mese è il 31, allora (*then*) la data dovrebbe essere il 30, non il 31.

Come potete vedere, vi è una regola generale che si nasconde fra questi test. Quando si incrementa il mese, la data non può essere maggiore dell'ultimo giorno del mese. Questo implica che incrementando il mese il 28 febbraio si dovrebbe ottenere il 28 marzo. Tale test manca e sarebbe utile scriverlo.

Pertanto non sono le richieste multiple per ogni sezione a rendere problematico il Listato 9.8. Piuttosto è il fatto che si sottopone a test più di un concetto. Pertanto, probabilmente la regola migliore suggerirebbe di ridurre al minimo il numero di richiese per concetto e sottoporre a test un solo concetto per ogni funzione di test.

F.I.R.S.T.

NOTA

Quanto segue fa parte del materiale di training di Object Mentor Inc.

Un test è “pulito” quando obbedisce alle cinque regole che formano l’acronimo F.I.R.S.T.

- *Fast* – I test dovrebbero essere veloci. Dovrebbero essere eseguiti rapidamente. Se i test sono lenti, eviterete di eseguirli con la dovuta frequenza. In tal caso, non scoprirete i problemi con la dovuta tempestività, tanto da poterli correggere con facilità. Non vi sentirete sufficientemente liberi di raffinare il codice. Alla fine il codice inizierà a sgretolarsi.
- *Independent* – I test non dovrebbero dipendere l’uno dall’altro. Un test non dovrebbe predisporre le condizioni per il test successivo. Dovreste essere in grado di eseguire ogni test in modo indipendente e di eseguire i test in qualsiasi ordine. Quando i test dipendono l’uno dall’altro, il primo che fallirà provocherà altri fallimenti in cascata, complicando ogni diagnosi e celando gli effetti che si producono come conseguenza.
- *Repeatable* – I test dovrebbero essere ripetibili in qualsiasi ambiente. Dovreste essere in grado di eseguire i test nell’ambiente di produzione, nell’ambiente QA (Quality Assurance) e sul vostro portatile mentre state tornando a casa in treno senza una rete. Se i vostri test non sono ripetibili in qualsiasi ambiente, avrete sempre

una “scusa” per i loro fallimenti. Sarete anche nell’impossibilità di eseguire i test dove l’ambiente non è disponibile.

- *Self-Validating* – I test dovrebbero essere autoconclusivi e avere un output booleano. Il test o ha successo o fallisce. Non si deve essere costretti a leggere un file log per scoprire se i test hanno avuto successo. Non si devono confrontare manualmente due diversi file di testo per scoprire se i test hanno avuto successo. Se i test non sono autoconclusivi, il fallimento può diventare soggettivo e l’esecuzione dei test può richiedere una lunga valutazione manuale.
- *Timely* – I test devono essere aggiornati. Gli unit test dovrebbero essere scritti appena prima del codice di produzione cui fanno riferimento. Se scrivete i test dopo il codice di produzione, potreste trovare che è troppo difficile sottoporre a test il codice di produzione. Potreste anche decidere che una qualche parte del codice di produzione è semplicemente troppo difficile da sottoporre a test. In altre parole finireste per progettare il codice di produzione in modo incollaudabile.

Conclusioni

Abbiamo solo scalfito la superficie di questo argomento. In effetti, penso che si potrebbe scrivere un intero libro sulle tecniche di pulizia dei test. Per la “salute” di un progetto, i test sono altrettanto importanti del codice di produzione. Forse sono ancora più importanti, perché i test preservano e migliorano la flessibilità, manutenibilità e riutilizzabilità del codice di produzione. Pertanto curate sempre la pulizia dei vostri test.

Fate di tutto per renderli espressivi e succinti. Inventate delle API di test che costruiscano un linguaggio specifico del dominio, che faciliti la

scrittura dei test.

Se trascurate i test, il primo che ne pagherà le conseguenze sarà il vostro codice. Quindi, curate la pulizia dei vostri test.

Bibliografia

- [RSpec]: Aslak Hellesøy, David Chelimsky, RSpec: Behavior Driven Development for Ruby Programmers, Pragmatic Bookshelf, 2008.
- [GOF]: Gamma et al., Design Patterns: Elements of Reusable Object Oriented Software, Addison-Wesley, Boston 1996.

Capitolo 10

Classi di Jeff Langr



Finora, in questo libro, ci siamo concentrati sulle regole di buona scrittura di righe e blocchi del codice. Abbiamo parlato di corretta composizione delle funzioni e di come correlarle. Ma anche con tutta questa attenzione all'espressività del codice e delle funzioni, non possiamo ancora dire che il nostro codice sia pulito finché non

dedichiamo la nostra attenzione ai livelli più elevati dell'organizzazione del codice. Parliamo quindi di pulizia delle classi.

Organizzazione delle classi

In base alle convenzioni standard di Java, una classe dovrebbe iniziare con un elenco di variabili. Poi devono venire le eventuali costanti statiche pubbliche. Poi le variabili statiche private, seguite dalle variabili private di istanza. È raro che vi sia un buon motivo per avere una variabile pubblica.

Le funzioni pubbliche vengono subito dopo la lista delle variabili. È bene porre gli strumenti di servizio privati richiamati da una funzione pubblica appena dopo la funzione pubblica stessa. Questo in base alla regola dei passi, per poter leggere il programma come l'articolo di una rivista.

Incapsulazione

È bene mantenere private le variabili e le funzioni di servizio, ma senza fanatismi. Talvolta dobbiamo rendere `protected` una variabile o una funzione di servizio, in modo da potervi accedere con un test. Secondo noi, comandano i test. Se un test deve richiamare una funzione o accedere a una variabile nello stesso package, la renderemo `protected` o le daremo una visibilità a livello del package. Tuttavia, prima dobbiamo aver esplorato ogni possibilità utile per mantenere la privacy. Allentare l'incapsulazione deve essere sempre solo un'ultima risorsa.

Le classi dovrebbero essere piccole!

La prima regola delle classi è che dovrebbero essere piccole. La seconda regola delle classi è che dovrebbero essere ancora più piccole. No, non intendiamo ripetere le cose già dette nel capitolo sulle funzioni. Ma come per le funzioni, “più piccole” è la regola principale quando si parla di classi. Come con le funzioni, sorge spontanea la domanda: “Piccole quanto?”.

Con le funzioni abbiamo misurato le dimensioni contando, semplicemente, le righe. Con le classi usiamo una metrica differente: le responsabilità [RDD].

Il Listato 10.1 presenta una classe, `superDashboard`, che espone circa settanta metodi pubblici. La maggior parte degli sviluppatori concorderebbe sul fatto che le sue dimensioni sono eccessive. Alcuni sviluppatori considererebbero `SuperDashboard` una sorta di divinità fra le classi.

Listato 10.1 Troppe Responsabilità.

```
public class SuperDashboard extends JFrame implements MetaDataUser
    public String getCustomizerLanguagePath()
    public void setSystemConfigPath(String systemConfigPath)
    public String getSystemConfigDocument()
    public void setSystemConfigDocument(String systemConfigDocument)
    public boolean getGuruState()
    public boolean getNoviceState()
    public boolean getOpenSourceState()
    public void showObject(MetaObject object)
    public void showProgress(String s)
    public boolean isMetadataDirty()
    public void setIsMetadataDirty(boolean isMetadataDirty)
    public Component getLastFocusedComponent()
    public void setLastFocused(Component lastFocused)
    public void setMouseSelectState(boolean isMouseSelected)
    public boolean isMouseSelected()
    public LanguageManager getLanguageManager()
    public Project getProject()
    public Project getFirstProject()
    public Project getLastProject()
    public String getNewProjectName()
    public void setComponentSizes(Dimension dim)
    public String getCurrentDir()
    public void setCurrentDir(String newDir)
    public void updateStatus(int dotPos, int markPos)
    public Class[] getDataBaseClasses()
    public MetadataFeeder getMetadataFeeder()
    public void addProject(Project project)
    public boolean setCurrentProject(Project project)
    public boolean removeProject(Project project)
```

```

public MetaProjectHeader getProgramMetadata()
public void resetDashboard()
public Project loadProject(String fileName, String projectName)
public void setCanSaveMetadata(boolean canSave)
public MetaObject getSelectedObject()
public void deselectObjects()
public void setProject(Project project)
public void editorAction(String actionPerformed, ActionEvent event)
public void setMode(int mode)
public FileManager getFileManager()
public void setFileManager(FileManager fileManager)
public ConfigManager getConfigManager()
public void setConfigManager(ConfigManager configManager)
public ClassLoader getClassLoader()
public void setClassLoader(ClassLoader classLoader)
public Properties getProps()
public String getUserHome()
public String getBaseDir()
public int getMajorVersionNumber()
public int getMinorVersionNumber()
public int getBuildNumber()
public MetaObject pasting(
    MetaObject target, MetaObject pasted, MetaProject project)
public void processMenuItems(MetaObject metaObject)
public void processMenuSeparators(MetaObject metaObject)
public void processTabPages(MetaObject metaObject)
public void processPlacement(MetaObject object)
public void processCreateLayout(MetaObject object)
public void updateDisplayLayer(MetaObject object, int layerIndex)
public void propertyEditedRepaint(MetaObject object)
public void processDeleteObject(MetaObject object)
public boolean getAttachedToDesigner()
public void processProjectChangedState(boolean hasProjectChanged)
public void processObjectNameChanged(MetaObject object)
public void runProject()
public void setAgowDragging(boolean allowDragging)
public boolean allowDragging()
public boolean isCustomizing()
public void setTitle(String title)
public IdeMenuBar getIdeMenuBar()
public void showHelper(MetaObject metaObject, String propertyName)
// ... seguono molti altri metodi non pubblici ...
}

}

```

Ma... e se SuperDashboard contenesse solo i metodi presentati nel Listato 10.2?

Listato 10.2 È abbastanza piccola?

```

public class SuperDashboard extends JFrame implements MetaDataUser
    public Component getLastFocusedComponent()
    public void setLastFocused(Component lastFocused)
    public int getMajorVersionNumber()
    public int getMinorVersionNumber()
    public int getBuildNumber()

}

```

Cinque metodi non sono troppi, vero? In questo caso però sì, perché, nonostante il piccolo numero di metodi, `SuperDashboard` ha troppe responsabilità.

Il nome di una classe dovrebbe descrivere le sue responsabilità. Pensandoci bene, il nome è probabilmente il primo modo con cui si possono determinare le dimensioni di una classe. Se non possiamo concepire un nome conciso per una classe, probabilmente è perché è troppo grande. Più è ambiguo il nome della classe, più probabilmente essa ha troppe responsabilità. Per esempio, i nomi di classi che includono parole come `Processor` o `Manager` o `Super` spesso sottintendono uno sfortunato eccesso di *responsabilità*.

Dovremmo anche essere in grado di scrivere una breve descrizione della classe in circa 25 parole, senza mai usare le parole “se”, “e”, “o” e “ma”. Come descrivereste `SuperDashboard`?

La classe `SuperDashboard` fornisce l’accesso al componente che per ultimo aveva il focus e ci consente anche di risalire al numero di versione e di build.

La prima “e” è già un indizio che `SuperDashboard` ha troppe responsabilità.

Il principio SRP (Single Responsibility Principle)

Tale principio (vedi [PPP] per maggiori informazioni) stabilisce che una classe o un modulo dovrebbe avere uno e un solo *motivo per dover cambiare*. Questo principio ci dà sia una definizione delle responsabilità sia un’indicazione delle dimensioni della classe. Le classi dovrebbero avere una sola responsabilità: un solo motivo per dover cambiare. La classe `SuperDashboard`, solo apparentemente piccola, presentata nel Listato 10.2 ha due motivi per dover cambiare. Innanzitutto, registra le informazioni sulla versione, che dovrebbero

essere aggiornate a ogni nuova uscita del software. In secondo luogo, gestisce i componenti di Java Swing (si tratta di una derivata di JFrame, la rappresentazione Swing di una finestra di alto livello della GUI). Senza dubbio vorremo aggiornare il numero di versione se dovremo modificare il codice di Swing, ma non necessariamente accade il contrario: potremmo cambiare il numero di versione sulla base di interventi su altre parti del sistema.

Il fatto di tentare di identificare le responsabilità (i motivi per cambiare) spesso ci aiuta a riconoscere e creare migliori astrazioni nel nostro codice. Possiamo estrarre con facilità da `SuperDashboard` i tre metodi che si occupano del numero di versione in una classe distinta di nome `Version` (Listato 10.3). La classe `Version` è un costrutto che ha molte probabilità di essere riutilizzato in altre applicazioni!

Listato 10.3 Una classe con una sola responsabilità.

```
public class Version {  
    public int getMajorVersionNumber()  
    public int getMinorVersionNumber()  
    public int getBuildNumber()  
}
```

Il principio SRP è uno dei concetti più importanti nella progettazione OO. È anche uno dei concetti più facili da comprendere e seguire. Tuttavia, purtroppo, spesso è anche il più violato dei principi di progettazione delle classi. Incontriamo regolarmente classi che fanno davvero troppe cose. Perché?

“Far funzionare” il software e “rendere pulito” il software sono due attività molto differenti. La maggior parte di noi ha capacità di ragionamento non illimitate, pertanto normalmente ci concentriamo più sul far funzionare il nostro codice che sulla sua organizzazione e pulizia. Questo è del tutto logico. Mantenere una separazione degli ambiti è altrettanto importante nelle nostre *attività* di programmazione quanto lo è nei nostri programmi.

Il problema è che troppi, fra noi, sono convinti che una volta che il programma “funziona”, il lavoro sia finito. In tal modo ignoriamo il problema dell’organizzazione e della pulizia. Così passiamo al problema successivo invece di tornare indietro e spezzare le classi troppo congestionate in unità disaccoppiate, dotate di un’unica responsabilità.

Contemporaneamente, molti sviluppatori temono che la presenza di una grande quantità di piccole classi, mono-scopo complichi la comprensione dell’immagine generale. Temono di dover vagare di classe in classe per scoprire come si comporta una determinata parte di un sistema.

Tuttavia, un sistema con molte piccole classi non ha più meccanismi di un sistema con poche grandi classi. Le cose da capire sono le stesse di un sistema con poche grandi classi. Pertanto la domanda è: “Volete che i vostri strumenti siano organizzati in tanti piccoli compartimenti dai quali trarre componenti ben definiti e ben etichettati? O volete pochi compartimenti nei quali infilare un po’ di tutto?”.

Ogni sistema di dimensioni non banali avrà necessariamente una certa logica e la relativa complessità. L’obiettivo principale, nel gestire tale complessità, consiste nell’*organizzarla* in modo che uno sviluppatore sappia dove trovare le cose e debba preoccuparsi solo del livello di complessità delle cose sulle quali sta direttamente lavorando. Al contrario, un sistema dotato di classi grandi, multiuso ci costringerà sempre a vagare in un mare di cose che non sarebbe necessario considerare, in un ambito piccolo.

Per riaffermare quanto detto in precedenza, vogliamo che i nostri sistemi siano costituiti da tante piccole classi, non da poche e grandi classi. Ogni piccola classe incapsula un’unica responsabilità, ha un unico motivo per dover cambiare e collabora con poche altre per ottenere il comportamento desiderato del sistema.

Coesione

Le classi dovrebbero avere un piccolo numero di variabili di istanza. Ognuno dei metodi di una classe dovrebbe manipolare una o più di quelle variabili. In generale più variabili vengono manipolate da un metodo maggiore è la coesione fra tale metodo e la sua classe. Una classe in cui ogni variabile viene usata da ogni metodo è massimamente coesa.

In generale non è né consigliabile né possibile creare classi così massimamente coese; d'altra parte, vorremmo avere un'elevata coesione. Quando la coesione è alta, significa che i metodi e le variabili della classe sono co-dipendenti e formano una solida unità logica.

Considerate l'implementazione di uno `Stack` presentata nel Listato 10.4. Questa è una classe molto coesa. Dei tre metodi, solo `size()` non usa entrambe le variabili.

Listato 10.4 Stack.java: una classe coesa.

```
public class Stack {
    private int topOfStack = 0;
    List<Integer> elements = new LinkedList<Integer>();
    public int size() {
        return topOfStack;
    }
    public void push(int element) {
        topOfStack++;
        elements.add(element);
    }
    public int pop() throws PoppedWhenEmpty {
        if (topOfStack == 0)
            throw new PoppedWhenEmpty();
        int element = elements.get(--topOfStack);
        elements.remove(topOfStack);
        return element;
    }
}
```

La strategia che punta ad avere funzioni piccole ed elenchi di parametri brevi, talvolta porta a una proliferazione di variabili di istanza che vengono impiegate solo da un sottoinsieme dei metodi.

Quando si verifica questa situazione, significa quasi sempre che vi è quanto meno un'altra classe che dovrebbe uscire da questa grande classe. Dovreste provare a separare le variabili e i metodi in due o più classi, in modo che le nuove classi siano più coese.

Curando la coesione si generano tante piccole classi

Anche solo il fatto di suddividere le funzioni più grandi in funzioni più piccole provoca una proliferazione di classi. Considerate per esempio una grossa funzione nella quale sono dichiarate molte variabili. Immaginate di voler estrarre una piccola parte di tale funzione, a formare una funzione distinta. Tuttavia, il codice che volete estrarre usa quattro delle variabili dichiarate nella funzione. Dovreste passare tutte e quattro tali variabili nella nuova funzione sotto forma di argomenti?

Niente affatto! Se promuovessimo queste quattro variabili in variabili di istanza della classe, potremmo estrarre il codice senza passargli alcuna variabile. Sarebbe facile spezzare la funzione in tante piccole parti.

Sfortunatamente, questo significa anche che le nostre classi perderanno coesione, perché accumuleranno sempre più variabili di istanza che esistono solo per consentire a una manciata di funzioni di condividerle. Ma un attimo! Se vi sono alcune funzioni che vogliono condividere determinate variabili, non potrebbero formare una classe a se stante? Naturalmente sì. Quando le classi perdono coesione, spezzatele!

Pertanto, il fatto di spezzare una grossa funzione in varie funzioni più piccole spesso ci dà anche l'opportunità di suddividere una classe. Ciò dà al nostro programma un'organizzazione molto migliore e una struttura molto più trasparente.

Per dimostrare cosa intendo, proviamo a usare un esempio davvero datato, tratto dal meraviglioso libro *Literate Programming* di Donald Knuth [Knuth92]. Il Listato 10.5 presenta una traduzione in Java del programma `PrintPrimes` di Knuth. Per dare a Knuth quel che è di Knuth, questo non è un programma suo, ma è l'output del suo strumento di nome WEB. L'ho usato perché rappresenta un ottimo punto di partenza per suddividere una grossa funzione in più funzioni e classi più piccole.

Listato 10.5 PrintPrimes.java.

```
package literatePrimes;
public class PrintPrimes {
    public static void main(String[] args) {
        final int M = 1000;
        final int RR = 50;
        final int CC = 4;
        final int WW = 10;
        final int ORDMAX = 30;
        int P[] = new int[M + 1];
        int PAGENUMBER;
        int PAGEOFFSET;
        int ROWOFFSET;
        int C;
        int J;
        int K;
        boolean JPRIME;
        int ORD;
        int SQUARE;
        int N;
        int MULT[] = new int[ORDMAX + 1];
        J = 1;
        K = 1;
        P[1] = 2;
        ORD = 2;
        SQUARE = 9;
        while (K < M) {
            do {
                J = J + 2;
                if (J == SQUARE) {
                    ORD = ORD + 1;
                    SQUARE = P[ORD] * P[ORD];
                    MULT[ORD - 1] = J;
                }
                N = 2;
                JPRIME = true;
                while (N < ORD && JPRIME) {
                    while (MULT[N] < J)
                        MULT[N] = MULT[N] + P[N] + P[N];
                    if (MULT[N] == J)
                        JPRIME = false;
                    N = N + 1;
                }
            } while (JPRIME);
            K = K + 1;
        }
    }
}
```

```

        P[K] = J;
    }
    {
        PAGENUMBER = 1;
        PAGEOFFSET = 1;
        while (PAGEOFFSET <= M) {
            System.out.println("The First " + M +
                               " Prime Numbers - Page " + PAGENUMBER);
            System.out.println("");
            for (ROWOFFSET = PAGEOFFSET; ROWOFFSET < PAGEOFFSET + RR; ROWOFFSET++) {
                for (C = 0; C < CC; C++)
                    if (ROWOFFSET + C * RR <= M)
                        System.out.format("%10d", P[ROWOFFSET + C * RR]);
                System.out.println("");
            }
            System.out.println("\f");
            PAGENUMBER = PAGENUMBER + 1;
            PAGEOFFSET = PAGEOFFSET + RR * CC;
        }
    }
}

```

Questo programma, scritto come un'unica funzione, è un vero groviglio. Ha una struttura profondamente annidata, una pletora di orribili variabili e una struttura fortemente accoppiata. Come minimo, la grossa funzione dovrebbe essere suddivisa in più funzioni più piccole.

I Listati da 10.6 a 10.8 mostrano il risultato della suddivisione del codice presentato nel Listato 10.5 in classi e funzioni più piccole e della scelta di nomi significativi per tali classi, funzioni e variabili.

Listato 10.6 PrimePrinter.java (dopo il refactoring).

```

package literatePrimes;
public class PrimePrinter {
    public static void main(String[] args) {
        final int NUMBER_OF_PRIMES = 1000;
        int[] primes = PrimeGenerator.generate(NUMBER_OF_PRIMES);
        final int ROWS_PER_PAGE = 50;
        final int COLUMNS_PER_PAGE = 4;
        RowColumnPagePrinter tablePrinter =
            new RowColumnPagePrinter(ROWS_PER_PAGE, COLUMNS_PER_PAGE,
                                   "The First " + NUMBER_OF_PRIMES + " Prime
Numbers");
        tablePrinter.print(primes);
    }
}

```

Listato 10.7 RowColumnPagePrinter.java.

```

package literatePrimes;
import java.io.PrintStream;
public class RowColumnPagePrinter {
    private int rowsPerPage;
    private int columnsPerPage;
    private int numbersPerPage;
    private String pageHeader;
    private PrintStream printStream;
    public RowColumnPagePrinter(int rowsPerPage, int columnsPerPage,
                               String pageHeader) {
        this.rowsPerPage = rowsPerPage;
        this.columnsPerPage = columnsPerPage;
        this.pageHeader = pageHeader;
        numbersPerPage = rowsPerPage * columnsPerPage;
        printStream = System.out;
    }
    public void print(int data[]) {
        int pageNumber = 1;
        for (int firstIndexOnPage = 0;
             firstIndexOnPage < data.length;
             firstIndexOnPage += numbersPerPage) {
            int lastIndexOnPage =
                Math.min(firstIndexOnPage + numbersPerPage - 1, data.length - 1);
            printPageHeader(pageHeader, pageNumber);
            printPage(firstIndexOnPage, lastIndexOnPage, data);
            printStream.println("\f");
            pageNumber++;
        }
    }
    private void printPage(int firstIndexOnPage,
                          int lastIndexOnPage,
                          int[] data) {
        int firstIndexOfLastRowOnPage =
            firstIndexOnPage + rowsPerPage - 1;
        for (int firstIndexInRow = firstIndexOnPage;
             firstIndexInRow <= firstIndexOfLastRowOnPage;
             firstIndexInRow++) {
            printRow(firstIndexInRow, lastIndexOnPage, data);
            printStream.println("");
        }
    }
    private void printRow(int firstIndexInRow,
                         int lastIndexOnPage,
                         int[] data) {
        for (int column = 0;
             column < columnsPerPage;
             column++) {
            int index = firstIndexInRow + column * rowsPerPage;
            if (index <= lastIndexOnPage)
                printStream.format("%10d", data[index]);
        }
    }
    private void printPageHeader(String pageHeader,
                                int pageNumber) {
        printStream.println(pageHeader + " - - - Page " + pageNumber);
        printStream.println("");
    }
    public void setOutput(PrintStream printStream) {
        this.printStream = printStream;
    }
}

```

```
}
```

Listato 10.8 PrimeGenerator.java.

```
package literatePrimes;
import java.util.ArrayList;
public class PrimeGenerator {
    private static int[] primes;
    private static ArrayList<Integer> multiplesOfPrimeFactors;
    protected static int[] generate(int n) {
        primes = new int[n];
        multiplesOfPrimeFactors = new ArrayList<Integer>();
        set2AsFirstPrime();
        checkOddNumbersForSubsequentPrimes();
        return primes;
    }
    private static void set2AsFirstPrime() {
        primes[0] = 2;
        multiplesOfPrimeFactors.add(2);
    }
    private static void checkOddNumbersForSubsequentPrimes() {
        int primeIndex = 1;
        for (int candidate = 3;
             primeIndex < primes.length;
             candidate += 2) {
            if (isPrime(candidate))
                primes[primeIndex++] = candidate;
        }
    }
    private static boolean isPrime(int candidate) {
        if (isLeastRelevantMultipleOfNextLargerPrimeFactor(candidate)) {
            multiplesOfPrimeFactors.add(candidate);
            return false;
        }
        return isNotMultipleOfAnyPreviousPrimeFactor(candidate);
    }
    private static boolean
    isLeastRelevantMultipleOfNextLargerPrimeFactor(int candidate) {
        int nextLargerPrimeFactor = primes[multiplesOfPrimeFactors.size()];
        int leastRelevantMultiple = nextLargerPrimeFactor * nextLargerPrimeFactor;
        return candidate == leastRelevantMultiple;
    }
    private static boolean
    isNotMultipleOfAnyPreviousPrimeFactor(int candidate) {
        for (int n = 1; n < multiplesOfPrimeFactors.size(); n++) {
            if (isMultipleOfNthPrimeFactor(candidate, n))
                return false;
        }
        return true;
    }
    private static boolean
    isMultipleOfNthPrimeFactor(int candidate, int n) {
        return
            candidate == smallestOddNthMultipleNotLessThanCandidate(candidate, n);
    }
    private static int
    smallestOddNthMultipleNotLessThanCandidate(int candidate, int n) {
        int multiple = multiplesOfPrimeFactors.get(n);
        while (multiple < candidate)
            multiple += 2 * primes[n];
    }
}
```

```
        multiplesOfPrimeFactors.set(n, multiple);
        return multiple;
    }
}
```

La prima cosa che potete notare è che il programma è diventato molto più lungo: da una pagina e mezza ne ha riempite più di tre. Tale crescita è dovuta a vari motivi. Innanzitutto, il programma dopo il refactoring usa nomi di variabili più lunghi e descrittivi. In secondo luogo, il programma dopo il refactoring usa le dichiarazioni di funzioni e classi come un modo per commentare il codice. Infine abbiamo usato la spaziatura e la formattazione per migliorare la leggibilità del programma.

Notate come il programma sia stato suddiviso in tre grandi “responsabilità”. Il programma principale è contenuto nella classe `PrimePrinter`. Ha il compito di gestire l’ambiente di esecuzione. Cambierà solo se cambierà il metodo di chiamata. Per esempio, se questo programma venisse convertito in un servizio SOAP (*Simple Object Access Protocol*), questa sarebbe la classe da adattare.

La classe `RowColumnPagePrinter` sa formattare un elenco di numeri nelle pagine, con un determinato numero di righe e colonne. Se la formattazione dell’output deve cambiare, questa è la classe da adattare.

La classe `PrimeGenerator` sa come generare un elenco di numeri primi. Notate che non è fatta per essere istanziata da un oggetto. La classe è solo un blocco utile per dichiarare e tenere nascoste le variabili. Questa classe cambierà qualora l’algoritmo di calcolo dei numeri primi dovesse cambiare.

Questa non era una riscrittura! Non siamo ripartiti da zero e scritto il programma da capo. In effetti, se osservate attentamente i due programmi, vedrete che impiegano lo stesso algoritmo e gli stessi meccanismi.

La modifica è stata svolta scrivendo un pacchetto di test che ha verificato l’esatto comportamento del primo programma. Poi è stata

fatta una miriade di piccole modifiche, una alla volta. Dopo ogni modifica il programma è stato eseguito per garantire che il suo comportamento non fosse cambiato. Un passo dopo l’altro, il primo programma è stato ripulito e trasformato nel secondo.

Organizzare gli interventi di modifica

Per la maggior parte dei sistemi, gli interventi sono continui. Ogni modifica ci sottopone al rischio che la parte rimanente del sistema non funzioni più a dovere. In un sistema pulito organizziamo le nostre classi in modo da ridurre il rischio di dover apportare cambiamenti.

La classe `Sql` presentata nel Listato 10.9 viene usata per generare stringhe SQL correttamente formate partendo dai metadati appropriati. È un lavoro ancora provvisorio e, di conseguenza, non supporta ancora alcune funzionalità SQL come le istruzioni `update`. Quando alla classe `Sql` occorre aggiungere il supporto delle istruzioni `update`, dobbiamo “aprire” questa classe per apportare le modifiche richieste. Il problema dell’aprire una classe è che introduce dei rischi. Ogni modifica alla classe potrebbe pregiudicare il funzionamento di un’altra parte della classe. Occorre ripetere interamente i test.

Listato 10.9 Una classe da modificare.

```
public class Sql {  
    public Sql(String table, Column[] columns)  
    public String create()  
    public String insert(Object[] fields)  
    public String selectAll()  
    public String findByKey(String keyColumn, String keyValue)  
    public String select(Column column, String pattern)  
    public String select(Criteria criteria)  
    public String preparedInsert()  
    private String columnList(Column[] columns)  
    private String valuesList(Object[] fields, final Column[] columns)  
    private String selectWithCriteria(String criteria)  
    private String placeholderList(Column[] columns)  
}
```

La classe `Sql` deve cambiare ogni volta che aggiungiamo un nuovo tipo di istruzione. Deve cambiare anche quando modifichiamo i dettagli

di un unico tipo di istruzione, per esempio se dobbiamo modificare la funzionalità `select` per supportare le subselezioni. Il fatto che siano due i motivi per cambiare la classe `Sql` già dice che essa viola il principio SRP.

Possiamo individuare questa violazione anche da un semplice punto di vista organizzativo. La struttura dei metodi di `Sql` mostra che vi sono dei metodi privati, come `selectWithCriteria`, che sembrano correlati solo alle istruzioni `select`.

Il fatto che il comportamento di un metodo privato si applichi solo a un piccolo sottoinsieme di una classe può essere un'utile euristica per individuare i possibili miglioramenti. Tuttavia, la principale motivazione per intervenire dovrebbe essere il fatto stesso di modificare il sistema. Se la classe `Sql` fosse completa in senso logico, non dovremmo preoccuparci di separarne le responsabilità. Se non avessimo bisogno della funzionalità `update` per il prossimo futuro, dovremmo lasciar stare `Sql`. Ma non appena ci troviamo ad aprire una classe, dovremmo considerare anche l'idea di correggerne la struttura.

E se considerassimo una soluzione come quella presentata nel Listato 10.10? Ogni metodo dell'interfaccia pubblica definito nella classe `Sql` del Listato 10.9 viene rifattorizzato, entrando a far parte di una derivata della classe `Sql`. Notate che i metodi privati, come `valuesList`, si spostano esattamente dove sono richiesti. Il comportamento privato comune viene isolato da una coppia di classi di servizio: `Where` e `ColumnList`.

Listato 10.10 Un insieme di classi chiuse.

```
abstract public class Sql {  
    public Sql(String table, Column[] columns)  
    abstract public String generate();  
}  
public class CreateSql extends Sql {  
    public CreateSql(String table, Column[] columns)  
    @Override public String generate()  
}  
public class SelectSql extends Sql {  
    public SelectSql(String table, Column[] columns)  
    @Override public String generate()
```

```

}
public class InsertSql extends Sql {
    public InsertSql(String table, Column[] columns, Object[] fields)
        @Override public String generate()
    private String valuesList(Object[] fields, final Column[] columns)
}
public class SelectWithCriteriaSql extends Sql {
    public SelectWithCriteriaSql(
        String table, Column[] columns, Criteria criteria)
        @Override public String generate()
}
public class SelectWithMatchSql extends Sql {
    public SelectWithMatchSql(
        String table, Column[] columns, Column column, String pattern)
        @Override public String generate()
}
public class FindByKeySql extends Sql {
    public FindByKeySql(
        String table, Column[] columns, String keyColumn, String keyValue)
        @Override public String generate()
}
public class PreparedInsertSql extends Sql {
    public PreparedInsertSql(String table, Column[] columns)
        @Override public String generate()
    private String placeholderList(Column[] columns)
}
public class Where {
    public Where(String criteria)
    public String generate()
}
public class ColumnList {
    public ColumnList(Column[] columns)
    public String generate()
}

```

Il codice di ogni classe diviene estremamente semplice. Il tempo necessario per comprendere ogni classe si riduce praticamente a zero. Il rischio che una funzione possa pregiudicare il funzionamento di un'altra quasi svanisce. Dal punto di vista dei test, diviene molto più facile dimostrare ogni dettaglio della logica di questa soluzione, in quanto le classi sono tutte isolate l'una dall'altra.

Altrettanto importante, quando sarà il momento di aggiungere le istruzioni `update`, non sarà necessario modificare nessuna delle classi esistenti! Programmeremo la logica per realizzare le istruzioni `update` in una nuova sottoclassificazione di `sql` chiamata `UpdateSql`. Il resto del codice del sistema non verrà interessato da questa modifica e quindi continuerà a funzionare come prima.

La nuova logica di `sql` rappresenta un punto di ottimo. Supporta il principio SRP. Supporta anche un altro importante principio di progettazione delle classi OO chiamato Open-Closed Principle o OCP [PPP]: le classi dovrebbero essere aperte alle estensioni ma chiuse alle modifiche. La nostra nuova classe `sql` permette l'inserimento di nuove funzionalità tramite delle sottoclassi, ma possiamo apportare questa modifica mantenendo chiusa ogni altra classe. Non dobbiamo fare altro che scrivere la classe `UpdateSql`.

Vogliamo strutturare i nostri sistemi in modo da doverli modificare il meno possibile quando li aggiorniamo con funzionalità nuove o differenti. In un sistema ideale, incorporiamo le nuove funzionalità estendendo il sistema, non modificando il codice esistente.

Isolamento dalle modifiche

Le esigenze cambiano, pertanto anche il codice deve cambiare. Sappiamo che in OO vi sono le classi concrete, che contengono i dettagli implementativi (il codice) e le classi astratte, che rappresentano solo i concetti. Una classe client che dipende da dettagli concreti corre dei rischi qualora tali dettagli dovessero cambiare. Possiamo introdurre delle interfacce e delle classi astratte per cercare di isolare l'impatto di tali dettagli.

Le dipendenze dai dettagli concreti creano problemi anche per i test del sistema. Se creiamo una classe `Portfolio` che dipende dall'API esterna `TokyoStockExchange` per ottenere un valore, i nostri casi di test subiranno l'impatto della sua volatilità. È difficile scrivere un test quando si potrebbe ottenere una risposta differente ogni cinque minuti!

Invece di progettare `Portfolio` in modo che dipenda direttamente da `TokyoStockExchange`, creiamo un'interfaccia, `StockExchange`, che dichiara un unico metodo:

```

public interface StockExchange {
    Money currentPrice(String symbol);
}

```

Progettiamo `TokyoStockExchange` in modo da implementare questa interfaccia. Dobbiamo anche assicurarci che il costruttore di `Portfolio` accetti come argomento uno `stockExchange`:

```

public Portfolio {
    private StockExchange exchange;
    public Portfolio(StockExchange exchange) {
        this.exchange = exchange;
    }
    // ...
}

```

Ora il nostro test può creare un'implementazione verificabile dell'interfaccia `StockExchange` che emula `TokyoStockExchange`. Questa implementazione di test correggerà l'attuale valore del simbolo che usiamo nel test. Se il nostro test mostra l'acquisto di cinque azioni Microsoft nel nostro portafoglio, programmiamo l'implementazione di test in modo che restituisca sempre \$100 per ogni azione Microsoft. La nostra implementazione di test dell'interfaccia `StockExchange` si riduce a una semplice ricerca in una tabella. Possiamo così scrivere un test che si aspetta il valore \$500 per il valore del nostro portafoglio.

```

public class PortfolioTest {
    private FixedStockExchangeStub exchange;
    private Portfolio portfolio;
    @Before
    protected void setUp() throws Exception {
        exchange = new FixedStockExchangeStub();
        exchange.fix("MSFT", 100);
        portfolio = new Portfolio(exchange);
    }
    @Test
    public void GivenFiveMSFTTotalShouldBe500() throws Exception {
        portfolio.add(5, "MSFT");
        Assert.assertEquals(500, portfolio.value());
    }
}

```

Se un sistema è sufficientemente disaccoppiato da poter essere sottoposto a test in questo modo, sarà sicuramente flessibile e faciliterà ogni riutilizzo. La mancanza di accoppiamento fa sì che gli elementi del

nostro sistema siano più isolati fra loro e da ogni modifica. Questo isolamento facilita la comprensione di ogni elemento del sistema.

Riducendo al minimo l'accoppiamento, le nostre classi obbediranno anche a un altro principio di progettazione: il DIP (*Dependency Inversion Principle*) [PPP]. In pratica, il principio DIP dice che le nostre classi dovrebbero dipendere da astrazioni, non da dettagli concreti.

Invece di dipendere dai dettagli implementativi della classe `TokyoStockExchange`, la nostra classe `Portfolio` ora dipende dall'interfaccia `StockExchange`, la quale rappresenta il concetto astratto di chiedere il valore di un determinato simbolo. Questa astrazione isola da tutti i dettagli necessari per ottenere tale prezzo, compresa l'origine di tale informazione.

Bibliografia

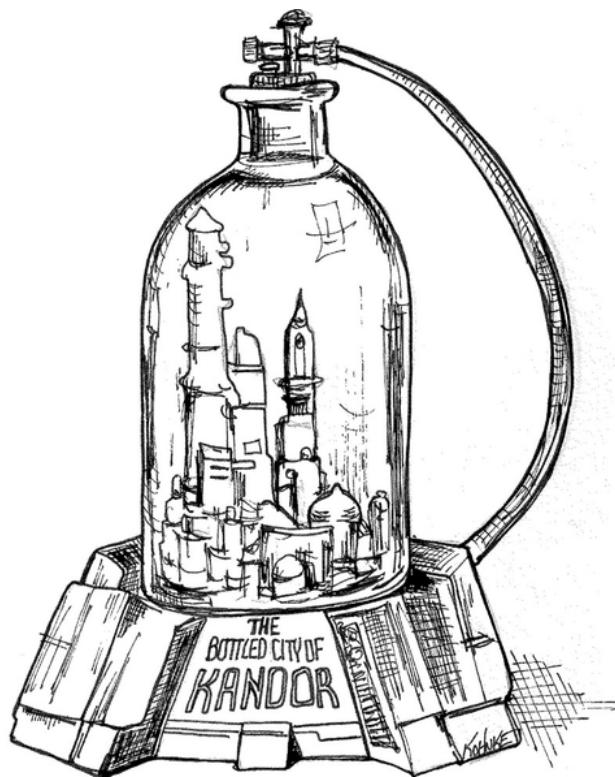
- [RDD]: Rebecca Wirfs- Brock et al., *Object Design: Roles, Responsibilities, and Collaborations*, Addison-Wesley, Boston 2002.
- [PPP]: Robert C. Martin, *Agile Software Development: Principles, Patterns, and Practices*, Prentice Hall, Upper Saddle River, New Jersey 2002.
- [Knuth92]: Donald E. Knuth, *Literate Programming*, Center for the Study of Language and Information, Leland Stanford Junior University, Stanfor 1992.

Capitolo 11

Sistemi *di Kevin Dean Wampler*

La complessità uccide. Succhia la linfa vitale degli sviluppatori, genera prodotti difficili da progettare, realizzare e collaudare.

Ray Ozzie, CTO, Microsoft Corporation



Come edifichereste una città?

Riuscireste, da soli, a considerare tutti i dettagli? Probabilmente no. Anche solo gestire una città è un’impresa eccessiva per una sola persona. Tuttavia, le città funzionano (il più delle volte). Funzionano perché le città hanno dei team di persone che gestiscono specifiche parti della città: l’acquedotto, i sistemi di alimentazione, di circolazione del traffico, di pubblica sicurezza, di edificazione e così via. Alcune di queste persone sono responsabili del *piano generale*, mentre altre si occupano dei dettagli.

Le città funzionano anche perché hanno sviluppato appropriati livelli di astrazione e modularità, che consentono a delle persone e ai “componenti” che esse gestiscono di operare efficacemente, anche senza conoscere il piano generale.

Sebbene spesso i team di sviluppo software siano ben organizzati, i sistemi sui quali lavorano spesso non hanno gli stessi livelli di separazione degli ambiti e di astrazione. La pulizia del codice ci aiuta in questo senso anche ai più bassi livelli di astrazione. In questo capitolo vedremo come mantenere la pulizia ai livelli di astrazione più elevati, quelli del *sistema*.

Separate la realizzazione dall’uso di un sistema

Innanzitutto, considerate che la *realizzazione* è un processo molto differente dall’uso. Mentre scrivo, vi è un nuovo hotel in costruzione, che vedo dalla mia finestra di Chicago. Oggi è solo un blocco di cemento, con delle impalcature e una gru. Gli addetti indossano tutti il casco e abiti da lavoro. Entro un anno o giù di lì l’hotel sarà ultimato. Le impalcature e la gru non ci saranno più. L’edificio sarà pulito, ricoperto di pannelli di vetro e colorato. Anche gli addetti e gli ospiti avranno un aspetto molto differente.

I sistemi software dovrebbero separare il processo di avvio, quando vengono realizzati gli oggetti dell'applicazione e vengono stabilite le dipendenze, dalla logica runtime che si esprime dopo l'avvio.

Il processo di avvio è un ambito che deve essere considerato da ogni applicazione. Si tratta del principale ambito che esamineremo in questo capitolo. La separazione degli ambiti è una delle più antiche e importanti tecniche progettuali della nostra attività.

Sfortunatamente, la maggior parte delle applicazioni non considera questa separazione. Il codice del processo di avvio è *ad hoc* e si mescola con la logica runtime. Ecco un tipico esempio:

```
public Service getService() {  
    if (service == null)  
        service = new MyServiceImpl(...);  
        // Default sufficiente per la maggior parte dei casi?  
    return service;  
}
```

Questa si chiama *lazy initialization/evaluation* e la “pigrizia” di questa inizializzazione e valutazione ha numerosi meriti. Non incorriamo nei tempi di costruzione, a meno che usiamo effettivamente l'oggetto, e di conseguenza i nostri tempi di avvio possono essere più rapidi. Inoltre ci assicuriamo che non venga mai restituito `null`.

Tuttavia, ora abbiamo una dipendenza diretta da `MyServiceimpl` e da tutto ciò che richiede il suo costruttore (che ho tralasciato). Non possiamo compilare senza risolvere queste dipendenze, anche se in realtà a runtime non useremo mai un oggetto di questo tipo!

Il collaudo può essere un problema. Se `MyServiceimpl` è un oggetto “pesante”, dovremo assicurarci che al campo del servizio sia assegnato un oggetto *Test Double* [Mezzaros07] o *Mock* prima di richiamare questo metodo durante lo unit test. Poiché abbiamo mescolato la logica costruttiva alla normale elaborazione runtime, dovremo sottoporre a test tutti i percorsi di esecuzione (per esempio, il test del `null` e il suo blocco). Il fatto di avere entrambe queste responsabilità significa che il

metodo sta facendo più di una cosa, ovvero che stiamo infrangendo il principio *SRP* (della responsabilità unica).

Peggio ancora: non sappiamo se `MyServiceimpl` è l'oggetto giusto in tutte le situazioni, come ho scritto nel commento. Perché la classe con questo metodo deve conoscere il contesto globale? *Davvero* possiamo sapere qual è l'oggetto giusto da usare qui? È possibile che esista davvero un unico tipo adatto a tutti i contesti possibili?

Un'unica occorrenza di inizializzazione lazy non rappresenta un problema, naturalmente. Tuttavia, nelle applicazioni in genere vi sono più istanze di piccoli idiom di setup come questo. Pertanto, la *strategia* di setup globale (se esiste) risulta *frammentata* in tutta l'applicazione, con poca modularità e spesso molta duplicazione.

Se siamo *diligenti* nella realizzazione di sistemi ben strutturati e solidi, non dovremmo mai consentire che alcuni piccoli e *comodi* idiom infrangano la modularità. Il processo di avvio che prevede la costruzione e il collegamento degli oggetti non fa eccezione. Dovremmo modularizzare questo processo separatamente rispetto alla normale logica runtime e dovremmo assicurarci di avere una strategia globale e coerente per risolvere le principali dipendenze.

Separazione di main

Un modo per separare la costruzione dall'uso consiste semplicemente nel trasferire tutti gli aspetti della costruzione in `main` o nei moduli richiamati da `main` e nel progettare il resto del sistema supponendo che tutti gli oggetti siano già stati costruiti e collegati in modo appropriato (Figura 11.1).

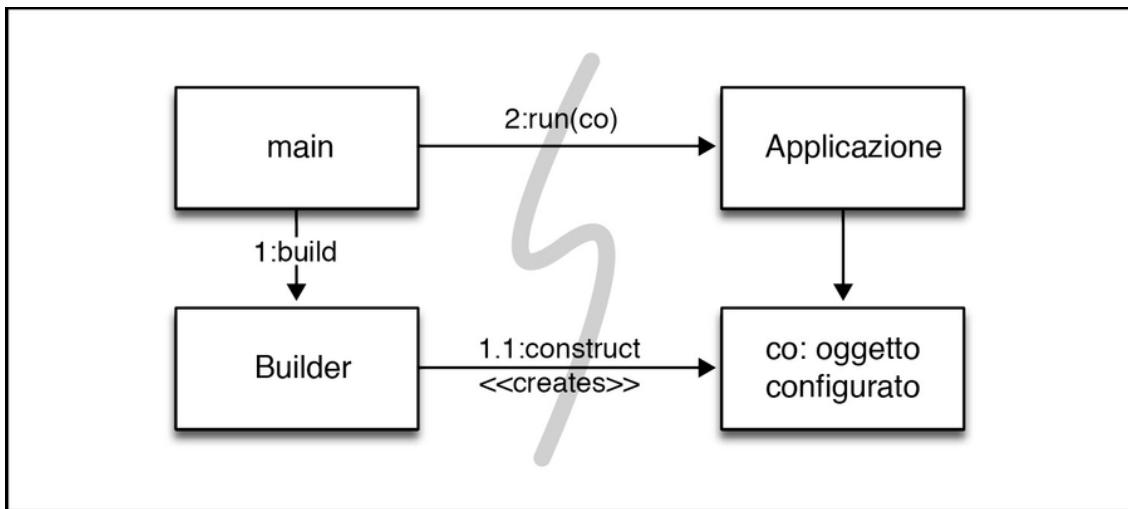


Figura 11.1 Separazione della costruzione, in main().

Il flusso di controllo è facile da seguire. La funzione `main` costruisce gli oggetti necessari per il sistema, poi li passa all'applicazione, la quale, semplicemente, li utilizza. Notate la direzione delle frecce di dipendenza che attraversano la barriera fra `main` e l'applicazione. Vanno tutte in un'unica direzione: si allontanano da `main`. Questo significa che l'applicazione non ha alcuna conoscenza di `main` o del processo di costruzione. Semplicemente si aspetta che tutto sia già stato costruito correttamente.

Factory

Talvolta, naturalmente, dobbiamo far sì che l'applicazione sia responsabile del momento in cui l'oggetto viene creato. Per esempio, in un sistema di elaborazione degli ordini l'applicazione deve creare le istanze di `LineItem` da aggiungere a un `Order`. In questo caso possiamo usare il pattern Abstract Factory [GOF] per dare all'applicazione il controllo del momento in cui costruire i `LineItem`, ma mantenendo tutti i dettagli di tale costruzione separati rispetto al codice dell'applicazione (Figura 11.2).

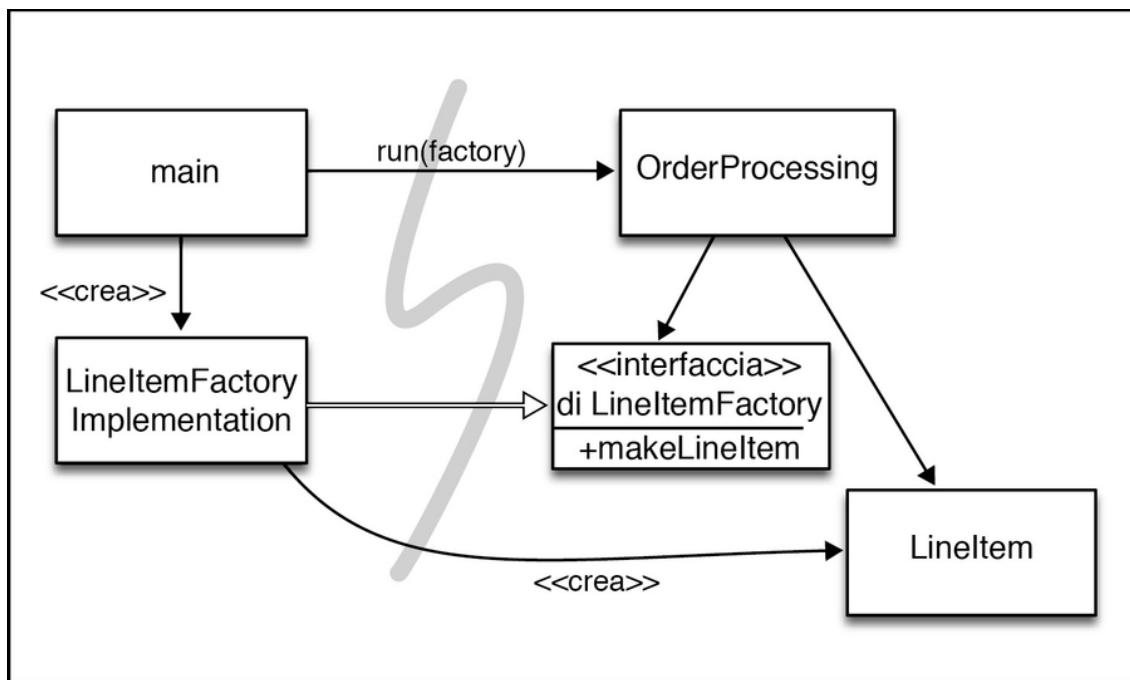


Figura 11.2 Separazione della costruzione con factory.

Di nuovo, notate come tutte le dipendenze puntino da `main` verso l'applicazione `OrderProcessing`. Questo significa che l'applicazione è disaccoppiata dai dettagli costruttivi di `LineItem`. Tale funzionalità è contenuta in `LineItemFactoryImplementation`, che è dalla stessa parte di `main` nello schema. E tuttavia l'applicazione mantiene il completo controllo della costruzione delle istanze di `LineItem` e può anche fornire al costruttore gli argomenti specifici dell'applicazione.

Dependency Injection

Un potente meccanismo per separare la costruzione dall'uso è chiamato *Dependency Injection* (DI), che è l'applicazione dell'inversione del controllo (Inversion of Control - IoC) alla gestione delle dipendenze (vedi, per esempio, [Fowler]). L'inversione del controllo trasferisce le responsabilità secondarie da un oggetto ad altri oggetti dedicati a tale scopo, supportando in tal modo il principio SRP,

dell'unica responsabilità. Nel contesto della gestione delle dipendenze, un oggetto non dovrebbe assumersi esso stesso le responsabilità dell'istanziazione delle dipendenze. Al contrario, dovrebbe passare questa responsabilità a un altro meccanismo “autorevole”, invertendo pertanto il controllo. Poiché il setup si trova in un ambito globale, questo meccanismo autorevole solitamente si troverà o nella routine `main` o in un apposito container.

Le ricerche JNDI sono un esempio di implementazione parziale della DI, dove un oggetto domanda a un *directory server* di fornire un “servizio” corrispondente a un determinato nome.

```
MyService myService = (MyService) (jndiContext.lookup("NameOfMyService")) ;
```

L'oggetto chiamante non controlla quale genere di oggetto viene restituito (gli basta che implementi l'interfaccia appropriata, naturalmente), ma in ogni caso è l'oggetto chiamante a risolvere la dipendenza.

Una vera *Dependency Injection* si spinge un passo oltre. La classe non intraprende alcun passo diretto per risolvere le sue dipendenze; è completamente passiva. Al contrario, fornisce dei metodi *setter* o degli argomenti del costruttore (o entrambi) che vengono impiegati per iniettare le dipendenze. Durante il processo di costruzione, il container DI istanzia gli oggetti richiesti (solitamente *on demand*) e usa gli argomenti del costruttore o i metodi setter forniti per risolvere le dipendenze. Quali oggetti dipendenti vengano usati in realtà viene specificato tramite un file di configurazione o da programma in un modulo dedicato alla costruzione.

Il framework Spring fornisce il miglior container DI attualmente disponibile per Java (vedi [Spring]; esiste anche il framework Spring.NET). Si definiscono tramite un file di configurazione XML gli oggetti da collegare fra loro, poi si richiedono specifici oggetti per nome nel codice Java. Ne vedremo un esempio fra breve.

Ma che cosa si può dire sui pregi dell'inizializzazione “pigra”, *lazy*? Questo idioma talvolta è ancora utile nella DI. Innanzitutto, la maggior parte dei container DI non costruisce un oggetto finché non è necessario. In secondo luogo, molti di questi container forniscono dei meccanismi per richiamare le factory o per costruire dei proxy, i quali possono essere utilizzati per la valutazione *lazy* e altre ottimizzazioni analoghe (non dimenticate che l'istanziazione/valutazione *lazy* è solo un'ottimizzazione e forse è anche prematura!).

Estensione di scala

Le città un tempo furono paesi e prima ancora furono semplici insediamenti. All'inizio le strade sono vicoli sterrati, ma poi vengono lastricate e asfaltate, sempre allargandole. I piccoli edifici e le aree vuote lasciano progressivamente il posto a edifici più grandi, alcuni dei quali alla fine verranno sostituiti da grattacieli.

All'inizio non vi sono servizi di energia elettrica, acqua, fognatura o per Internet (gasp!). Questi servizi vengono aggiunti mano a mano che cresce la densità di abitanti ed edifici.

Questa crescita non è indolare. Quante volte avete guidato, buca dopo buca, in una zona sottoposta a ristrutturazioni e ampiamenti domandandovi: “Ma perché non l'hanno fatta più grande già da subito?”.

Ma come poteva essere previsto? Come giustificare la spesa per un viale a sei corsie che attraversa il centro di un piccolo abitato? Chi mai vorrebbe un'arteria così davanti a casa?

È solo un mito che i sistemi possano essere “buoni fin da subito”. Piuttosto, dovremmo prepararci a implementare bene oggi le necessità dell'oggi, e a rifattorizzare ed espandere il sistema domani per implementare le necessità del domani. Questa è l'essenza dell'agilità iterativa e incrementale. Lo sviluppo test-driven, il refactoring e il

codice pulito che essi producono svolgono questo lavoro a livello del codice.

Ma che dire del livello del sistema? L'architettura del sistema non richiede una pre-pianificazione? Certamente, anch'essa può crescere incrementalmente di complessità.

I sistemi software hanno alcune peculiarità rispetto ai sistemi fisici. La loro architettura può crescere incrementalmente, ma solo se manteniamo una corretta separazione degli ambiti.

È la natura effimera dei sistemi software a renderlo possibile, come vedremo. Consideriamo innanzitutto un controsenso di un'architettura che non separa adeguatamente gli ambiti.

Le architetture EJB1 ed EJB2 non separavano gli ambiti in modo appropriato e pertanto imponevano inutili barriere a una crescita organica. Considerate un *Entity Bean* di una classe persistente `Bank`. Un entity bean è una rappresentazione in memoria di dati relazionali; in altre parole è la riga di una tabella.

Innanzitutto, dovreste definire un'interfaccia locale (nel processo) o remota (con una JVM distinta), utilizzabile dai client. Il Listato 11.1 presenta un esempio di interfaccia locale.

Listato 11.1 Un'interfaccia locale EJB2 per un EJB Bank.

```
package com.example.banking;
import java.util.Collections;
import javax.ejb.*;
public interface BankLocal extends java.ejb.EJBLocalObject {
    String getStreetAddr1() throws EJBException;
    String getStreetAddr2() throws EJBException;
    String getCity() throws EJBException;
    String getState() throws EJBException;
    String getZipCode() throws EJBException;
    void setStreetAddr1(String street1) throws EJBException;
    void setStreetAddr2(String street2) throws EJBException;
    void setCity(String city) throws EJBException;
    void setState(String state) throws EJBException;
    void setZipCode(String zip) throws EJBException;
    Collection getAccounts() throws EJBException;
    void setAccounts(Collection accounts) throws EJBException;
    void addAccount(AccountDTO accountDTO) throws EJBException;
}
```

Qui mostro numerosi attributi dell'indirizzo di `Bank` e una `collection` di conti della banca, ognuna delle quali avrà dei dati gestiti da un `Account` EJB distinto. Il Listato 11.2 presenta la classe di implementazione corrispondente per il bean `Bank`.

Listato 11.2 L'implementazione corrispondente EJB2 Entity Bean.

```
package com.example.banking;
import java.util.Collections;
import javax.ejb.*;
public abstract class Bank implements javax.ejb.EntityBean {
    // Logica operativa...
    public abstract String getStreetAddr1();
    public abstract String getStreetAddr2();
    public abstract String getCity();
    public abstract String getState();
    public abstract String getZipCode();
    public abstract void setStreetAddr1(String street1);
    public abstract void setStreetAddr2(String street2);
    public abstract void setCity(String city);
    public abstract void setState(String state);
    public abstract void setZipCode(String zip);
    public abstract Collection getAccounts();
    public abstract void setAccounts(Collection accounts);
    public void addAccount(AccountDTO accountDTO) {
        InitialContext context = new InitialContext();
        AccountHomeLocal accountHome = context.lookup("AccountHomeLocal");
        AccountLocal account = accountHome.create(accountDTO);
        Collection accounts = getAccounts();
        accounts.add(account);
    }
    // logica del container EJB
    public abstract void setId(Integer id);
    public abstract Integer getId();
    public Integer ejbCreate(Integer id) { ... }
    public void ejbPostCreate(Integer id) { ... }
    // Il resto dovrebbe essere implementato, ma solitamente è vuoto:
    public void setEntityContext(EntityContext ctx) {}
    public void unsetEntityContext() {}
    public void ejbActivate() {}
    public void ejbPassivate() {}
    public void ejbLoad() {}
    public void ejbStore() {}
    public void ejbRemove() {}
}
```

Non mostro la corrispondente interfaccia `LocalHome`, sostanzialmente una factory usata per creare gli oggetti, né uno dei tanti possibili metodi finder (query) per `Bank` che potreste aggiungere.

Infine, dovreste scrivere uno o più descrittori XML di deployment che specificano i dettagli del mappaggio oggetto-relazionale su una

memoria persistente, il comportamento desiderato per le transazioni, i vincoli di sicurezza e così via.

La logica operativa è strettamente accoppiata all'applicazione EJB2 "container". Dovrete prevedere sottoclassi per i tipi del container e fornire i vari metodi operativi richiesti dal container.

Questo accoppiamento con un pesante container, complica l'esecuzione di unit test isolate. È necessario simulare il container, e può essere difficile o uno spreco di tempo predisporre gli EJB e i test su un vero server. Il riutilizzo all'esterno dell'architettura EJB2 è praticamente impossibile, a causa di questo stretto accoppiamento.

Infine, anche la programmazione a oggetti risulta pregiudicata. Un bean non può ereditare da un altro bean. Notate la logica per l'aggiunta di un nuovo account. È normale per i bean EJB2 definire degli oggetti DTO (*Data Transfer Object*) che, sostanzialmente, sono "struct" senza alcun comportamento. Questo, solitamente, produce tipi ridondanti, che contengono sostanzialmente gli stessi dati e richiedono l'impiego di codice specifico per copiare i dati da un oggetto a un altro.

Confusione di ambiti

L'architettura EJB2 si avvicina a una vera separazione degli ambiti, in alcune aree. Per esempio, i comportamenti transazionali, di sicurezza e (in parte) di persistenza desiderati sono dichiarati nei descrittori di deployment, indipendentemente dal codice sorgente.

Notate che *ambiti* come la persistenza tendono ad attraversare la naturale delimitazione degli oggetti di un dominio. Volete che la persistenza di tutti i vostri oggetti usi generalmente la stessa strategia; per esempio, che usi un determinato DBMS (*Database management system*) o un puro file di testo, che segua determinate convenzioni di denominazione per le tabelle e le colonne, che usi una semantica coerente per le transazioni e così via.

In linea di principio, potete ragionare sulla strategia di persistenza in modo modulare, incapsulato. Ma all'atto pratico, vi troverete sostanzialmente a distribuire lo stesso codice che implementa la strategia di persistenza in molti oggetti. Abbiamo pertanto una confusione di ambiti. Di nuovo, il framework di persistenza deve essere modulare e la nostra logica di dominio, in isolamento, deve essere modulare. Il problema è la fine granularità dell'intersezione di questi domini.

In effetti, il modo in cui l'architettura EJB gestiva la persistenza, la sicurezza e le transazioni, “anticipava” la programmazione AOP (*Aspect-Oriented Programming*) (vedi [AOSD], per informazioni generali sugli aspetti, e [AspectJ] e [Colyer], per informazioni specifiche su AspectJ), il quale è un approccio generalistico alla conservazione della modularità negli ambiti che prevedono questo tipo di confusione.

Nella AOP, dei costrutti modulari chiamati *aspetti* specificano su quali punti del sistema dovremmo intervenire in modo coerente per supportare un determinato ambito. Questa specifica si estrinseca tramite un meccanismo sintetico, dichiaratico o programmatico.

Considerando, per esempio, la persistenza, dovreste dichiarare quali oggetti e attributi (o pattern) dovrebbero essere resi persistenti e poi delegare i compiti di persistenza al framework che si occupa della persistenza. Le modifiche al comportamento vengono eseguite in modo *non invasivo* (ovvero non richiedono interventi manuali sul codice sorgente di destinazione) al codice in questione da parte del framework AOP. Esaminiamo i tre aspetti e meccanismi in Java.

Proxy Java

I proxy Java sono adatti alle situazioni più semplici, come l'incorporazione di chiamate a metodi nei singoli oggetti o nelle classi.

Tuttavia, i proxy dinamici forniti dal JDK funzionano solo con le interfacce. Quanto alle classi, occorre impiegare una libreria di manipolazione del *byte-code*, come CGLIB, ASM o Javassist (vedi [CGLIB], [ASM] e [Javassist]).

Il Listato 11.3 presenta la struttura di un proxy JDK che fornisce il supporto della persistenza per l'applicazione `Bank`, indicando solo i metodi per ottenere e impostare la lista dei conti.

Listato 11.3 Un esempio di proxy JDK.

```
// Bank.java (eliminati i nomi dei package...)
import java.util.*;
// L'astrazione di una banca.
public interface Bank {
    Collection<Account> getAccounts();
    void setAccounts(Collection<Account> accounts);
}
// BankImpl.java
import java.util.*;
// L'oggetto POJO (Plain Old Java Object) che implementa l'astrazione.
public class BankImpl implements Bank {
    private List<Account> accounts;
    public Collection<Account> getAccounts() {
        return accounts;
    }
    public void setAccounts(Collection<Account> accounts) {
        this.accounts = new ArrayList<Account>();
        for (Account account: accounts) {
            this.accounts.add(account);
        }
    }
}
// BankProxyHandler.java
import java.lang.reflect.*;
import java.util.*;
// "InvocationHandler" richiesto dall'API.
public class BankProxyHandler implements InvocationHandler {
    private Bank bank;
    public BankHandler (Bank bank) {
        this.bank = bank;
    }
    // Metodo definito in InvocationHandler
    public Object invoke(Object proxy, Method method, Object[] args)
        throws Throwable {
        String methodName = method.getName();
        if (methodName.equals("getAccounts")) {
            bank.setAccounts(getAccountsFromDatabase());
            return bank.getAccounts();
        } else if (methodName.equals("setAccounts")) {
            bank.setAccounts((Collection<Account>) args[0]);
            setAccountsToDatabase(bank.getAccounts());
            return null;
        } else {
            ...
        }
    }
}
```

```

        }
    }
    // Qui vanno molti dettagli:
    protected Collection<Account> getAccountsFromDatabase() { ... }
    protected void setAccountsToDatabase(Collection<Account> accounts) { ... }
    // Da qualche altra parte...
    Bank bank = (Bank) Proxy.newProxyInstance(
        Bank.class.getClassLoader(),
        new Class[] { Bank.class },
        new BankProxyHandler(new BankImpl())));

```

Abbiamo definito un’interfaccia `Bank`, che verrà incorporata dal proxy e un oggetto POJO (*Plain-Old Java Object*), `BankImpl`, che implementa la logica operativa. (Parleremo fra poco dei POJO.)

L’API Proxy richiede un oggetto `InvocationHandler` che richiama per implementare ogni chiamata a metodo di `Bank` al proxy. Il nostro oggetto `BankProxyHandler` usa l’API di reflection Java per mappare le chiamate a metodo generiche sui corrispondenti metodi di `BankImpl` e così via.

Vi è *molto* codice qui ed è anche relativamente complicato, anche per questo semplice caso (per esempi più dettagliati sull’API Proxy e vari esempi d’uso, vedi, per esempio, [Goetz]). Usare una libreria di manipolazione del byte-code è altrettanto complicato. La massa e la complessità di questo codice sono due dei difetti dei proxy. Rendono quasi impossibile creare codice pulito! Inoltre, i proxy non forniscono un meccanismo per specificare dei “punti” di interesse dell’esecuzione a livello dell’intero sistema, il che è necessario per realizzare una vera soluzione AOP. (La AOP viene talvolta confusa con le tecniche usate per implementarla, come l’intercettazione dei metodi e il “wrapping” tramite proxy. Il vero valore di un sistema di AOP è la capacità di specificare comportamenti sistematici in modo conciso e modulare.)

Framework AOP puri Java

Fortunatamente, la maggior parte delle attività di delega dei proxy può essere svolta automaticamente tramite appositi strumenti. I proxy

vengono impiegati internamente in numerosi framework Java, per esempio Spring AOP e JBoss AOP, per implementare gli *aspetti* in puro codice Java (vedi [Spring] e [JBoss]; con “puro Java” si intende Java senza l’uso di AspectJ). In Spring, si scrive la logica operativa come oggetti POJO (*Plain-Old Java Objects*) riguardanti esclusivamente quel dominio. Essi non hanno alcuna dipendenza dagli altri framework impiegati (o da qualsiasi altro dominio). Pertanto, sono concettualmente più semplici e più facili da sottoporre a test. La relativa semplicità fa sì che sia più facile garantire la corretta implementazione delle richieste degli utenti e poi anche manutenere e far evolvere il codice in futuro.

Si incorpora l’infrastruttura dell’applicazione, compresi gli ambiti “confusi” come la persistenza, le transazioni, la sicurezza, la gestione della cache, i failover e così via, usando i file di configurazione dichiarativi o API. In molti casi, in realtà si specificano *aspetti* della libreria Spring o JBoss, dove il framework gestisce in modo trasparente rispetto all’utente tutti i meccanismi legati all’uso dei proxy Java o delle librerie byte-code. Queste dichiarazioni a loro volta controllano il container DI (*Dependency Injection*), che istanzia i principali oggetti e li collega fra loro *on demand*.

Il Listato 11.4 presenta un tipico frammento di un file di configurazione Spring V2.5, `app.xml` (adattato da

<http://www.theserverside.com/tt/articles/article.tss?l=IntrotoSpring25>).

Listato 11.4 File di configurazione di Spring 2.X.

```
<beans>
  ...
  <bean id="appDataSource"
    class="org.apache.commons.dbcp.BasicDataSource"
    destroy-method="close"
    p:driverClassName="com.mysql.jdbc.Driver"
    p:url="jdbc:mysql://localhost:3306/mydb"
    p:username="me"/>
  <bean id="bankDataAccessObject"
    class="com.example.banking.persistence.BankDataAccessObject"
    p:dataSource-ref="appDataSource"/>
  <bean id="bank"
    class="com.example.banking.model.Bank"
```

```

p:dataAccessObject-ref="bankDataAccessObject"/>
...
</beans>

```

Ogni “bean” è come una parte di una sorta di “matrioska”, dove un oggetto del dominio (`Bank`) viene preso in carico (l’essenza del proxy) e incorporato (*wrapped*) da un oggetto DAO (*Data Accessor Object*), che a sua volta viene preso in carico da un driver JDBC (Figura 11.3).

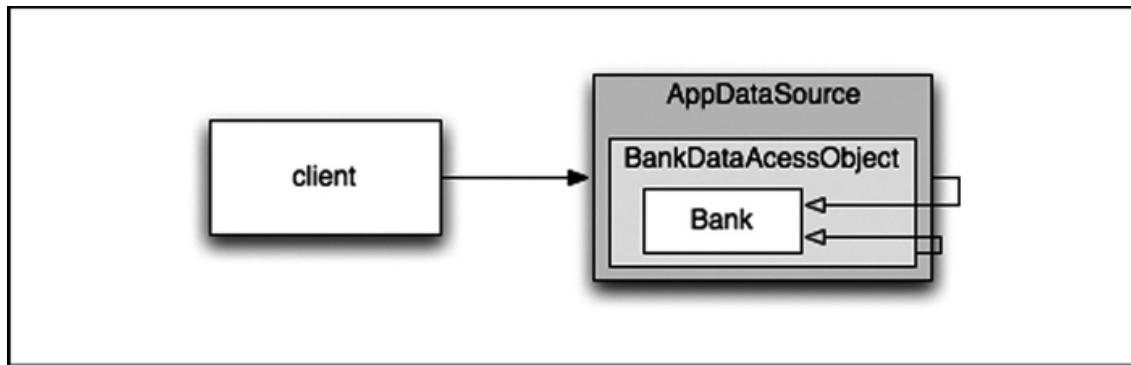


Figura 11.3 La “matrioska” di decoratori.

Il client è convinto di richiamare `getAccounts()` su un oggetto `Bank`, ma in realtà comunica solo con quello più esterno di un insieme di oggetti DECORATOR ([GOF]) annidati, che estendono il comportamento di base del POJO `Bank`. Potremmo aggiungere altri decoratori per le transazioni, per la cache e così via.

Nell’applicazione, bastano poche righe per chiedere al container DI (Dependency Injection) gli oggetti di livello più elevato del sistema, come specificato nel file XML.

```

XmlBeanFactory bf =
    new XmlBeanFactory(new ClassPathResource("app.xml", getClass()));
Bank bank = (Bank) bf.getBean("bank");

```

Dal momento che sono necessarie così poche righe di codice Java specifico per Spring, l’applicazione risulta quasi completamente disaccoppiata da Spring, eliminando così i problemi di accoppiamento troppo stretto di sistemi come EJB2.

Sebbene le direttive XML possano essere prolixe e difficili da leggere (l'esempio può essere semplificato utilizzando meccanismi che sfruttano la tecnica *convention over configuration* e le annotazioni Java 5 per ridurre la quantità di logica di "connessione" esplicita richiesta), la "politica" specificata in questi file di configurazione è più semplice della complessa logica del proxy e degli aspetti, che risulta nascosta e creata automaticamente. Questo tipo di architettura è così apprezzata che framework come Spring hanno portato a una completa revisione dello standard EJB per la versione 3. EJB3 segue in larga misura il modello Spring, che consiste nel supportare in modo dichiarativo quegli ambiti che prevedono confusione di ruoli, usando file di configurazione XML e/o annotazioni Java 5.

Il Listato 11.5 presenta il nostro oggetto `Bank` riscritto in EJB3 (adattato da <http://www.onjava.com/pub/a/onjava/2006/05/17/standardizing-with-ejb3-java-persistence-api.html>).

Listato 11.5 L'EJB Bank in EJB3.

```
package com.example banking.model;
import javax.persistence.*;
import java.util.ArrayList;
import java.util.Collection;
@Entity
@Table(name = "BANKS")
public class Bank implements java.io.Serializable {
    @Id @GeneratedValue(strategy=GenerationType.AUTO)
    private int id;
    @Embeddable // Un oggetto "inline" nella riga del DB per Bank
    public class Address {
        protected String streetAddr1;
        protected String streetAddr2;
        protected String city;
        protected String state;
        protected String zipCode;
    }
    @Embedded
    private Address address;
    @OneToMany(cascade = CascadeType.ALL, fetch = FetchType.EAGER, mappedBy="bank")
    private Collection<Account> accounts = new ArrayList<Account>();
    public int getId() {
        return id;
    }
    public void setId(int id) {
        this.id = id;
    }
    public void addAccount(Account account) {
```

```

        account.setBank(this);
        accounts.add(account);
    }
    public Collection<Account> getAccounts() {
        return accounts;
    }
    public void setAccounts(Collection<Account> accounts) {
        this.accounts = accounts;
    }
}

```

Questo codice è molto più pulito dell'EJB2 originale. Vi permangono alcuni dettagli sulle entità, contenuti nelle annotazioni. Tuttavia, poiché nessuna di tali informazioni si trova all'esterno delle annotazioni, il codice è pulito, chiaro e pertanto facile da provare, manutenere e così via.

Una parte delle informazioni nelle annotazioni relative alla persistenza (o anche tutte) possono essere trasferite nei descrittori XML di deployment, se lo si desidera, col risultato di avere puro codice POJO. Se i dettagli di mappaggio della persistenza non cambiano frequentemente, molti team possono decidere di mantenerli nelle annotazioni, ma con molti meno rischi di effetti negativi rispetto all'invasività di EJB2.

AspectJ

Infine, lo strumento più avanzato per separare gli ambiti tramite gli aspetti è il linguaggio AspectJ (vedi [AspectJ] e [Colyer]), un'estensione di Java che fornisce un supporto eccezionale per gli aspetti, come i costrutti rivolti alla modularità. Gli approcci in puro codice Java consentiti da Spring AOP e JBoss AOP sono sufficienti per l'80/90 percento dei casi in cui gli aspetti sono più utili. Tuttavia, AspectJ fornisce un ricchissimo e potente insieme di strumenti per separare gli ambiti. Il difetto di AspectJ è la necessità di adottare alcuni nuovi strumenti e imparare nuovi costrutti e idiomì del linguaggio.

I problemi di adozione sono stati parzialmente limitati da una recente introduzione: La *annotation form* di AspectJ, nella quale le annotazioni Java 5 vengono impiegate per definire gli aspetti usando puro codice Java. Inoltre, il framework Spring offre varie funzionalità che semplificano molto l'incorporazione di annotazioni per un team dotato di un'esperienza limitata nell'uso di AspectJ.

Una descrizione approfondita di AspectJ non rientra negli scopi di questo libro. A tale proposito consultate [AspectJ], [Colyer] e [Spring].

Sottoporre a test l'architettura del sistema

Il vantaggio di separare gli ambiti con un approccio ad aspetti è davvero notevole. Se potete scrivere la logica di dominio della vostra applicazione usando comune codice POJO, disaccoppiato da ogni ambito dell'architettura a livello del codice, allora diventa davvero possibile sottoporre a *test* l'architettura. Potete farla evolvere in base alle necessità, adottando le nuove tecnologie disponibili, senza rivoluzioni in stile BDUF (Big Design Up Front), da non confondersi con la pratica della progettazione anticipata, BDUF è la pratica che consiste nel progettare tutto in anticipo, prima ancora di iniziare ogni implementazione). A dire il vero, le attività BDUF sono anche pericolose, perché non favoriscono gli adattamenti, a causa della resistenza psicologica al fatto di scartare quanto già pianificato e per il modo in cui le scelte in termini di architettura influenzano ogni successiva riflessione riguardante la struttura.

Gli architetti veri e propri *devono* sviluppare il proprio lavoro in termini BDUF, perché non è possibile apportare radicali cambiamenti a un progetto architettonico e quindi a una struttura fisica una volta che la costruzione è in corso (vi è comunque una significativa quantità di esplorazione iterativa e di discussione dei dettagli, anche dopo l'avvio della costruzione). Anche se il software ha una sua “fisicità” (il termine

fisicità è stato usato da [Kolence]), spesso consente di intervenire radicalmente sulla struttura, sempre che tale struttura separi efficacemente i suoi ambiti.

Significa che possiamo avviare un progetto software con un'architettura “ingenua” ma ben disaccoppiata, implementando rapidamente le richieste degli utenti e poi aggiungendo una maggior dose di infrastruttura mentre il progetto cresce di dimensioni. Alcuni dei più grandi siti web del mondo hanno ottenuto elevate prestazioni in termini di disponibilità e prestazioni, usando tecniche sofisticate di cache dei dati, di sicurezza, di virtualizzazione e così via, il tutto in modo efficiente e flessibile perché la loro struttura ad accoppiamento minimo era adeguatamente semplice a ogni livello di astrazione e visibilità.

Naturalmente, questo non significa che ci buttiamo in un progetto “senza direzione”. Abbiamo alcune aspettative in termini di ampiezza, obiettivi e tempi, come pure in termini di struttura generale del sistema. Tuttavia, dobbiamo mantenere la capacità di cambiare rotta in risposta alle circostanze. L'architettura EJB degli albori è solo una delle tante API che, pur ben progettate, compromettevano la separazione degli ambiti. Anche le API ben progettate possono essere fin esagerate quando non sono davvero necessarie. Una buona API dovrebbe essere quasi “invisibile” la maggior parte delle volte, in modo che il team dedichi il proprio impegno creativo all'implementazione delle richieste del cliente. In caso contrario, i vincoli imposti dall'architettura inibiranno la possibilità di fornire al cliente un oggetto ottimale. Per ricapitolare...

Un'architettura di sistema ottimale è costituita da ambiti modularizzati, ognuno dei quali è *implementato* con oggetti POJO (Plain Old Java Objects). I vari domini vengono integrati fra loro impiegano strumenti Aspects minimamente invasivi. Questa architettura può anche essere test-driven, come il codice.

Ottimizzazione delle decisioni

La modularità e la separazione degli ambiti rendono possibili la gestione decentralizzata e la capacità decisionale. In un sistema di dimensioni non banali, che si tratti di una città o un progetto software, non esiste un'unica persona in grado di prendere tutte le decisioni.

Sappiamo bene che è meglio dare la responsabilità alle persone più qualificate. Spesso però ci dimentichiamo che è anche meglio postporre le decisioni fino all'ultimo momento possibile. Questo non è un atteggiamento pigro o irresponsabile; ci consente di prendere decisioni informate con le migliori informazioni possibili. Una decisione prematura è una decisione presa sulla base di conoscenze non ottimali. Avremo a disposizione un feedback più limitato dai clienti, meno riflessioni sul progetto e meno esperienza con le nostre scelte implementative.

L'agilità garantita da un sistema POJO con ambiti modularizzati ci consente di prendere decisioni ottimali, *just-in-time*, basate sulle conoscenze più recenti. Si riduce anche la complessità di queste decisioni.

Usate gli standard con cura, solo se dimostrano il proprio valore

La costruzione di edifici è uno spettacolo per gli occhi, data la velocità con cui vengono edificati nuovi palazzi (anche in pieno inverno) e anche perché le nuove tecnologie rendono possibili delle costruzioni ardite. L'edilizia è un settore maturo con parti, metodi e standard ben ottimizzati, che si sono evoluti nell'arco di secoli.

Molti team usavano l'architettura EJB2 perché era uno standard, anche quando sarebbe stato sufficiente impiegare strutture più leggere e semplici. Ho visto team talmente ossessionati dagli standard da

dimenticare che l’obiettivo è implementare un valore per i propri clienti.

Gli standard facilitano il riutilizzo di idee e componenti, l’assunzione di personale con un’esperienza mirata, l’incapsulazione di buone idee e la connessione fra componenti. Tuttavia, il processo di creazione degli standard talvolta impone tempi di attesa troppo lunghi e alcuni standard perdono contatto con le reali esigenze di coloro che li adotteranno e che dovrebbero rappresentare il loro scopo.

I sistemi richiedono l’impiego di linguaggi specifici del dominio

La costruzione di edifici, come la maggior parte dei domini, ha sviluppato un ricco linguaggio, con un vocabolario, degli idiom e dei pattern (il lavoro di [Alexander] ha avuto una grande influenza sulla comunità software) in grado di fornire le informazioni essenziali in modo chiaro e conciso. Nel software, recentemente vi è un rinnovato interesse nella creazione di linguaggi specifici di un dominio (DSL – Domain Specific Language) (vedi, per esempio, [DSL]; [JMock] è un buon esempio di API Java that crea un linguaggio DSL), sotto forma di piccoli linguaggi script o API per linguaggi standard che permettono di scrivere il codice in modo che possa essere “letto” quasi come se fosse una prosa da un esperto del dominio.

Un buon DSL minimizza il “gap comunicativo” fra un concetto del dominio e il codice che lo implementa, così come le tecniche agili ottimizzano le comunicazioni in un team e con i finanziatori del progetto. Se riuscite a implementare la logica del dominio nello stesso linguaggio impiegato da un esperto del dominio, ridurrete i rischi di tradurre erroneamente il dominio nell’implementazione.

I linguaggi DSL, se usati in modo efficace, elevano il livello di astrazione sopra gli idiom del codice e gli schemi progettuali.

Consentono allo sviluppatore di rivelare lo scopo del codice al livello di astrazione appropriato.

I linguaggi specifici di un dominio consentono di esprimere tutti i livelli di astrazione e tutti i domini dell'applicazione sotto forma di POJO, dalla politica di alto livello fino ai più bassi dettagli implementativi.

Conclusioni

Anche i sistemi devono essere “puliti”. Un’architettura invasiva cela la logica del dominio e ha un impatto negativo sull’agilità. Quando la logica del dominio è offuscata, quella che soffre è la qualità, perché lì trovano spazio per nascondersi i bug e le “storie” divengono più difficili da implementare. Compromettendo l’agilità, ne soffre la produttività e si perdono i vantaggi dello sviluppo TDD (Test-Driven Development).

A tutti i livelli di astrazione, lo scopo dovrebbe essere chiaro. Questo è possibile solo se scrivete del codice POJO e usate i meccanismi della programmazione ad aspetto per incorporare in modo non invasivo gli altri ambiti dell’implementazione.

Che stiate progettando sistemi o singoli moduli, non dimenticatevi mai di usare le cose più semplici che siano in grado di funzionare.

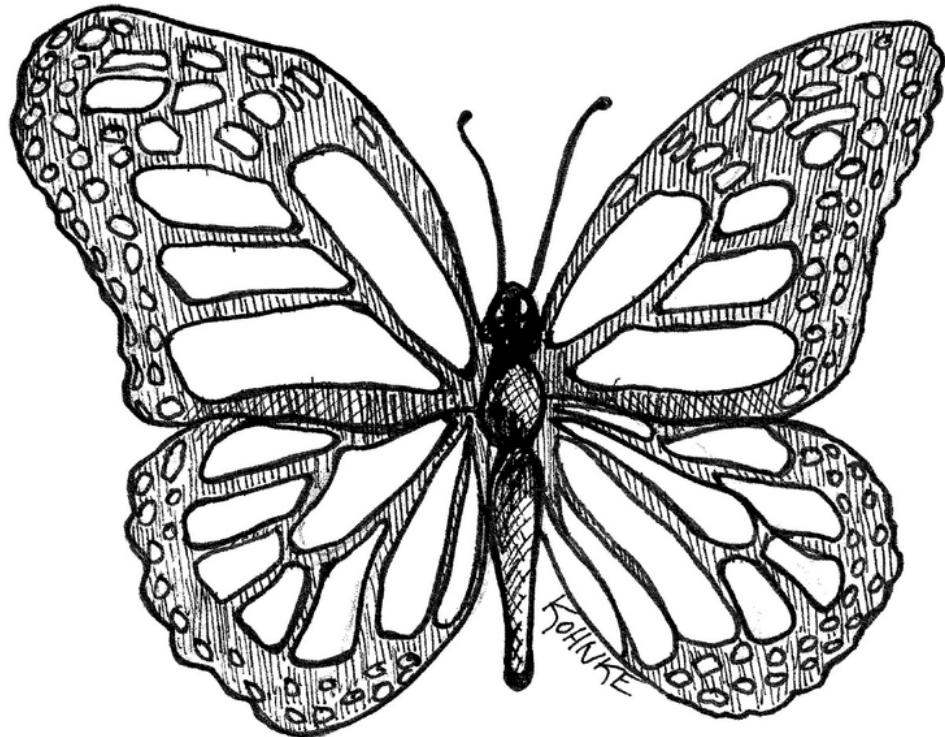
Bibliografia

- [Alexander]: Christopher Alexander, A Timeless Way of Building, Oxford University Press, New York 1979.
- [AOSD]: Aspect-Oriented Software Development port, <http://aosd.net>
- [ASM]: ASM Home Page, <http://asm.objectweb.org/>
- [AspectJ]: <http://eclipse.org/aspectj>

- [CGLIB]: Code Generation Library, <http://cglib.sourceforge.net/>
- [Colyer]: Adrian Colyer, Andy Clement, George Hurley, Mathew Webster, Eclipse AspectJ, Person Education, Inc., Upper Saddle River, New Jersey 2005.
- [DSL]: Domain-specific programming language,
http://en.wikipedia.org/wiki/Domain-specific_programming_language
- [Fowler]: Inversion of Control Containers and the Dependency Injection pattern, <http://martinfowler.com/articles/injection.html>
- [Goetz]: Brian Goetz, *Java Theory and Practice: Decorating with Dynamic Proxies*, <http://www.ibm.com/developerworks/java/library/j-jtp08305.html>
- [Javassist]: Javassist Home Page,
<http://www.csg.is.titech.ac.jp/~chiba/javassist/>
- [JBoss]: JBoss Home Page, <http://jboss.org>
- [JMock]: JMock – A Lightweight Mock Object Library for Java,
<http://jmock.org>
- [Kolence]: Kenneth W. Kolence, “Software physics and computer performance measurements”, in *Proceedings of the ACM annual conference*, Volume 2, Boston, Massachusetts 1972, pp. 1024-1040.
- [Spring]: The Spring Framework, <http://www.springframework.org>
- [Mezzaros07]: Gerard Mezzaros, XUnit Patterns, Addison-Wesley, Boston 2007.
- [GOF]: Gamma et al., Design Patterns: Elements of Reusable Object Oriented Software, Addison-Wesley, Boston 1996.

Capitolo 12

Simple Design *di Jeff Langr*



Come far emergere un codice pulito

Esistono quattro semplici regole da seguire per poter creare buoni progetti? E se queste regole fossero utili per migliorare la struttura del codice, facilitando l'applicazione di principi come quello dell'unica

responsabilità (SRP) e quello dell'inversione delle dipendenze (DIP)? E se queste quattro regole facilitassero l'emergere di buoni progetti?

Molti di noi ritengono che le quattro regole di Kent Beck di Simple Design [XPE] siano di grande aiuto nella creazione di software ben progettato.

Secondo Kent, un progetto è “semplice” se adotta le seguenti regole.

- Passa tutti i test.
- Non contiene duplicazioni.
- Esprime lo scopo del programmatore.
- Minimizza il numero di classi e metodi.

Le regole sono elencate in ordine di importanza.

Regola 1 di Simple Design – Passa tutti i test

Innanzitutto, un progetto deve produrre un sistema che operi come previsto. Un sistema può anche essere perfetto su carta, ma se non vi è alcun modo semplice per verificare che operi nel modo previsto, allora tutto quell'impegno di progettazione è di dubbio valore.

Un sistema che è ampiamente collaudato e passa sempre tutti i suoi test è un sistema collaudabile. Può sembrare ovvio, ma è un'affermazione importante. I sistemi che non sono collaudabili non possono essere verificati. E un sistema che non può essere verificato non dovrebbe mai essere messo in opera.

Fortunatamente, il fatto stesso di rendere collaudabili i nostri sistemi favorisce una progettazione in cui le nostre classi sono piccole e svolgono un unico compito. È più facile sottoporre a test delle classi che seguono il principio SRP. Più test scriviamo, più procediamo verso cose facili da collaudare. Pertanto il fatto stesso di assicurarci che il

nostro sistema sia completamente collaudabile ci aiuta a creare progetti migliori.

Uno stretto accoppiamento complica la creazione di test. Pertanto, ancora una volta, più test scriviamo e più usiamo principi progettuali come il DIP e strumenti come la *Dependency Injection*, le interfacce e l’astrazione per ridurre al minimo l’accoppiamento. Così i nostri progetti migliorano sempre più.

È interessante notare che basta una regola semplice e ovvia che dice che dobbiamo prevedere dei test ed eseguirli continuativamente per avere un impatto sull’aderenza del sistema agli obiettivi principali della programmazione OO: basso accoppiamento ed elevata coesione. Il solo fatto di scrivere test migliora i nostri progetti.

Regole 2-4 di Simple Design – Refactoring

Una volta che abbiamo i test, abbiamo la possibilità di mantenere pulito il nostro codice e le nostre classi. Per farlo eseguiamo un refactoring incrementale del codice. Ogni volta che aggiungiamo qualche riga di codice, ci fermiamo a riflettere sulla nuova struttura del codice. Abbiamo causato un degrado? In caso affermativo, lo ripuliamo ed eseguiamo i nostri test per verificare che tutto funzioni. *La presenza di questi test elimina il timore che le attività di pulizia del codice non abbiano causato problemi!*

Durante questo passo di refactoring, possiamo applicare tutto quello che conosciamo sulla buona progettazione del software. Possiamo migliorare la coesione, ridurre l’accoppiamento, separare gli ambiti, modularizzare gli ambiti del sistema, far “dimagrire” le nostre funzioni e classi, scegliere nomi migliori e così via. Questa è anche la fase in cui applichiamo le ultime tre regole di Simple Design: eliminare le duplicazioni, garantire l’espressività e ridurre al minimo il numero di classi e metodi.

Niente duplicazione

La duplicazione è il nemico numero uno di un sistema ben progettato. Introduce inutilmente più lavoro, più rischi e più complessità. La duplicazione si manifesta in varie forme. Le righe di codice esattamente identiche, naturalmente, sono un esempio plateale di duplicazione. Le righe di codice simili, spesso possono essere rese ancora più simili, al punto da consentirne il refactoring. Ma la duplicazione può esistere anche in altre forme, come la duplicazione di implementazioni. Per esempio, in una classe `collection` potremmo avere i due metodi seguenti:

```
int size() {}

boolean isEmpty() {}
```

con implementazioni distinte per ciascun metodo. Il metodo `isEmpty` potrebbe restituire un valore booleano, mentre `size` potrebbe restituire un contatore. Possiamo eliminare questa duplicazione collegando `isEmpty` alla definizione di `size`:

```
boolean isEmpty() {
    return 0 == size();
}
```

Ma per creare un sistema pulito dobbiamo eliminare ogni duplicazione, anche solo di poche righe di codice. Per esempio, considerate il seguente codice.

```
public void scaleToOneDimension(float desiredDimension, float imageDimension) {
    if (Math.abs(desiredDimension - imageDimension) < errorThreshold)
        return;
    float scalingFactor = desiredDimension / imageDimension;
    scalingFactor = (float)(Math.floor(scalingFactor * 100) * 0.01f);
    RenderedOp newImage = ImageUtilities.getScaledImage(
        image, scalingFactor, scalingFactor);
    image.dispose();
    System.gc();
    image = newImage;
}

public synchronized void rotate(int degrees) {
    RenderedOp newImage = ImageUtilities.getRotatedImage(
        image, degrees);
    image.dispose();
    System.gc();
    image = newImage;
```

```
}
```

Per garantire la pulizia del sistema, dovremmo eliminare anche la piccola duplicazione fra i metodi `scaleToOneDimension` e `rotate`:

```
public void scaleToOneDimension(float desiredDimension, float imageDimension) {
    if (Math.abs(desiredDimension - imageDimension) < errorThreshold)
        return;
    float scalingFactor = desiredDimension / imageDimension;
    scalingFactor = (float)(Math.floor(scalingFactor * 100) * 0.01f);
    replaceImage(ImageUtilities.getScaledImage(
        image, scalingFactor, scalingFactor));
}
public synchronized void rotate(int degrees) {
    replaceImage(ImageUtilities.getRotatedImage(image, degrees));
}
private void replaceImage(RenderedOp newImage) {
    image.dispose();
    System.gc();
    image = newImage;
}
```

Estraendo gli elementi comuni a un livello così minuto, iniziamo a riconoscere le violazioni del principio SRP. Pertanto potremmo estrarre un metodo in un'altra classe. Questo ne eleva la visibilità. Qualcun altro, nel team, potrebbe riconoscere l'opportunità di astrarre ulteriormente il nuovo metodo e riutilizzarlo in un altro contesto. Questo “riutilizzo anche nelle piccole cose” può ridurre notevolmente la complessità del sistema. Capire come riutilizzare le piccole cose è fondamentale per capire come riutilizzare anche le grandi cose.

Il pattern TEMPLATE METHOD [GOF] è una tecnica molto utilizzata per eliminare la duplicazione ad alto livello. Per esempio:

```
public class VacationPolicy {
    public void accrueUSDivisionVacation() {
        // codice per calcolare le vacanze sulla base del numero di ore lavorate
        // ...
        // codice per garantire il minimo di legge USA in giorni di vacanza
        // ...
        // codice per considerare il periodo di vacanza nella paga
        // ...
    }
    public void accrueEUDivisionVacation() {
        // codice per calcolare le vacanze sulla base del numero di ore lavorate
        // ...
        // codice per garantire il minimo di legge EU in giorni di vacanza
        // ...
        // codice per considerare il periodo di vacanza nella paga
        // ...
    }
}
```

}

Il codice di `accrueUSDivisionVacation` e `accrueEuropeanDivisionVacation` è sostanzialmente lo stesso, con l'eccezione del calcolo dei minimi di legge. Quella parte dell'algoritmo cambia sulla base del tipo di dipendente.

Possiamo eliminare tale duplicazione applicando il pattern TEMPLATE METHOD.

```
abstract public class VacationPolicy {  
    public void accrueVacation() {  
        calculateBaseVacationHours();  
        alterForLegalMinimums();  
        applyToPayroll();  
    }  
    private void calculateBaseVacationHours() { /* ... */ };  
    abstract protected void alterForLegalMinimums();  
    private void applyToPayroll() { /* ... */ };  
}  
public class USVacationPolicy extends VacationPolicy {  
    @Override protected void alterForLegalMinimums() {  
        // Caso US: logica specifica  
    }  
}  
public class EUVacationPolicy extends VacationPolicy {  
    @Override protected void alterForLegalMinimums() {  
        // Caso EU: logica specifica  
    }  
}
```

Le sottoclassi riempiono i “buchi” dell’algoritmo di `accrueVacation`, fornendo le uniche informazioni che non sono duplicate.

Espressività

La maggior parte di noi ha sicuramente avuto la brutta esperienza di lavorare su codice molto involuto. Molti di noi hanno anche prodotto codice della cui pulizia non andavano troppo fieri. È facile scrivere del codice che ci sembra di capire, perché al momento in cui lo scriviamo abbiamo ben presente il problema che stiamo risolvendo. Ma le altre persone che lavorano sullo stesso codice potrebbero non contare sulle stesse conoscenze.

La maggior parte del costo di un progetto software è la sua manutenzione a lungo termine. Per ridurre al minimo l'insorgere di difetti mentre introduciamo nuove modifiche, è fondamentale che siamo in grado di comprendere che cosa fa un sistema. Aumentando la complessità dei sistemi, uno sviluppatore avrà bisogno di sempre più tempo per comprenderlo e cresce anche la probabilità di un'errata interpretazione del codice. Pertanto, il codice deve esprimere chiaramente lo scopo del suo autore. Più l'autore riesce a curare la chiarezza del codice, meno tempo impiegheranno gli altri per comprenderne il funzionamento. Questo riduce i difetti e anche i costi di manutenzione.

Potete curare l'espressività scegliendo buoni nomi. Vogliamo essere in grado di capire che cosa fa una classe o una funzione e non scoprire con sorpresa quali sono le sue responsabilità.

Potete curare l'espressività anche mantenendo piccole le vostre funzioni e classi. Classi e funzioni più piccole sono solitamente più facili da denominare, da scrivere e da comprendere.

Potete curare l'espressività anche usando sistemi di denominazione standard. I pattern progettuali, per esempio, pongono molto l'accento sulla comunicazione e l'espressività. Usando nomi standard, come COMMAND o VISITOR, nei nomi delle classi che implementano tali pattern, potete descrivere in forma sintetica la vostra idea agli altri sviluppatori.

Gli unit test ben scritti sono anch'essi espressivi. Un obiettivo importante dei test è quello di fungere da documentazione basata sull'esempio. Chi leggerà i nostri test dovrà essere in grado di capire a che cosa serve una determinata classe.

Ma il modo più importante per dimostrare l'espressività è quello di *provare*. Troppo spesso ci accontentiamo che il nostro codice funzioni e poi passiamo al problema successivo senza dedicare un tempo

sufficiente a garantire che tale codice sia comprensibile da parte di chi lo leggerà. Ma ricordate: la persona che con più probabilità si troverà a leggere il vostro codice siete proprio voi.

Pertanto dedicate al lavoro il tempo che merita. Passate qualche istante di più su ognuna delle vostre funzioni e classi. Scegliete nomi migliori, suddividete le funzioni troppo grandi in funzioni più piccole e, in generale, prendetevi cura di ciò che avete creato. È una risorsa preziosa.

Minimizzare classi e metodi

Anche concetti fondamentali come l'eliminazione delle duplicazioni, l'espressività del codice e il principio SRP possono essere presi fin troppo alla lettera. Nel tentativo di compattare le nostre classi e i nostri metodi, potremmo creare classi e metodi troppo minuscoli. Pertanto questa regola suggerisce anche di contenere il numero di funzioni e classi.

Un numero eccessivo di classi e metodi talvolta è il risultato di un eccesso di dogmatismo. Considerate, per esempio, uno standard di programmazione che insiste nel creare un'interfaccia per ogni singola classe. O considerate quegli sviluppatori che insistono sul fatto che i campi e i comportamenti vengano sempre separati in classi per i dati e classi per i comportamenti. Dale dogma dovrebbe essere abbandonato, in nome di un approccio più pragmatico.

Il nostro obiettivo è quello di garantire la compattezza del sistema, mantenendo piccole anche le nostre funzioni e classi. Ricordate, tuttavia, che questa regola quella che ha la priorità più bassa delle quattro regole di *Simple Design*. Pertanto, sebbene sia importante contenere il numero di classi e funzioni, è più importante curare i test, eliminare le duplicazioni e curare l'espressività.

Conclusioni

Esiste un insieme di semplici tecniche in grado di sostituire l'esperienza? Chiaramente, no. D'altra parte, le tecniche descritte in questo capitolo e in questo libro sono come una forma cristallizzata dei molti decenni di esperienza dei suoi autori. Seguendo le tecniche di *Simple Design* vi incoraggiamo ad aderire a buoni principi e schemi progettuali il cui apprendimento richiederebbe anni.

Bibliografia

- [XPE]: Kent Beck, Extreme Programming Explained: Embrace Change, Addison- Wesley, Boston 1999.
- [GOF]: Gamma et al., Design Patterns: Elements of Reusable Object Oriented Software, Addison-Wesley, Boston 1996.

Capitolo 13

Concorrenza *di Brett L. Schuchert*

Gli oggetti sono le astrazioni dell’elaborazione. I thread sono le astrazioni dello scheduling.

James O. Coplien (corrispondenza privata)



Scrivere programmi concorrenti in modo “pulito” è difficile, molto difficile. È molto più facile scrivere codice che operi in un unico thread. È anche facile scrivere codice multi-thread che sembri

apparentemente “buono”, ma che manifesti problemi a un livello più profondo. Il codice che si comporta perfettamente finché il sistema non viene posto sotto stress.

In questo capitolo parleremo delle esigenze della programmazione concorrente e le difficoltà che essa introduce. Presenteremo poi vari consigli utili per affrontare tali difficoltà e scrivere codice concorrente pulito. Infine, concluderemo con alcuni argomenti legati al testing del codice concorrente.

La concorrenza “pulita” è un argomento complesso, che meriterebbe un libro a sé. La nostra strategia in questo libro è di presentare qui una panoramica, che riprenderemo in modo più dettagliato nell’Appendice A, “Concorrenza II”. Se siete solo curiosi sull’argomento concorrenza, questo capitolo sarà sufficiente per le vostre esigenze. Se invece avete la necessità di comprendere la concorrenza a un livello più profondo, vi rimandiamo all’Appendice A.

A che cosa serve la concorrenza?

La concorrenza è una strategia di disaccoppiamento. Ci aiuta a disaccoppiare *che cosa* viene fatto da *quando* viene fatto. Nelle applicazioni mono-thread, *che cosa* e *quando* sono così strettamente accoppiate che spesso lo stato dell’intera applicazione può essere determinato semplicemente osservando lo stack. Un programmatore che esegua il debug di un sistema di tale tipo può impostare un breakpoint o una sequenza di breakpoint e conoscere lo stato del sistema in base ai breakpoint raggiunti.

Il disaccoppiamento del *che cosa* dal *quando* può migliorare radicalmente sia la produttività, sia la struttura di un’applicazione. Dal punto di vista strutturale, l’applicazione assume l’aspetto di tanti piccoli computer che collaborano invece di un unico grosso ciclo.

Questo può rendere il sistema più facile da comprendere e offre alcune tecniche davvero potenti per separare gli ambiti.

Considerate, per esempio, il modello standard a “Servlet” delle applicazioni web. Questi sistemi operano sotto l’ombrellino di un container web o EJB che gestisce, *parzialmente*, la concorrenza al posto vostro. I servlet vengono eseguiti in modo asincrono ogni volta che giunge una richiesta dal Web. Il programmatore di servlet non deve preoccuparsi di gestire tutte le richieste in arrivo. *In linea di principio*, l’esecuzione di ogni servlet opera in un suo piccolo mondo ed è disaccoppiata rispetto a quelle di ogni altro servlet.

Naturalmente se tutto fosse così facile, questo capitolo non sarebbe necessario. In effetti, il disaccoppiamento garantito dai container web è tutt’altro che perfetto. I programmatori di servlet devono procedere con molta cautela e devono fare attenzione ad assicurarsi che i loro programmi concorrenti siano corretti. Ciononostante, i vantaggi strutturali del modello a servlet sono significativi.

Ma la struttura delle applicazioni non è l’unico motivo che spinge ad adottare la concorrenza. Alcuni sistemi hanno tempi di risposta e vincoli di throughput (“produttività”) che richiedono l’impiego di soluzioni concorrenti “manuali”. Per esempio, considerate un aggregatore di informazioni mono-thread che acquisisca informazioni da più siti web e che riunisca tali informazioni in un riepilogo quotidiano. Poiché questo sistema è mono-thread, raggiunge un sito web alla volta, e termina il precedente prima di passare al successivo. L’esecuzione giornaliera richiede meno di 24 ore. Tuttavia, dal momento che vengono aggiunti sempre più siti web, i tempi si allungano, fino a superare le 24 ore per la sola raccolta di tutti i dati. Quell’unico thread incorre in lunghi tempi di attesa ai socket web, in attesa del completamento delle operazioni di I/O. Potremmo migliorare

le prestazioni usando un algoritmo multi-thread che interroga più di un sito web per volta.

O possiamo considerare un sistema che gestisca un utente alla volta, richiedendo un solo secondo per utente. Questo sistema risulta piuttosto reattivo per alcuni utenti, ma aumentando il numero di utenti, i tempi di risposta del sistema si allungano. Nessun utente ama trovarsi in coda dopo altri 150! Potremmo migliorare il tempo di risposta di questo sistema gestendo più utenti in concorrenza.

Oppure considerate un sistema che interpreti grosse masse di dati ma che possa fornire una soluzione solo dopo averle elaborate tutte. Forse ogni gruppo di dati potrebbe essere elaborato da un computer distinto, in modo che in parallelo possano essere elaborati più gruppi di dati contemporaneamente.

Miti e fraintendimenti

Come abbiamo visto, vi sono motivazioni molto reali per adottare la concorrenza. Tuttavia, come abbiamo detto, la concorrenza è difficile. Se non si fa attenzione, si possono sviluppare situazioni davvero intricate. Considerate questi miti e fraintendimenti molto comuni.

- *La concorrenza migliora sempre le prestazioni.*
Talvolta la concorrenza può migliorare le prestazioni, ma solo quando vi è una quota importante di tempo di attesa che può essere condivisa fra più thread o più CPU. Nessuna di queste situazioni è banale.
- *La struttura dell'applicazione non cambia quando si scrivono programmi concorrenti.*
In realtà, la struttura di un algoritmo concorrente può essere radicalmente differente da quella di un sistema mono-thread. Il

disaccoppiamento del *che cosa* dal *quando* solitamente ha un effetto notevole sulla struttura del sistema.

- *Conoscere gli aspetti della concorrenza non è importante quando si lavora con un container web o EJB.*

In realtà, fareste bene a scoprire che cosa sta facendo il container e a imparare a proteggersi dai problemi dell'aggiornamento concorrente e dei *deadlock* descritti in questo capitolo.

Ecco invece alcune affermazioni più equilibrate riguardanti la scrittura di software concorrente.

- *La concorrenza introduce un certo sovraccarico*, in termini sia prestazionali sia di codice aggiuntivo.
- *La concorrenza, se gestita correttamente*, è complessa, anche per problemi semplici.
- *I bug introdotti dalla concorrenza sono difficilmente ripetibili*, pertanto spesso vengono considerati semplici situazioni sporadiche (raggi cosmici, problemi tecnici e così via) invece dei difetti che in realtà sono.
- *La concorrenza spesso richiede un cambiamento radicale nella strategia di progettazione.*

Sfide

Che cosa rende così difficile la programmazione concorrente?

Considerate la seguente classe, banalissima:

```
public class X {  
    private int lastIdUsed;  
    public int getNextId() {  
        return ++lastIdUsed;  
    }  
}
```

Immaginiamo di creare un'istanza di `x`, di impostare il campo `lastIdUsed` a 42 e poi di condividere l'istanza fra i due thread. Ora

supponiamo che entrambi i thread richiamino il metodo `getNextId()`; vi sono tre possibili risultati.

- Il thread uno ottiene il valore 43, il thread due ottiene il valore 44, `lastIdUsed` vale 44.
- Il thread uno ottiene il valore 44, il thread due ottiene il valore 43, `lastIdUsed` vale 44.
- Il thread uno ottiene il valore 43, il thread due ottiene il valore 43, `lastIdUsed` vale 43.

Il terzo risultato (vedi “Approfondimento”, nell’Appendice A), sorprendente, si verifica quando i due thread procedono insieme. Questo si verifica perché vi sono più percorsi che possono essere seguiti dai due thread per eseguire quell’unica riga di codice Java e alcuni di essi generano risultati errati. Quanti percorsi vi sono? Per rispondere davvero a tale domanda, dobbiamo comprendere che cosa fa il compilatore Just-In-Time con il byte-code generato e capire anche che cosa viene considerato “atomico” dal modello di memoria Java.

Una risposta rapida, considerando solo il byte-code generato, è che vi sono ben 12.870 percorsi di esecuzione possibili (vedi “Percorsi possibili di esecuzione”, nell’Appendice A) per questi due thread in esecuzione nel metodo `getNextId`. Se il tipo di `lastIdUsed` cambia da `int` a `long`, il numero di percorsi possibili incrementa a 2.704.156.

Naturalmente la maggior parte di questi percorsi genera risultati validi. Il problema è che *alcuni di essi no...*

Concorrenza: principi di difesa

Quella che segue è una serie di principi e tecniche per aiutarvi a difendere i vostri sistemi dai problemi introdotti dal codice concorrente.

Il principio SRP (Single Responsibility Principle)

Il principio SRP [PPP] stabilisce che un determinato metodo/classe/componente dovrebbe avere un unico motivo per dover cambiare. La struttura che genera la concorrenza è un elemento sufficientemente complesso per essere, per l'appunto, un motivo per dover apportare modifiche e pertanto merita di essere separata dal resto del codice. Sfortunatamente, troppo spesso i dettagli implementativi della concorrenza si trovano incorporati direttamente insieme ad altro codice di produzione. Ecco alcuni elementi da considerare.

- Il codice che gestisce la concorrenza ha un suo proprio ciclo di vita di sviluppo, modifica e ottimizzazione.
- Il codice che gestisce la concorrenza ha le sue proprie sfide, che sono differenti e spesso più difficili di quelle del codice non concorrente.
- Il numero di problemi in cui può incorrere del codice di concorrenza mal realizzato è già di per sé notevole, senza aggiungergli il carico del codice dell'applicazione, circostante.

SUGGERIMENTO

Mantenete il codice che gestisce la concorrenza separato dall'altro codice (vedi “Un esempio client/server”, nell'Appendice A).

Corollario: limitate il livello di visibilità (scope) dei dati

Come abbiamo visto, due thread che modificano lo stesso campo di un oggetto condiviso possono interferire fra loro, e il loro comportamento è imprevedibile. Una soluzione consiste nell'usare la parola chiave `synchronized` per proteggere una *sezione critica* del codice che impiega l'oggetto condiviso. È importante limitare il numero di tali

sezioni critiche. Più sono i punti in cui vengono aggiornati i dati condivisi, più probabilmente:

- dimenticherete di proteggere uno o più di questi punti, pregiudicando il funzionamento di tutto il codice che modifica tali dati condivisi;
- vi sarà una duplicazione degli sforzi per assicurarvi che tutto sia controllato in modo efficace (violazione del principio DRY, *Don't Repeat Yourself*) [PRAG];
- sarà difficile determinare l'origine dei fallimenti, che già normalmente è difficile da trovare.

SUGGERIMENTO

Curate l'incapsulazione dei dati; limitate il più possibile l'accesso ai dati che possono dover essere condivisi.

Corollario: usate copie dei dati

Un buon modo per evitare l'uso di dati condivisi consiste nell'evitare di condividerli. In alcune situazioni è possibile copiare gli oggetti e trattarli in modo *read-only*. In altri casi deve essere possibile copiare gli oggetti, raccogliere in queste copie i risultati forniti da più thread e poi unire i risultati in un unico thread.

Se vi è un modo semplice per evitare di condividere degli oggetti, il codice risultante causerà probabilmente molti meno problemi. Potreste temere il costo di creazione di tutti questi oggetti. Vale la pena di sperimentare per scoprire se questo è davvero un problema. Tuttavia, se l'uso di copie degli oggetti consente al codice di evitare la sincronizzazione, il risparmio dovuto al fatto di evitare il relativo bloccaggio intrinseco probabilmente supererà di gran lunga l'impegno dovuto all'attività di creazione e di successiva eliminazione delle “scorie”.

Corollario: i thread dovrebbero essere il più possibile indipendenti

Scrivete il vostro codice a thread in modo che ogni thread operi in un proprio mondo, che non condivide dati con nessun altro thread. Ogni thread elabora una sola richiesta client, dove tutti i dati necessari provengono da una risorsa non condivisa e vengono memorizzati in variabili locali. Questo fa sì che ognuno di questi thread si comporti come se fosse l'unico thread al mondo e come se non vi fossero requisiti di sincronizzazione.

Per esempio, delle classi derivate da `HttpServlet` ricevono tutte le informazioni necessarie sotto forma di parametri passati ai metodi `doGet` e `doPost`. Questo fa sì che ogni `Servlet` si comporti come se avesse a disposizione una propria macchina. Pertanto, se il codice del `Servlet` impiega solo variabili locali, non vi è alcuna possibilità che il `Servlet` provochi problemi di sincronizzazione. Naturalmente, la maggior parte delle applicazioni che usa i `Servlet` alla fine dovrà operare su risorse condivise, per esempio un database.

SUGGERIMENTO

Cercate di partizionare i dati in sottoinsiemi indipendenti che possano essere gestiti da thread indipendenti, possibilmente su CPU differenti.

Studiate la vostra libreria

Java 5 offre molti miglioramenti per lo sviluppo concorrente rispetto alle versioni precedenti. Vi sono vari aspetti da considerare quando si scrive del codice a thread in Java 5.

- Usate le apposite collection thread-safe.
- Usate il framework `Executor` per l'esecuzione di task non correlati.
- Quando possibile usate soluzioni senza blocaggi.

- Molte classi della libreria non sono thread-safe.

Collection thread-safe

Ai primi tempi di Java, Doug Lea scrisse un libro fondativo: *Concurrent Programming in Java* [Lea99]. Insieme al libro sviluppò varie *collection thread-safe*, che successivamente divennero parte del JDK nel package `java.util.concurrent`. Le collection di tale package sono sicure ed efficaci per le situazioni multi-thread. Addirittura, l'implementazione di `ConcurrentHashMap` si comporta meglio di `HashMap` in quasi tutte le situazioni. Consente anche di eseguire letture e scritture concorrenti in simultanea e offre dei metodi che supportano le più comuni operazioni composite che altrimenti non sarebbero thread-safe. Se l'ambiente di deployment è Java 5, iniziate con `ConcurrentHashMap`.

Vi sono molti altri tipi di classi, aggiunte per supportare le funzionalità avanzate di concorrenza. Ecco alcuni esempi.

<code>ReentrantLock</code>	Un bloccaggio che può essere attivato in un metodo e rilasciato in un altro.
<code>Semaphore</code>	Un'implementazione di un classico meccanismo a semaforo, un sistema di bloccaggio a contatore.
<code>CountDownLatch</code>	Un bloccaggio che attende un certo numero di eventi prima di rilasciare tutti i thread in attesa. Questo consente a tutti i thread di avere una buona possibilità di partire più o meno contemporaneamente.

SUGGERIMENTO

Studiate le classi disponibili. Nel caso di Java, studiate `java.util.concurrent`, `java.util.concurrent.atomic` e `java.util.concurrent.locks`.

Studiate i modelli di esecuzione

Vi sono vari modi per controllare i comportamenti in un'applicazione concorrente. Per poterne parlare dobbiamo conoscere alcune

definizioni di base.

Bound Resources	Risorse di dimensioni fisse o in quantità fissata usate in un ambiente concorrente. Esempi: connessioni a database e buffer di lettura/scrittura di dimensioni fisse.
Mutual Exclusion	Un solo thread per volta può accedere ai dati condivisi o alla risorsa condivisa.
Starvation	A un thread o a un gruppo di thread viene impedito di proseguire per un tempo eccessivamente lungo o per sempre. Per esempio, il fatto di consentire sempre ai thread rapidi di passare davanti può impedire l'esecuzione ai thread lenti, se i thread rapidi non finiscono mai.
Deadlock	Due (o più) thread attendono il termine dell'altro. Ogni thread ha una risorsa richiesta dall'altro e nessuno dei due (o più) può procedere finché non ottiene la risorsa occupata dall'altro.
Livelock	I thread sono bloccati: ognuno di essi tenta di procedere, ma non può perché trova sempre l'altro "fra le scatole". Si innesca una risonanza: i thread continuano a tentare di progredire, ma restano bloccati per un tempo eccessivamente lungo, o per sempre.

Sulla base di queste definizioni, possiamo ora parlare dei vari modelli di esecuzione usati nella programmazione concorrente.

Produttori-Consumatori

Uno o più thread produttori creano del lavoro e lo collocano in un buffer o una coda. Uno o più thread consumatori acquisiscono tale lavoro dalla coda e lo completano. La coda fra i produttori e i consumatori è una *bound resource*. Questo significa che i produttori devono attendere che si liberi dello spazio nella coda prima di scrivervi e che i consumatori devono attendere che vi sia qualcosa nella coda per consumarla. La coordinazione fra i produttori e i consumatori attraverso la coda richiede che i produttori e i consumatori si inviano segnali. I produttori scrivono sulla coda e segnalano che la coda non è più vuota. I consumatori leggono dalla coda e segnalano che la coda non è più piena. Potenzialmente entrambi attendono una notifica per poter procedere.

(https://it.wikipedia.org/wiki/Problema_del_prodottore/consumatore).

Lettori-Scrittori

Quando vi è una risorsa condivisa che funge principalmente da fonte di informazioni per i lettori, ma che talvolta viene aggiornata dagli scrittori, il *throughput* (la “produttività”) diviene problematico.

Spingere sul *throughput* può far ricadere nella *starvation* e nell’accumulo di informazioni obsolete. Privilegiare gli aggiornamenti può avere un impatto sul *throughput*. Coordinare i lettori in modo che non leggano qualcosa che uno scrittore sta aggiornando (e viceversa) è un’opera di bilanciamento drastica. Gli scrittori tendono a bloccare più lettori per molto tempo, a scapito del *throughput*.

La sfida consiste nell’equilibrare le esigenze dei lettori e degli scrittori per soddisfare le esigenze dell’applicazione, per garantire un *throughput* ragionevole e per evitare la *starvation*. Una semplice strategia chiede agli scrittori di attendere finché non vi sono lettori, prima di consentire allo scrittore di eseguire un aggiornamento. Se però continuano a esservi lettori, gli scrittori entreranno in *starvation*.

D’altra parte, se vi sono troppi scrittori e hanno sempre la priorità, si ha un calo di *throughput*. Il problema, quindi, è trovare tale equilibrio ed evitare i problemi di aggiornamento concorrente.

(http://en.wikipedia.org/wiki/Readers-writers_problem).

La cena dei filosofi

Immaginate che un certo numero di filosofi siano seduti a una tavola rotonda. A destra di ogni filosofo si trova una forchetta. Al centro della tavola si trova una grossa ciotola di spaghetti. I filosofi trascorrono il loro tempo a pensare, ma a un certo punto hanno fame. Quando hanno fame, prendono le forchette a destra e a sinistra e iniziano a mangiare. Ma un filosofo può mangiare solo se ha due forchette. Se il filosofo alla sua destra o alla sua sinistra sta già usando una delle forchette, dovrà

attendere finché tale filosofo non avrà finito di mangiare e riporrà le forchette a lato. Quando un filosofo ha finito di mangiare, ripone entrambe le forchette ai due lati, sulla tavola, e attende di avere ancora fame.

Al posto dei filosofi mettete i thread e al posto delle forchette mettete le risorse e questo problema diventa simile a molte applicazioni in cui i processi entrano in competizione per l'accesso alle risorse. Se non vengono programmati con attenzione, questi sistemi possono andare incontro a deadlock, a livelock, e a un degrado del throughput e dell'efficienza.

La maggior parte dei problemi di concorrenza che vi troverete ad affrontare sarà probabilmente una variante di questi tre problemi. Studiate questi algoritmi e scrivete delle soluzioni che li usano, in modo che quando incontrerete questo genere di problemi, sarete più preparati a risolverli. (https://it.wikipedia.org/wiki/Problema_dei_filosofi_a_cena).

SUGGERIMENTO

Studiate questi algoritmi di base e le loro soluzioni.

Attenzione alle dipendenze fra metodi sincronizzati

Le dipendenze fra metodi sincronizzati provocano l'insorgere di subdoli bug nel codice concorrente. Il linguaggio Java ha il concetto di `synchronized`, che protegge un singolo metodo. Tuttavia, se in una classe vi è più di un metodo sincronizzato, significa che il sistema è scritto in modo errato (vedi “Le dipendenze fra i metodi possono pregiudicare il funzionamento del codice concorrente”, nell’Appendice A).

SUGGERIMENTO

Evitate di usare più di un metodo su un oggetto condiviso.

Vi saranno situazioni in cui dovete usare più di un metodo su un oggetto condiviso. In questo caso, vi sono tre modi per correggere il codice.

- *Client-Based Locking* – Il client blocca il server prima di richiamare il primo metodo e si assicura che il bloccaggio comprenda il codice che richiama l'ultimo metodo.
- *Server-Based Locking* - Nel server, create un metodo che blocchi il server, richiami tutti i metodi e poi lo sblocchi. Poi il client potrà richiamare il nuovo metodo.
- *Adapted Server* - Create un intermediario che applichi il bloccaggio. Questo è un esempio di *server-based locking*, in cui non si può intervenire sul server originario.

Riducete al minimo le sezioni sincronizzate

La parola chiave `synchronized` introduce un blocco. Tutte le sezioni del codice controllate dallo stesso blocco hanno la garanzia di avere un solo thread interno in esecuzione in un determinato momento. I blocchi sono costosi perché introducono ritardi. Pertanto non vogliamo congestionare il nostro codice con istruzioni `synchronized`. D'altra parte, le sezioni critiche devono essere protette (una sezione critica è qualsiasi sezione di codice che deve essere protetta dall'uso concorrente affinché il programma sia corretto). Pertanto vogliamo progettare il nostro codice con il minor numero possibile di sezioni critiche.

Alcuni programmatori alle prime armi fanno in modo che le loro sezioni critiche siano molto estese. Tuttavia, il fatto di estendere la sincronizzazione oltre il minimo necessario aumenta le contese e

introduce un degrado prestazionale (vedi “Migliorare la produttività (throughput)”, nell’Appendice A).

SUGGERIMENTO

Fate in modo che le sezioni sincronizzate siano le più piccole possibili.

Scrivere il codice di chiusura corretto è difficile

Scrivere un sistema che debba rimanere attivo e funzionante per sempre non è la stessa cosa di scrivere qualcosa che funziona per un po’ e poi termina adeguatamente.

Una corretta chiusura può essere difficile da ottenere. Fra i problemi più comuni vi sono i deadlock (vedi “Deadlock”, nell’Appendice A), dove i thread restano bloccati in attesa di un segnale che non arriva mai.

Per esempio, immaginate un sistema con un thread genitore che genera più thread figli e poi attende il loro completamento prima di rilasciare le risorse impegnate e chiudersi. E se uno dei thread generati fosse bloccato in un deadlock? Il genitore attenderà per sempre e il sistema non si chiuderà mai.

O considerate un sistema simile, al quale sia stato chiesto di chiudersi. Il genitore dice a tutti i figli che ha generato di abbandonare le attività e di terminare. Ma cosa accadrebbe se due di questi figli fossero legati in una coppia produttore/consumatore? Supponete che il produttore riceva il segnale dal genitore e si chiuda rapidamente. Il consumatore attenderà invano un messaggio dal produttore e rimarrà bloccato in uno stato in cui non può ricevere il segnale di chiusura. Rimarrà in attesa del produttore e non si chiuderà mai, bloccando nel contempo anche la chiusura del genitore.

Situazioni come questa non sono affatto insolite. Pertanto se dovete scrivere del codice concorrente che preveda una chiusura composta,

preparatevi a dedicare del tempo ad assicurarvi che la chiusura possa avvenire correttamente.

SUGGERIMENTO

Iniziate presto a ragionare e anche a lavorare sul codice di chiusura. Vi richiederà più tempo di quanto possiate immaginare. Considerate anche gli algoritmi usati, perché questo compito è probabilmente più difficile del preventivato.

Collaudo del codice di un thread

Non è affatto facile dimostrare la correttezza del codice. Il collaudo non garantisce la correttezza. Tuttavia, la disponibilità di buoni test aiuta a ridurre al minimo i rischi. Questo vale per una soluzione mono-thread. Ma quando vi sono due o più thread che usano lo stesso codice e operano su dati condivisi, le cose si complicano notevolmente.

SUGGERIMENTO

Scrivete dei test in grado di evidenziare i problemi e poi lanciateli frequentemente, con diverse configurazioni del programma e del sistema e carichi di lavoro. Se i test falliscono, trovatene la causa. Non ignorate un fallimento solo perché i test sono passati a una successiva esecuzione.

Ci sono molte cose da tenere in considerazione. Ecco alcune raccomandazioni dettagliate.

- Trattate i fallimenti occasionali come possibili problemi dei thread.
- Curate innanzitutto il funzionamento del codice mono-thread.
- Rendete versatile il codice a thread. >
- Rendete configurabile il codice a thread.
- Provate a generare più thread delle CPU disponibili.
- Eseguite i test su piattaforme differenti.
- Mettete alla prova il vostro codice per provare a forzare i fallimenti.

Trattate i fallimenti occasionali come possibili problemi dei thread

Il codice a thread provoca il fallimento anche di elementi “assolutamente sicuri”. La maggior parte degli sviluppatori (compresi noi autori) non ha un’idea esatta di come i thread possono interagire con l’altro codice. I bug nel codice a thread possono manifestare i loro sintomi una volta in un migliaio o in un milione di esecuzioni. I tentativi di replicare i sistemi possono essere davvero frustranti. Questo spesso porta gli sviluppatori ad attribuire i problemi ai raggi cosmici, a un problema hardware o a qualche altro evento fortuito. È bene fare finta che questi eventi fortuiti non esistano. Più a lungo vengono ignorati, maggiore sarà la quantità di codice costruito su un approccio potenzialmente difettoso.

SUGGERIMENTO

Non ignorate i difetti del sistema e neppure i cosiddetti “eventi fortuiti”.

Curate innanzitutto il funzionamento del codice mono-thread

Questo può sembrare ovvio, ma vale la pena di ribadirlo. Assicuratevi che il codice funzioni indipendentemente dal suo uso nei thread. Generalmente, questo significa creare dei POJO che vengono richiamati dai thread. I POJO sono estranei ai thread e pertanto possono essere sottoposti a test all’esterno dell’ambiente dei thread. Più potete collocare il sistema in questi POJO e meglio sarà.

SUGGERIMENTO

Non tentate di risolvere contemporaneamente i bug del codice a thread e di quello mono-thread. Assicuratevi che il vostro codice funzioni indipendentemente dai thread.

Rendete versatile il codice a thread

Scrivete il codice che supporta la concorrenza in modo che possa essere eseguito in più configurazioni:

- un thread, più thread, variando le cose;
- il codice a thread interagisce con qualcosa che può essere reale o realizzato per un test;
- predisponete test fittizi che operino rapidamente, lentamente, a velocità variabile;
- configurate i test in modo da poter svolgere un certo numero di iterazioni.

SUGGERIMENTO

Curate la versatilità del codice a thread, in modo da poterlo eseguire con varie configurazioni.

Rendete configurabile il codice a thread

Trovare l'equilibrio corretto nei thread richiede varie prove. Fin da subito, trovate dei modi per cronometrare le prestazioni del vostro sistema in configurazioni differenti. Fate in modo che il numero di threads possa essere agevolmente regolato. Considerate la possibilità di cambiare le cose anche mentre il sistema è in funzione. Considerate anche le possibilità di auto-regolazione del sistema in base al throughput e al livello di carico.

Provate a generare più thread delle CPU disponibili

Alcune cose si verificano quando il sistema esegue una commutazione di task. Per favorire tali commutazioni, eseguite più thread delle CPU o dei core disponibili. Più frequentemente verrà

eseguita una commutazione di task, più probabilmente vi accorgerete che del codice salta una sezione critica o entra in un deadlock.

Eseguite i test su piattaforme differenti

A metà del 2007 abbiamo tenuto un corso sulla programmazione concorrente. Il corso si svolgeva principalmente in OS X. Le lezioni venivano tenute usando un sistema Windows XP operante in una VM. I test scritti per dimostrare le condizioni di fallimento, fallivano molto meno in ambiente XP che in OS X.

In entrambe le situazioni il codice sottoposto a test era errato. Questo ribadiva il fatto che sistemi operativi differenti possono imporre politiche differenti ai thread, le quali hanno un impatto sull'esecuzione del codice. Il codice multi-thread si comporta in modo differente in ambienti differenti. (Lo sapevate che il modello di thread di Java non garantisce il threading preemptive? Il supporto degli odierni sistemi operativi offre il threading preemptive, quindi lo potete avere “gratuitamente”. Ciononostante, non è garantito da JVM.) Dovreste pertanto eseguire i vostri test in ogni possibile ambiente di deployment.

SUGGERIMENTO

Eseguite il vostro codice a thread su tutte le piattaforme possibili, fin da subito e spesso.

Mettete alla prova il vostro codice per provare a forzare i fallimenti

È solo normale che i difetti presenti nel codice concorrente tendano a nascondersi. I semplici test spesso non sono sufficienti a evidenziarli. In effetti, spesso rimangono nascosti durante la normale elaborazione. Possono però manifestarsi ogni *tot* ore o giorni o settimane!

Il motivo per il quale i bug legati ai thread possono essere infrequenti, sporadici e difficilmente replicabili, è che solo pochissimi percorsi di esecuzione delle molte migliaia di percorsi possibili in una sezione vulnerabile del codice possono avere problemi. Pertanto la probabilità di riuscire a seguire un percorso fallimentare può essere irrisoria. Questo può rendere molto difficoltoso il rilevamento e il debugging.

Come è possibile aumentare le possibilità di ricreare evenienze così rare? Potete mettere alla prova il vostro codice e costringerlo a operare in modi differenti aggiungendo chiamate a metodi come `Object.wait()`,

`Object.sleep()`, `Object.yield()` e `Object.priority()`.

Ognuno di questi metodi può alterare l'ordine di esecuzione, aumentando di conseguenza le probabilità di rilevare un difetto. È meglio che il codice manifesti i suoi problemi il più precocemente e frequentemente possibile. Vi sono due opzioni per mettere alla prova il codice:

- test manuali;
- test automatici.

Test manuali

Potete inserire nel vostro codice delle chiamate manuali a `wait()`, `sleep()`, `yield()` e `priority()`. È il caso di farlo quando dovete sottoporre a test un frammento di codice particolarmente ostico.

Ecco un esempio di questo tipo:

```
public synchronized String nextUrlOrNull() {  
    if(hasNext()) {  
        String url = urlGenerator.next();  
        Thread.yield(); // inserito per i test.  
        updateHasNext();  
        return url;  
    }  
    return null;  
}
```

La chiamata a `yield()` cambierà i percorsi di esecuzione seguiti dal codice e potrebbe far fallire il codice laddove non aveva manifestato problemi. Se il codice si blocca, non è per l'inserimento della chiamata a `yield()`. (Non è esattamente così. Poiché la JVM non garantisce il threading preemptive, un particolare algoritmo potrebbe sempre funzionare su un sistema operativo non preemptive. È possibile anche il contrario, ma per ragioni differenti.) Piuttosto, il vostro codice aveva un difetto e, semplicemente, la chiamata gli ha consentito di manifestarsi.

Vi sono alcuni problemi in questo approccio.

- Occorre trovare manualmente i punti appropriati del codice.
- Come sapere dove inserire la chiamata e quale chiamata usare?
- Lasciando questo codice nell'ambiente di produzione lo si rallenta inutilmente.
- È un approccio che conta troppo sulla fortuna. Potreste anche non trovare nessuno dei difetti presenti. In effetti, non avete dalla vostra molte probabilità di trovarli.

Ciò di cui abbiamo bisogno è un modo per operare durante i test ma non in produzione. Dobbiamo anche poter cambiare le configurazioni durante le varie esecuzioni, per aumentare la probabilità totale di trovare errori.

Chiaramente, se dividiamo il nostro sistema in più POJO che non sanno nulla dei thread e delle classi che controllano i thread, sarà più facile trovare i punti appropriati in cui mettere alla prova il codice. Inoltre, potremmo creare più test che richiamano i POJO in situazioni differenti.

Test automatici

Potete usare strumenti come un framework Aspect-Oriented, CGLIB o ASM per mettere alla prova da programma il vostro codice. Per esempio, potete usare una classe con un unico metodo:

```
public class ThreadJigglePoint {  
    public static void jiggle() {  
    }  
}
```

Potete poi aggiungere delle chiamate in vari punti del vostro codice:

```
public synchronized String nextUrlOrNull() {  
    if(hasNext()) {  
        ThreadJigglePoint.jiggle();  
        String url = urlGenerator.next();  
        ThreadJigglePoint.jiggle();  
        updateHasNext();  
        ThreadJigglePoint.jiggle();  
        return url;  
    }  
    return null;  
}
```

Ora avete a disposizione un semplice aspetto che seleziona casualmente vari comportamenti: nulla, `sleep()` o `yield()`.

O immaginate che la classe `ThreadJigglePoint` abbia due implementazioni. La prima implementa `jiggle` in modo che non faccia nulla e viene usata in produzione. La seconda genera un numero casuale per scegliere fra `sleep()`, `yield()` o nulla. Se eseguite i vostri test un migliaio di volte con questa casualità, potreste riuscire a ottenere i difetti. Se i test vengono passati, quanto meno potete dire di averci provato con diligenza. Sebbene sia un po' semplicistica, questa potrebbe essere un'opzione ragionevole, al posto di uno strumento più sofisticato.

Esiste uno strumento chiamato ConTest (<http://www.alphaworks.ibm.com/tech/contest>), sviluppato da IBM, che fa qualcosa di simile, ma in modo un po' più sofisticato.

Il punto è che occorre variare le cose in modo che i thread operino in tempi e sequenze differenti. La combinazione di test ben scritti e di

queste variabilità può aumentare enormemente la probabilità di individuare gli errori.

SUGGERIMENTO

Usate queste strategie di variabilità per individuare gli errori.

Conclusioni

Il codice concorrente è difficile da far funzionare. Anche quel codice che sembra semplice da seguire può diventare un incubo quando entrano in gioco i thread e i dati condivisi. Se vi capita di dover scrivere del codice concorrente, curate al massimo la pulizia del codice o incorrerete in frequenti e subdoli problemi.

Innanzitutto, seguite sempre il principio SRP (*Single Responsibility Principle*). Suddividete il vostro sistema in più POJO che separano il codice a thread dal resto. Quando sottponete a test il vostro codice a thread, assicuratevi che non venga impiegato nient’altro. Questo vuole anche dire che il vostro codice a thread dovrebbe essere piccolo e mirato.

Studiate tutti i possibili problemi dovuti alla concorrenza: considerate più thread che operano su dati condivisi o su una risorsa comune. I casi limite, come il momento della chiusura o la fine dell’iterazione di un ciclo, possono essere particolarmente problematici.

Studiate la libreria che impiegate e i principali algoritmi. Alcune delle funzionalità offerte dalla libreria supportano proprio la soluzione di problemi tipici degli algoritmi fondamentali.

Imparate a individuare quelle regioni del codice che devono essere bloccate e poi bloccate. Non bloccate, invece quelle regioni del codice che non ne hanno necessità. Evitate di richiamare una sezione bloccata da un’altra. Questo richiede una conoscenza approfondita di cosa è condiviso e cosa no. Riducete il più possibile la quantità di

oggetti condivisi e il loro livello di visibilità (scope). Modificate la struttura degli oggetti con dati condivisi in base alle esigenze dei client, invece di costringere i client a gestire uno stato condiviso.

I problemi emergeranno. Quelli che non emergono subito potrebbero emergere anche una sola volta. Questi eventi “fortuiti” in genere si verificano solo sotto carico o a intervalli apparentemente casuali. Pertanto, dovete essere in grado di eseguire il codice a thread in più configurazioni e su più piattaforme, ripetutamente e continuamente. La collaudabilità, che è il frutto naturale delle tre leggi dello sviluppo TDD (Test-Driven Development), implica un certo livello di versatilità, la quale permette di eseguire il codice in un’ampia gamma di configurazioni.

Potete migliorare notevolmente le probabilità di trovare errori se vi preoccuperete di mettere alla prova il vostro codice. Potete riuscirci manualmente o usando una tecnologia automatizzata. Ma dedicatevi il più presto possibile a questa attività. Volete mettere alla prova il codice a thread molto prima che entri in produzione.

Se adotterete un approccio “pulito”, le probabilità di riuscita cresceranno enormemente.

Bibliografia

- [Lea99]: Doug Lea, Concurrent Programming in Java: Design Principles and Patterns, 2d. ed., Prentice Hall, Upper Saddle River, New Jersey 1999.
- [PPP]: Robert C. Martin, Agile Software Development: Principles, Patterns, and Practices, Prentice Hall, Upper Saddle River, New Jersey 2002.
- [PRAG]: Andrew Hunt, Dave Thomas, The Pragmatic Programmer, Addison-Wesley, Boston 2000. Edizione italiana, Il>

Pragmatic Programmer, Apogeo, Milano 2018.

Raffinamento progressivo (caso di studio di un parser di argomenti da riga di comando)



Questo capitolo è un caso di studio riguardante un raffinamento progressivo. Sotto esame è un modulo che è partito anche bene, ma non

permetteva estensioni di scala. Poi vedrete come il modulo è stato rifattorizzato e ripulito.

A molti di noi capita ogni tanto di dover analizzare argomenti inviati dalla riga di comando. Se non abbiamo uno strumento adeguato, dobbiamo semplicemente attraversare l'array di stringhe passato alla funzione `main`. Per farlo esistono molti ottimi strumenti di servizio, da più fonti, ma nessuno di essi faceva esattamente quello che volevo. Pertanto, naturalmente, ho deciso di scriverne uno mio. L'ho chiamato `Args`.

`Args` è molto semplice da usare. Basta costruire la classe `Args` con gli argomenti di input e una stringa di formattazione e poi interrogare l'istanza di `Args` chiedendole il valore degli argomenti. Considerate il seguente semplice esempio:

Listato 14.1 Semplice uso di `Args`.

```
public static void main(String[] args) {
    try {
        Args arg = new Args("l,p#,d*", args);
        boolean logging = arg.getBoolean('l');
        int port = arg.getInt('p');
        String directory = arg.getString('d');
        executeApplication(logging, port, directory);
    } catch (ArgsException e) {
        System.out.printf("Argument error: %s\n", e.errorMessage());
    }
}
```

Potete vedere quanto sia semplice. Abbiamo semplicemente creato un'istanza della classe `Args` con due parametri. Il primo parametro è la stringa di formato, o schema: `"l,p#,d*"`. Essa definisce tre argomenti inviati dalla riga di comando. Il primo, `-l`, è un argomento booleano. Il secondo, `-p`, è un argomento intero. Il terzo, `-d`, è un argomento stringa. Il secondo parametro del costruttore di `Args` è semplicemente l'array degli argomenti della riga di comando passati a `main`.

Se il costruttore si chiude senza lanciare una `ArgsException`, significa che la riga di comando in arrivo è stata sottoposta con successo a

parsing e l'istanza di `Args` è pronta per essere interrogata. Metodi come `getBoolean`, `getInteger` e `getString` ci consentono di accedere al valore degli argomenti in base al loro nome.

Se vi è un problema, nella stringa di formattazione o negli argomenti inviati dalla riga di comando, viene lanciata una `ArgsException`. Per ottenere una descrizione del problema si può usare il metodo `errorMessage` dell'eccezione.

Implementazione di Args

Il Listato 14.2 è l'implementazione della classe `Args`. Vi prego di leggerla con molta attenzione. Ho fatto estrema attenzione allo stile e alla struttura e spero che valga la pena di emularli.

Listato 14.2 `Args.java`.

```
package com.objectmentor.utilities.args;
import static com.objectmentor.utilities.args.ArgsException.ErrorCode.*;
import java.util.*;
public class Args {
    private Map<Character, ArgumentMarshaler> marshalers;
    private Set<Character> argsFound;
    private ListIterator<String> currentArgument;
    public Args(String schema, String[] args) throws ArgsException {
        marshalers = new HashMap<Character, ArgumentMarshaler>();
        argsFound = new HashSet<Character>();
        parseSchema(schema);
        parseArgumentStrings(Arrays.asList(args));
    }
    private void parseSchema(String schema) throws ArgsException {
        for (String element : schema.split(","))
            if (element.length() > 0)
                parseSchemaElement(element.trim());
    }
    private void parseSchemaElement(String element) throws ArgsException {
        char elementId = element.charAt(0);
        String elementTail = element.substring(1);
        validateSchemaElementId(elementId);
        if (elementTail.length() == 0)
            marshalers.put(elementId, new BooleanArgumentMarshaler());
        else if (elementTail.equals("*"))
            marshalers.put(elementId, new StringArgumentMarshaler());
        else if (elementTail.equals("#"))
            marshalers.put(elementId, new IntegerArgumentMarshaler());
        else if (elementTail.equals("##"))
            marshalers.put(elementId, new DoubleArgumentMarshaler());
        else if (elementTail.equals("[*]"))
            marshalers.put(elementId, new StringArrayArgumentMarshaler());
    }
}
```

```

        else
            throw new ArgsException(INVALID_ARGUMENT_FORMAT, elementId, elementTail);
    }
    private void validateSchemaElementId(char elementId) throws ArgsException {
        if (!Character.isLetter(elementId))
            throw new ArgsException(INVALID_ARGUMENT_NAME, elementId, null);
    }
    private void parseArgumentStrings(List<String> argsList) throws ArgsException
    {
        for (currentArgument = argsList.listIterator(); currentArgument.hasNext();)
        {
            String argString = currentArgument.next();
            if (argString.startsWith("-"))
                parseArgumentCharacters(argString.substring(1));
            } else {
                currentArgument.previous();
                break;
            }
        }
    }
    private void parseArgumentCharacters(String argChars) throws ArgsException {
        for (int i = 0; i < argChars.length(); i++)
            parseArgumentCharacter(argChars.charAt(i));
    }
    private void parseArgumentCharacter(char argChar) throws ArgsException {
        ArgumentMarshaler m = marshalers.get(argChar);
        if (m == null)
            throw new ArgsException(UNEXPECTED_ARGUMENT, argChar, null);
        } else {
            argsFound.add(argChar);
            try {
                m.set(currentArgument);
            } catch (ArgsException e) {
                e.setErrorArgumentId(argChar);
                throw e;
            }
        }
    }
    public boolean has(char arg) {
        return argsFound.contains(arg);
    }
    public int nextArgument() {
        return currentArgument.nextInt();
    }
    public boolean getBoolean(char arg) {
        return BooleanArgumentMarshaler.getValue(marshalers.get(arg));
    }
    public String getString(char arg) {
        return StringArgumentMarshaler.getValue(marshalers.get(arg));
    }
    public int getInt(char arg) {
        return IntegerArgumentMarshaler.getValue(marshalers.get(arg));
    }
    public double getDouble(char arg) {
        return DoubleArgumentMarshaler.getValue(marshalers.get(arg));
    }
    public String[] getStringArray(char arg) {
        return StringArrayArgumentMarshaler.getValue(marshalers.get(arg));
    }
}

```

Notate come sia possibile leggere questo codice da cima a fondo senza mai saltare o cercare nulla. L'unica cosa che si può dover andare a cercare in avanti è la definizione di `ArgumentMarshaler`, che ho tralasciato intenzionalmente. Dopo aver letto con attenzione questo codice, dovreste già sapere che cos'è l'interfaccia `ArgumentMarshaler` e che cosa fanno le sue derivate. Nei Listati da 14.3 a 14.6 ne presento alcune.

Listato 14.3 ArgumentMarshaler.java.

```
public interface ArgumentMarshaler {  
    void set(Iterator<String> currentArgument) throws ArgsException;  
}
```

Listato 14.4 BooleanArgumentMarshaler.java.

```
public class BooleanArgumentMarshaler implements ArgumentMarshaler {  
    private boolean booleanValue = false;  
    public void set(Iterator<String> currentArgument) throws ArgsException {  
        booleanValue = true;  
    }  
    public static boolean getValue(ArgumentMarshaler am) {  
        if (am != null && am instanceof BooleanArgumentMarshaler)  
            return ((BooleanArgumentMarshaler) am).booleanValue;  
        else  
            return false;  
    }  
}
```

Listato 14.5 StringArgumentMarshaler.java.

```
import static com.objectmentor.utilities.args.ArgsException.ErrorCode.*;  
public class StringArgumentMarshaler implements ArgumentMarshaler {  
    private String stringValue = "";  
    public void set(Iterator<String> currentArgument) throws ArgsException {  
        try {  
            stringValue = currentArgument.next();  
        } catch (NoSuchElementException e) {  
            throw new ArgsException(MISSING STRING);  
        }  
    }  
    public static String getValue(ArgumentMarshaler am) {  
        if (am != null && am instanceof StringArgumentMarshaler)  
            return ((StringArgumentMarshaler) am).stringValue;  
        else  
            return "";  
    }  
}
```

Listato 14.6 IntegerArgumentMarshaler.java.

```

import static com.objectmentor.utilities.args.ArgsException.ErrorCode.*;
public class IntegerArgumentMarshaler implements ArgumentMarshaler {
    private int intValue = 0;
    public void set(Iterator<String> currentArgument) throws ArgsException {
        String parameter = null;
        try {
            parameter = currentArgument.next();
            intValue = Integer.parseInt(parameter);
        } catch (NoSuchElementException e) {
            throw new ArgsException(MISSING_INTEGER);
        } catch (NumberFormatException e) {
            throw new ArgsException(INVALID_INTEGER, parameter);
        }
    }
    public static int getValue(ArgumentMarshaler am) {
        if (am != null && am instanceof IntegerArgumentMarshaler)
            return ((IntegerArgumentMarshaler) am).intValue;
        else
            return 0;
    }
}

```

Le altre classi derivate da `ArgumentMarshaler` replicano semplicemente questo schema per array di `double` e `String` e non farebbero altro che congestionare il capitolo. Ve le lascio da svolgere come esercizio.

Vi è un'altra cosa che può lasciarvi perplessi: la definizione delle costanti per i codici d'errore. Si trovano tutte nella classe `ArgsException` (Listato 14.7).

Listato 14.7 `ArgsException.java`.

```

import static com.objectmentor.utilities.args.ArgsException.ErrorCode.*;
public class ArgsException extends Exception {
    private char errorArgumentId = '\0';
    private String errorParameter = null;
    private ErrorCode errorCode = OK;
    public ArgsException() {}
    public ArgsException(String message) {super(message);}
    public ArgsException(ErrorCode errorCode) {
        this.errorCode = errorCode;
    }
    public ArgsException(ErrorCode errorCode, String errorParameter) {
        this.errorCode = errorCode;
        this.errorParameter = errorParameter;
    }
    public ArgsException(ErrorCode errorCode,
                        char errorArgumentId, String errorParameter) {
        this.errorCode = errorCode;
        this.errorParameter = errorParameter;
        this.errorArgumentId = errorArgumentId;
    }
    public char getErrorArgumentId() {
        return errorArgumentId;
    }
}

```

```

public void setErrorArgumentId(char errorArgumentId) {
    this.errorArgumentId = errorArgumentId;
}
public String getErrorParameter() {
    return errorParameter;
}
public void setErrorParameter(String errorParameter) {
    this.errorParameter = errorParameter;
}
public ErrorCode getErrorCode() {
    return errorCode;
}
public void setErrorCode(ErrorCode errorCode) {
    this.errorCode = errorCode;
}
public String errorMessage() {
    switch (errorCode) {
        case OK:
            return "TILT: Should not get here.";
        case UNEXPECTED_ARGUMENT:
            return String.format("Argument -%c unexpected.", errorArgumentId);
        case MISSING_STRING:
            return String.format("Could not find string parameter for -%c.",
                errorArgumentId);
        case INVALID_INTEGER:
            return String.format("Argument -%c expects an integer but was '%s'.",
                errorArgumentId, errorParameter);
        case MISSING_INTEGER:
            return String.format("Could not find integer parameter for -%c.",
                errorArgumentId);
        case INVALID_DOUBLE:
            return String.format("Argument -%c expects a double but was '%s'.",
                errorArgumentId, errorParameter);
        case MISSING_DOUBLE:
            return String.format("Could not find double parameter for -%c.",
                errorArgumentId);
        case INVALID_ARGUMENT_NAME:
            return String.format("'%c' is not a valid argument name.",
                errorArgumentId);
        case INVALID_ARGUMENT_FORMAT:
            return String.format("'%s' is not a valid argument format.",
                errorParameter);
    }
    return "";
}
public enum ErrorCode {
    OK, INVALID_ARGUMENT_FORMAT, UNEXPECTED_ARGUMENT, INVALID_ARGUMENT_NAME,
    MISSING_STRING,
    MISSING_INTEGER, INVALID_INTEGER,
    MISSING_DOUBLE, INVALID_DOUBLE
}

```

È impressionante notare quanto codice sia necessario per esprimere i dettagli di questo semplice concetto. Uno dei motivi è che stiamo usando un linguaggio particolarmente prolioso. Java, essendo un linguaggio a tipi statici, richiede molte parole per soddisfare il sistema

di gestione dei tipi. In un linguaggio come Ruby, Python o Smalltalk, questo programma sarebbe molto più compatto (recentemente ho riscritto questo modulo in Ruby. Le sue dimensioni sono state pari a un settimo e aveva una struttura un po' migliore).

Vi prego di leggere il codice ancora una volta. Dedicate particolare attenzione alla denominazione degli elementi, alle dimensioni delle funzioni e alla formattazione del codice. Se siete programmati esperti, potreste avere da ridire, qua e là, in termini di stile o strutture. Ma, in generale, spero che conveniate sul fatto che questo programma è ben scritto e ha una *struttura pulita*.

Per esempio, dovrebbe essere evidente come si aggiunge un nuovo tipo di argomento, per esempio una data o un numero complesso, e sarete d'accordo nel dire che tale aggiunta richiederebbe un impegno davvero minimo. In breve, basterebbe semplicemente produrre una nuova derivata di `ArgumentMarshaler`, una nuova funzione `getxxx` e una nuova istruzione `case` nella funzione `parseSchemaElement`. Probabilmente vi sarà anche un nuovo `ArgsException.ErrorCode` e un nuovo messaggio d'errore.

Come ci sono arrivato?

Non temete. Non ho scritto questo programma, semplicemente, dall'inizio alla fine nella sua attuale forma. Ma ciò che conta ancora di più è che non mi aspetto che voi siate in grado di scrivere programmi puliti ed eleganti in un solo passo. Se abbiamo imparato qualcosa nel corso degli ultimi due decenni, è che programmare è un'abilità più che una scienza. Per scrivere codice pulito, dovete partire dal codice mal realizzato e poi ripulirlo.

Questo non dovrebbe sorprendervi. Abbiamo imparato questa verità a scuola, quando i nostri insegnanti hanno tentato (solitamente invano) di farci scrivere prima "la brutta" dei nostri temi. Il processo, ci hanno detto, prevedeva la scrittura di una prima bozza, poi di una seconda

bozza, poi di molte altre bozze fino a ottenere la versione finale. La scrittura di un tema “pulito”, hanno tentato di insegnarci, è frutto di un *raffinamento progressivo*.

La maggior parte dei programmatori alle prime armi (e anche degli studenti) non segue questo consiglio con la dovuta attenzione. Sono convinti che l’obiettivo principale sia quello di far funzionare il programma. Quando “funziona”, passano al compito successivo, lasciando il programma “funzionante” nello stato che gli ha permesso di “funzionare”. La maggior parte dei programmatori esperti sa che questo comportamento ha un nome: *suicidio professionale*.

Args: “la brutta”

Il Listato 14.8 presenta una versione preliminare della classe `Args`. Diciamo che “funziona”. Ed è un groviglio.

Listato 14.8 `Args.java` (“la brutta”).

```
import java.text.ParseException;
import java.util.*;
public class Args {
    private String schema;
    private String[] args;
    private boolean valid = true;
    private Set<Character> unexpectedArguments = new TreeSet<Character>();
    private Map<Character, Boolean> booleanArgs =
        new HashMap<Character, Boolean>();
    private Map<Character, String> stringArgs = new HashMap<Character, String>();
    private Map<Character, Integer> intArgs = new HashMap<Character, Integer>();
    private Set<Character> argsFound = new HashSet<Character>();
    private int currentArgument;
    private char errorArgumentId = '\0';
    private String errorParameter = "TILT";
    private ErrorCode errorCode = ErrorCode.OK;
    private enum ErrorCode {
        OK, MISSING_STRING, MISSING_INTEGER, INVALID_INTEGER, UNEXPECTED_ARGUMENT}
    public Args(String schema, String[] args) throws ParseException {
        this.schema = schema;
        this.args = args;
        valid = parse();
    }
    private boolean parse() throws ParseException {
        if (schema.length() == 0 && args.length == 0)
            return true;
        parseSchema();
        try {
```

```

        parseArguments();
    } catch (ArgsException e) {
    }
    return valid;
}
private boolean parseSchema() throws ParseException {
    for (String element : schema.split(",")) {
        if (element.length() > 0) {
            String trimmedElement = element.trim();
            parseSchemaElement(trimmedElement);
        }
    }
    return true;
}
private void parseSchemaElement(String element) throws ParseException {
    char elementId = element.charAt(0);
    String elementTail = element.substring(1);
    validateSchemaElementId(elementId);
    if (isBooleanSchemaElement(elementTail))
        parseBooleanSchemaElement(elementId);
    else if (isStringSchemaElement(elementTail))
        parseStringSchemaElement(elementId);
    else if (isIntegerSchemaElement(elementTail)) {
        parseIntegerSchemaElement(elementId);
    } else {
        throw new ParseException(
            String.format("Argument: %c has invalid format: %s.", elementId,
                         elementTail), 0);
    }
}
private void validateSchemaElementId(char elementId) throws ParseException {
    if (!Character.isLetter(elementId)) {
        throw new ParseException(
            "Bad character:" + elementId + "in Args format: " + schema, 0);
    }
}
private void parseBooleanSchemaElement(char elementId) {
    booleanArgs.put(elementId, false);
}
private void parseIntegerSchemaElement(char elementId) {
    intArgs.put(elementId, 0);
}
private void parseStringSchemaElement(char elementId) {
    stringArgs.put(elementId, "");
}
private boolean isStringSchemaElement(String elementTail) {
    return elementTail.equals("*");
}
private boolean isBooleanSchemaElement(String elementTail) {
    return elementTail.length() == 0;
}
private boolean isIntegerSchemaElement(String elementTail) {
    return elementTail.equals("#");
}
private boolean parseArguments() throws ArgsException {
    for (currentArgument = 0; currentArgument < args.length; currentArgument++)
    {
        String arg = args[currentArgument];
        parseArgument(arg);
    }
    return true;
}

```

```

}

private void parseArgument(String arg) throws ArgsException {
    if (arg.startsWith("-"))
        parseElements(arg);
}

private void parseElements(String arg) throws ArgsException {
    for (int i = 1; i < arg.length(); i++)
        parseElement(arg.charAt(i));
}

private void parseElement(char argChar) throws ArgsException {
    if (setArgument(argChar))
        argsFound.add(argChar);
    else {
        unexpectedArguments.add(argChar);
        errorCode = ErrorCode.UNEXPECTED_ARGUMENT;
        valid = false;
    }
}

private boolean setArgument(char argChar) throws ArgsException {
    if (isBooleanArg(argChar))
        setBooleanArg(argChar, true);
    else if (isStringArg(argChar))
        setStringArg(argChar);
    else if (isIntArg(argChar))
        setIntArg(argChar);
    else
        return false;
    return true;
}

private boolean isIntArg(char argChar) {return intArgs.containsKey(argChar);}

private void setIntArg(char argChar) throws ArgsException {
    currentArgument++;
    String parameter = null;
    try {
        parameter = args[currentArgument];
        intArgs.put(argChar, new Integer(parameter));
    } catch (ArrayIndexOutOfBoundsException e) {
        valid = false;
        errorArgumentId = argChar;
        errorCode = ErrorCode.MISSING_INTEGER;
        throw new ArgsException();
    } catch (NumberFormatException e) {
        valid = false;
        errorArgumentId = argChar;
        errorParameter = parameter;
        errorCode = ErrorCode.INVALID_INTEGER;
        throw new ArgsException();
    }
}

private void setStringArg(char argChar) throws ArgsException {
    currentArgument++;
    try {
        stringArgs.put(argChar, args[currentArgument]);
    } catch (ArrayIndexOutOfBoundsException e) {
        valid = false;
        errorArgumentId = argChar;
        errorCode = ErrorCode.MISSING_STRING;
        throw new ArgsException();
    }
}

private boolean isStringArg(char argChar) {

```

```

        return stringArgs.containsKey(argChar);
    }
    private void setBooleanArg(char argChar, boolean value) {
        booleanArgs.put(argChar, value);
    }
    private boolean isBooleanArg(char argChar) {
        return booleanArgs.containsKey(argChar);
    }
    public int cardinality() {
        return argsFound.size();
    }
    public String usage() {
        if (schema.length() > 0)
            return "-[" + schema + "]";
        else
            return "";
    }
    public String errorMessage() throws Exception {
        switch (errorCode) {
            case OK:
                throw new Exception("TIILT: Should not get here.");
            case UNEXPECTED_ARGUMENT:
                return unexpectedArgumentMessage();
            case MISSING_STRING:
                return String.format("Could not find string parameter for -%c.",
                        errorArgumentId);
            case INVALID_INTEGER:
                return String.format("Argument -%c expects an integer but was '%s'.",
                        errorArgumentId, errorParameter);
            case MISSING_INTEGER:
                return String.format("Could not find integer parameter for -%c.",
                        errorArgumentId);
        }
        return "";
    }
    private String unexpectedArgumentMessage() {
        StringBuffer message = new StringBuffer("Argument(s) -");
        for (char c : unexpectedArguments) {
            message.append(c);
        }
        message.append(" unexpected.");
        return message.toString();
    }
    private boolean falseIfNull(Boolean b) {
        return b != null && b;
    }
    private int zeroIfNull(Integer i) {
        return i == null ? 0 : i;
    }
    private String blankIfNull(String s) {
        return s == null ? "" : s;
    }
    public String getString(char arg) {
        return blankIfNull(stringArgs.get(arg));
    }
    public int getInt(char arg) {
        return zeroIfNull(intArgs.get(arg));
    }
    public boolean getBoolean(char arg) {
        return falseIfNull(booleanArgs.get(arg));
    }
}

```

```

    public boolean has(char arg) {
        return argsFound.contains(arg);
    }
    public boolean isValid() {
        return valid;
    }
    private class ArgsException extends Exception {
    }
}

```

Spero che la vostra reazione iniziale a questo ammasso di codice sia: “Per fortuna che non l’ha lasciato così!”. Se la pensate così, provate a immaginare cosa mai potrà pensare di voi chi dovesse trovarsi a lavorare su del codice che avete lasciato in “brutta”, solo perché funzionava…

In realtà probabilmente “la brutta” è la cosa più gentile che potete dire di questo codice. Chiaramente è un passo (molto) provvisorio. Il numero di variabili di istanza è da incubo. Il tutto condito da stringhe ostiche come "TILT", da `HashSet` e `TreeSet` e da blocchi `try-catch-catch`.

Non volevo scrivere un tale ammasso di righe di codice. In effetti, ho anche cercato di curare un po’ l’organizzazione. Potete vederlo dalle mie scelte per i nomi delle funzioni e delle variabili e dal fatto che il programma ha una sua qual struttura. Ma, chiaramente, lo scopo evidente era quello di togliersi di torno sbrigativamente il problema.

Il groviglio si è sviluppato gradualmente. Le prime versioni non erano così brutte. Per esempio, il Listato 14.9 presenta una versione preliminare in cui funzionavano solo gli argomenti `Boolean`.

Listato 14.9 Args.java (solo Boolean).

```

package com.objectmentor.utilities.getopts;
import java.util.*;
public class Args {
    private String schema;
    private String[] args;
    private boolean valid;
    private Set<Character> unexpectedArguments = new TreeSet<Character>();
    private Map<Character, Boolean> booleanArgs =
        new HashMap<Character, Boolean>();
    private int numberofArguments = 0;
    public Args(String schema, String[] args) {
        this.schema = schema;
        this.args = args;
        valid = parse();
    }
}

```

```

    }
    public boolean isValid() {
        return valid;
    }
    private boolean parse() {
        if (schema.length() == 0 && args.length == 0)
            return true;
        parseSchema();
        parseArguments();
        return unexpectedArguments.size() == 0;
    }
    private boolean parseSchema() {
        for (String element : schema.split(",")) {
            parseSchemaElement(element);
        }
        return true;
    }
    private void parseSchemaElement(String element) {
        if (element.length() == 1) {
            parseBooleanSchemaElement(element);
        }
    }
    private void parseBooleanSchemaElement(String element) {
        char c = element.charAt(0);
        if (Character.isLetter(c)) {
            booleanArgs.put(c, false);
        }
    }
    private boolean parseArguments() {
        for (String arg : args) parseArgument(arg);
        return true;
    }
    private void parseArgument(String arg) {
        if (arg.startsWith("-")) parseElements(arg);
    }
    private void parseElements(String arg) {
        for (int i = 1; i < arg.length(); i++)
            parseElement(arg.charAt(i));
    }
    private void parseElement(char argChar) {
        if (isBoolean(argChar)) {
            numberOfArguments++;
            setBooleanArg(argChar, true);
        } else
            unexpectedArguments.add(argChar);
    }
    private void setBooleanArg(char argChar, boolean value) {
        booleanArgs.put(argChar, value);
    }
    private boolean isBoolean(char argChar) {
        return booleanArgs.containsKey(argChar);
    }
    public int cardinality() {
        return numberOfArguments;
    }
    public String usage() {
        if (schema.length() > 0)
            return "-["+schema+"]";
        else
            return "";
    }
}

```

```

public String errorMessage() {
    if (unexpectedArguments.size() > 0) {
        return unexpectedArgumentMessage();
    } else
        return "";
}
private String unexpectedArgumentMessage() {
    StringBuffer message = new StringBuffer("Argument(s) -");
    for (char c : unexpectedArguments) {
        message.append(c);
    }
    message.append(" unexpected.");
    return message.toString();
}
public boolean getBoolean(char arg) {
    return booleanArgs.get(arg);
}
}

```

Potete certamente trovare numerosi motivi per criticare questo codice, ma in realtà non è poi così male. È compatto, semplice e facile da comprendere. Tuttavia, in questo codice è facile vedere i semi del suo problema. È abbastanza evidente che questo codice non può crescere.

Notate che il prossimo passo aggiunge solo altri due tipi di argomenti: `String` e `integer`. L'aggiunta di due soli tipi di argomenti ha un impatto molto negativo sul codice. Ha trasformato un qualcosa che sarebbe stato ragionevolmente manutenibile in un qualcosa che è facile immaginare zeppo di bug e problemi.

Ho aggiunto i due tipi di argomenti incrementalmente. Innanzitutto ho aggiunto l'argomento `String`, ottenendo questo:

Listato 14.10 Args.java (Boolean e String).

```

package com.objectmentor.utilities.getopts;
import java.text.ParseException;
import java.util.*;
public class Args {
    private String schema;
    private String[] args;
    private boolean valid = true;
    private Set<Character> unexpectedArguments = new TreeSet<Character>();
    private Map<Character, Boolean> booleanArgs =
        new HashMap<Character, Boolean>();
    private Map<Character, String> stringArgs = new HashMap<Character, String>();
    private Set<Character> argsFound = new HashSet<Character>();
    private int currentArgument;
    private char errorArgument = '\0';
    enum ErrorCode {

```

```

    OK, MISSING_STRING}
private ErrorCode errorCode = ErrorCode.OK;
public Args(String schema, String[] args) throws ParseException {
    this.schema = schema;
    this.args = args;
    valid = parse();
}
private boolean parse() throws ParseException {
    if (schema.length() == 0 && args.length == 0)
        return true;
    parseSchema();
    parseArguments();
    return valid;
}
private boolean parseSchema() throws ParseException {
    for (String element : schema.split(",")) {
        if (element.length() > 0) {
            String trimmedElement = element.trim();
            parseSchemaElement(trimmedElement);
        }
    }
    return true;
}
private void parseSchemaElement(String element) throws ParseException {
    char elementId = element.charAt(0);
    String elementTail = element.substring(1);
    validateSchemaElementId(elementId);
    if (isBooleanSchemaElement(elementTail))
        parseBooleanSchemaElement(elementId);
    else if (isStringSchemaElement(elementTail))
        parseStringSchemaElement(elementId);
}
private void validateSchemaElementId(char elementId) throws ParseException {
    if (!Character.isLetter(elementId)) {
        throw new ParseException(
            "Bad character:" + elementId + "in Args format: " + schema, 0);
    }
}
private void parseStringSchemaElement(char elementId) {
    stringArgs.put(elementId, "");
}
private boolean isStringSchemaElement(String elementTail) {
    return elementTail.equals("*");
}
private boolean isBooleanSchemaElement(String elementTail) {
    return elementTail.length() == 0;
}
private void parseBooleanSchemaElement(char elementId) {
    booleanArgs.put(elementId, false);
}
private boolean parseArguments() {
    for (currentArgument = 0; currentArgument < args.length; currentArgument++)
    {
        String arg = args[currentArgument];
        parseArgument(arg);
    }
    return true;
}
private void parseArgument(String arg) {
    if (arg.startsWith("-"))
        parseElements(arg);
}

```

```

private void parseElements(String arg) {
    for (int i = 1; i < arg.length(); i++)
        parseElement(arg.charAt(i));
}
private void parseElement(char argChar) {
    if (setArgument(argChar))
        argsFound.add(argChar);
    else {
        unexpectedArguments.add(argChar);
        valid = false;
    }
}
private boolean setArgument(char argChar) {
    boolean set = true;
    if (isBoolean(argChar))
        setBooleanArg(argChar, true);
    else if (isString(argChar))
        setStringArg(argChar, "");
    else
        set = false;
    return set;
}
private void setStringArg(char argChar, String s) {
    currentArgument++;
    try {
        stringArgs.put(argChar, args[currentArgument]);
    } catch (ArrayIndexOutOfBoundsException e) {
        valid = false;
        errorArgument = argChar;
        errorCode = ErrorCode.MISSING_STRING;
    }
}
private boolean isString(char argChar) {
    return stringArgs.containsKey(argChar);
}
private void setBooleanArg(char argChar, boolean value) {
    booleanArgs.put(argChar, value);
}
private boolean isBoolean(char argChar) {
    return booleanArgs.containsKey(argChar);
}
public int cardinality() {
    return argsFound.size();
}
public String usage() {
    if (schema.length() > 0)
        return "-[" + schema + "]";
    else
        return "";
}
public String errorMessage() throws Exception {
    if (unexpectedArguments.size() > 0) {
        return unexpectedArgumentMessage();
    } else switch (errorCode) {
        case MISSING_STRING:
            return String.format("Could not find string parameter for -%c.",
                                errorArgument);
        case OK:
            throw new Exception("TILT: Should not get here.");
    }
    return "";
}

```

```

    }
    private String unexpectedArgumentMessage() {
        StringBuffer message = new StringBuffer("Argument(s) -");
        for (char c : unexpectedArguments) {
            message.append(c);
        }
        message.append(" unexpected.");
        return message.toString();
    }
    public boolean getBoolean(char arg) {
        return falseIfNull(booleanArgs.get(arg));
    }
    private boolean falseIfNull(Boolean b) {
        return b == null ? false : b;
    }
    public String getString(char arg) {
        return blankIfNull(stringArgs.get(arg));
    }
    private String blankIfNull(String s) {
        return s == null ? "" : s;
    }
    public boolean has(char arg) {
        return argsFound.contains(arg);
    }
    public boolean isValid() {
        return valid;
    }
}

```

Come potete vedere, le cose cominciano a sfuggire di mano. Non è ancora nulla di terribile, ma il problema sta sicuramente cominciando a crescere. È bastata l'aggiunta del tipo di argomento `integer` per assestargli il “colpo di grazia”.

E così mi sono fermato qui

Avevo almeno altri due tipi di argomento da aggiungere ed era evidente che le cose sarebbero peggiorate. Se mi ci fossi messo a capofitto, probabilmente sarei riuscito a farlo funzionare, ma avrei lasciato dietro di me un groviglio di codice quasi impossibile da sistemare. Se la struttura di questo codice doveva essere manutenibile, questo era il momento di cambiare rotta.

Pertanto ho smesso di aggiungere nuove funzionalità e ho iniziato il refactoring. Avendo appena aggiunto gli argomenti `String` e `integer`, sapevo che ogni tipo di argomento richiedeva l'aggiunta di nuovo

codice principalmente in tre punti. Innanzitutto, ogni tipo di argomento richiedeva un modo per eseguire il parsing del suo elemento `schema`, per selezionare l'`HashMap` di tale tipo. Poi, ogni tipo di argomento doveva essere sottoposto a parsing nella stringa della riga di comando e convertito nel tipo corretto. Infine, ogni tipo di argomento aveva bisogno di un metodo `getxxx` in modo da poterlo restituire al chiamante con il suo vero tipo.

Più tipi... tutti con metodi simili... tutto faceva pensare a una classe. E così è nata l'idea della classe `ArgumentMarshaler`.

L'incrementalismo

Uno dei modi migliori per rovinare un programma consiste nel sottoporlo a massicci interventi strutturali nel nome del progresso. Alcuni programmi non si riprendono più da tali “progressi”. Il problema? È molto difficile fare in modo che il programma torni a funzionare come faceva prima dei “progressi”.

Per evitarlo, impiego la disciplina dello sviluppo TDD (Test-Driven Development). Uno dei punti fermi di questo approccio consiste nel mantenere il sistema sempre in uno stato funzionante. In altre parole, con lo sviluppo TDD, non posso apportare al sistema una modifica che ne pregiudica il funzionamento. Ogni modifica apportata deve comunque permettere il funzionamento del sistema, come prima.

Per riuscirci, devo predisporre un insieme di test automatici da eseguire agevolmente per verificare che il comportamento del sistema non sia cambiato. Per la classe `Args` avevo creato un insieme di unit test e di accettazione mentre stavo realizzando la versione precedente, difettosa. Gli unit test erano scritti in Java e amministrati da JUnit. I test di accettazione erano scritti sotto forma di pagine wiki in FitNesse. Potevo eseguire questi test ogni volta che volevo e se il sistema li

passava, potevo essere tranquillo: il sistema funzionava ancora come previsto.

Pertanto ho proceduto ad apportare una grande quantità di piccolissimi interventi. Ognuna di esse faceva evolvere la struttura del sistema verso il concetto `ArgumentMarshaler`. E tuttavia ogni modifica manteneva il sistema funzionante. La prima modifica prevedeva l'aggiunta di un abbozzo di `ArgumentMarshaler` alla fine del codice (Listato 14.11).

Listato 14.11 `ArgumentMarshaler` aggiunta a `Args.java`.

```
private class ArgumentMarshaler {  
    private boolean booleanValue = false;  
    public void setBoolean(boolean value) {  
        booleanValue = value;  
    }  
    public boolean getBoolean() {return booleanValue;}  
}  
private class BooleanArgumentMarshaler extends ArgumentMarshaler {}  
private class StringArgumentMarshaler extends ArgumentMarshaler {}  
private class IntegerArgumentMarshaler extends ArgumentMarshaler {}  
}
```

Chiaramente, questa aggiunta non pregiudicava nulla. Pertanto ho poi eseguito la più semplice modifica che potevo, che avrebbe sovertito il meno possibile il programma. Ho cambiato l'`HashMap` per gli argomenti `Boolean` **in un** `ArgumentMarshaler`.

```
private Map<Character, ArgumentMarshaler> booleanArgs =  
new HashMap<Character, ArgumentMarshaler>();
```

Questo ha avuto un effetto su un paio di istruzioni, che ho prontamente adattato.

```
...  
private void parseBooleanSchemaElement(char elementId) {  
    booleanArgs.put(elementId, new BooleanArgumentMarshaler());  
}  
...  
private void setBooleanArg(char argChar, boolean value) {  
    booleanArgs.get(argChar).setBoolean(value);  
}  
...
```

```

public boolean getBoolean(char arg) {
    return falseIfNull(booleanArgs.get(arg).getBoolean());
}

```

Notate come queste modifiche siano esattamente nelle aree che ho appena menzionato: il `parse`, il `set` e il `get` del tipo di argomento.

Sfortunatamente, per quanto sia piccola questa modifica, alcuni dei test hanno cominciato a fallire. Se osservate con attenzione `getBoolean`, vedrete che se la richiamate con '`y`' ma non vi è alcun argomento `y`, allora `booleanArgs.get('y')` restituirà `null` e la funzione lancerà una `NullPointerException`. La funzione `falseIfNull` aveva proprio lo scopo di proteggere, ma la modifica che ho apportato ha reso tale funzione irrilevante.

L'incrementalismo richiedeva che intervenissi rapidamente, prima di applicare qualsiasi altra modifica. In effetti, l'intervento non è stato troppo difficile. Mi è bastato spostare il controllo del `null`. Non dovevo più controllare che il booleano fosse `null`; ci doveva pensare l'`ArgumentMarshaler`.

Innanzitutto, ho tolto la chiamata a `falseIfNull` dalla funzione `getBoolean`. Ormai era inutile, così ho eliminato anche la funzione. I test fallivano ancora nello stesso modo, così potevo essere abbastanza sicuro di non aver introdotto nuovi errori.

```

public boolean getBoolean(char arg) {
    return booleanArgs.get(arg).getBoolean();
}

```

Poi, ho suddiviso la funzione in due righe e ho inserito l'`ArgumentMarshaler` in una sua variabile chiamata `argumentMarshaler`. Non mi interessava che la variabile avesse un nome lungo; era davvero ridondante e congestionava la funzione. Così l'ho abbreviato in `am` [N5].

```

public boolean getBoolean(char arg) {
    Args.ArgumentMarshaler am = booleanArgs.get(arg);
    return am.getBoolean();
}

```

Quindi ho scritto la logica di rilevamento del `null`.

```
public boolean getBoolean(char arg) {
    Args.ArgumentMarshaler am = booleanArgs.get(arg);
    return am != null && am.getBoolean();
}
```

Argomenti String

L'aggiunta di argomenti di tipo `String` è stata molto simile a quella di argomenti `boolean`. Ho dovuto cambiare la `HashMap` e far sì che le funzioni `parse`, `set` e `get` funzionassero. Non dovreste trovare sorprese in ciò che segue, tranne forse il fatto che sembro porre tutta l'implementazione di “smistamento”, (*marshaling*) nella classe base `ArgumentMarshaler` invece di distribuirla nelle sue derivate.

```
private Map<Character, ArgumentMarshaler> stringArgs =
    new HashMap<Character, ArgumentMarshaler>();
...
private void parseStringSchemaElement(char elementId) {
    stringArgs.put(elementId, new StringArgumentMarshaler());
}
...
private void setStringArg(char argChar) throws ArgsException {
    currentArgument++;
    try {
        stringArgs.get(argChar).setString(args[currentArgument]);
    } catch (ArrayIndexOutOfBoundsException e) {
        valid = false;
        errorArgumentId = argChar;
        errorCode = ErrorCode.MISSING_STRING;
        throw new ArgsException();
    }
}
...
public String getString(char arg) {
    Args.ArgumentMarshaler am = stringArgs.get(arg);
    return am == null ? "" : am.getString();
}
...
private class ArgumentMarshaler {
    private boolean booleanValue = false;
    private String stringValue;
    public void setBoolean(boolean value) {
        booleanValue = value;
    }
    public boolean getBoolean() {
        return booleanValue;
    }
    public void setString(String s) {
        stringValue = s;
    }
}
```

```

    }
    public String getString() {
        return stringValue == null ? "" : stringValue;
    }
}

```

Di nuovo, queste modifiche sono state fatte una alla volta e sempre facendo in modo che i test funzionassero e, possibilmente, venissero superati. Quando non passava un test, mi assicuravo sempre di correggere il problema (e far passare il test) prima di procedere con la modifica successiva.

A questo punto dovreste essere in grado di vedere chiaramente qual è il mio intento. Dopo aver inserito tutto il comportamento di smistamento nella classe base `ArgumentMarshaler`, inizierò a delegare tale comportamento nelle derivate. Questo mi consentirà di mantenere il software funzionante mentre cambio, gradualmente, l'assetto del programma.

Il passo successivo è consistito nel porre le funzionalità per l'argomento `int` in `ArgumentMarshaler`. Di nuovo, non vi sono sorprese.

```

private Map<Character, ArgumentMarshaler> intArgs =
    new HashMap<Character, ArgumentMarshaler>();
...
private void parseIntegerSchemaElement(char elementId) {
    intArgs.put(elementId, new IntegerArgumentMarshaler());
}
...
private void setIntArg(char argChar) throws ArgsException {
    currentArgument++;
    String parameter = null;
    try {
        parameter = args[currentArgument];
        intArgs.get(argChar).setInteger(Integer.parseInt(parameter));
    } catch (ArrayIndexOutOfBoundsException e) {
        valid = false;
        errorArgumentId = argChar;
        errorCode = ErrorCode.MISSING_INTEGER;
        throw new ArgsException();
    } catch (NumberFormatException e) {
        valid = false;
        errorArgumentId = argChar;
        errorParameter = parameter;
        errorCode = ErrorCode.INVALID_INTEGER;
        throw new ArgsException();
    }
}
...
public int getInt(char arg) {

```

```

Args.ArgumentMarshaler am = intArgs.get(arg);
    return am == null ? 0 : am.getInteger();
}

...
private class ArgumentMarshaler {
    private boolean booleanValue = false;
    private String stringValue;
private int integerValue;
    public void setBoolean(boolean value) {
        booleanValue = value;
    }
    public boolean getBoolean() {
        return booleanValue;
    }
    public void setString(String s) {
        stringValue = s;
    }
    public String getString() {
        return stringValue == null ? "" : stringValue;
    }
public void setInteger(int i) {
    integerValue = i;
}
public int getInteger() {
    return integerValue;
}
}

```

Ora che tutto il marshaling è in `ArgumentMarshaler`, ho iniziato a distribuire le funzionalità nelle derivate. Il primo passo è consistito nel trasferire la funzione `setBoolean` in `BooleanArgumentMarshaler` e nell'assicurarmi che venisse richiamata correttamente. Pertanto ho creato un metodo astratto `set`.

```

private abstract class ArgumentMarshaler {
    protected boolean booleanValue = false;
    private String stringValue;
    private int integerValue;
    public void setBoolean(boolean value) {
        booleanValue = value;
    }
    public boolean getBoolean() {
        return booleanValue;
    }
    public void setString(String s) {
        stringValue = s;
    }
    public String getString() {
        return stringValue == null ? "" : stringValue;
    }
    public void setInteger(int i) {
        integerValue = i;
    }
    public int getInteger() {
        return integerValue;
    }
}

```

```

    }
    public abstract void set(String s);
}

}

```

Poi ho implementato il metodo `set` in `BooleanArgumentMarshaler`.

```

private class BooleanArgumentMarshaler extends ArgumentMarshaler {
    public void set(String s) {
        booleanValue = true;
    }
}

```

E infine ho sostituito la chiamata a `setBoolean` con una chiamata a `set`.

```

private void setBooleanArg(char argChar, boolean value) {
    booleanArgs.get(argChar).set("true");
}

```

I test passavano. Poiché questa modifica richiedeva che `set` passasse al `BooleanArgumentMarshaler`, ho rimosso il metodo `setBoolean` dalla classe base `ArgumentMarshaler`.

Notate che la funzione astratta `set` accetta un argomento `String`, ma l'implementazione in `BooleanArgumentMarshaler` non lo usa. Ho inserito tale argomento perché sapevo che `StringArgumentMarshaler` e `IntegerArgumentMarshaler` lo avrebbero usato.

Poi, volevo inserire il metodo `get` in `BooleanArgumentMarshaler`. Le funzioni `get` sono sempre ostiche, perché il tipo restituito deve essere `Object` e in questo caso deve essere convertito in un `Boolean`.

```

public boolean getBoolean(char arg) {
    Args.ArgumentMarshaler am = booleanArgs.get(arg);
    return am != null && (Boolean)am.get();
}

```

Per garantire la compilabilità, ho aggiunto la funzione `get` all'`ArgumentMarshaler`.

```

private abstract class ArgumentMarshaler {
    ...
    public Object get() {
        return null;
    }
}

```

La compilazione funzionava e, ovviamente, i test fallivano. Per accontentare i test è bastato rendere `get` astratta e implementarla in `BooleanArgumentMarshaler`.

```
private abstract class ArgumentMarshaler {  
    protected boolean booleanValue = false;  
    ...  
    public abstract Object get();  
}  
private class BooleanArgumentMarshaler extends ArgumentMarshaler {  
    public void set(String s) {  
        booleanValue = true;  
    }  
    public Object get() {  
        return booleanValue;  
    }  
}
```

Ora passava i test. Pertanto `get` e `set` sono entrambe passate a `BooleanArgumentMarshaler`! Questo mi ha consentito di rimuovere da `ArgumentMarshaler` la vecchia funzione `getBoolean`, di spostare in `BooleanArgumentMarshaler` la variabile protetta `booleanValue` e di renderla private.

Ho provveduto a eseguire le stesse modifiche per le `String`. Ho spostato `set` e `get`, cancellato le funzioni inutilizzate e spostato le variabili.

```
private void setStringArg(char argChar) throws ArgsException {  
    currentArgument++;  
    try {  
        stringArgs.get(argChar).set(args[currentArgument]);  
    } catch (ArrayIndexOutOfBoundsException e) {  
        valid = false;  
        errorArgumentId = argChar;  
        errorCode = ErrorCode.MISSING_STRING;  
        throw new ArgsException();  
    }  
}  
...  
public String getString(char arg) {  
    Args.ArgumentMarshaler am = stringArgs.get(arg);  
    return am == null ? "" : (String) am.get();  
}  
...  
private abstract class ArgumentMarshaler {  
    private int integerValue;  
  
    public void setInteger(int i) {  
        integerValue = i;  
    }
```

```

        public int getInteger() {
            return integerValue;
        }

        public abstract void set(String s);

        public abstract Object get();
    }

    private class BooleanArgumentMarshaler extends ArgumentMarshaler {
private boolean booleanValue = false;

        public void set(String s) {
            booleanValue = true;
        }

        public Object get() {
            return booleanValue;
        }
    }
    private class StringArgumentMarshaler extends ArgumentMarshaler {
private String stringValue = "";

        public void set(String s) {
stringValue = s;
        }

        public Object get() {
return stringValue;
        }
    }
    private class IntegerArgumentMarshaler extends ArgumentMarshaler {
        public void set(String s) {

        }

        public Object get() {
            return null;
        }
    }
}

```

Infine, ho ripetuto l'operazione per gli interi. L'operazione è stata un po' più complicata per il fatto che gli interi devono subire un parsing, e parse può lanciare un'eccezione. Ma il risultato è migliore, perché l'intero concetto di `NumberFormatException` finisce dentro

`IntegerArgumentMarshaler`.

```

private boolean isIntArg(char argChar) {return intArgs.containsKey(argChar);}
private void setIntArg(char argChar) throws ArgsException {
    currentArgument++;
    String parameter = null;
    try {
        parameter = args[currentArgument];
        intArgs.get(argChar).set(parameter);
    } catch (ArrayIndexOutOfBoundsException e) {

```

```

        valid = false;
        errorArgumentId = argChar;
        errorCode = ErrorCode.MISSING_INTEGER;
        throw new ArgsException();
    } catch (ArgsException e) {
        valid = false;
        errorArgumentId = argChar;
        errorParameter = parameter;
        errorCode = ErrorCode.INVALID_INTEGER;
        throw e;
    }
}
...
private void setBooleanArg(char argChar) {
    try {
        booleanArgs.get(argChar).set("true");
    } catch (ArgsException e) {
    }
}
...
public int getInt(char arg) {
    Args.ArgumentMarshaler am = intArgs.get(arg);
    return am == null ? 0 : (Integer) am.get();
}
...
private abstract class ArgumentMarshaler {
    public abstract void set(String s) throws ArgsException;
    public abstract Object get();
}
...
private class IntegerArgumentMarshaler extends ArgumentMarshaler {
    private int intValue = 0;
    public void set(String s) throws ArgsException {
        try {
            intValue = Integer.parseInt(s);
        } catch (NumberFormatException e) {
            throw new ArgsException();
        }
    }
    public Object get() {
        return intValue;
    }
}
}

```

Naturalmente, il tutto continuava a passare i test. Poi, mi sono sbarazzato delle tre diverse mappe in cima all'algoritmo. Questo ha reso l'intero sistema molto più generico. Tuttavia, non potevo eliminarli semplicemente cancellandoli, perché il sistema non avrebbe più funzionato. Piuttosto, ho aggiunto una nuova `Map` per l'`ArgumentMarshaler` e poi ho cambiato uno per uno i metodi impiegati, invece delle tre mappe.

```

public class Args {
    ...
    private Map<Character, ArgumentMarshaler> booleanArgs =
        new HashMap<Character, ArgumentMarshaler>();

```

```

private Map<Character, ArgumentMarshaler> stringArgs =
    new HashMap<Character, ArgumentMarshaler>();
private Map<Character, ArgumentMarshaler> intArgs =
    new HashMap<Character, ArgumentMarshaler>();
private Map<Character, ArgumentMarshaler> marshalers =
new HashMap<Character, ArgumentMarshaler>();
...
private void parseBooleanSchemaElement(char elementId) {
ArgumentMarshaler m = new BooleanArgumentMarshaler();
    booleanArgs.put(elementId, m);
marshalers.put(elementId, m);
}
private void parseIntegerSchemaElement(char elementId) {
ArgumentMarshaler m = new IntegerArgumentMarshaler();
    intArgs.put(elementId, m);
marshalers.put(elementId, m);
}
private void parseStringSchemaElement(char elementId) {
ArgumentMarshaler m = new StringArgumentMarshaler();
    stringArgs.put(elementId, m);
marshalers.put(elementId, m);
}
}

```

Naturalmente i test non hanno avuto problemi. Poi, ho cambiato `isBooleanArg` da questa:

```

private boolean isBooleanArg(char argChar) {
    return booleanArgs.containsKey(argChar);
}

```

a questa:

```

private boolean isBooleanArg(char argChar) {
ArgumentMarshaler m = marshalers.get(argChar);
    return m instanceof BooleanArgumentMarshaler;
}

```

Nessun problema con i test. Pertanto ho applicato la stessa modifica

a `isIntArg` **e** `isStringArg`.

```

private boolean isIntArg(char argChar) {
ArgumentMarshaler m = marshalers.get(argChar);
    return m instanceof IntegerArgumentMarshaler;
}
private boolean isStringArg(char argChar) {
ArgumentMarshaler m = marshalers.get(argChar);
    return m instanceof StringArgumentMarshaler;
}

```

Ancora nessun problema con i test. Pertanto ho eliminato tutte le chiamate duplicate a `marshalers.get`:

```

private boolean setArgument(char argChar) throws ArgsException {
ArgumentMarshaler m = marshalers.get(argChar);

```

```

        if (isBooleanArg(m))
            setBooleanArg(argChar);
        else if (isStringArg(m))
            setStringArg(argChar);
        else if (isIntArg(m))
            setIntArg(argChar);
        else
            return false;
        return true;
    }
    private boolean isIntArg(ArgumentMarshaler m) {
        return m instanceof IntegerArgumentMarshaler;
    }
    private boolean isStringArg(ArgumentMarshaler m) {
        return m instanceof StringArgumentMarshaler;
    }
    private boolean isBooleanArg(ArgumentMarshaler m) {
        return m instanceof BooleanArgumentMarshaler;
    }
}

```

Questo intervento non ha lasciato alcun buon motivo per conservare i tre metodi `isxxxxArg`. Pertanto li ho resi inline:

```

private boolean setArgument(char argChar) throws ArgsException {
    ArgumentMarshaler m = marshalers.get(argChar);
    if (m instanceof BooleanArgumentMarshaler)
        setBooleanArg(argChar);
    else if (m instanceof StringArgumentMarshaler)
        setStringArg(argChar);
    else if (m instanceof IntegerArgumentMarshaler)
        setIntArg(argChar);
    else
        return false;
    return true;
}

```

Poi, ho iniziato a usare la mappa `marshalers` nelle funzioni `set`, interrompendo l'uso delle altre tre mappe. Sono partito dai `boolean`.

```

private boolean setArgument(char argChar) throws ArgsException {
    ArgumentMarshaler m = marshalers.get(argChar);
    if (m instanceof BooleanArgumentMarshaler)
        setBooleanArg(m);
    else if (m instanceof StringArgumentMarshaler)
        setStringArg(argChar);
    else if (m instanceof IntegerArgumentMarshaler)
        setIntArg(argChar);
    else
        return false;
    return true;
}
...
private void setBooleanArg(ArgumentMarshaler m) {
    try {
        m.set("true"); // era: booleanArgs.get(argChar).set("true");
    } catch (ArgsException e) {
    }
}

```

}

Tutto ok per i test, così ho fatto lo stesso per le `String` e gli `Integer`.

Questo mi ha consentito di integrare un po' di codice di gestione delle eccezioni nella funzione `setArgument`.

```
private boolean setArgument(char argChar) throws ArgsException {
    ArgumentMarshaler m = marshalers.get(argChar);
    try {
        if (m instanceof BooleanArgumentMarshaler)
            setBooleanArg(m);
        else if (m instanceof StringArgumentMarshaler)
            setStringArg(m);
        else if (m instanceof IntegerArgumentMarshaler)
            setIntArg(m);
        else
            return false;
    } catch (ArgsException e) {
        valid = false;
        errorArgumentId = argChar;
        throw e;
    }
    return true;
}
private void setIntArg(ArgumentMarshaler m) throws ArgsException {
    currentArgument++;
    String parameter = null;
    try {
        parameter = args[currentArgument];
        m.set(parameter);
    } catch (ArrayIndexOutOfBoundsException e) {
        errorCode = ErrorCode.MISSING_INTEGER;
        throw new ArgsException();
    } catch (ArgsException e) {
        errorParameter = parameter;
        errorCode = ErrorCode.INVALID_INTEGER;
        throw e;
    }
}
private void setStringArg(ArgumentMarshaler m) throws ArgsException {
    currentArgument++;
    try {
        m.set(args[currentArgument]);
    } catch (ArrayIndexOutOfBoundsException e) {
        errorCode = ErrorCode.MISSING_STRING;
        throw new ArgsException();
    }
}
```

A questo punto era quasi possibile rimuovere le tre vecchie map. Innanzitutto, dovevo cambiare la funzione `getBoolean` da così:

```
public boolean getBoolean(char arg) {
    Args.ArgumentMarshaler am = booleanArgs.get(arg);
    return am != null && (Boolean) am.get();
}
```

a così:

```
public boolean getBoolean(char arg) {
    Args.ArgumentMarshaler am = marshalers.get(arg);
    boolean b = false;
    try {
        b = am != null && (Boolean) am.get();
    } catch (ClassCastException e) {
        b = false;
    }
    return b;
}
```

Quest'ultima modifica può essere un po' sorprendente. Perchè decido improvvisamente di occuparmi di `ClassCastException`? Il motivo è che ho un insieme di unit test e un insieme distinto di test di accettazione scritti in FitNesse. Di conseguenza, i test FitNesse si assicurano che se ho richiamato `getBoolean` su un argomento non `boolean`, il risultato sia `false`. Gli unit test no. Fino a questo punto avevo eseguito solo unit test (per evitare ulteriori sorprese di questo tipo, ho aggiunto un nuovo unit test, che ha richiamato tutti i test di FitNesse).

Quest'ultima modifica mi ha consentito di impiegare un'altra caratteristica delle mappe di `boolean`:

```
private void parseBooleanSchemaElement(char elementId) {
    ArgumentMarshaler m = new BooleanArgumentMarshaler();
    booleanArgs.put(elementId, m);
    marshalers.put(elementId, m);
}
```

E ora possiamo cancellare la mappa di `boolean`.

```
public class Args {
    ...
    private Map<Character, ArgumentMarshaler> booleanArgs =
    new HashMap<Character, ArgumentMarshaler>();
    private Map<Character, ArgumentMarshaler> stringArgs =
        new HashMap<Character, ArgumentMarshaler>();
    private Map<Character, ArgumentMarshaler> intArgs =
        new HashMap<Character, ArgumentMarshaler>();
    private Map<Character, ArgumentMarshaler> marshalers =
        new HashMap<Character, ArgumentMarshaler>();
    ...
}
```

Poi, ho eseguito allo stesso modo la migrazione degli argomenti `String` e `Integer` e ho fatto un po' di pulizia con i `boolean`.

```

private void parseBooleanSchemaElement(char elementId) {
    marshalers.put(elementId, new BooleanArgumentMarshaler());
}
private void parseIntegerSchemaElement(char elementId) {
    marshalers.put(elementId, new IntegerArgumentMarshaler());
}
private void parseStringSchemaElement(char elementId) {
    marshalers.put(elementId, new StringArgumentMarshaler());
}
public String getString(char arg) {
    Args.ArgumentMarshaler am = marshalers.get(arg);
    try {
        return am == null ? "" : (String) am.get();
    } catch (ClassCastException e) {
        return "";
    }
}
public int getInt(char arg) {
    Args.ArgumentMarshaler am = marshalers.get(arg);
    try {
        return am == null ? 0 : (Integer) am.get();
    } catch (Exception e) {
        return 0;
    }
}
...
public class Args {
    ...
    private Map<Character, ArgumentMarshaler> stringArgs
        = new HashMap<Character, ArgumentMarshaler>(),
    private Map<Character, ArgumentMarshaler> intArgs
        = new HashMap<Character, ArgumentMarshaler>(),
    private Map<Character, ArgumentMarshaler> marshalers
        = new HashMap<Character, ArgumentMarshaler>();
    ...
}

```

Poi, ho reso inline i tre metodi `parse` perché non facevano più molto:

```

private void parseSchemaElement(String element) throws ParseException {
    char elementId = element.charAt(0);
    String elementTail = element.substring(1);
    validateSchemaElementId(elementId);
    if (isBooleanSchemaElement(elementTail))
        marshalers.put(elementId, new BooleanArgumentMarshaler());
    else if (isStringSchemaElement(elementTail))
        marshalers.put(elementId, new StringArgumentMarshaler());
    else if (isIntegerSchemaElement(elementTail)) {
        marshalers.put(elementId, new IntegerArgumentMarshaler());
    } else {
        throw new ParseException(String.format(
            "Argument: %c has invalid format: %s.", elementId, elementTail), 0);
    }
}

```

Okay, ora diamo un'occhiata d'insieme. Il Listato 14.12 presenta lo stato attuale della classe `Args`.

Listato 14.12 Args.java (dopo il primo refactoring).

```
package com.objectmentor.utilities.getopts;
import java.text.ParseException;
import java.util.*;
public class Args {
    private String schema;
    private String[] args;
    private boolean valid = true;
    private Set<Character> unexpectedArguments = new TreeSet<Character>();
    private Map<Character, ArgumentMarshaler> marshalers =
        new HashMap<Character, ArgumentMarshaler>();
    private Set<Character> argsFound = new HashSet<Character>();
    private int currentArgument;
    private char errorArgumentId = '\0';
    private String errorParameter = "TILT";
    private ErrorCode errorCode = ErrorCode.OK;
    private enum ErrorCode {
        OK, MISSING_STRING, MISSING_INTEGER, INVALID_INTEGER, UNEXPECTED_ARGUMENT
    }
    public Args(String schema, String[] args) throws ParseException {
        this.schema = schema;
        this.args = args;
        valid = parse();
    }
    private boolean parse() throws ParseException {
        if (schema.length() == 0 && args.length == 0)
            return true;
        parseSchema();
        try {
            parseArguments();
        } catch (ArgsException e) {
        }
        return valid;
    }
    private boolean parseSchema() throws ParseException {
        for (String element : schema.split(",")) {
            if (element.length() > 0) {
                String trimmedElement = element.trim();
                parseSchemaElement(trimmedElement);
            }
        }
        return true;
    }
    private void parseSchemaElement(String element) throws ParseException {
        char elementId = element.charAt(0);
        String elementTail = element.substring(1);
        validateSchemaElementId(elementId);
        if (isBooleanSchemaElement(elementTail))
            marshalers.put(elementId, new BooleanArgumentMarshaler());
        else if (isStringSchemaElement(elementTail))
            marshalers.put(elementId, new StringArgumentMarshaler());
        else if (isIntegerSchemaElement(elementTail)) {
            marshalers.put(elementId, new IntegerArgumentMarshaler());
        } else {
            throw new ParseException(String.format(
                "Argument: %c has invalid format: %s.", elementId, elementTail), 0);
        }
    }
    private void validateSchemaElementId(char elementId) throws ParseException {
        if (!Character.isLetter(elementId)) {
```

```

        throw new ParseException(
            "Bad character:" + elementId + "in Args format: " + schema, 0);
    }
}

private boolean isStringSchemaElement(String elementTail) {
    return elementTail.equals("*");
}

private boolean isBooleanSchemaElement(String elementTail) {
    return elementTail.length() == 0;
}

private boolean isIntegerSchemaElement(String elementTail) {
    return elementTail.equals("#");
}

private boolean parseArguments() throws ArgsException {
    for (currentArgument=0; currentArgument<args.length; currentArgument++) {
        String arg = args[currentArgument];
        parseArgument(arg);
    }
    return true;
}

private void parseArgument(String arg) throws ArgsException {
    if (arg.startsWith("-"))
        parseElements(arg);
}

private void parseElements(String arg) throws ArgsException {
    for (int i = 1; i < arg.length(); i++)
        parseElement(arg.charAt(i));
}

private void parseElement(char argChar) throws ArgsException {
    if (setArgument(argChar))
        argsFound.add(argChar);
    else {
        unexpectedArguments.add(argChar);
        errorCode = ErrorCode.UNEXPECTED_ARGUMENT;
        valid = false;
    }
}

private boolean setArgument(char argChar) throws ArgsException {
    ArgumentMarshaler m = marshalers.get(argChar);
    try {
        if (m instanceof BooleanArgumentMarshaler)
            setBooleanArg(m);
        else if (m instanceof StringArgumentMarshaler)
            setStringArg(m);
        else if (m instanceof IntegerArgumentMarshaler)
            setIntArg(m);
        else
            return false;
    } catch (ArgsException e) {
        valid = false;
        errorArgumentId = argChar;
        throw e;
    }
    return true;
}

private void setIntArg(ArgumentMarshaler m) throws ArgsException {
    currentArgument++;
    String parameter = null;
    try {
        parameter = args[currentArgument];
        m.set(parameter);
    }
}

```

```

        } catch (ArrayIndexOutOfBoundsException e) {
            errorCode = ErrorCode.MISSING_INTEGER;
            throw new ArgsException();
        } catch (ArgsException e) {
            errorParameter = parameter;
            errorCode = ErrorCode.INVALID_INTEGER;
            throw e;
        }
    }
    private void setStringArg(ArgumentMarshaler m) throws ArgsException {
        currentArgument++;
        try {
            m.set(args[currentArgument]);
        } catch (ArrayIndexOutOfBoundsException e) {
            errorCode = ErrorCode.MISSING_STRING;
            throw new ArgsException();
        }
    }
    private void setBooleanArg(ArgumentMarshaler m) {
        try {
            m.set("true");
        } catch (ArgsException e) {
        }
    }
    public int cardinality() {
        Return argsFound.size();
    }
    public String usage() {
        if (schema.length() > 0)
            return "-[" + schema + "]";
        else
            return "";
    }
    public String errorMessage() throws Exception {
        switch (errorCode) {
            case OK:
                throw new Exception("TILT: Should not get here.");
            case UNEXPECTED_ARGUMENT:
                return unexpectedArgumentMessage();
            case MISSING_STRING:
                return String.format("Could not find string parameter for -%c.",
                        errorArgumentId);
            case INVALID_INTEGER:
                return String.format("Argument -%c expects an integer but was '%s'.",
                        errorArgumentId, errorParameter);
            case MISSING_INTEGER:
                return String.format("Could not find integer parameter for -%c.",
                        errorArgumentId);
        }
        return "";
    }
    private String unexpectedArgumentMessage() {
        StringBuffer message = new StringBuffer("Argument(s) -");
        for (char c : unexpectedArguments) {
            message.append(c);
        }
        message.append(" unexpected.");
        return message.toString();
    }
    public boolean getBoolean(char arg) {
        Args.ArgumentMarshaler am = marshalers.get(arg);

```

```

        boolean b = false;
        try {
            b = am != null && (Boolean) am.get();
        } catch (ClassCastException e) {
            b = false;
        }
        return b;
    }

    public String getString(char arg) {
        Args.ArgumentMarshaler am = marshalers.get(arg);
        try {
            return am == null ? "" : (String) am.get();
        } catch (ClassCastException e) {
            return "";
        }
    }

    public int getInt(char arg) {
        Args.ArgumentMarshaler am = marshalers.get(arg);
        try {
            return am == null ? 0 : (Integer) am.get();
        } catch (Exception e) {
            return 0;
        }
    }

    public boolean has(char arg) {
        return argsFound.contains(arg);
    }

    public boolean isValid() {
        return valid;
    }

    private class ArgsException extends Exception {
    }

    private abstract class ArgumentMarshaler {
        public abstract void set(String s) throws ArgsException;
        public abstract Object get();
    }

    private class BooleanArgumentMarshaler extends ArgumentMarshaler {
        private boolean booleanValue = false;
        public void set(String s) {
            booleanValue = true;
        }
        public Object get() {
            return booleanValue;
        }
    }

    private class StringArgumentMarshaler extends ArgumentMarshaler {
        private String stringValue = "";
        public void set(String s) {
            stringValue = s;
        }
        public Object get() {
            return stringValue;
        }
    }

    private class IntegerArgumentMarshaler extends ArgumentMarshaler {
        private int intValue = 0;
        public void set(String s) throws ArgsException {
            try {
                intValue = Integer.parseInt(s);
            } catch (NumberFormatException e) {
                throw new ArgsException();
            }
        }
    }
}

```

```

        }
    public Object get() {
        return intValue;
    }
}
}

```

Dopo tutto questo lavoro, la soluzione è deludente. La struttura è un po' migliore, ma abbiamo ancora tutte quelle variabili in cima; c'è ancora quell'orribile “`case`” sui tipi in `setArgument`. E poi tutte quelle funzioni `set` sono davvero terribili. Per non parlare dell'elaborazione degli errori. Abbiamo ancora molto lavoro da fare.

Vorrei, soprattutto, eliminare quella selezione dei tipi in `setArgument` [G23]. Quello che vorrei in `setArgument` è un'unica chiamata a `ArgumentMarshaler.set`. Questo significa che devo inserire `setIntArg`, `setStringArg` e `setBooleanArg` nelle classi derivate appropriate di `ArgumentMarshaler`. Ma vi è un problema.

Se osservate attentamente `setIntArg`, noterete che impiega due variabili di istanza: `args` e `currentArg`. Per spostare `setIntArg` in `BooleanArgumentMarshaler`, dovrò passare `args` e `currentArgs` come argomenti di funzione. Non è bello [F1]. Piuttosto vorrei passare un solo argomento anziché due. Fortunatamente, vi è una soluzione semplice. Possiamo convertire l'array `args` in una `list` e passare un `Iterator` alle funzioni `set`. Ciò che segue mi ha richiesto dieci passi, assicurandomi ogni volta il passaggio di tutti i test. Ma ora vi mostrerò solo il risultato. Ormai dovreste essere in grado di immaginarvi la maggior parte dei miei piccoli passi.

```

public class Args {
    private String schema;
    private String[] args;
    private boolean valid = true;
    private Set<Character> unexpectedArguments = new TreeSet<Character>();
    private Map<Character, ArgumentMarshaler> marshalers =
        new HashMap<Character, ArgumentMarshaler>();
    private Set<Character> argsFound = new HashSet<Character>();
    private Iterator<String> currentArgument;
    private char errorArgumentId = '\0';
    private String errorParameter = "TILT";
}

```

```

private ErrorCode errorCode = ErrorCode.OK;
private List<String> argsList;
private enum ErrorCode {
    OK, MISSING_STRING, MISSING_INTEGER, INVALID_INTEGER, UNEXPECTED_ARGUMENT}
public Args(String schema, String[] args) throws ParseException {
    this.schema = schema;
argsList = Arrays.asList(args);
    valid = parse();
}
private boolean parse() throws ParseException {
    if (schema.length() == 0 && argsList.size() == 0)
        return true;
    parseSchema();
    try {
        parseArguments();
    } catch (ArgsException e) {
    }
    return valid;
}
...
private boolean parseArguments() throws ArgsException {
    for (currentArgument = argsList.iterator(); currentArgument.hasNext();) {
        String arg = currentArgument.next();
        parseArgument(arg);
    }
    return true;
}
...
private void setIntArg(ArgumentMarshaler m) throws ArgsException {
    String parameter = null;
    try {
        parameter = currentArgument.next();
        m.set(parameter);
    } catch (NoSuchElementException e) {
        errorCode = ErrorCode.MISSING_INTEGER;
        throw new ArgsException();
    } catch (ArgsException e) {
        errorParameter = parameter;
        errorCode = ErrorCode.INVALID_INTEGER;
        throw e;
    }
}
private void setStringArg(ArgumentMarshaler m) throws ArgsException {
    try {
        m.set(currentArgument.next());
    } catch (NoSuchElementException e) {
        errorCode = ErrorCode.MISSING_STRING;
        throw new ArgsException();
    }
}

```

Queste semplici modifiche hanno passato tutti i test. Ora possiamo iniziare a spostare le funzioni `set` nelle derivate appropriate. Innanzitutto, devo applicare la seguente modifica a `setArgument`:

```

private boolean setArgument(char argChar) throws ArgsException {
    ArgumentMarshaler m = marshalers.get(argChar);
    if (m == null)

```

```

    return false;
    try {
        if (m instanceof BooleanArgumentMarshaler)
            setBooleanArg(m);
        else if (m instanceof StringArgumentMarshaler)
            setStringArg(m);
        else if (m instanceof IntegerArgumentMarshaler)
            setIntArg(m);
    else
        return false;
    } catch (ArgsException e) {
        valid = false;
        errorArgumentId = argChar;
        throw e;
    }
    return true;
}

```

Questa modifica è importante, perché vogliamo eliminare completamente la catena `if-else`. Pertanto, dovevamo estrarre la condizione d'errore.

Ora possiamo iniziare a spostare le funzioni `set`. La funzione `setBooleanArg` è banale, pertanto la faremo per prima. Il nostro obiettivo consiste nel modificare la funzione `setBooleanArg` in modo che non faccia altro che rimandare a `BooleanArgumentMarshaler`.

```

private boolean setArgument(char argChar) throws ArgsException {
    ArgumentMarshaler m = marshalers.get(argChar);
    if (m == null)
        return false;
    try {
        if (m instanceof BooleanArgumentMarshaler)
            setBooleanArg(m, currentArgument);
        else if (m instanceof StringArgumentMarshaler)
            setStringArg(m);
        else if (m instanceof IntegerArgumentMarshaler)
            setIntArg(m);
    } catch (ArgsException e) {
        valid = false;
        errorArgumentId = argChar;
        throw e;
    }
    return true;
}
...
private void setBooleanArg(ArgumentMarshaler m,
    Iterator<String> currentArgument)
    throws ArgsException {
try {
    m.set("true");
catch (ArgsException e) {
    }
}

```

Ma non avevamo appena inserito tale elaborazione delle eccezioni? Ma inserire delle cose da qualche parte e poi estrarle è piuttosto comune durante un refactoring. Basta un piccolo passo e la necessità di far funzionare i test costringe a spostare nuovamente tutto. Il refactoring è un po' come risolvere un cubo di Rubik: tanti piccoli passi avanti e indietro in vista dell'obiettivo finale. Ma ogni passo rende possibile il successivo.

Perché abbiamo passato quell'`Iterator` quando `setBooleanArg` certamente non ne aveva bisogno? Perché invece `setIntArg` e `setStringArg` sì, ne avranno bisogno! E poiché voglio realizzare tutte e tre queste funzioni tramite un metodo astratto in `ArgumentMarshaler`, devo passarlo a `setBooleanArg`.

Pertanto ora `setBooleanArg` è inutile. Se vi fosse una funzione `set` in `ArgumentMarshaler`, potremmo richiamarla direttamente. Pertanto è giunto il momento di creare tale funzione! Il primo passo è aggiungere a `ArgumentMarshaler` il nuovo metodo astratto.

```
private abstract class ArgumentMarshaler {  
    public abstract void set(Iterator<String> currentArgument)  
        throws ArgsException;  
    public abstract void set(String s) throws ArgsException;  
    public abstract Object get();  
}
```

Naturalmente questo pregiudica il funzionamento di tutte le derivate. Pertanto implementiamo il nuovo metodo in ognuna di esse.

```
private class BooleanArgumentMarshaler extends ArgumentMarshaler {  
    private boolean booleanValue = false;  
    public void set(Iterator<String> currentArgument) throws ArgsException {  
        booleanValue = true;  
    }  
    public void set(String s) {  
        booleanValue = true;  
    }  
    public Object get() {  
        return booleanValue;  
    }  
}  
private class StringArgumentMarshaler extends ArgumentMarshaler {  
    private String stringValue = "";  
    public void set(Iterator<String> currentArgument) throws ArgsException {  
    }
```

```

    public void set(String s) {
        stringValue = s;
    }
    public Object get() {
        return stringValue;
    }
}
private class IntegerArgumentMarshaler extends ArgumentMarshaler {
    private int intValue = 0;
    public void set(Iterator<String> currentArgument) throws ArgsException {
    }
    public void set(String s) throws ArgsException {
        try {
            intValue = Integer.parseInt(s);
        } catch (NumberFormatException e) {
            throw new ArgsException();
        }
    }
    public Object get() {
        return intValue;
    }
}

```

E ora possiamo eliminare `setBooleanArg!`

```

private boolean setArgument(char argChar) throws ArgsException {
    ArgumentMarshaler m = marshalers.get(argChar);
    if (m == null)
        return false;
    try {
        if (m instanceof BooleanArgumentMarshaler)
            m.set(currentArgument);
        else if (m instanceof StringArgumentMarshaler)
            setStringArg(m);
        else if (m instanceof IntegerArgumentMarshaler)
            setIntArg(m);
    } catch (ArgsException e) {
        valid = false;
        errorArgumentId = argChar;
        throw e;
    }
    return true;
}

```

I test sono soddisfatti e la funzione `set` rimanda a

`BooleanArgumentMarshaler!`

Ora possiamo fare la stessa cosa per le stringhe e gli interi.

```

private boolean setArgument(char argChar) throws ArgsException {
    ArgumentMarshaler m = marshalers.get(argChar);
    if (m == null)
        return false;
    try {
        if (m instanceof BooleanArgumentMarshaler)
            m.set(currentArgument);
        else if (m instanceof StringArgumentMarshaler)
            m.set(currentArgument);
    }
}

```

```

        else if (m instanceof IntegerArgumentMarshaler)
m.set(currentArgument);
    } catch (ArgsException e) {
        valid = false;
        errorArgumentId = argChar;
        throw e;
    }
    return true;
}
...
private class StringArgumentMarshaler extends ArgumentMarshaler {
    private String stringValue = "";
    public void set(Iterator<String> currentArgument) throws ArgsException {
try {
    stringValue = currentArgument.next();
} catch (NoSuchElementException e) {
    errorCode = ErrorCode.MISSING_STRING;
    throw new ArgsException();
}
}
public void set(String s) {
}
public Object get() {
    return stringValue;
}
}
private class IntegerArgumentMarshaler extends ArgumentMarshaler {
    private int intValue = 0;
    public void set(Iterator<String> currentArgument) throws ArgsException {
String parameter = null;
try {
    parameter = currentArgument.next();
    set(parameter);
} catch (NoSuchElementException e) {
    errorCode = ErrorCode.MISSING_INTEGER;
    throw new ArgsException();
} catch (ArgsException e) {
    errorParameter = parameter;
    errorCode = ErrorCode.INVALID_INTEGER;
    throw e;
}
}
public void set(String s) throws ArgsException {
    try {
        intValue = Integer.parseInt(s);
    } catch (NumberFormatException e) {
        throw new ArgsException();
    }
}
public Object get() {
    return intValue;
}
}
}

```

E ora il *colpo di grazia*: il `case` sui tipi può essere eliminato!

```

private boolean setArgument(char argChar) throws ArgsException {
    ArgumentMarshaler m = marshalers.get(argChar);
    if (m == null)
        return false;

```

```

        try {
            m.set(currentArgument);
            return true;
        } catch (ArgsException e) {
            valid = false;
            errorArgumentId = argChar;
            throw e;
        }
    }
}

```

Ora possiamo sbarazzarci anche di alcune funzioni in `IntegerArgumentMarshaler` e ripulire un po' le cose.

```

private class IntegerArgumentMarshaler extends ArgumentMarshaler {
    private int intValue = 0
    public void set(Iterator<String> currentArgument) throws ArgsException {
        String parameter = null;
        try {
            parameter = currentArgument.next();
            intValue = Integer.parseInt(parameter);
        } catch (NoSuchElementException e) {
            errorCode = ErrorCode.MISSING_INTEGER;
            throw new ArgsException();
        } catch (NumberFormatException e) {
            errorParameter = parameter;
            errorCode = ErrorCode.INVALID_INTEGER;
            throw new ArgsException();
        }
    }
    public Object get() {
        return intValue;
    }
}

```

Possiamo anche trasformare `ArgumentMarshaler` in un'interfaccia.

```

private interface ArgumentMarshaler {
    void set(Iterator<String> currentArgument) throws ArgsException;
    Object get();
}

```

Ora possiamo vedere quanto sia facile aggiungere un nuovo tipo di argomento alla nostra struttura. Richiederà solo poche, isolate modifiche. Innanzitutto, iniziamo aggiungendo un nuovo caso di test per controllare che l'argomento `double` funzioni correttamente.

```

public void testSimpleDoublePresent() throws Exception {
    Args args = new Args("x##", new String[] {"-x", "42.3"});
    assertTrue(args.isValid());
    assertEquals(1, args.cardinality());
    assertTrue(args.has('x'));
    assertEquals(42.3, args.getDouble('x'), .001);
}

```

Ora ripuliamo il codice di parsing e aggiungiamo il rilevamento di `##` per un argomento di tipo `double`.

```
private void parseSchemaElement(String element) throws ParseException {
    char elementId = element.charAt(0);
    String elementTail = element.substring(1);
    validateSchemaElementId(elementId);
    if (elementTail.length() == 0)
        marshalers.put(elementId, new BooleanArgumentMarshaler());
    else if (elementTail.equals("*"))
        marshalers.put(elementId, new StringArgumentMarshaler());
    else if (elementTail.equals("#"))
        marshalers.put(elementId, new IntegerArgumentMarshaler());
    else if (elementTail.equals("##"))
        marshalers.put(elementId, new DoubleArgumentMarshaler());
    else
        throw new ParseException(String.format(
            "Argument: %c has invalid format: %s.", elementId, elementTail), 0);
}
```

Poi, scriviamo la classe `DoubleArgumentMarshaler`.

```
private class DoubleArgumentMarshaler implements ArgumentMarshaler {
    private double doubleValue = 0;
    public void set(Iterator<String> currentArgument) throws ArgsException {
        String parameter = null;
        try {
            parameter = currentArgument.next();
            doubleValue = Double.parseDouble(parameter);
        } catch (NoSuchElementException e) {
            errorCode = ErrorCode.MISSING_DOUBLE;
            throw new ArgsException();
        } catch (NumberFormatException e) {
            errorParameter = parameter;
            errorCode = ErrorCode.INVALID_DOUBLE;
            throw new ArgsException();
        }
    }
    public Object get() {
        return doubleValue;
    }
}
```

Questo ci costringe ad aggiungere un nuovo `ErrorCode`.

```
private enum ErrorCode {
    OK, MISSING_STRING, MISSING_INTEGER, INVALID_INTEGER,
    UNEXPECTED_ARGUMENT, MISSING_DOUBLE, INVALID_DOUBLE}
```

E abbiamo anche bisogno di una funzione `getDouble`.

```
public double getDouble(char arg) {
    Args.ArgumentMarshaler am = marshalers.get(arg);
    try {
        return am == null ? 0 : (Double) am.get();
    } catch (Exception e) {
```

```

    return 0.0;
}
}

```

E i test sono sempre soddisfatti! Tutto in modo quasi indolore.

Pertanto ora assicuriamoci che l'elaborazione degli errori funzioni correttamente. Il prossimo caso di test controlla che venga prodotto un errore nel caso in cui a un argomento `#` venga passata una stringa incomprensibile.

```

public void testInvalidDouble() throws Exception {
    Args args = new Args("x##", new String[] {"-x", "Forty two"});
    assertFalse(args.isValid());
    assertEquals(0, args.cardinality());
    assertFalse(args.has('x'));
    assertEquals(0, args.getInt('x'));
    assertEquals("Argument -x expects a double but was 'Forty two'.",
                args.errorMessage());
}

...
public String errorMessage() throws Exception {
    switch (errorCode) {
        case OK:
            throw new Exception("TILT: Should not get here.");
        case UNEXPECTED_ARGUMENT:
            return unexpectedArgumentMessage();
        case MISSING_STRING:
            return String.format("Could not find string parameter for -%c.",
                                 errorArgumentId);
        case INVALID_INTEGER:
            return String.format("Argument -%c expects an integer but was '%s'.",
                                 errorArgumentId, errorParameter);
        case MISSING_INTEGER:
            return String.format("Could not find integer parameter for -%c.",
                                 errorArgumentId);
        case INVALID_DOUBLE:
            return String.format("Argument -%c expects a double but was '%s'.",
                                 errorArgumentId, errorParameter);
        case MISSING_DOUBLE:
            return String.format("Could not find double parameter for -%c.",
                                 errorArgumentId);
    }
    return "";
}

```

E i test continuano a passare. Il test successivo verifica se un argomento `double` viene rilevato correttamente.

```

public void testMissingDouble() throws Exception {
    Args args = new Args("x##", new String[] {"-x"});
    assertFalse(args.isValid());
    assertEquals(0, args.cardinality());
    assertFalse(args.has('x'));
    assertEquals(0.0, args.getDouble('x'), 0.01);
    assertEquals("Could not find double parameter for -x.", args.errorMessage());
}

```

}

Questo passa come previsto. L'abbiamo scritto semplicemente per completezza.

Il codice per le eccezioni non è bellissimo e in realtà non appartiene neppure alla classe `Args`. Lanciamo anche una `ParseException`, che in realtà non ci compete. Pertanto uniamo tutte le eccezioni in un'unica classe `ArgsException` e trasferiamola in un suo modulo.

```
public class ArgsException extends Exception {
    private char errorArgumentId = '\0';
    private String errorParameter = "TILT";
    private ErrorCode errorCode = ErrorCode.OK;
    public ArgsException() {}
    public ArgsException(String message) {super(message);}
    public enum ErrorCode {
        OK, MISSING_STRING, MISSING_INTEGER, INVALID_INTEGER, UNEXPECTED_ARGUMENT,
        MISSING_DOUBLE, INVALID_DOUBLE}
    ...
}

public class Args {
    ...
    private char errorArgumentId = '\0';
    private String errorParameter = "TILT";
    private ArgsException.ErrorCode errorCode = ArgsException.ErrorCode.OK;
    private List<String> argsList;
    public Args(String schema, String[] args) throws ArgsException {
        this.schema = schema;
        argsList = Arrays.asList(args);
        valid = parse();
    }
    private boolean parse() throws ArgsException {
        if (schema.length() == 0 && argsList.size() == 0)
            return true;
        parseSchema();
        try {
            parseArguments();
        } catch (ArgsException e) {
        }
        return valid;
    }
    private boolean parseSchema() throws ArgsException {
        ...
    }
    private void parseSchemaElement(String element) throws ArgsException {
        ...
        else
            throw new ArgsException(
                String.format("Argument: %c has invalid format: %s.",
                             elementId, elementTail));
    }
    private void validateSchemaElementId(char elementId) throws ArgsException {
        if (!Character.isLetter(elementId)) {
            throw new ArgsException(
                "Bad character:" + elementId + "in Args format: " + schema);
        }
    }
}
```

```

        }
    }

...
private void parseElement(char argChar) throws ArgsException {
    if (setArgument(argChar))
        argsFound.add(argChar);
    else {
        unexpectedArguments.add(argChar);
        errorCode = ArgsException.ErrorCode.UNEXPECTED_ARGUMENT;
        valid = false;
    }
}

...
private class StringArgumentMarshaler implements ArgumentMarshaler {
    private String stringValue = "";
    public void set(Iterator<String> currentArgument) throws ArgsException {
        try {
            stringValue = currentArgument.next();
        } catch (NoSuchElementException e) {
            errorCode = ArgsException.ErrorCode.MISSING_STRING;
            throw new ArgsException();
        }
    }
    public Object get() {
        return stringValue;
    }
}
private class IntegerArgumentMarshaler implements ArgumentMarshaler {
    private int intValue = 0;
    public void set(Iterator<String> currentArgument) throws ArgsException {
        String parameter = null;
        try {
            parameter = currentArgument.next();
            intValue = Integer.parseInt(parameter);
        } catch (NoSuchElementException e) {
            errorCode = ArgsException.ErrorCode.MISSING_INTEGER;
            throw new ArgsException();
        } catch (NumberFormatException e) {
            errorParameter = parameter;
            errorCode = ArgsException.ErrorCode.INVALID_INTEGER;
            throw new ArgsException();
        }
    }
    public Object get() {
        return intValue;
    }
}
private class DoubleArgumentMarshaler implements ArgumentMarshaler {
    private double doubleValue = 0;
    public void set(Iterator<String> currentArgument) throws ArgsException {
        String parameter = null;
        try {
            parameter = currentArgument.next();
            doubleValue = Double.parseDouble(parameter);
        } catch (NoSuchElementException e) {
            errorCode = ArgsException.ErrorCode.MISSING_DOUBLE;
            throw new ArgsException();
        } catch (NumberFormatException e) {
            errorParameter = parameter;
            errorCode = ArgsException.ErrorCode.INVALID_DOUBLE;
            throw new ArgsException();
        }
    }
}

```

```

        }
    }
    public Object get() {
        return doubleValue;
    }
}
}

```

Ora è tutto a posto. Adesso l'unica eccezione lanciata da `Args` è `ArgsException`. Spostando `ArgsException` in un suo modulo possiamo spostare molto del codice di supporto degli errori in tale modulo e fuori dal modulo `Args`. Questa è la collocazione naturale di quel codice e ciò aiuta anche a ripulire il modulo `Args` e a farlo evolvere.

Pertanto ora abbiamo separato completamente il codice per la gestione delle eccezioni degli errori dal modulo `Args` (Listati da 14.13 a 14.16). Tutto ciò è stato ottenuto tramite una serie di circa 30 piccoli passi, ognuno dei quali è stato sottoposto ai test.

Listato 14.13 ArgsTest.java.

```

package com.objectmentor.utilities.args;
import junit.framework.TestCase;
public class ArgsTest extends TestCase {
    public void testCreateWithNoSchemaOrArguments() throws Exception {
        Args args = new Args("", new String[0]);
        assertEquals(0, args.cardinality());
    }
    public void testWithNoSchemaButWithOneArgument() throws Exception {
        try {
            new Args("", new String[]{"-x"});
            fail();
        } catch (ArgsException e) {
            assertEquals(ArgsException.ErrorCode.UNEXPECTED_ARGUMENT,
                        e.getErrorCode());
            assertEquals('x', e.getErrorArgumentId());
        }
    }
    public void testWithNoSchemaButWithMultipleArguments() throws Exception {
        try {
            new Args("", new String[]{"-x", "-y"});
            fail();
        } catch (ArgsException e) {
            assertEquals(ArgsException.ErrorCode.UNEXPECTED_ARGUMENT,
                        e.getErrorCode());
            assertEquals('x', e.getErrorArgumentId());
        }
    }
    public void testNonLetterSchema() throws Exception {
        try {
            new Args("*", new String[]{});
            fail("Args constructor should have thrown exception");
        }
    }
}

```

```

        } catch (ArgsException e) {
            assertEquals(ArgsException.ErrorCode.INVALID_ARGUMENT_NAME,
                        e.getErrorCode());
            assertEquals('*', e.getErrorArgumentId());
        }
    }
    public void testInvalidArgumentFormat() throws Exception {
        try {
            new Args("f~", new String[] {});
            fail("Args constructor should have throws exception");
        } catch (ArgsException e) {
            assertEquals(ArgsException.ErrorCode.INVALID_FORMAT, e.getErrorCode());
            assertEquals('f', e.getErrorArgumentId());
        }
    }
    public void testSimpleBooleanPresent() throws Exception {
        Args args = new Args("x", new String[]{"-x"});
        assertEquals(1, args.cardinality());
        assertEquals(true, args.getBoolean('x'));
    }
    public void testSimpleStringPresent() throws Exception {
        Args args = new Args("x*", new String[]{"-x", "param"});
        assertEquals(1, args.cardinality());
        assertTrue(args.has('x'));
        assertEquals("param", args.getString('x'));
    }
    public void testMissingStringArgument() throws Exception {
        try {
            new Args("x*", new String[]{"-x"});
            fail();
        } catch (ArgsException e) {
            assertEquals(ArgsException.ErrorCode.MISSING_STRING, e.getErrorCode());
            assertEquals('x', e.getErrorArgumentId());
        }
    }
    public void testSpacesInFormat() throws Exception {
        Args args = new Args("x, y", new String[]{"-xy"});
        assertEquals(2, args.cardinality());
        assertTrue(args.has('x'));
        assertTrue(args.has('y'));
    }
    public void testSimpleIntPresent() throws Exception {
        Args args = new Args("x#", new String[]{"-x", "42"});
        assertEquals(1, args.cardinality());
        assertTrue(args.has('x'));
        assertEquals(42, args.getInt('x'));
    }
    public void testInvalidInteger() throws Exception {
        try {
            new Args("x#", new String[]{"-x", "Forty two"});
            fail();
        } catch (ArgsException e) {
            assertEquals(ArgsException.ErrorCode.INVALID_INTEGER, e.getErrorCode());
            assertEquals('x', e.getErrorArgumentId());
            assertEquals("Forty two", e.getErrorParameter());
        }
    }
    public void testMissingInteger() throws Exception {
        try {
            new Args("x#", new String[]{"-x"});
            fail();
        }
    }
}

```

```

        } catch (ArgsException e) {
            assertEquals(ArgsException.ErrorCode.MISSING_INTEGER, e.getErrorCode());
            assertEquals('x', e.getErrorArgumentId());
        }
    }

    public void testSimpleDoublePresent() throws Exception {
        Args args = new Args("x##", new String[]{"-x", "42.3"});
        assertEquals(1, args.cardinality());
        assertTrue(args.has('x'));
        assertEquals(42.3, args.getDouble('x'), .001);
    }

    public void testInvalidDouble() throws Exception {
        try {
            new Args("x##", new String[]{"-x", "Forty two"});
            fail();
        } catch (ArgsException e) {
            assertEquals(ArgsException.ErrorCode.INVALID_DOUBLE, e.getErrorCode());
            assertEquals('x', e.getErrorArgumentId());
            assertEquals("Forty two", e.getErrorParameter());
        }
    }

    public void testMissingDouble() throws Exception {
        try {
            new Args("x##", new String[]{"-x"});
            fail();
        } catch (ArgsException e) {
            assertEquals(ArgsException.ErrorCode.MISSING_DOUBLE, e.getErrorCode());
            assertEquals('x', e.getErrorArgumentId());
        }
    }
}
}

```

Listato 14.14 ArgsExceptionTest.java.

```

public class ArgsExceptionTest extends TestCase {
    public void testUnexpectedMessage() throws Exception {
        ArgsException e =
            new ArgsException(ArgsException.ErrorCode.UNEXPECTED_ARGUMENT,
                'x', null);
        assertEquals("Argument -x unexpected.", e.errorMessage());
    }

    public void testMissingStringMessage() throws Exception {
        ArgsException e = new ArgsException(ArgsException.ErrorCode.MISSING_STRING,
            'x', null);
        assertEquals("Could not find string parameter for -x.", e.errorMessage());
    }

    public void testInvalidIntegerMessage() throws Exception {
        ArgsException e =
            new ArgsException(ArgsException.ErrorCode.INVALID_INTEGER,
                'x', "Forty two");
        assertEquals("Argument -x expects an integer but was 'Forty two'.",
            e.errorMessage());
    }

    public void testMissingIntegerMessage() throws Exception {
        ArgsException e =
            new ArgsException(ArgsException.ErrorCode.MISSING_INTEGER, 'x', null);
        assertEquals("Could not find integer parameter for -x.", e.errorMessage());
    }

    public void testInvalidDoubleMessage() throws Exception {
        ArgsException e = new ArgsException(ArgsException.ErrorCode.INVALID_DOUBLE,

```

```

        'x', "Forty two");
    assertEquals("Argument -x expects a double but was 'Forty two'.",
                e.errorMessage());
}
public void testMissingDoubleMessage() throws Exception {
    ArgsException e = new ArgsException(ArgsException.ErrorCode.MISSING_DOUBLE,
                                         'x', null);
    assertEquals("Could not find double parameter for -x.", e.errorMessage());
}
}

```

Listato 14.15 ArgsException.java.

```

public class ArgsException extends Exception {
    private char errorArgumentId = '\0';
    private String errorParameter = "TILT";
    private ErrorCode errorCode = ErrorCode.OK;
    public ArgsException() {}
    public ArgsException(String message) {super(message);}
    public ArgsException(ErrorCode errorCode) {
        this.errorCode = errorCode;
    }
    public ArgsException(ErrorCode errorCode, String errorParameter) {
        this.errorCode = errorCode;
        this.errorParameter = errorParameter;
    }
    public ArgsException(ErrorCode errorCode, char errorArgumentId,
                         String errorParameter) {
        this.errorCode = errorCode;
        this.errorParameter = errorParameter;
        this.errorArgumentId = errorArgumentId;
    }
    public char getErrorArgumentId() {
        return errorArgumentId;
    }
    public void setErrorArgumentId(char errorArgumentId) {
        this.errorArgumentId = errorArgumentId;
    }
    public String getErrorParameter() {
        return errorParameter;
    }
    public void setErrorParameter(String errorParameter) {
        this.errorParameter = errorParameter;
    }
    public ErrorCode getErrorCode() {
        return errorCode;
    }
    public void setErrorCode(ErrorCode errorCode) {
        this.errorCode = errorCode;
    }
    public String errorMessage() throws Exception {
        switch (errorCode) {
            case OK:
                throw new Exception("TILT: Should not get here.");
            case UNEXPECTED_ARGUMENT:
                return String.format("Argument -%c unexpected.", errorArgumentId);
            case MISSING_STRING:
                return String.format("Could not find string parameter for -%c.",
                                    errorArgumentId);
            case INVALID_INTEGER:

```

```

        return String.format("Argument -%c expects an integer but was '%s'.",
                           errorArgumentId, errorParameter);
    case MISSING_INTEGER:
        return String.format("Could not find integer parameter for -%c.",
                           errorArgumentId);
    case INVALID_DOUBLE:
        return String.format("Argument -%c expects a double but was '%s'.",
                           errorArgumentId, errorParameter);
    case MISSING_DOUBLE:
        return String.format("Could not find double parameter for -%c.",
                           errorArgumentId);
    }
    return "";
}
public enum ErrorCode {
    OK, INVALID_FORMAT, UNEXPECTED_ARGUMENT, INVALID_ARGUMENT_NAME,
    MISSING_STRING,
    MISSING_INTEGER, INVALID_INTEGER,
    MISSING_DOUBLE, INVALID_DOUBLE
}
}

```

Listato 14.16 Args.java.

```

public class Args {
    private String schema;
    private Map<Character, ArgumentMarshaler> marshalers =
        new HashMap<Character, ArgumentMarshaler>();
    private Set<Character> argsFound = new HashSet<Character>();
    private Iterator<String> currentArgument;
    private List<String> argsList;
    public Args(String schema, String[] args) throws ArgsException {
        this.schema = schema;
        argsList = Arrays.asList(args);
        parse();
    }
    private void parse() throws ArgsException {
        parseSchema();
        parseArguments();
    }
    private boolean parseSchema() throws ArgsException {
        for (String element : schema.split(",")) {
            if (element.length() > 0) {
                parseSchemaElement(element.trim());
            }
        }
        return true;
    }
    private void parseSchemaElement(String element) throws ArgsException {
        char elementId = element.charAt(0);
        String elementTail = element.substring(1);
        validateSchemaElementId(elementId);
        if (elementTail.length() == 0)
            marshalers.put(elementId, new BooleanArgumentMarshaler());
        else if (elementTail.equals("*"))
            marshalers.put(elementId, new StringArgumentMarshaler());
        else if (elementTail.equals("#"))
            marshalers.put(elementId, new IntegerArgumentMarshaler());
        else if (elementTail.equals("##"))
            marshalers.put(elementId, new DoubleArgumentMarshaler());
    }
}

```

```

        else
            throw new ArgsException(ArgsException.ErrorCode.INVALID_FORMAT,
                                   elementId, elementTail);
    }
    private void validateSchemaElementId(char elementId) throws ArgsException {
        if (!Character.isLetter(elementId)) {
            throw new ArgsException(ArgsException.ErrorCode.INVALID_ARGUMENT_NAME,
                                   elementId, null);
        }
    }
    private void parseArguments() throws ArgsException {
        for (currentArgument = argsList.iterator(); currentArgument.hasNext();) {
            String arg = currentArgument.next();
            parseArgument(arg);
        }
    }
    private void parseArgument(String arg) throws ArgsException {
        if (arg.startsWith("-"))
            parseElements(arg);
    }
    private void parseElements(String arg) throws ArgsException {
        for (int i = 1; i < arg.length(); i++)
            parseElement(arg.charAt(i));
    }
    private void parseElement(char argChar) throws ArgsException {
        if (setArgument(argChar))
            argsFound.add(argChar);
        else {
            throw new ArgsException(ArgsException.ErrorCode.UNEXPECTED_ARGUMENT,
                                   argChar, null);
        }
    }
    private boolean setArgument(char argChar) throws ArgsException {
        ArgumentMarshaler m = marshalers.get(argChar);
        if (m == null)
            return false;
        try {
            m.set(currentArgument);
            return true;
        } catch (ArgsException e) {
            e.setErrorArgumentId(argChar);
            throw e;
        }
    }
    public int cardinality() {
        return argsFound.size();
    }
    public String usage() {
        if (schema.length() > 0)
            return "-[" + schema + "]";
        else
            return "";
    }
    public boolean getBoolean(char arg) {
        ArgumentMarshaler am = marshalers.get(arg);
        boolean b = false;
        try {
            b = am != null && (Boolean) am.get();
        } catch (ClassCastException e) {
            b = false;
        }
    }
}

```

```

        return b;
    }
    public String getString(char arg) {
        ArgumentMarshaler am = marshalers.get(arg);
        try {
            return am == null ? "" : (String) am.get();
        } catch (ClassCastException e) {
            return "";
        }
    }
    public int getInt(char arg) {
        ArgumentMarshaler am = marshalers.get(arg);
        try {
            return am == null ? 0 : (Integer) am.get();
        } catch (Exception e) {
            return 0;
        }
    }
    public double getDouble(char arg) {
        ArgumentMarshaler am = marshalers.get(arg);
        try {
            return am == null ? 0 : (Double) am.get();
        } catch (Exception e) {
            return 0.0;
        }
    }
    public boolean has(char arg) {
        return argsFound.contains(arg);
    }
}

```

La maggior parte delle modifiche alla classe `Args` riguardava cancellazioni. Molto del codice è stato semplicemente estratto da `Args` e inserito in `ArgsException`. Bene. Abbiamo anche spostato tutti gli `ArgumentMarshaler` nei rispettivi file. Meglio!

La progettazione del software è soprattutto corretto partizionamento: creare posizioni appropriate in cui inserire le varie parti del codice. Questa separazione degli ambiti rende il codice molto più semplice da comprendere e da manutenere.

Di particolare interesse è il metodo `errorMessage` di `ArgsException`. Chiaramente era una violazione del principio SRP inserire la formattazione del messaggio d'errore in `Args`. `Args` dovrebbe occuparsi solo dell'elaborazione degli argomenti, non della formattazione dei messaggi d'errore. Tuttavia, ha davvero senso porre il codice di formattazione dei messaggi d'errore in `ArgsException`?

Francamente, si tratta di un compromesso. Gli utenti che non gradiscono i messaggi d'errore forniti da `ArgsException` dovranno scrivere i propri. Ma la comodità di avere dei messaggi d'errore già pronti all'uso non è da sottovalutare.

A questo punto dovrebbe essere chiaro che siamo davvero vicini alla soluzione finale che abbiamo presentato all'inizio di questo capitolo. Le ultime trasformazioni ve le lascio come esercizio.

Conclusioni

Non è sufficiente che il codice funzioni. Il codice che, semplicemente, “funziona” spesso è molto difettoso. I programmatore che si accontentano di “codice funzionante” non si comportano in modo professionale. Forse temono di non avere il tempo per migliorare la struttura e la composizione del loro codice, ma non sono d'accordo. Nulla, più del codice scritto male, ha un effetto più profondo e più deleterio a lungo termine su un progetto in corso di sviluppo. Le scadenze sottovalutate possono essere riconsiderate, i cattivi requisiti possono essere ridefiniti. La cattive dinamiche interne del team possono essere risolte. Ma il cattivo codice resta in agguato e “fermenta”, divenendo un peso inesorabile che trascina verso il basso tutto il team. Quante volte ho visto dei team procedere a passo di lumaca perché, per la fretta, hanno fatto del codice una sorta di palude malsana, che dominerà per sempre il loro destino.

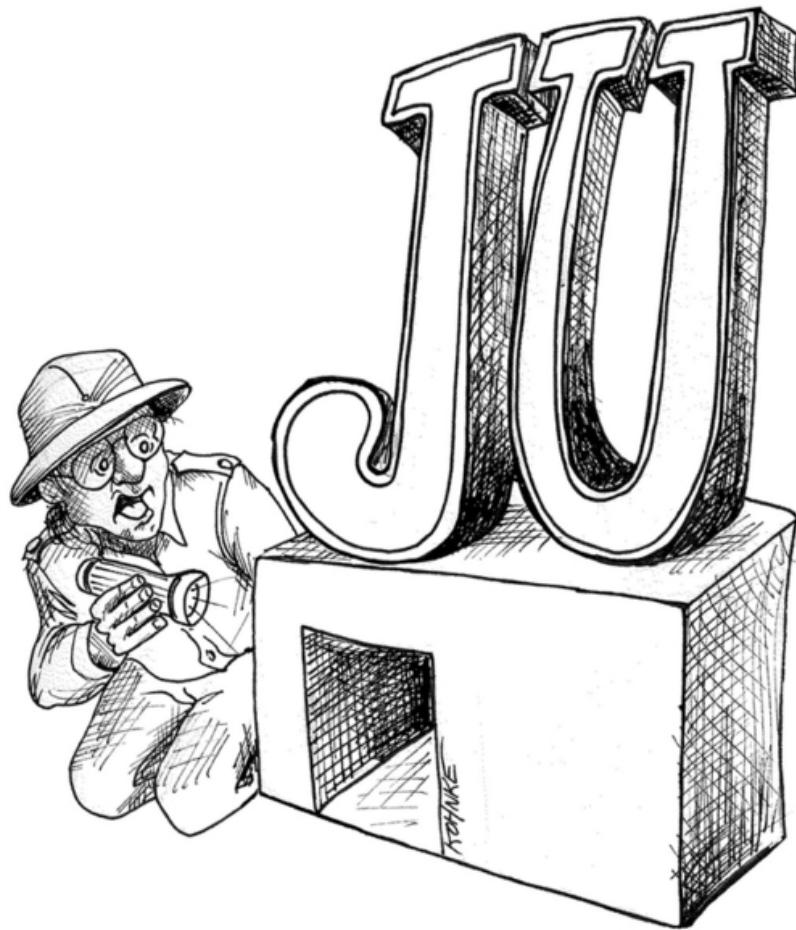
Naturalmente il cattivo codice può essere ripulito. Ma ripulire costa. Nel codice mal scritto, i moduli si insinuano l'uno nell'altro, creando una grande quantità di dipendenze complesse e nascoste. Trovare e risolvere le vecchie dipendenze è un compito lungo e arduo. D'altra parte, gestire del codice pulito è relativamente semplice. Se avete combinato un pasticcio in un modulo in una mattina, è facile sistemarlo

in un pomeriggio. Ancora meglio, se avete combinato un pasticcio cinque minuti fa, è facile risolverlo subito.

Pertanto la soluzione consiste nel mantenere continuamente il vostro codice il più possibile pulito e semplice. Non concedete mai a un difetto il tempo di prendere piede.

Capitolo 15

JUnit



JUnit è uno dei più famosi fra i framework Java. Si tratta di un framework semplice come concetti, preciso nelle definizioni ed elegante nell'implementazione. Ma qual è l'aspetto del suo codice? In questo capitolo tratteremo un esempio tratto dal framework JUnit.

Il framework JUnit

JUnit ha avuto molti autori, ma è nato da un'idea di Kent Beck ed Eric Gamma mentre erano in volo verso Atlanta. Kent voleva imparare il linguaggio Java ed Eric voleva imparare a usare il framework di test Smalltalk di Kent. “Che cosa c’è di più naturale per un paio di nerd costretti a stare in uno spazio ristretto se non tirare fuori il portatile e iniziare a programmare?” (Kent Beck, *JUnit Pocket Guide*, O'Reilly, California 2004, p. 43.) Dopo tre ore di lavoro ad alta quota, avevano scritto gli elementi base di JUnit.

Il modulo che esamineremo è un interessante frammento di codice che aiuta a identificare gli errori di confronto fra stringhe. Questo modulo si chiama `ComparisonCompactor`. Date due stringhe differenti, come `ABCDE` e `ABXDE`, mostra la differenza generando una stringa come `<...B[X]D...`.

Potrei proseguire nella spiegazione, ma i casi di test svolgeranno meglio il compito. Pertanto date un’occhiata al Listato 15.1 e capirete meglio i requisiti di questo modulo. Già che ci siete, sottoponete a critica la struttura dei test. Potrebbero essere più semplici o più ovvi?

Listato 15.1 ComparisonCompactorTest.java.

```
package junit.tests.framework;
import junit.framework.ComparisonCompactor;
import junit.framework.TestCase;
public class ComparisonCompactorTest extends TestCase {
    public void testMessage() {
        String failure = new ComparisonCompactor(0, "b", "c").compact("a");
        assertTrue("a expected:<[b]> but was:<[c]>".equals(failure));
    }
    public void testStartSame() {
        String failure= new ComparisonCompactor(1, "ba", "bc").compact(null);
        assertEquals("expected:<b[a]> but was:<b[c]>", failure);
    }
    public void testEndSame() {
        String failure= new ComparisonCompactor(1, "ab", "cb").compact(null);
        assertEquals("expected:<[a]b> but was:<[c]b>", failure);
    }
    public void testSame() {
        String failure= new ComparisonCompactor(1, "ab", "ab").compact(null);
        assertEquals("expected:<ab> but was:<ab>", failure);
    }
}
```

```

public void testNoContextStartAndEndSame() {
    String failure= new ComparisonCompactor(0, "abc", "adc").compact(null);
    assertEquals("expected:<...[b]...> but was:<...[d]...>", failure);
}
public void testStartAndEndContext() {
    String failure= new ComparisonCompactor(1, "abc", "adc").compact(null);
    assertEquals("expected:<a[b]c> but was:<a[d]c>", failure);
}
public void testStartAndEndContextWithEllipses() {
    String failure=
        new ComparisonCompactor(1, "abcde", "abfde").compact(null);
    assertEquals("expected:<...b[c]d...> but was:<...b[f]d...>", failure);
}
public void testComparisonErrorStartSameComplete() {
    String failure= new ComparisonCompactor(2, "ab", "abc").compact(null);
    assertEquals("expected:<ab[]> but was:<ab[c]>", failure);
}
public void testComparisonErrorEndSameComplete() {
    String failure= new ComparisonCompactor(0, "bc", "abc").compact(null);
    assertEquals("expected:<[]...> but was:<[a]...>", failure);
}
public void testComparisonErrorEndSameCompleteContext() {
    String failure= new ComparisonCompactor(2, "bc", "abc").compact(null);
    assertEquals("expected:<[]bc> but was:<[a]bc>", failure);
}
public void testComparisonErrorOverlappingMatches() {
    String failure= new ComparisonCompactor(0, "abc", "abbc").compact(null);
    assertEquals("expected:<...[]...> but was:<...[b]...>", failure);
}
public void testComparisonErrorOverlappingMatchesContext() {
    String failure= new ComparisonCompactor(2, "abc", "abbc").compact(null);
    assertEquals("expected:<ab[]c> but was:<ab[b]c>", failure);
}
public void testComparisonErrorOverlappingMatches2() {
    String failure= new ComparisonCompactor(0, "abcdde", "abcde").compact(null);
    assertEquals("expected:<...[d]...> but was:<...[]...>", failure);
}
public void testComparisonErrorOverlappingMatches2Context() {
    String failure=
        new ComparisonCompactor(2, "abcdde", "abcde").compact(null);
    assertEquals("expected:<...cd[d]e> but was:<...cd[]e>", failure);
}
public void testComparisonErrorWithActualNull() {
    String failure= new ComparisonCompactor(0, "a", null).compact(null);
    assertEquals("expected:<a> but was:<null>", failure);
}
public void testComparisonErrorWithActualNullContext() {
    String failure= new ComparisonCompactor(2, "a", null).compact(null);
    assertEquals("expected:<a> but was:<null>", failure);
}
public void testComparisonErrorWithExpectedNull() {
    String failure= new ComparisonCompactor(0, null, "a").compact(null);
    assertEquals("expected:<null> but was:<a>", failure);
}
public void testComparisonErrorWithExpectedNullContext() {
    String failure= new ComparisonCompactor(2, null, "a").compact(null);
    assertEquals("expected:<null> but was:<a>", failure);
}
public void testBug609972() {
    String failure= new ComparisonCompactor(10, "S&P500", "0").compact(null);
}

```

```

        assertEquals("expected:<[S&P50]0> but was:<[]0>", failure);
    }
}

```

Ho eseguito su `ComparisonCompactor` un'analisi di copertura del codice usando questi test. Il codice è coperto al 100 per cento. Ogni riga di codice, ogni istruzione `if` e ciclo `for` viene eseguito dai test. Ciò mi dà un'elevata sicurezza che il codice funzioni ed eleva il mio rispetto per l'abilità dei suoi autori.

Il codice di `ComparisonCompactor` è riportato nel Listato 15.2. Dedicate qualche istante a considerare questo codice. Penso che lo troverete ben partizionato, ragionevolmente espressivo e semplice nella struttura. Quando avrete finito, torneremo a parlarne.

Listato 15.2 ComparisonCompactor.java (originale).

```

package junit.framework;
public class ComparisonCompactor {
    private static final String ELLIPSIS = "...";
    private static final String DELTA_END = "]";
    private static final String DELTA_START = "[";
    private int fContextLength;
    private String fExpected;
    private String fActual;
    private int fPrefix;
    private int fSuffix;
    public ComparisonCompactor(int contextLength,
                               String expected, String actual) {
        fContextLength = contextLength;
        fExpected = expected;
        fActual = actual;
    }
    public String compact(String message) {
        if (fExpected == null || fActual == null || areStringsEqual())
            return Assert.format(message, fExpected, fActual);
        findCommonPrefix();
        findCommonSuffix();
        String expected = compactString(fExpected);
        String actual = compactString(fActual);
        return Assert.format(message, expected, actual);
    }
    private String compactString(String source) {
        String result = DELTA_START +
                       source.substring(fPrefix, source.length() - fSuffix + 1) +
                       DELTA_END;
        if (fPrefix > 0)
            result = computeCommonPrefix() + result;
        if (fSuffix > 0)
            result = result + computeCommonSuffix();
        return result;
    }
}

```

```

private void findCommonPrefix() {
    fPrefix = 0;
    int end = Math.min(fExpected.length(), fActual.length());
    for (; fPrefix < end; fPrefix++) {
        if (fExpected.charAt(fPrefix) != fActual.charAt(fPrefix))
            break;
    }
}
private void findCommonSuffix() {
    int expectedSuffix = fExpected.length() - 1;
    int actualSuffix = fActual.length() - 1;
    for (; actualSuffix >= fPrefix && expectedSuffix >= fPrefix;
          actualSuffix--, expectedSuffix--) {
        if (fExpected.charAt(expectedSuffix) != fActual.charAt(actualSuffix))
            break;
    }
    fSuffix = fExpected.length() - expectedSuffix;
}
private String computeCommonPrefix() {
    return (fPrefix > fContextLength ? ELLIPSIS : "") +
           fExpected.substring(Math.max(0, fPrefix - fContextLength), fPrefix);
}
private String computeCommonSuffix() {
    int end = Math.min(fExpected.length() - fSuffix + 1 + fContextLength,
                       fExpected.length());
    return fExpected.substring(fExpected.length() - fSuffix + 1, end) +
           (fExpected.length() - fSuffix + 1 < fExpected.length() - fContextLength ? ELLIPSIS : "");
}
private boolean areStringsEqual() {
    return fExpected.equals(fActual);
}
}

```

Forse avrete qualche dubbio su questo modulo. Vi sono alcune lunghe espressioni e alcuni strani `+ 1` e così via. Ma in generale questo modulo è piuttosto buono. A parte questo, poteva anche avere l'aspetto del Listato 15.3.

Listato 15.3 ComparisonCompactor.java (defattorizzato).

```

package junit.framework;
public class ComparisonCompactor {
    private int ctxt;
    private String s1;
    private String s2;
    private int pfx;
    private int sfx;
    public ComparisonCompactor(int ctxt, String s1, String s2) {
        this.ctxt = ctxt;
        this.s1 = s1;
        this.s2 = s2;
    }
    public String compact(String msg) {
        if (s1 == null || s2 == null || s1.equals(s2))
            return Assert.format(msg, s1, s2);
    }
}

```

```

    pfx = 0;
    for (; pfx < Math.min(s1.length(), s2.length()); pfx++) {
        if (s1.charAt(pfx) != s2.charAt(pfx))
            break;
    }
    int sfx1 = s1.length() - 1;
    int sfx2 = s2.length() - 1;
    for (; sfx2 >= pfx && sfx1 >= pfx; sfx2--, sfx1--) {
        if (s1.charAt(sfx1) != s2.charAt(sfx2))
            break;
    }
    sfx = s1.length() - sfx1;
    String cmp1 = compactString(s1);
    String cmp2 = compactString(s2);
    return Assert.format(msg, cmp1, cmp2);
}
private String compactString(String s) {
    String result =
        "[" + s.substring(pfx, s.length() - sfx + 1) + "]";
    if (pfx > 0)
        result = (pfx > ctxt ? "... " : "") +
            s1.substring(Math.max(0, pfx - ctxt), pfx) + result;
    if (sfx > 0) {
        int end = Math.min(s1.length() - sfx + 1 + ctxt, s1.length());
        result = result + (s1.substring(s1.length() - sfx + 1, end) +
            (s1.length() - sfx + 1 < s1.length() - ctxt ? "... " : ""));
    }
    return result;
}
}

```

Anche se gli autori hanno lasciato questo modulo in un'ottima forma, la *regola dei boy-scout* (vedi “La regola dei boy-scout”, nel Capitolo 1) ci dice che dovremmo lasciare le cose più pulite di come le abbiamo trovate. Pertanto, come possiamo migliorare il codice originale presentato nel Listato 15.2?

La prima cosa che toglierei è il prefisso `f` delle variabili membro [N6]. Gli ambienti odierni rendono ridondante questo tipo di codifica dello scope. Pertanto eliminiamo tutte quelle `f`.

```

private int contextLength;
private String expected;
private String actual;
private int prefix;

private int suffix;

```

Poi,abbiamo un costrutto condizionale non incapsulato all'inizio della funzione `compact` [G28].

```

public String compact(String message) {
    if (expected == null || actual == null || areStringsEqual())

```

```

        return Assert.format(message, expected, actual);
    findCommonPrefix();
    findCommonSuffix();
    String expected = compactString(this.expected);
    String actual = compactString(this.actual);
    return Assert.format(message, expected, actual);

}

```

Questa condizione dovrebbe essere incapsulata per chiarirne l'intento. Pertanto estraiamola in un metodo che ne spieghi il significato.

```

public String compact(String message) {
    if (shouldNotCompact())
        return Assert.format(message, expected, actual);
    findCommonPrefix();
    findCommonSuffix();
    String expected = compactString(this.expected);
    String actual = compactString(this.actual);
    return Assert.format(message, expected, actual);
}

private boolean shouldNotCompact() {
    return expected == null || actual == null || areStringsEqual();
}

```

Non mi piace molto la notazione `this.expected` e `this.actual` nella funzione `compact`. Questo si è verificato quando abbiamo cambiato il nome di `fExpected` in `expected`. Perché in questa funzione vi sono variabili che hanno lo stesso nome delle variabili membro? Non rappresentano qualcos'altro [N4]? Dovremmo rendere non ambiguo il loro nome.

```

String compactExpected = compactString(expected);
String compactActual = compactString(actual);

```

Le negazioni sono un po' più difficili da comprendere delle affermazioni [G29]. Pertanto trasformiamo quell'istruzione `if` in cima e invertiamo il senso della condizione.

```

public String compact(String message) {
    if (canBeCompacted()) {
        findCommonPrefix();
        findCommonSuffix();
        String compactExpected = compactString(expected);
        String compactActual = compactString(actual);
        return Assert.format(message, compactExpected, compactActual);
    } else {
        return Assert.format(message, expected, actual);
    }
}
private boolean canBeCompacted() {
    return expected != null && actual != null && !areStringsEqual();
}

```

}

Il nome della funzione è strano [N7]. Anche se compatta le stringhe, in realtà potrebbe non compattarle affatto se `canBeCompacted` restituisce `false`. Pertanto, chiamare questa funzione `compact` cela l'effetto collaterale del controllo degli errori. Notate anche che la funzione restituisce un messaggio formattato, non solo le stringhe compattate. Pertanto il nome della funzione in realtà dovrebbe essere `formatCompactedComparison`. Questo nome è più logico, se letto insieme all'argomento della funzione:

```
public String formatCompactedComparison(String message) {
```

Il corpo dell'istruzione `if` è il punto in cui si verifica davvero il compattamento delle stringhe `expected` e `actual`. Dovremmo estrarre in un metodo chiamato `compactExpectedAndActual`. Tuttavia, vogliamo che sia la funzione `formatCompactedComparison` a svolgere tutta la formattazione. La funzione `compact...` non dovrebbe fare altro che compattare [G30].

Pertanto suddividiamola come segue:

```
...
private String compactExpected;
private String compactActual;
...
public String formatCompactedComparison(String message) {
    if (canBeCompacted()) {
        compactExpectedAndActual();
        return Assert.format(message, compactExpected, compactActual);
    } else {
        return Assert.format(message, expected, actual);
    }
}
private void compactExpectedAndActual() {
    findCommonPrefix();
    findCommonSuffix();
    compactExpected = compactString(expected);
    compactActual = compactString(actual);
}
```

Notate che questo ci ha spinto a promuovere `compactExpected` e `compactActual` in variabili membro. Non mi piace il modo in cui le ultime due righe della nuova funzione restituiscono le variabili, mentre le prime due no. Non usano convenzioni coerenti [G11]. Pertanto

dovremmo modificare `findCommonPrefix` e `findCommonSuffix` in modo che restituiscano i valori prefisso e suffisso.

```
private void compactExpectedAndActual() {
    prefixIndex = findCommonPrefix();
    suffixIndex = findCommonSuffix();
    compactExpected = compactString(expected);
    compactActual = compactString(actual);
}
private int findCommonPrefix() {
    int prefixIndex = 0;
    int end = Math.min(expected.length(), actual.length());
    for (; prefixIndex < end; prefixIndex++) {
        if (expected.charAt(prefixIndex) != actual.charAt(prefixIndex))
            break;
    }
    return prefixIndex;
}
private int findCommonSuffix() {
    int expectedSuffix = expected.length() - 1;
    int actualSuffix = actual.length() - 1;
    for (; actualSuffix >= prefixIndex && expectedSuffix >= prefixIndex;
        actualSuffix--, expectedSuffix--) {
        if (expected.charAt(expectedSuffix) != actual.charAt(actualSuffix))
            break;
    }
    return expected.length() - expectedSuffix;
}
```

Dovremmo anche modificare il nome delle variabili membro per renderlo un po' preciso [N1]; dopotutto, entrambi sono indici.

Un'attenta ispezione di `findCommonSuffix` evidenzia un *accoppiamento temporale nascosto* [G31]; dipende dal fatto che `prefixIndex` viene calcolato da `findCommonPrefix`. Se queste due funzioni venissero chiamate fuori ordine, produrrebbero una complessa sessione di debug. Pertanto, per evidenziare questo accoppiamento temporale, facciamo in modo che `findCommonSuffix` prenda il `prefixIndex` come argomento.

```
private void compactExpectedAndActual() {
    prefixIndex = findCommonPrefix();
    suffixIndex = findCommonSuffix(prefixIndex);
    compactExpected = compactString(expected);
    compactActual = compactString(actual);
}
private int findCommonSuffix(int prefixIndex) {
    int expectedSuffix = expected.length() - 1;
    int actualSuffix = actual.length() - 1;
    for (; actualSuffix >= prefixIndex && expectedSuffix >= prefixIndex;
        actualSuffix--, expectedSuffix--) {
        if (expected.charAt(expectedSuffix) != actual.charAt(actualSuffix))
            break;
    }
}
```

```

        }
        return expected.length() - expectedSuffix;
    }
}

```

Non ne sono molto soddisfatto. Il passaggio dell'argomento `prefixIndex` è un po' arbitrario [G32]. Riesce a stabilire l'ordinamento, ma non fa nulla per spiegare il motivo di tale ordinamento. Un altro programmatore potrebbe annullare ciò che abbiamo fatto perché non vi è alcuna indicazione del fatto che il parametro in realtà è obbligatorio. Pertanto seguiamo un'altra strada.

```

private void compactExpectedAndActual() {
    findCommonPrefixAndSuffix();
    compactExpected = compactString(expected);
    compactActual = compactString(actual);
}

private void findCommonPrefixAndSuffix() {
    findCommonPrefix();
    int expectedSuffix = expected.length() - 1;
    int actualSuffix = actual.length() - 1;
    for (; actualSuffix >= prefixIndex && expectedSuffix >= prefixIndex;
          actualSuffix--, expectedSuffix--) {
        if (expected.charAt(expectedSuffix) != actual.charAt(actualSuffix))
            break;
    }
    suffixIndex = expected.length() - expectedSuffix;
}

private void findCommonPrefix() {
    prefixIndex = 0;
    int end = Math.min(expected.length(), actual.length());
    for (; prefixIndex < end; prefixIndex++)
        if (expected.charAt(prefixIndex) != actual.charAt(prefixIndex))
            break;
}

```

Rimettiamo `findCommonPrefix` e `findCommonSuffix` dove si trovavano, cambiando il nome di `findCommonSuffix` in `findCommonPrefixAndSuffix` e facendole richiamare `findCommonPrefix` prima di ogni altra cosa. Questo chiarisce molto meglio rispetto alla soluzione precedente la natura temporale delle due funzioni. Inoltre evidenzia i difetti di `findCommonPrefixAndSuffix`. Ripuliamola un po'.

```

private void findCommonPrefixAndSuffix() {
    findCommonPrefix();
    int suffixLength = 1;
    for (; !suffixOverlapsPrefix(suffixLength); suffixLength++) {
        if (charFromEnd(expected, suffixLength) !=
            charFromEnd(actual, suffixLength))
            break;
    }
}

```

```

        }
        suffixIndex = suffixLength;
    }
    private char charFromEnd(String s, int i) {
        return s.charAt(s.length()-i);
    }
    private boolean suffixOverlapsPrefix(int suffixLength) {
        return actual.length() - suffixLength < prefixLength ||
            expected.length() - suffixLength < prefixLength;
    }
}

```

Molto meglio. Ora chiarisce che `suffixIndex` in realtà è la lunghezza del suffisso e non ha un buon nome. Lo stesso dicasi per `prefixIndex`. Sarebbe più coerente usare “length” al posto di “index”. Il problema è che la variabile `suffixIndex` non parte da zero, ma da 1 e così non è davvero una lunghezza. Questo è inoltre il motivo per cui vi sono tutti quei `+ 1` in `computeCommonSuffix` [G33]. Correggiamo il tutto. Il risultato è presentato nel Listato 15.4.

Listato 15.4 ComparisonCompactor.java (intermedio).

```

public class ComparisonCompactor {
    ...
    private int suffixLength;
    ...
    private void findCommonPrefixAndSuffix() {
        findCommonPrefix();
        suffixLength = 0;
        for (; !suffixOverlapsPrefix(suffixLength); suffixLength++) {
            if (charFromEnd(expected, suffixLength) !=
                charFromEnd(actual, suffixLength))
                break;
        }
    }
    private char charFromEnd(String s, int i) {
        return s.charAt(s.length() - i - 1);
    }
    private boolean suffixOverlapsPrefix(int suffixLength) {
        return actual.length() - suffixLength <= prefixLength ||
            expected.length() - suffixLength <= prefixLength;
    }
    private String compactString(String source) {
        String result = DELTA_START +
            source.substring(prefixLength, source.length() -
suffixLength) + DELTA_END;
        if (prefixLength > 0)
            result = computeCommonPrefix() + result;
        if (suffixLength > 0)
            result = result + computeCommonSuffix();
        return result;
    }
    private String computeCommonSuffix() {
        int end = Math.min(expected.length() - suffixLength +

```

```

        contextLength, expected.length());
    return expected.substring(expected.length() - suffixLength, end) +
        (expected.length() - suffixLength < expected.length() -
        contextLength ? ELLIPSIS : "");
}

```

Abbiamo sostituito `i + 1` in `computeCommonSuffix` con un `- 1` in `charFromEnd`, che ha perfettamente senso, e due operatori `<=` in `suffixOverlapsPrefix`, anch'essi perfettamente sensati. Questo ci ha consentito di cambiare il nome di `suffixIndex` in `suffixLength`, il che migliora notevolmente la leggibilità del codice.

Vi è però un problema. Eliminando `i + 1`, ho notato la seguente riga in `compactString`:

```
if (suffixLength > 0)
```

Dategli un'occhiata nel Listato 15.4. Ora `suffixLength` è inferiore di un'unità rispetto a prima, così devo cambiare l'operatore `>` in un `>=`. Ma questo non ha senso. Sembra sensato solamente *ora*, in questo momento! Questo significa che probabilmente si trattava di un bug. Non esattamente un bug. Dopo un'ulteriore analisi vediamo che l'istruzione `if ora` impedisce che venga aggiunto un suffisso di lunghezza zero. Prima di eseguire la modifica, l'istruzione `if` non interveniva mai, perché `suffixIndex` non poteva mai essere minore di uno!

Questo richiama in causa entrambe le istruzioni `if` di `compactString`! Sembra che entrambe debbano essere eliminate. Pertanto trasformiamole in commento ed eseguiamo i test. Passati! Pertanto ristrutturiamo `compactString` per eliminare le istruzioni `if` di troppo e semplifichiamo la funzione [G9].

```

private String compactString(String source) {
    return computeCommonPrefix() +
        DELTA_START +
        source.substring(prefixLength, source.length() - suffixLength) +
        DELTA_END +
        computeCommonSuffix();
}

```

Molto meglio! Ora vediamo che la funzione `compactString` non fa altro che comporre i frammenti. Possiamo probabilmente chiarirlo ancora di più. In effetti, potremmo apportare tante piccole migliorie. Ma invece di accompagnarvi nel resto delle modifiche, vi mostro solo il risultato, nel Listato 15.5.

Listato 15.5 ComparisonCompactor.java (finale).

```
package junit.framework;
public class ComparisonCompactor {
    private static final String ELLIPSIS = "...";
    private static final String DELTA_END = "]";
    private static final String DELTA_START = "[";
    private int contextLength;
    private String expected;
    private String actual;
    private int prefixLength;
    private int suffixLength;
    public ComparisonCompactor(int contextLength, String expected, String actual )
    {
        this.contextLength = contextLength;
        this.expected = expected;
        this.actual = actual;
    }
    public String formatCompactedComparison(String message) {
        String compactExpected = expected;
        String compactActual = actual;
        if (shouldBeCompacted()) {
            findCommonPrefixAndSuffix();
            compactExpected = compact(expected);
            compactActual = compact(actual);
        }
        return Assert.format(message, compactExpected, compactActual);
    }
    private boolean shouldBeCompacted() {
        return !shouldNotBeCompacted();
    }
    private boolean shouldNotBeCompacted() {
        return expected == null || actual == null || expected.equals(actual);
    }
    private void findCommonPrefixAndSuffix() {
        findCommonPrefix();
        suffixLength = 0;
        for (; !suffixOverlapsPrefix(); suffixLength++) {
            if (charFromEnd(expected, suffixLength) !=
                charFromEnd(actual, suffixLength))
                break;
        }
    }
    private char charFromEnd(String s, int i) {
        return s.charAt(s.length() - i - 1);
    }
    private boolean suffixOverlapsPrefix() {
        return actual.length() - suffixLength <= prefixLength ||
               expected.length() - suffixLength <= prefixLength;
    }
}
```

```

private void findCommonPrefix() {
    prefixLength = 0;
    int end = Math.min(expected.length(), actual.length());
    for (; prefixLength < end; prefixLength++)
        if (expected.charAt(prefixLength) != actual.charAt(prefixLength)) break;
}
private String compact(String s) {
    return new StringBuilder()
        .append(startingEllipsis())
        .append(startingContext())
        .append(DELTA_START)
        .append(delta(s))
        .append(DELTA_END)
        .append(endingContext())
        .append(endingEllipsis())
        .toString();
}
private String startingEllipsis() {
    return prefixLength > contextLength ? ELLIPSIS : "";
}
private String startingContext() {
    int contextStart = Math.max(0, prefixLength - contextLength);
    int contextEnd = prefixLength;
    return expected.substring(contextStart, contextEnd);
}
private String delta(String s) {
    int deltaStart = prefixLength;
    int deltaEnd = s.length() - suffixLength;
    return s.substring(deltaStart, deltaEnd);
}
private String endingContext() {
    int contextStart = expected.length() - suffixLength;
    int contextEnd =
        Math.min(contextStart + contextLength, expected.length());
    return expected.substring(contextStart, contextEnd);
}
private String endingEllipsis() {
    return (suffixLength > contextLength ? ELLIPSIS : "");
}
}

```

Davvero un buon risultato. Il modulo è separato in un gruppo di funzioni di analisi e un altro gruppo di funzioni di sintesi. Sono suddivise per tipologia in modo che la definizione di ogni funzione si trovi appena dopo il punto in cui essa viene usata. Tutte le funzioni di analisi compaiono per prime e tutte le funzioni di sintesi per ultime.

Se osservate con attenzione, noterete che ho annullato molte delle decisioni che ho preso all'inizio del capitolo. Per esempio, ho reso inline alcuni metodi estratti rimettendoli in `formatCompactedComparison` e ho cambiato il senso dell'espressione `shouldNotBeCompacted`. Questo è normale. Spesso un refactoring ne suggerisce un altro che porta ad

annullare il primo. Il refactoring è un processo iterativo, composto da tentativi ed errori, che poi converge verso qualcosa che ha l’aspetto di un lavoro professionale.

Conclusioni

E così abbiamo soddisfatto la regola dei boy-scout. Abbiamo lasciato questo modulo un po’ più pulito di come l’abbiamo trovato. Ed era già piuttosto “pulito”. Gli autori avevano svolto un lavoro eccellente. Ma nessun modulo è immune ai miglioramenti e ognuno di noi ha la responsabilità di lasciare il codice un po’ meglio di come l’ha trovato.

Refactoring di `SerialDate`



Su <http://www.jfree.org/jcommon/>, troverete la libreria JCommon.

All'interno di tale libreria vi è un package chiamato `org.jfree.date`. In tale package si trova una classe chiamata `SerialDate`. In questo capitolo ci occuperemo di tale classe.

L'autore di `SerialDate` è David Gilbert, un programmatore esperto e molto competente. Come vedremo, il suo codice presenta un livello significativo di professionalità e disciplina. A tutti gli effetti, si tratta di “buon codice”, ma intendo farlo a pezzettini.

Non è per cattiveria, né penso di essere migliore di David, tanto da potermi permettere di giudicare il suo codice. In effetti, se considerate molto del mio codice, sono sicuro che trovereste molte cose da sistemare.

Quindi non è per maleducazione o arroganza. Quello che sto per fare non è niente di più e niente di meno di una revisione professionale. Un'attività alla quale dovremmo essere abituati. E dovremmo essere felici quando viene svolta sul nostro lavoro. È solo grazie alle critiche che si migliora. Vale per i medici, i piloti e gli avvocati. E probabilmente anche noi programmati abbiamo sempre bisogno di imparare.

Un'ultima cosa su David Gilbert: David è molto di più di un buon programmatore. Ha avuto il coraggio e la benevolenza di offrire il suo codice alla comunità intera in modo del tutto gratuito. Ha messo il suo codice in chiaro in modo che tutti potessero vederlo e ha invitato a usarlo e studiarlo. Questa è davvero una bella cosa!

`SerialDate` (vedi Appendice B, Listato B.1) è una classe che rappresenta una data in Java. Perché avere una classe che rappresenta una data, quando Java ha già `java.util.Date` e `java.util.Calendar` e così via? L'autore ha scritto questa classe in risposta a un problema che spesso anch'io ho avvertito. Il commento nel suo Javadoc (riga 67) lo spiega molto bene. Potremmo disquisire sulle sue intenzioni, ma certamente il problema richiedeva una soluzione ed è davvero benvenuta una classe che manipola date anziché ore.

Innanzitutto, facciamola funzionare

Vi sono alcuni unit test, che si trovano in una classe chiamata `SerialDateTests` (vedi Appendice B, Listato B.2). I test vengono tutti superati. Sfortunatamente una rapida ispezione ai test mostra che essi

non controllano proprio tutto [T1]. Per esempio, una ricerca “Find Usages” sul metodo `MonthCodeToQuarter` (riga 334) indica che non viene usato [F4]. In pratica, gli unit test non lo sottopongono a test.

Pertanto ho lanciato Clover per scoprire che cosa coprissero e non coprissero gli unit test. Clover ha rilevato che gli unit test eseguivano solo 91 delle 185 istruzioni eseguibili in `SerialDate` (circa il 50 percento) [T2]. La mappa della copertura sembrava una sorta di ricamo, con grossi varchi di codice non considerato, qua e là in tutta la classe.

Era mia intenzione comprendere appieno e anche eseguire il refactoring di questa classe. Non potevo farlo senza un test dalla copertura più ampia. Pertanto ho scritto un mio set di unit test completamente indipendente (vedi Appendice B, Listato B.4).

Mentre vedrete questi test, noterete che molti di essi sono in commento. Questi test non venivamo passati. Essi rappresentano comportamenti che, secondo me, `SerialDate` dovrebbe avere. Pertanto, mentre eseguivo il refactoring di `SerialDate`, ho fatto in modo che passasse anche questi test.

Pur con alcuni dei test in commento, Clover rileva che i nuovi unit test eseguono solo 170 (il 92 percento) delle 185 istruzioni eseguibili. Questo è abbastanza positivo e penso che sia possibile innalzare tale valore.

I primi test in commento (righe 23-63) erano un po’ una mia “fissa”. Il programma non era concepito per passarli, ma quel comportamento mi sembrava solo ovvio [G2].

Non so perché sia stato scritto il metodo `testWeekdayCodeToString`, ma visto che esiste, sembra ovvio che non debba essere case sensitive. Scrivere questi test è stato facile [T3]. Farli passare, ancora più facile; ho solo modificato le righe 259 e 263 per usare `equalsIgnoreCase`.

Ho lasciato i test alle righe 32 e 45 in commento perché non mi è chiaro perché debbano essere supportate le abbreviazioni “tues” e “thurs”.

I test alle righe 153 e 154 non vengono passati. Chiaramente, dovrebbero [G2]. Possiamo correggere con facilità il problema e i test alle righe dalla 163 alla 213, apportando le seguenti modifiche alla funzione `stringToMonthCode`.

```
457     if ((result < 1) || (result > 12)) {
458         result = -1;
459         for (int i = 0; i < monthNames.length; i++) {
460             if (s.equalsIgnoreCase(shortMonthNames[i])) {
461                 result = i + 1;
462                 break;
463             }
464             if (s.equalsIgnoreCase(monthNames[i])) {
465                 result = i + 1;
466                 break;
467             }
468         }
```

Il test comentato alla riga 318 evidenzia un bug nel metodo `getFollowingDayOfWeek` (riga 672). Il 25 dicembre 2004 era un sabato. Il sabato successivo era il 1 gennaio 2005. Ma quando eseguiamo il test, vediamo che `getFollowingDayOfWeek` restituisce ancora il 25 dicembre per il sabato successivo al 25 dicembre. Chiaramente, questo è sbagliato [G3], [T1]. Vediamo il problema alla riga 685. Si tratta di un tipico errore della condizione di delimitazione [T5]. Dovrebbe essere come segue:

```
685     if (baseDOW >= targetWeekday) {
```

È interessante notare che questa funzione è stato l’obiettivo di un primo intervento. La cronologia delle modifiche (riga 43) mostra che i “bug” sono stati corretti in `getPreviousDayOfWeek`, `getFollowingDayOfWeek` e `getNearestDayOfWeek` [T6].

Il test `testGetNearestDayOfWeek` (riga 329), che considera il metodo `getNearestDayOfWeek` (riga 705), non è stato fin da subito così articolato ed esaustivo come è attualmente. Gli ho aggiunto molti casi di test perché

non tutti i miei casi di test iniziali lo superavano [T6]. Potete vedere il tipo di problemi osservando quali casi di test sono in commento. Lo schema dei fallimenti è rivelatore [T7]. Mostra che l'algoritmo fallisce se il giorno successivo è nel futuro. Chiaramente vi è un qualche tipo di errore nella condizione di delimitazione [T5].

La copertura dei test rilevata da Clover è anch'essa interessante [T8]. La riga 719 non viene mai eseguita! Ciò significa che l'istruzione `if` alla riga 718 è sempre `false`. Basta un'occhiata al codice per capire che dovrebbe essere `true`. La variabile `adjust` è sempre negativa e così non può mai essere maggiore o uguale a 4. Pertanto questo algoritmo è sbagliato.

Ecco l'algoritmo corretto:

```
int delta = targetDOW - base.getDayOfWeek();
int positiveDelta = delta + 7;
int adjust = positiveDelta % 7;
if (adjust > 3)
    adjust -= 7;

return SerialDate.addDays(adjust, base);
```

Infine, i test alle righe 417 e 429 possono essere superati lanciando semplicemente una `IllegalArgumentException` invece di far restituire una stringa d'errore a `weekInMonthToString` e `relativeToString`.

Con queste modifiche tutti gli unit test vengono superati e penso che ora `SerialDate` funzioni. Pertanto è giunto il momento di farla anche funzionare “bene”.

Ora raffiniamola

Ora ricon sidereremo `SerialDate` da cima a fondo, migliorandola passo dopo passo. Anche se non si vede in questa discussione, dopo ogni singola modifica eseguirò tutti gli unit test di `JCommon`, compresi i miei nuovi test per `SerialDate`. Pertanto state tranquilli: ogni modifica che vedete funziona per tutta `JCommon`.

Partendo dalla riga 1, vediamo una grande quantità di commenti, con informazioni sulla licenza, i copyright, gli autori e la cronologia delle modifiche. Riconosco che vi sono vincoli legali da assecondare e così i copyright e le licenze devono restare al loro posto. D'altro canto, la cronologia delle modifiche è un residuato degli anni Sessanta. Oggi abbiamo degli strumenti di controllo del codice sorgente in grado di farlo per noi. Questa cronologia dovrebbe essere cancellata [C1].

L'elenco di `import` a partire dalla riga 61 può essere abbreviato usando `java.text.*` e `java.util.*` [J1].

Mi infastidisce la formattazione HTML nel Javadoc (riga 67). Il fatto di avere un file di codice sorgente con più di un linguaggio è sgradevole. Questo commento impiega *quattro* fra lingue e linguaggi: inglese, Java, Javadoc e HTML [G1]. Con così tanti linguaggi, è difficile garantire la pulizia. Per esempio, il corretto posizionamento delle righe 71 e 72 si perde con la generazione del Javadoc e d'altra parte, chi mai vuol vedere degli `` e `` nel codice sorgente? Una strategia migliore potrebbe essere circondare l'intero commento con un `<pre>` in modo che la formattazione visibile nel codice sorgente permanga anche nel Javadoc. (Una soluzione ancora migliore sarebbe stata che il Javadoc presentasse tutti i commenti preformati, in modo che i commenti appaiano uguali sia nel codice sia nel documento).

La riga 86 contiene la dichiarazione della classe. Perché questa classe si chiama `SerialDate`? Qual è il significato della parte “`serial`”? È forse perché la classe è derivata da `Serializable`? Non sembra probabile.

Penso di sapere perché sia stata usata la parola “`serial`”. L'indizio è nelle costanti `SERIAL_LOWER_BOUND` e `SERIAL_UPPER_BOUND` alle righe 98 e 101. Un indizio perfino migliore si trova nel commento che inizia alla riga 830. Questa classe si chiama `SerialDate` perché è implementata usando un “numero seriale” che si dà il caso sia il numero di giorni dal 30 dicembre 1899.

Vedo due problemi. Innanzitutto, il termine “numero seriale” non è del tutto corretto. Sembra un dettaglio, ma la rappresentazione somiglia più a un *offset*, uno “scostamento” che a numero seriale. Il termine “numero seriale” ha più a che fare con il codice di un prodotto che con una data. Pertanto non lo trovo un nome particolarmente descrittivo [N1]. Un termine più descrittivo potrebbe essere “ordinal”.

Il secondo problema è più significativo. Il nome `SerialDate` implica un’implementazione. Questa è una classe astratta. Non vi è alcuna necessità di specificare nulla sull’implementazione. In effetti, vi sarebbe anche un buon motivo per nasconderla, l’implementazione! Pertanto trovo che questo nome sia sbagliato anche in termini di livello di astrazione [N2]. Secondo me, il nome di questa classe dovrebbe essere, semplicemente, `Date`.

Sfortunatamente, vi sono già troppe classi nella libreria Java di nome `Date`, e pertanto questo non è, probabilmente, il miglior nome da scegliere. Poiché questa classe parla di giorni e non di ore, ho pensato anche di chiamarla `Day`, ma questo nome è già abbondantemente usato. Alla fine ho scelto `DayDate`: un buon compromesso.

Da qui in poi, nella discussione userò il termine `DayDate`. Vi lascio il compito di ricordare che nei listati che leggerete troverete al suo posto `SerialDate`.

Capisco perché `DayDate` eredita da `Comparable` e `Serializable`. Ma perché eredita anche da `MonthConstants`? La classe `MonthConstants` (vedi Appendice B, Listato B.3) è solo un cumulo di costanti finali statiche che definiscono i mesi. Ereditare da classi di costanti è un vecchio trucco che i programmati Java usavano per poter evitare di usare espressioni come `MonthConstants.January`, ma non è comunque una buona idea [J2].

`MonthConstants` dovrebbe essere una `enum`.

```
public abstract class DayDate implements Comparable, Serializable {  
    public static enum Month {  
        JANUARY(1),
```

```

FEBRUARY(2),
MARCH(3),
APRIL(4),
MAY(5),
JUNE(6),
JULY(7),
AUGUST(8),
SEPTEMBER(9),
OCTOBER(10),
NOVEMBER(11),
DECEMBER(12);
Month(int index) {
    this.index = index;
}
public static Month make(int monthIndex) {
    for (Month m : Month.values()) {
        if (m.index == monthIndex)
            return m;
    }
    throw new IllegalArgumentException("Invalid month index " + monthIndex);
}
public final int index;
}

```

Cambiare `MonthConstants` in questa `enum` costringe ad apportare alcune modifiche alla classe `DayDate` e a tutti i suoi utilizzatori. C'è voluta un'ora intera per eseguire tutte le modifiche. Tuttavia, ogni funzione che prima accettava un `int` per un mese, ora accetta un enumeratore `Month`. Ciò significa che possiamo sbarazzarci del metodo `isValidMonthCode` (riga 326) e di tutto il controllo degli errori del codice del mese come quello in `monthCodeToQuarter` (riga 356) [G5].

Poi abbiamo la riga 91, `serialVersionUID`. Questa variabile viene usata per controllare il serializer. Se la modifichiamo, ogni `DayDate` scritta con una vecchia versione del software non sarà più leggibile e produrrà una `InvalidClassException`. Se non dichiarate la variabile `serialVersionUID`, il compilatore ne genererà automaticamente una, che sarà differente ogni volta che apporterete una modifica al modulo. Lo so che tutti i documenti raccomandano di controllare manualmente questa variabile, ma mi sembra che il controllo automatico della serializzazione sia molto più sicuro [G4]. Dopotutto, è meglio eseguire il debug di una `InvalidClassException` rispetto al comportamento che si otterrebbe

dimenticando di modificare il `serialVersionUID`. Pertanto cancellerò la variabile, quanto meno da qui in poi.

NOTA

Molti revisori di questo testo hanno contestato questa decisione. Sostengono che in un framework open-source sia meglio affermare il controllo manuale su questo ID, in modo che piccole modifiche al software non comportino l'invalidità delle vecchie date serializzate. Questo è un argomento sensato. Tuttavia, quanto meno il fallimento, per quanto possa essere fastidioso, avrà una causa ben definita. D'altra parte, se l'autore della classe dimenticasse di aggiornare l'ID, la modalità di errore sarebbe indefinita e finirebbe per essere molto silenziosa. Penso che la vera morale di questa storia sia che non doresti aspettarti di deserializzare fra le versioni.

Trovo ridondante il commento alla riga 93. Nei commenti ridondanti spesso si annidano falsità e disinformazioni [C2]. Pertanto lo tolgo di mezzo.

I commenti alle righe 97 e 100 parlano dei numeri seriali, di cui abbiamo già parlato [C1]. Le variabili che descrivono sono la prima e l'ultima data descrivibile da `DayDate`. Il tutto può essere chiarito meglio [N1].

```
public static final int EARLIEST_DATE_ORDINAL = 2;      // 1/1/1900  
public static final int LATEST_DATE_ORDINAL = 2958465; // 12/31/9999
```

Non mi è chiaro perché `EARLIEST_DATE_ORDINAL` sia 2 invece di 0. Vi è un suggerimento nel commento alla riga 829 che suggerisce che questo ha qualcosa a che fare con il modo in cui i dati sono rappresentati in Microsoft Excel. Vi sono più informazioni in una derivata di `DayDate` chiamata `SpreadsheetDate` (vedi Appendice B, Listato B.5). Il commento alla riga 71 descrive piuttosto bene l'argomento.

Il problema sembra essere correlato all'implementazione di `SpreadsheetDate` e non ha niente a che vedere con `DayDate`. Ne concludo che `EARLIEST_DATE_ORDINAL` e `LATEST_DATE_ORDINAL` in realtà non appartengono a `DayDate` e dovrebbero essere spostate in `SpreadsheetDate` [G6].

In effetti, una ricerca nel codice mostra che queste variabili vengono impiegate solo in `SpreadsheetDate`. Per nulla in `DayDate`, né in qualsiasi altra classe del framework `JCommon`. Pertanto le sposterò in `SpreadsheetDate`.

Le variabili successive, `MINIMUM_YEAR_SUPPORTED` e `MAXIMUM_YEAR_SUPPORTED` (righe 104 e 107), sono un po' un dilemma. Sembra chiaro che se `DayDate` è una classe astratta che non fornisce alcuna informazione sull'implementazione, non dovrebbe neanche informarci dell'anno minimo o massimo. Di nuovo, sarei tentato di spostare queste variabili in `SpreadsheetDate` [G6]. Tuttavia, una rapida ricerca degli utilizzatori di queste variabili indica un'altra classe: `RelativeDayOfWeekRule` (vedi Appendice B, Listato B.6). Vediamo che l'uso è riportato alle righe 177 e 178 della funzione `getDate`, dove vengono usate per controllare che l'argomento di `getDate` sia un anno valido. Il dilemma è che l'utilizzatore di una classe astratta abbia bisogno di informazioni sulla sua implementazione.

Quello che dobbiamo fare è fornire questa informazione senza "inquinare" `DayDate`. Solitamente, otterremmo le informazioni sull'implementazione da un'istanza di una derivata. Tuttavia, alla funzione `getDate` non viene passata un'istanza di `DayDate`. Però restituisce un'istanza di tale tipo, il che significa che da qualche parte la deve creare. Le righe da 187 a 205 forniscono un indizio. L'istanza di `DayDate` viene creata da una delle tre funzioni `getPreviousDayOfWeek`, `getNearestDayOfWeek` o `getFollowingDayOfWeek`. Osservando il listato di `DayDate`, vediamo che queste funzioni (righe 638-724) restituiscono tutte una data creata da `addDays` (riga 571), che richiama `createInstance` (riga 808), che a sua volta crea una `SpreadsheetDate`! [G7].

Generalmente non è una buona idea che le classi base sappiano qualcosa delle loro derivate. Per sistemare il problema, dovremmo usare il pattern ABSTRACT FACTORY [GOF] e creare una

`DayDateFactory`. Questa factory creerà le istanze di `DayDate` di cui abbiamo bisogno e potrà anche rispondere a domande sull'implementazione, come la data massima e minima.

```
public abstract class DayDateFactory {  
    private static DayDateFactory factory = new SpreadsheetDateFactory();  
    public static void setInstance(DayDateFactory factory) {  
        DayDateFactory.factory = factory;  
    }  
    protected abstract DayDate _makeDate(int ordinal);  
    protected abstract DayDate _makeDate(int day, DayDate.Month month, int year);  
    protected abstract DayDate _makeDate(int day, int month, int year);  
    protected abstract DayDate _makeDate(java.util.Date date);  
    protected abstract int _getMinimumYear();  
    protected abstract int _getMaximumYear();  
    public static DayDate makeDate(int ordinal) {  
        return factory._makeDate(ordinal);  
    }  
    public static DayDate makeDate(int day, DayDate.Month month, int year) {  
        return factory._makeDate(day, month, year);  
    }  
    public static DayDate makeDate(int day, int month, int year) {  
        return factory._makeDate(day, month, year);  
    }  
    public static DayDate makeDate(java.util.Date date) {  
        return factory._makeDate(date);  
    }  
    public static int getMinimumYear() {  
        return factory._getMinimumYear();  
    }  
    public static int getMaximumYear() {  
        return factory._getMaximumYear();  
    }  
}
```

Questa classe factory sostituisce i metodi `createInstance` con dei metodi `makeDate`, cambiando un po' i nomi [N1]. Come default ha `SpreadsheetDateFactory` che però può essere cambiata in qualsiasi momento con un'altra factory. I metodi statici che delegano ai metodi astratti usano una combinazione di pattern SINGLETON [GOF], DECORATOR [GOF] e ABSTRACT FACTORY, che si sono rivelati utili.

La classe `SpreadsheetDateFactory` ha il seguente aspetto.

```
public class SpreadsheetDateFactory extends DayDateFactory {  
    public DayDate _makeDate(int ordinal) {  
        return new SpreadsheetDate(ordinal);  
    }  
    public DayDate _makeDate(int day, DayDate.Month month, int year) {  
        return new SpreadsheetDate(day, month, year);  
    }  
    public DayDate _makeDate(int day, int month, int year) {
```

```

        return new SpreadsheetDate(day, month, year);
    }
    public DayDate _makeDate(Date date) {
        final GregorianCalendar calendar = new GregorianCalendar();
        calendar.setTime(date);
        return new SpreadsheetDate(calendar.get(Calendar.DATE),
            DayDate.Month.make(calendar.get(Calendar.MONTH) + 1),
            calendar.get(Calendar.YEAR));
    }
    protected int _getMinimumYear() {
        return SpreadsheetDate.MINIMUM_YEAR_SUPPORTED;
    }
    protected int _getMaximumYear() {
        return SpreadsheetDate.MAXIMUM_YEAR_SUPPORTED;
    }
}

```

Come potete vedere, ho già spostato le variabili `MINIMUM_YEAR_SUPPORTED` e `MAXIMUM_YEAR_SUPPORTED` in `SpreadsheetDate`, cui appartengono [G6].

Il problema successivo di `DayDate` riguarda le costanti dei giorni, a partire dalla riga 109. Queste dovrebbero diventare un'altra `enum` [J3]. Abbiamo già visto come fare, quindi non mi ripeterò. La vedrete comunque nei listati finali.

Poi, troviamo una serie di tabelle che partono con `LAST_DAY_OF_MONTH` alla riga 140. Il mio primo problema con queste tabelle è che i commenti che le descrivono sono ridondanti [C3]. Basta il nome. Pertanto cancellerò i commenti.

Non sembra esservi alcun buon motivo per il quale questa tabella non sia privata [G8], perché vi è una funzione statica `lastDayOfMonth` che fornisce gli stessi dati.

La tabella successiva, `AGGREGATE_DAYS_TO_END_OF_MONTH`, è un po' più misteriosa, perché non viene usata da nessuna parte nel framework JCommon [G9]. Pertanto l'ho cancellata.

Lo stesso vale per `LEAP_YEAR_AGGREGATE_DAYS_TO_END_OF_MONTH`.

La tabella successiva, `AGGREGATE_DAYS_TO_END_OF_PRECEDING_MONTH`, viene usata solo in `SpreadsheetDate` (righe 434 e 473). Questo fa sorgere la domanda se non debba essere spostata in `SpreadsheetDate`. Il motivo per non spostarla è che la tabella non è specifica di alcuna implementazione

[G6]. D'altra parte, non esiste nessuna implementazione se non `SpreadsheetDate`, e così la tabella dovrebbe essere spostata più vicina a dove viene usata [G10].

Quello che penso è che per coerenza [G11], dovremmo rendere la tabella privata ed esporla tramite una funzione come

`julianDateOfLastDayOfMonth`. Nessuno sembra aver bisogno di una funzione come quella. Inoltre, la tabella può essere risposta con facilità in `DayDate` nel caso in cui una nuova implementazione di `DayDate` ne avesse bisogno. Pertanto l'ho spostata.

Lo stesso vale per la tabella `LEAP_YEAR_AGGREGATE_DAYS_TO_END_OF_MONTH`.

Poi, vediamo tre set di costanti che possono essere trasformate in `enum` (righe 162-205). Il primo dei tre seleziona una settimana in un mese. L'ho cambiato in un' `enum`: `WeekInMonth`.

```
public enum WeekInMonth {  
    FIRST(1), SECOND(2), THIRD(3), FOURTH(4), LAST(0);  
    public final int index;  
    WeekInMonth(int index) {  
        this.index = index;  
    }  
}
```

Il significato del secondo set di costanti (righe 177-187) è un po' più oscuro. Le costanti `INCLUDE_NONE`, `INCLUDE_FIRST`, `INCLUDE_SECOND` e `INCLUDE_BOTH` vengono impiegate per indicare se le date agli estremi che definiscono un intervallo dovrebbero essere incluse in tale intervallo.

Matematicamente, i termini corretti sarebbero “intervallo aperto”, “intervallo semi-aperto” e “intervallo chiuso”. Penso che sia meglio usare la nomenclatura matematica [N3], così le ho trasformate in un' `enum` di nome `DateInterval` contenente gli enumeratori `CLOSED`, `CLOSED_LEFT`, `CLOSED_RIGHT` e `OPEN`.

Il terzo set di costanti (righe 18-205) descrive se la ricerca di un determinato giorno della settimana debba indicare l'ultima, la prossima

o la più vicina delle istanze. Decidere il nome è stato difficile. Alla fine, ho optato per `WeekdayRange`, con gli enumeratori `LAST`, `NEXT` e `NEAREST`.

Potreste non concordare con i nomi che ho scelto. Per me hanno senso, ma per voi magari no. Il punto è che ora sono in una forma che permette di cambiarli con facilità [J3]. Non vengono più passati come interi, ma come simboli. Ora posso usare la funzione di “cambio nome” del mio IDE per cambiarne il nome o il tipo, senza preoccuparmi di perdere un `-1` o un `2` da qualche parte nel codice o che qualche dichiarazione di un argomento `int` rimanga mal descritta.

Il campo `description` alla riga 208 non sembra essere usato da nessuno. L’ho cancellato insieme ai suoi metodi accessor e mutator [G9].

Ho cancellato anche l’ormai inutile costruttore di default alla riga 213 [G12]. Il compilatore lo genererà per noi.

Possiamo saltare il metodo `isValidWeekdayCode` (righe 216-238), perché l’abbiamo cancellato quando abbiamo creato l’enumerazione `Day`.

Questo ci porta al metodo `stringToWeekdayCode` (righe 242-270). I Javadoc che non dicono niente di più della signature del metodo sono solo di troppo [C3], [G12]. L’unica informazione utile fornita da questo Javadoc è la descrizione del valore restituito `-1`. Tuttavia, poiché ora abbiamo l’enumerazione `Day`, in realtà ora il commento è errato [C2]. Il metodo ora lancia una `IllegalArgumentException`. Pertanto ho cancellato il Javadoc.

Ho cancellato anche tutte le parole chiave `final` nelle dichiarazioni di argomenti e variabili. Per quanto si vede, non aggiungevano nulla, se non congestione [G12]. Eliminare i `final` è una sorta di convenzione, ormai. Per esempio, Robert Simmons ([Simmons04] p. 73) ci consiglia fortemente di “... mettere il `final` dappertutto nel vostro codice”.

Chiaramente non sono d’accordo. Penso che vi sono alcuni buoni utilizzi del `final`, per esempio un’occasionale costante `final`, ma

altrimenti la parola chiave non aggiunge quasi nulla e crea molta congestione. Forse la penso così perché tutti gli errori individuabili da `final` vengono già trovati dagli unit test che ho scritto.

Non mi piacevano le istruzioni `if` duplicate [G5] all'interno del ciclo `for` (righe 259 e 263), così le ho connesse in un'unica istruzione `if` usando l'operatore `||`. Inoltre ho usato l'enumerazione `Day` per controllare il ciclo `for` e ho fatto alcuni altri interventi di natura estetica.

Mi sono accorto che questo metodo in realtà non appartiene a `DayDate`. In realtà si tratta della funzione di parsing di `Day`. Pertanto l'ho spostata nell'enumerazione `Day`. Tuttavia, ciò ha reso l'enumerazione `Day` un po' troppo grande. Poiché il concetto di `Day` non dipende da `DayDate`, ho spostato l'enumerazione `Day` all'esterno della classe `DayDate` in un proprio file di codice sorgente [G13].

Ho spostato nell'enumerazione `Day` anche la funzione successiva, `weekdayCodeToString` (righe 272-286) e l'ho chiamata `toString`.

```
public enum Day {
    MONDAY(Calendar.MONDAY),
    TUESDAY(Calendar.TUESDAY),
    WEDNESDAY(Calendar.WEDNESDAY),
    THURSDAY(Calendar.THURSDAY),
    FRIDAY(Calendar.FRIDAY),
    SATURDAY(Calendar.SATURDAY),
    SUNDAY(Calendar.SUNDAY);
    public final int index;
    private static DateFormatSymbols dateSymbols = new DateFormatSymbols();
    Day(int day) {
        index = day;
    }
    public static Day make(int index) throws IllegalArgumentException {
        for (Day d : Day.values())
            if (d.index == index)
                return d;
        throw new IllegalArgumentException(
            String.format("Illegal day index: %d.", index));
    }
    public static Day parse(String s) throws IllegalArgumentException {
        String[] shortWeekdayNames = dateSymbols.getShortWeekdays();
        String[] weekDayNames = dateSymbols.getWeekdays();
        s = s.trim();
        for (Day day : Day.values()) {
            if (s.equalsIgnoreCase(shortWeekdayNames[day.index]) ||
                s.equalsIgnoreCase(weekDayNames[day.index])) {
                return day;
            }
        }
    }
}
```

```

        }
        throw new IllegalArgumentException(
            String.format("%s is not a valid weekday string", s));
    }
    public String toString() {
        return dateSymbols.getWeekdays()[index];
    }
}

```

Vi sono due funzioni `getMonths` (righe 288-316). La prima richiama la seconda. La seconda non è mai chiamata da nessuno, se non la prima. Pertanto, le ho fatte “collassare” in una e le ho molto semplificate [G9], [G12], [F4]. Infine, ho cambiato il nome, per renderlo un po’ più descrittivo [N1].

```

public static String[] getMonthNames() {
    return dateFormatSymbols.getMonths();
}

```

La funzione `isValidMonthCode` (righe 326-346) è stata resa irrilevante dall’enum `Month`, così l’ho cancellata [G9].

La funzione `monthCodeToQuarter` (righe 356-375) sa un po’ di FEATURE ENVY [Refactoring] [G14] e probabilmente appartiene all’enum `Month`, come il metodo `quarter`. L’ho sostituita.

```

public int quarter() {
    return 1 + (index-1)/3;
}

```

Questo ha reso l’enum `Month` sufficientemente grande da formare una propria classe. Pertanto l’ho spostata fuori da `DayDate` per coerenza con l’enum `Day` [G11], [G13].

I successivi due metodi si chiamano `monthCodeToString` (righe 377-426). Di nuovo, vediamo lo schema di un metodo che richiama il suo gemello con un flag. Solitamente non è una buona idea passare un flag come argomento a una funzione, in particolare quando tale flag seleziona semplicemente il formato dell’output [G15]. Ho rinominato, semplificato e ristrutturato queste funzioni e le ho spostate nell’enum `Month` [N1], [N3], [C3], [G14].

```

    public String toString() {
        return dateFormatSymbols.getMonths()[index - 1];
    }
    public String toShortString() {
        return dateFormatSymbols.getShortMonths()[index - 1];
    }
}

```

Il metodo successivo è `stringToMonthCode` (righe 428-472). L'ho rinominato, spostato nell'enum `Month` e semplificato [N1], [N3], [C3], [G14], [G12].

```

public static Month parse(String s) {
    s = s.trim();
    for (Month m : Month.values())
        if (m.matches(s))
            return m;
    try {
        return make(Integer.parseInt(s));
    }
    catch (NumberFormatException e) {}
    throw new IllegalArgumentException("Invalid month " + s);
}
private boolean matches(String s) {
    return s.equalsIgnoreCase(toString()) ||
           s.equalsIgnoreCase(toShortString());
}

```

Il metodo `isLeapYear` (righe 495-517) può essere reso un po' più espressivo [G16].

```

public static boolean isLeapYear(int year) {
    boolean fourth = year % 4 == 0;
    boolean hundredth = year % 100 == 0;
    boolean fourHundredth = year % 400 == 0;
    return fourth && (!hundredth || fourHundredth);
}

```

La funzione successiva, `leapYearCount` (righe 519-536) in realtà non appartiene a `DayDate`. Nessuno la richiama, tranne due metodi in `SpreadsheetDate`. Pertanto l'ho spinta più in basso [G6].

La funzione `lastDayOfMonth` (righe 538-560) impiega l'array `LAST_DAY_OF_MONTH`. Questo array in realtà appartiene all'enum `Month` [G17], così l'ho spostata là. Ho anche semplificato la funzione e l'ho resa un po' più espressiva [G16].

```

public static int lastDayOfMonth(Month month, int year) {
    if (month == Month.FEBRUARY && isLeapYear(year))
        return month.lastDay() + 1;
}

```

```

        else
            return month.lastDay();

    }

```

Ora le cose iniziano a farsi un po' più interessanti. La funzione successiva è `addDays` (righe 562-576). Prima di tutto, poiché questa funzione opera sulle variabili di `DayDate`, non dovrebbe essere statica [G18]. Pertanto l'ho cambiata in un metodo di istanza. In secondo luogo, richiama la funzione `toSerial`. Questa funzione dovrebbe chiamarsi `toOrdinal` [N1]. Infine, il metodo può essere semplificato.

```

public DayDate addDays(int days) {
    return DayDateFactory.makeDate(toOrdinal() + days);
}

```

Lo stesso vale per `addMonths` (righe 578-602). Dovrebbe essere un metodo di istanza [G18]. L'algoritmo è un po' complicato, così ho usato il principio EXPLAINING TEMPORARY VARIABLES [Beck97] [G19] per renderlo più trasparente. Ho anche rinominato il metodo `getYYY` in `getYear` [N1].

```

public DayDate addMonths(int months) {
    int thisMonthAsOrdinal = 12 * getYear() + getMonth().index - 1;
    int resultMonthAsOrdinal = thisMonthAsOrdinal + months;
    int resultYear = resultMonthAsOrdinal / 12;
    Month resultMonth = Month.make(resultMonthAsOrdinal % 12 + 1);
    int lastDayOfResultMonth = lastDayOfMonth(resultMonth, resultYear);
    int resultDay = Math.min(getDayOfMonth(), lastDayOfResultMonth);
    return DayDateFactory.makeDate(resultDay, resultMonth, resultYear);
}

```

La funzione `addYears` (righe 604-626) non ha sorprese.

```

public DayDate plusYears(int years) {
    int resultYear = getYear() + years;
    int lastDayOfMonthInResultYear = lastDayOfMonth(getMonth(), resultYear);
    int resultDay = Math.min(getDayOfMonth(), lastDayOfMonthInResultYear);
    return DayDateFactory.makeDate(resultDay, getMonth(), resultYear);
}

```

Ho un piccolo dubbio nel cambiare questi metodi da statici a metodi dell'istanza. L'espressione `date.addDays(5)` chiarisce a sufficienza che l'oggetto `date` non cambia e che invece viene restituita una nuova istanza di `DayDate`? O suggerisce, erroneamente, che vogliamo aggiungere cinque

giorni all’oggetto `date`? Potreste pensare che sia un problema da poco, ma un frammento di codice con il seguente aspetto può essere molto ingannatore [G20].

```
DayDate date = DateFactory.makeDate(5, Month.DECEMBER, 1952);  
date.addDays(7); // fa avanzare la data di una settimana.
```

Chi leggerà questo codice, molto probabilmente accetterà semplicemente che `addDays` modifichi l’oggetto `date`. Pertanto il nome che scegliamo deve evitare questa ambiguità [N4]. Pertanto ho cambiato i nomi in `plusDays` e `plusMonths`. Mi sembra che lo scopo del metodo sia catturato adeguatamente da:

```
DayDate date = oldDate.plusDays(5);
```

Mentre la riga seguente non chiarisce sufficientemente al lettore che viene modificato l’oggetto `date`:

```
date.plusDays(5);
```

Gli algoritmi si fanno più interessanti. `getPreviousDayOfWeek` (righe 628-660) funziona, per carità, ma è un po’ complessa. Dopo un po’ di riflessioni sul suo significato [G21], l’ho semplificata e ho usato il principio EXPLAINING TEMPORARY VARIABLES [G19] per renderla più chiara. L’ho anche trasformata da un metodo statico a un metodo di istanza [G18] e mi sono sbarazzato del metodo di istanza [G5], duplicato (righe 997-1008).

```
public DayDate getPreviousDayOfWeek(Day targetDayOfWeek) {  
    int offsetToTarget = targetDayOfWeek.index - getDayOfWeek().index;  
    if (offsetToTarget >= 0)  
        offsetToTarget -= 7;  
    return plusDays(offsetToTarget);  
}
```

La stessa analisi (e lo stesso risultato) vale per `getFollowingDayOfWeek` (righe 662-693).

```
public DayDate getFollowingDayOfWeek(Day targetDayOfWeek) {  
    int offsetToTarget = targetDayOfWeek.index - getDayOfWeek().index;  
    if (offsetToTarget <= 0)  
        offsetToTarget += 7;  
    return plusDays(offsetToTarget);  
}
```

La funzione successiva è `getNearestDayOfWeek` (righe 695-726), che abbiamo corretto qualche pagina fa. Ma le modifiche che avevo apportato non sono coerenti con l'assetto scelto nelle ultime due funzioni [G11]. Pertanto l'ho resa coerente e ho usato il principio EXPLAINING TEMPORARY VARIABLES [G19] per rendere più chiaro l'algoritmo.

```
public DayDate getNearestDayOfWeek(final Day targetDay) {
    int offsetToThisWeeksTarget = targetDay.index - getDayOfWeek().index;
    int offsetToFutureTarget = (offsetToThisWeeksTarget + 7) % 7;
    int offsetToPreviousTarget = offsetToFutureTarget - 7;
    if (offsetToFutureTarget > 3)
        return plusDays(offsetToPreviousTarget);
    else
        return plusDays(offsetToFutureTarget);
}
```

Il metodo `getEndOfCurrentMonth` (righe 728-740) è un po' strano, perché è un metodo di istanza che [G14] vorrebbe avere un argomento `DayDate`. L'ho reso un vero metodo di istanza e ho reso più esplicativi alcuni nomi.

```
public DayDate getEndOfMonth() {
    Month month = getMonth();
    int year = getYear();
    int lastDay = lastDayOfMonth(month, year);
    return DayDateFactory.makeDate(lastDay, month, year);
}
```

Il refactoring di `weekInMonthToString` (righe 742-761) si è rivelato molto interessante. Usando gli strumenti di refactoring del mio IDE, innanzitutto ho spostato il metodo nell'enum `WeekInMonth` che avevo già creato. Poi ho chiamato il metodo `toString`. Quindi l'ho trasformato da un metodo statico a un metodo di istanza. Tutti i test rimanevano soddisfatti (pensate, ora, a dove voglio arrivare...).

Poi, ho cancellato interamente il metodo! Cinque assert fallivano (righe 411-415, Listato B.4). Ho modificato queste righe, in modo che usassero i nomi degli enumeratori (`FIRST`, `SECOND`, ...). Ora tutti i test passavano. Immaginate perché? Riuscite anche a vedere perché era necessario ognuno di questi passi? Lo strumento di refactoring si è assicurato che tutto ciò che prima richiamava `weekInMonthToString` ora

richiamasse `toString` sull'enumeratore `weekinMonth`, perché tutti gli enumeratori implementano `toString` in modo che, semplicemente, restituisca il proprio nome...

Sfortunatamente, ho un po' esagerato. Per quanto sia elegante questa magnifica catena di refactoring, mi sono reso conto che gli unici utilizzatori di questa funzione erano i test che avevo appena modificato, così ho cancellato anche i test.

“Mi freghi una volta? Sei una bestia! Mi freghi due volte? La bestia sono io!” Pertanto dopo avere determinato che solo i test richiamavano `relativeToString` (righe 765-781), ho semplicemente cancellato la funzione e i suoi test.

Infine abbiamo reso astratti i metodi di questa classe astratta. Il primo è stato `toSerial` (righe 838-844). Poche pagine fa ne ho cambiato il nome in `toordinal`. Valutando questo nuovo contesto, ho deciso che il suo nome doveva diventare `getOrdinalDay`.

Il metodo astratto successivo è `toDate` (righe 838-844). Converte un `DayDate` in un `java.util.Date`. Perché questo metodo è astratto? Se osserviamo la sua implementazione in `spreadsheetDate` (righe 198-207, Listato B.5), vediamo che non dipende affatto dall'implementazione di tale classe [G6]. Pertanto l'ho spostato in su.

I metodi `getYYYY`, `getMonth` e `getDayOfMonth` sono giustamente astratti. Tuttavia, `getDayOfWeek` è un altro di quei metodi che dovrebbero essere estratti da `spreadSheetDate` perché non dipende da nulla che non possa essere trovato in `DayDate` [G6]. O forse sì?

Se osservate con attenzione (riga 247, Listato B.5), vedrete che l'algoritmo dipende implicitamente dall'origine del giorno ordinale (in altre parole, il giorno della settimana del giorno “zero”). Pertanto anche se questa funzione non ha alcuna dipendenza fisica che non possa essere trasferita a `DayDate`, ha comunque una dipendenza logica.

Le dipendenze logiche come questa mi danno fastidio [G22]. Se qualcosa logica dipende dall'implementazione, allora anche qualcosa di fisico dovrebbe. Inoltre, mi sembra che l'algoritmo possa diventare più generico e che solo una piccola porzione di esso dipenda dall'implementazione [G6].

Pertanto ho creato in `DayDate` un metodo astratto di nome

`getDayOfWeekForOrdinalZero` e l'ho implementato in `SpreadsheetDate` in modo che restituisca `Day.SATURDAY`. Poi ho spostato il metodo `getDayOfWeek` in `DayDate` e l'ho modificato perché richiami `getOrdinalDay` e `getDayOfWeekForOrdinalZero`.

```
public Day getDayOfWeek() {
    Day startingDay = getDayOfWeekForOrdinalZero();
    int startingOffset = startingDay.index - Day.SUNDAY.index;
    return Day.make((getOrdinalDay() + startingOffset) % 7 + 1);
}
```

A proposito, osservate con attenzione il commento dalla riga 895 alla riga 899. Questa ripetizione è davvero necessaria? Come al solito, ho cancellato questo commento.

Il metodo successivo è `compare` (righe 902-913). Di nuovo, questo metodo non dovrebbe essere astratto [G6], ho messo l'implementazione in `DayDate`. Inoltre, il nome non è sufficientemente comunicativo [N1]. Questo metodo, in realtà, restituisce la differenza in giorni rispetto all'argomento. Pertanto ho cambiato il nome in `daysSince`. Inoltre, ho notato che non vi erano test per questo metodo, così li ho scritti.

Le sei funzioni successive (righe 915-980) sono tutte metodi astratti che dovrebbero essere implementati in `DayDate`. Pertanto li ho tolto tutti da `SpreadsheetDate`.

L'ultima funzione, `isInRange` (righe 982-995) deve anch'essa essere estratta ed essere sottoposta a refactoring. L'istruzione `switch` è bruttina [G23] e può essere sostituita trasferendo i `case` nell'enum `DateInterval`.

```
public enum DateInterval {
    OPEN {
```

```

        public boolean isIn(int d, int left, int right) {
            return d > left && d < right;
        }
    },
    CLOSED_LEFT {
        public boolean isIn(int d, int left, int right) {
            return d >= left && d < right;
        }
    },
    CLOSED_RIGHT {
        public boolean isIn(int d, int left, int right) {
            return d > left && d <= right;
        }
    },
    CLOSED {
        public boolean isIn(int d, int left, int right) {
            return d >= left && d <= right;
        }
    },
    public abstract boolean isIn(int d, int left, int right);
}
public boolean isInRange(DayDate d1, DayDate d2, DateInterval interval) {
    int left = Math.min(d1.getOrdinalDay(), d2.getOrdinalDay());
    int right = Math.max(d1.getOrdinalDay(), d2.getOrdinalDay());
    return interval.isIn(getOrdinalDay(), left, right);
}
}

```

Questo ci porta alla fine di `DayDate`. Pertanto ora daremo un'altra passata all'intera classe, per vedere come funziona.

Innanzitutto, il commento iniziale ormai non è più aggiornato, così l'ho accorciato e migliorato [C2]. Poi, ho spostato tutte le enum rimanenti all'interno ognuna di un proprio file [G12].

Poi, ho spostato la variabile statica (`dateFormatSymbols`) e i tre metodi statici (`getMonthNames`, `isLeapYear`, `lastDayOfMonth`) in una nuova classe: `DateUtil` [G6].

Ho spostato i metodi astratti in cima, dove è giusto che stiano [G24].

Ho cambiato `Month.make` in `Month.fromInt` [N1] e ho fatto lo stesso per tutte le altre enum. Ho anche creato un accessor `toInt()` per tutte le enum e ho reso privato il campo `index`.

Vi era qualche duplicazione [G5] in `plusYears` e `plusMonths`, che ho eliminato estraendo un nuovo metodo chiamato `correctLastDayOfMonth`, rendendo così tutti e tre i metodi molto più chiari.

Mi sono sbarazzato del numero magico ¹ [G25], sostituendolo con `Month.JANUARY.toInt() & Day.SUNDAY.toInt()`, in base alle esigenze. Mi sono occupato anche di `spreadsheetDate`, ripulendo un po' gli algoritmi. Il risultato finale è contenuto nei Listati da B.7 a B.16 dell'Appendice B.

È interessante notare che la copertura del codice in `DayDate` si è ridotta all'84,9 percento! Questo non è dovuto al fatto che vengono sottoposte a test meno funzionalità; piuttosto è perché la classe è "dimagrita" così tanto che le poche righe non considerate ora pesano molto di più. `DayDate` ora ha 45 istruzioni eseguibili su 53 coperte dai test. Le righe non considerate sono così banali da non averne bisogno, di test.

Conclusioni

Ancora una volta abbiamo seguito la regola dei boy-scout. Abbiamo lasciato il codice un po' più pulito di come l'abbiamo trovato. C'è voluto un po' di tempo, ma ne è valsa la pena. La copertura dei test è aumentata, alcuni bug sono stati corretti, il codice è stato semplificato e ridotto. La prossima persona che guarderà questo codice lo troverà (speriamo) più facile da usare rispetto a prima. Tale persona probabilmente sarà anche in grado di ripulirlo ancora di più di noi.

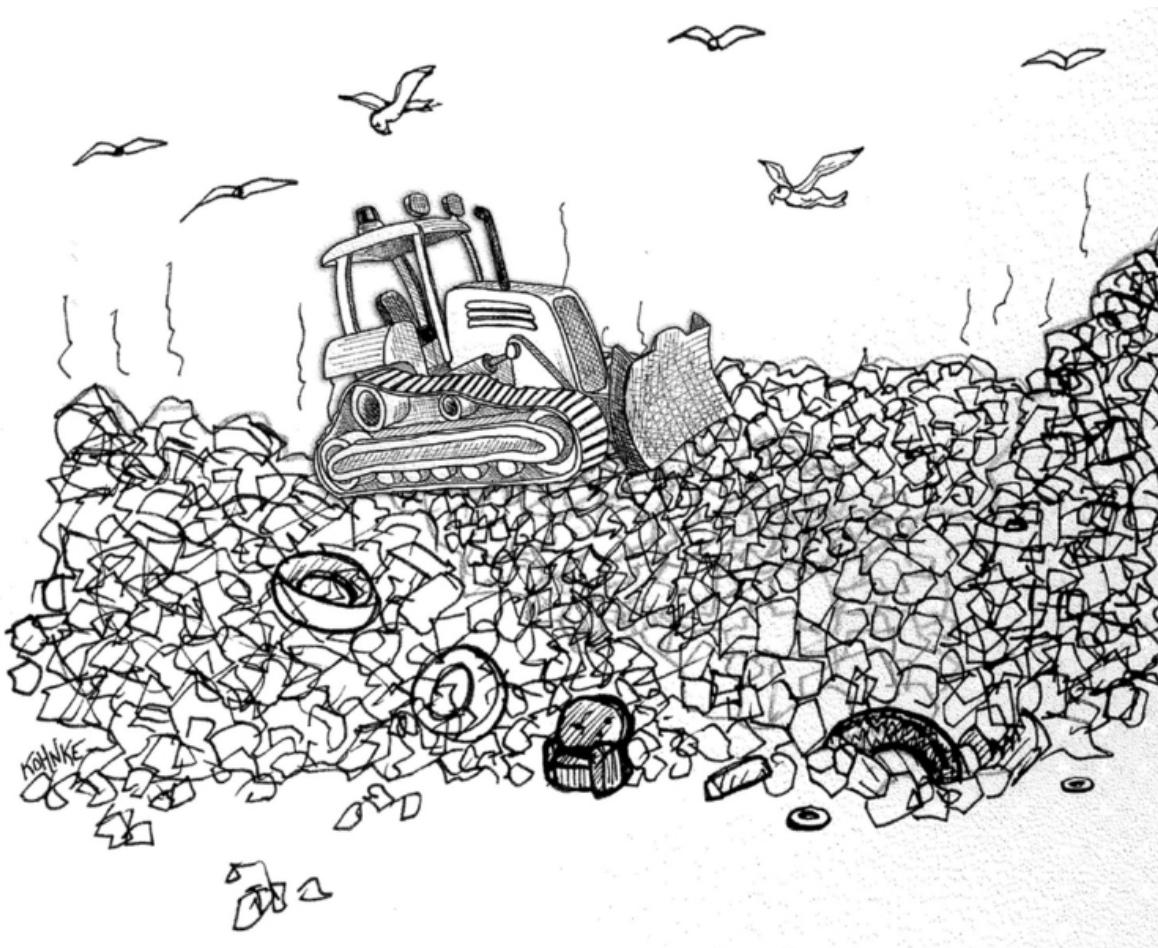
Bibliografia

- [GOF]: Gamma et al., Design Patterns: Elements of Reusable Object Oriented Software, Addison-Wesley, Boston 1996.
- [Simmons04]: Robert Simmons Jr., Hardcore Java, O'Reilly, California 2004.
- [Refactoring]: Martin Fowler et al., Refactoring: Improving the Design of Existing Code, Addison-Wesley, Boston 1999.

- [Beck97]: Kent Beck, Smalltalk Best Practice Patterns, Prentice Hall, New Jersey 1997.

Capitolo 17

Avvertenze ed euristiche



Nel suo meraviglioso libro Refactoring [Refactoring], Martin Fowler ha identificato molte “Avvertenze” [NdT: in originale il termine, che rendeva bene il senso, era “smells“, letteralmente “puzze“. Valeva la pena specificarlo]. La lista che segue include molte delle avvertenze di

Martin e ne aggiunge di nuove. Inoltre includo alcune “perle” ed euristiche che applico quotidianamente.

Ho compilato questo elenco occupandomi di svariati programmi ed eseguendone il refactoring. Mentre svolgevo ogni modifica, mi sono chiesto perché la stavo apportando e poi ho scritto il motivo. Il risultato è un elenco piuttosto lungo di cose che mi “puzzano” quando leggo del codice.

Questo elenco è concepito per essere letto da cima a fondo e anche per essere usato da riferimento.

Commenti

C1: Informazioni inappropriate

È inappropriate che un commento contenga informazioni che dovrebbero trovarsi in un altro elemento, come per esempio il sistema di controllo del codice sorgente, il sistema di controllo delle problematiche o qualsiasi altro sistema di registrazione. Le cronologie delle modifiche, per esempio, non fanno altro che congestionare i file di codice sorgente con ammassi di testo ormai vecchio e di scarso interesse. In generale, i metadati, come gli autori, la data di ultima modifica, il codice SPR e così via non dovrebbero comparire nei commenti. I commenti dovrebbero essere riservati a note tecniche sul codice e sulle scelte progettuali.

C2: Commenti obsoleti

Un commento ormai datato, irrilevante e impreciso è obsoleto. I commenti invecchiano rapidamente. È meglio non scrivere un commento che diverrà obsoleto. Se trovate un commento obsoleto, è meglio

aggiornarlo o eliminarlo il più rapidamente possibile. I commenti obsoleti tendono a spostarsi dal codice che un tempo descrivevano. Divengono come isole galleggianti di irrilevanza e di inganno sul codice.

C3: Commenti ridondanti

Un commento è ridondante se descrive qualcosa che descrive già adeguatamente se stesso. Per esempio:

```
i++; // incrementa i
```

Un altro esempio è un javadoc che non dice nulla di più (o magari anche meno) della signature della funzione:

```
/**  
 * @param sellRequest  
 * @return  
 * @throws ManagedComponentException  
 */  
public SellResponse beginSellItem(SellRequest sellRequest)  
  
throws ManagedComponentException
```

I commenti dovrebbero dire cose che il codice non è in grado di dire da solo.

C4: Commenti mal scritti

Un commento che merita di essere scritto, merita anche di essere scritto bene. Se pensate di scrivere un commento, assicuratevi che sia il miglior commento che possiate scrivere. Scegliete le parole con attenzione. Usate bene la grammatica e la punteggiatura. Non divagate. Non raccontate ovvietà. Siate concisi.

C5: Codice in commento

Mi fa imbestialire vedere delle parti di codice trasformate in commenti. Chissà a quando risale? Chissà se è ancora significativo?

Tuttavia nessuno lo cancellerà perché tutti penseranno che chiunque l'abbia scritto volesse farci qualcosa.

Tale codice se ne sta lì e “marcisce”, diventando sempre più irrilevante ogni giorno che passa. Richiama funzioni che non esistono più. Impiega variabili il cui nome è cambiato. Segue convenzioni da tempo obsolete. Inquina i moduli che lo contengono e distrae coloro che tentano di leggerlo. Il codice commentato è un *abominio*.

Quando vedete del codice in commento, *cancellatelo!* Non temete: il sistema di controllo del codice sorgente continuerà a ricordarlo. Se qualcuno dovesse mai averne bisogno, ripristinerà la versione precedente. Dategli il colpo di grazia.

Ambiente

E1: Build che richiedono più di un passo

La build di un progetto dovrebbe essere un'unica operazione semplice. Non bisogna essere costretti a considerare tanti piccoli pezzi di codice sorgente. Non deve essere necessaria una sequenza di arcani comandi o script contestuali per compilare i singoli elementi. Non si deve essere costretti a cercare in lungo e in largo tutti i vari file JAR, XML e così via richiesti dal sistema. *Dovreste* essere in grado di verificare il sistema con un unico comando e poi usare un altro comando per lanciare la build.

```
svn get mySystem  
cd mySystem  
ant all
```

E2: Test che richiedono più di un passo

Dovreste essere in grado di eseguire tutti gli unit test con un solo comando. Meglio ancora, potete lanciare tutti i test facendo clic su un

solo pulsante dell’IDE. Al limite dovreste essere in grado di usare un unico, semplice comando in una shell. La capacità di eseguire tutti i test è così importante che dovrebbe essere rapida, facile e banale.

Funzioni

F1: Troppi argomenti

Le funzioni dovrebbero avere un numero limitato di argomenti. Meglio nessun argomento; uno, due o tre argomenti vanno bene. Più di tre è troppo: dovrebbero essere evitate in assoluto. (Vedi “Argomenti di funzione” nel Capitolo 3.)

F2: Argomenti di output

Gli argomenti di output sono anti-intuitivi. Coloro che leggono il codice si aspettano che gli argomenti siano input, non output. Se la vostra funzione deve modificare lo stato di qualcosa, dovrà trattarsi dello stato dell’oggetto sul quale viene richiamata. (Vedi “Argomenti di output” nel Capitolo 3.)

F3: Flag usati come argomenti

Gli argomenti booleani dichiarano a gran voce che la funzione fa più di una cosa. Generano confusione e dovrebbero essere eliminati. (Vedi “Flag usati come argomenti” nel Capitolo 3.)

F4: Funzioni “morte”

I metodi che non vengono mai richiamati dovrebbero essere eliminati. Conservare del codice ormai inutile è uno spreco. Non temete

di cancellare la funzione. Ricordate: il sistema di controllo del codice sorgente continuerà a ricordarsela.

Generali

G1: Più linguaggi in un file di codice sorgente

Gli ambienti di programmazione di oggi consentono di inserire più linguaggi in un unico file di codice sorgente. Per esempio, un file di codice sorgente Java potrebbe contenere frammenti di XML, HTML, YAML, Javadoc, inglese, italiano, JavaScript e così via. Per un altro esempio, oltre all'HTML un file JSP potrebbe contenere codice Java, una sintassi a tag, commenti in linguaggio naturale, Javadoc, XML, JavaScript e così via.

Questo è quanto meno fonte di confusione, ma più probabilmente pura negligenza.

L'ideale è che un file di codice sorgente contenga un solo linguaggio. Realisticamente, è probabile che se ne debba usare più d'uno. Ma dovremmo cercare in tutti i modi di ridurre al minimo sia il numero sia l'estensione dei frammenti di altri linguaggi nel nostro file di codice sorgente.

G2: Non viene implementato un comportamento naturale

Seguendo il principio della “minima sorpresa” (*Principle of Least Astonishment*, http://en.wikipedia.org/wiki/Principle_of_least_astonishment), ogni funzione o classe dovrebbe implementare i comportamenti che un altro programmatore possa ragionevolmente aspettarsi. Per esempio,

considerate una funzione che traduce il nome di un giorno in un'enum che rappresenta il giorno.

```
Day day = DayDate.StringToDay(String dayName);
```

Ci aspetteremmo che la stringa "Monday" venga tradotta in `Day.MONDAY`. Ci aspetteremmo anche che venissero tradotte le abbreviazioni più comuni e anche che la funzione non distingua fra lettere maiuscole e minuscole.

Quando un comportamento ovvio non viene implementato, coloro che leggeranno il codice e gli utilizzatori del codice non potranno più contare sul loro intuito per i nomi di funzione. Perderanno la fiducia nell'autore e torneranno a leggere i dettagli del codice.

G3: Comportamento errato alle delimitazioni

Sembra solo ovvio dire che il codice dovrebbe comportarsi correttamente. Il problema è che raramente ci rendiamo conto di quanto sia complicato il comportamento corretto. Spesso gli sviluppatori scrivono funzioni che (in teoria) dovrebbero funzionare e poi si fidano del loro intuito anziché dimostrare che il codice funziona in tutti i suoi "angoli" e sui casi di limite.

Nulla può sostituire la diligenza nel lavoro. Ogni condizione di delimitazione, ogni caso limite, ogni dettaglio e ogni eccezione rappresenta qualcosa che può potenzialmente confondere un algoritmo altrimenti elegante e intuitivo. *Non contate ciecamente sul vostro intuito*. Individuate ogni condizione di delimitazione e scrivete un test apposito.

G4: Disattivazione delle "sicure"

L'incidente di Chernobyl si è verificato perché il direttore dell'impianto aveva disattivato i meccanismi di sicurezza, uno per uno. Questo perché non permettevano di condurre un determinato

esperimento. Il risultato è stato che l'esperimento non è stato condotto e che il mondo è andato incontro alla sua prima catastrofe nucleare civile.

È pericoloso disattivare i meccanismi di sicurezza. Esercitare un controllo manuale su `serialVersionUID` può essere necessario, ma è sempre pericoloso. Disattivare determinati warning del compilatore (o tutti i warning!) può produrre un risultato, ma al rischio di estenuanti sessioni di debugging. Disattivare i test che non passano e raccontarvi la storiella che ci penserete poi è un po' come convincersi che i soldi della carta di credito siano gratis.

G5: Duplicazioni

Questa è una delle più importanti regole di questo libro e dovreste applicarla scrupolosamente. Praticamente ogni autore di un libro sulla progettazione del software menziona questa regola. Dave Thomas e Andy Hunt l'hanno chiamata il Principio DRY (Don't Repeat Yourself, non ripetetevi [PRAG]). Kent Beck l'ha resa uno dei principi di base della "programmazione estrema" e l'ha chiamata: "Una e una sola volta". Ron Jeffries considera questa regola seconda solo al passaggio di tutti i test.

Ogni volta che vedete delle duplicazioni nel codice, avete trovato una mancata opportunità di astrazione. Tale duplicazione dovrebbe, probabilmente, diventare una subroutine o magari un'altra classe. Trasformando la duplicazione in un'astrazione, arricchirete il vocabolario del linguaggio del vostro progetto. Gli altri programmatore potranno usare le funzionalità astratte che avete creato. La programmazione diviene più veloce e meno soggetta a errori, proprio per il fatto che avete elevato il livello di astrazione.

La forma più ovvia di duplicazione si ha quando vi sono frammenti di codice identico, come se alcuni programmatore fossero impazziti e,

con il mouse, avessero incollato lo stesso codice più e più volte. Queste occorrenze dovrebbero essere sostituite da semplici metodi.

Una forma più subdola può essere in un costrutto `switch/case` o `if/else` che compare più e più volte in vari moduli, controllando sempre lo stesso set di condizioni. Queste occorrenze dovrebbero essere sostituite dal polimorfismo.

Ancora più subdoli sono i moduli che hanno algoritmi simili, ma che non hanno righe di codice simili. Anche questa è duplicazione e dovrebbe essere risolta applicando i pattern TEMPLATE METHOD [GOF] o STRATEGY [GOF].

In effetti, la maggior parte dei pattern progettuali che sono comparsi negli ultimi quindici anni sono semplicemente modi noti per eliminare le duplicazioni. Anche i Codd Normal Form sono una strategia per eliminare le duplicazioni nello schema di un database. La stessa programmazione OO è una strategia per organizzare i moduli ed eliminare le duplicazioni. E anche, non sorprenderà, la programmazione strutturata.

Credo di essermi spiegato. Trovate ed eliminate ogni duplicazione, ogni volta che potete.

G6: Codice posto al livello di astrazione errato

È importante creare astrazioni che separano i concetti generali di livello più elevato dai concetti dettagliati di basso livello. Talvolta possiamo farlo creando delle classi astratte, per contenere i concetti di alto livello, e delle classi derivate, per contenere i concetti di basso livello. Quando ci comportiamo in questo modo, dobbiamo assicurarci che la separazione sia completa. Vogliamo che *tutti* i concetti di basso livello siano nelle classi derivate e che *tutti* i concetti di alto livello siano nella classe base.

Per esempio, le costanti, le variabili e le funzioni di servizio che appartengono solo all'implementazione non dovrebbero trovarsi nella classe base. La classe base non dovrebbe saperne nulla.

Questa regola riguarda anch'essa i file di codice sorgente, i componenti e i moduli. Per una buona progettazione, dobbiamo separare i concetti ai vari livelli e collocarli in container differenti. Talvolta questi container sono classi base o derivate e talvolta sono file di codice sorgente, moduli o componenti. In qualsiasi caso, la separazione deve essere completa. Non vogliamo mescolare insieme concetti di alto e basso livello.

Considerate il seguente codice:

```
public interface Stack {  
    Object pop() throws EmptyException;  
    void push(Object o) throws FullException;  
    double percentFull();  
    class EmptyException extends Exception {}  
    class FullException extends Exception {}  
}
```

La funzione `percentFull` si trova al livello di astrazione errato. Sebbene vi siano molte implementazioni di `Stack` in cui il concetto di full (pieno) è ragionevole, vi sono altre implementazioni che semplicemente non hanno modi di sapere quanto sono piene. Pertanto, la funzione dovrebbe essere collocata in una interfaccia derivata, come `BoundedStack`.

Forse state pensando che l'implementazione possa restituire semplicemente zero se lo stack non ha limiti. Il problema però è che nessuno stack è davvero illimitato. Non potete prevenire una

`OutOfMemoryException` controllando

```
stack.percentFull() < 50.0.
```

Implementando la funzione in modo che restituisca uno 0, essa vi direbbe una bugia.

Il punto è che non potete mentire per un'astrazione mal collocata. Isolare le astrazioni è uno dei compiti più difficili per gli sviluppatori

di software e non esistono correzioni rapide per gli errori in questo senso.

G7: Classi base che dipendono dalle loro derivate

Il motivo più comune per partizionare i concetti fra classi base e derivate è quello di rendere i concetti delle classi base di livello più elevato indipendenti dai concetti delle classi derivate di livello più basso. Pertanto, quando vediamo delle classi base che menzionano i nomi delle loro derivate, immaginiamo già che ci sia un problema. In generale, le classi base non dovrebbero conoscere nulla delle loro derivate.

Vi sono delle eccezioni a questa regola, naturalmente. Talvolta il numero delle derivate è rigidamente fissato e la classe base contiene del codice che seleziona una di esse. Lo vediamo nelle implementazioni di macchine a stati finiti. Tuttavia, in tal caso le derivate e la classe base sono strettamente accoppiate e vanno sempre fornite insieme nello stesso file jar. Nel caso generale vogliamo essere in grado di fornire le classi derivate e basi in file jar distinti.

L'inserimento delle classi derivate e base in file jar distinti, assicurandosi che i file jar base non sappiano nulla del contenuto dei file jar derivati, ci consentono di fornire i nostri sistemi in componenti discreti e indipendenti. Quando tali componenti vengono modificati, possono nuovamente essere forniti senza i componenti base. Ciò significa che l'impatto di una modifica viene notevolmente attenuato e la manutenzione dei sistemi sul campo diviene molto più semplice.

G8: Eccesso di informazioni

I moduli ben definiti hanno interfacce molto piccole che consentono di fare molto con molto poco. I moduli mal definiti hanno interfacce

estese e profonde, che costringono a usare molte istruzioni anche per ottenere cose semplici. Un’interfaccia ben definita non offre moltissime funzioni da cui dipendere, e quindi l’accoppiamento è basso.

Un’interfaccia mal definita fornisce una gran quantità di funzioni che siete obbligati a richiamare, con un accoppiamento elevato.

I buoni sviluppatori di software imparano a limitare ciò che espongono alle interfacce delle loro classi e dei loro moduli. Meno metodi ha una classe, meglio è. Meno variabili conosce una funzione, meglio è. Meno variabili di istanza ha una classe, meglio è.

Nascondete i vostri dati. Nascondete le vostre funzioni di servizio. Nascondete le vostre costanti e gli oggetti temporanei. Non create classi con tanti metodi o tante variabili di istanza. Non create grandi quantità di variabili protette e funzioni per le vostre sottoclassi. Concentratevi sul mantenere le interfacce molto strette e molto leggere. Riducete al minimo ogni accoppiamento, limitando la quantità di informazioni.

G9: Il “dead code”

Il dead code è codice che non viene mai eseguito. Lo trovate nel corpo di un’istruzione `if` che controlla una condizione che non può mai verificarsi. Lo trovate nel blocco `catch` di un `try` che non lo lancia mai. Lo trovate in piccoli metodi di servizio che non vengono mai richiamati o nelle condizioni `switch/case` che non si verificano mai.

Il problema del dead code è che dopo un po’ inizia a marcire. Più vecchio è, più acre diviene il suo odore. Questo perché non viene aggiornato insieme al resto del codice. Viene comunque *compilato*, ma non segue più le nuove convenzioni o regole. È stato scritto in un’epoca in cui il sistema era *differente*. Quando trovate del dead code, fate la cosa giusta. Dategli una degna sepoltura. Cancellatelo dal sistema.

G10: Separazione verticale

Le variabili e le funzioni dovrebbero essere definite nei pressi di dove vengono usate. Le variabili locali dovrebbero essere dichiarate appena sopra il loro primo uso e dovrebbero avere una limitata visibilità “verticale”. Non vogliamo che esistano variabili locali dichiarate a centinaia di righe di distanza dal loro uso.

Le funzioni private dovrebbero essere definite appena sotto il loro primo uso. Le funzioni private appartengono al livello di visibilità (scope) dell’intera classe, ma è comunque bene limitarne la distanza verticale fra le chiamate e le definizioni. Trovare una funzione privata dovrebbe richiedere solo un’occhiata a partire dal suo primo uso.

G11: Incoerenza

Se fate qualcosa in un determinato modo, fate anche tutte le cose simili nello stesso modo. Questo rimanda al principio della minima sorpresa. Scegliete con cura le convenzioni e poi continuate a seguirle.

Se in una determinata funzione impiegate una variabile di nome `response` per contenere una `HttpServletResponse`, allora usate lo stesso nome di variabile coerentemente in tutte le altre funzioni che usano oggetti `HttpServletResponse`. Se denominate un metodo `processVerificationRequest`, allora usate un nome simile, per esempio `processDeletionRequest`, per i metodi che elaborano altri tipi di richieste.

Questa semplice coerenza, se applicata costantemente, può rendere il codice molto più facile da leggere e modificare.

G12: Congestione

A che cosa serve un costruttore di default senza implementazione? Non fa altro che congestionare il codice con artefatti inutili. Variabili

inutilizzate, funzioni mai richiamate, commenti non informativi e così via. Tutte queste cose dovrebbero essere eliminate. Mantenete i vostri file di codice sorgente puliti, ben organizzati e non congestionati.

G13: Accoppiamento artificioso

Le cose che non dipendono le une dalle altre non dovrebbero essere accoppiate artificiosamente. Per esempio, le `enum` generali non dovrebbero trovarsi all'interno di classi specifiche, perché questo costringe l'intera applicazione a sapere di più di queste classi specifiche. Lo stesso vale per le funzioni statiche dichiarate in classi specifiche.

In generale un accoppiamento è artificioso se è fra due moduli che non servono l'uno all'altro. È il risultato del fatto di aver inserito una variabile, una costante o una funzione in una posizione temporanea comoda, ma inappropriata. Questa è pigrizia e superficialità.

Cercate di determinare dove devono essere dichiarate le funzioni, le costanti e le variabili. Non “ficcatele” nel posto più comodo e immediato per poi lasciarle lì.

G14: Una questione di invidia (“feature envy”)

Questa è una delle avvertenze di Martin Fowler [Refactoring]. I metodi di una classe dovrebbero interessarsi alle variabili e alle funzioni della classe cui appartengono e non alle variabili e alle funzioni di altre classi. Quando un metodo usa gli accessor e mutator di qualche altro oggetto per manipolare i dati di tale oggetto, prova “invidia” per il livello di visibilità (scope) della classe di tale oggetto. Vorrebbe trovarsi all'interno di tale classe, in modo da poter avere un accesso diretto alle variabili che manipola. Per esempio:

```
public class HourlyPayCalculator {  
    public Money calculateWeeklyPay(HourlyEmployee e) {
```

```

        int tenthRate = e.getTenthRate().getPennies();
        int tenthsWorked = e.getTenthsWorked();
        int straightTime = Math.min(400, tenthsWorked);
        int overTime = Math.max(0, tenthsWorked - straightTime);
        int straightPay = straightTime * tenthRate;
        int overtimePay = (int) Math.round(overTime*tenthRate*1.5);
        return new Money(straightPay + overtimePay);
    }

}

```

Il metodo `calculateWeeklyPay` entra nell’oggetto `HourlyEmployee` per ottenere i dati sui quali opera. Il metodo `calculateWeeklyPay` “invidia” il livello di visibilità (scope) di `HourlyEmployee`. Vorrebbe trovarsi all’interno di `HourlyEmployee`.

Senza cambiare niente altro, vogliamo eliminare questa “invidia” perché espone a una classe i meccanismi interni di un’altra. Talvolta, tuttavia, questa “invidia” è un male necessario. Considerate il seguente:

```

public class HourlyEmployeeReport {
    private HourlyEmployee employee ;
    public HourlyEmployeeReport(HourlyEmployee e) {
        this.employee = e;
    }
    String reportHours() {
        return String.format(
            "Name: %s\tHours:%d.%1d\n",
            employee.getName(),
            employee.getTenthsWorked()/10, employee.getTenthsWorked()%10);
    }
}

```

Chiaramente, il metodo `reportHours` invidia la classe `HourlyEmployee`. D’altra parte, non vogliamo che `HourlyEmployee` sappia qualcosa del formato del report. Spostare tale stringa di formattazione nella classe `HourlyEmployee` violerebbe parecchi principi della progettazione OO (in particolare, il principio SRP, il principio OCP e il principio CCP, vedi [PPP]). Accoppierebbe `HourlyEmployee` con il formato del report, esponendola alle modifiche apportate a tale formato.

G15: Argomenti “a selettore”

Non esiste forse nulla di più abominevole di un argomento `false` che pende alla fine di una chiamata a funzione. Che cosa significa? Che

cosa cambierebbe se fosse `true`? Non solo lo scopo di un argomento a selettore è difficile da ricordare, ma ogni argomento a selettore combina più funzioni in una sola. Gli argomenti “a selettore” sono solo un modo pigro per evitare di suddividere una grossa funzione in più funzioni più piccole. Considerate il seguente esempio:

```
public int calculateWeeklyPay(boolean overtime) {  
    int tenthRate = getTenthRate();  
    int tenthsWorked = getTenthsWorked();  
    int straightTime = Math.min(400, tenthsWorked);  
    int overTime = Math.max(0, tenthsWorked - straightTime);  
    int straightPay = straightTime * tenthRate;  
    double overtimeRate = overtime ? 1.5 : 1.0 * tenthRate;  
    int overtimePay = (int) Math.round(overTime * overtimeRate);  
    return straightPay + overtimePay;  
}
```

Si richiama questa funzione con `true` se gli straordinari (`overtime`) vengono pagati una volta e mezza e con `false` se vengono pagati come tutto il resto. Questo costringe a ricordarsi che cosa significhi `calculateWeeklyPay(false)` ogni volta che lo incontrate. Ma il vero problema di una funzione come questa è che l'autore ha perso l'opportunità di scrivere la seguente:

```
public int straightPay() {  
    return getTenthsWorked() * getTenthRate();  
}  
public int overTimePay() {  
    int overTimeTenths = Math.max(0, getTenthsWorked() - 400);  
    int overTimePay = overTimeBonus(overTimeTenths);  
    return straightPay() + overTimePay;  
}  
private int overTimeBonus(int overTimeTenths) {  
    double bonus = 0.5 * getTenthRate() * overTimeTenths;  
    return (int) Math.round(bonus);  
}
```

Naturalmente, i selettori non sono necessariamente `boolean`. Possono essere `enum`, interi o qualsiasi altro tipo di argomento che viene usato per selezionare il comportamento della funzione. In generale è meglio avere più funzioni piuttosto che dover passare un codice a una funzione per selezionarne il comportamento.

G16: Offuscamento dello scopo

Vogliamo che il codice sia il più possibile espressivo. Le espressioni al volo, la notazione ungherese e i numeri magici offuscano lo scopo dell'autore. Per esempio, ecco un possibile aspetto della funzione

`overtimePay`:

```
public int m_otCalc() {  
    return iThsWkd * iThsRte +  
        (int) Math.round(0.5 * iThsRte *  
            Math.max(0, iThsWkd - 400))  
};  
}
```

Per quanto sia piccola e ricca, è anche praticamente impenetrabile. Vale però la pena di chiarire qual è lo scopo del nostro codice a tutti coloro che lo dovranno leggere.

G17: Responsabilità mal collocate

Una delle responsabilità più importanti di uno sviluppatore software è la scelta di dove collocare il codice. Per esempio, dove deve trovarsi la costante `PI`? Dovrà trovarsi nella classe `Math`? Forse appartiene alla classe `Trigonometry`? O magari alla classe `Circle`?

Qui entra in gioco il principio della minima sorpresa. Il codice dovrebbe essere collocato là dove chi legge si aspetta di trovarlo. La costante `PI` dovrà andare là dove sono dichiarate le funzioni trigonometriche. La costante `OVERTIME_RATE` dovrebbe essere dichiarata nella classe `HourlyPayCalculator`.

Talvolta però abbiamo le nostre idee sulla posizione corretta di determinate funzionalità. Le collochiamo in una funzione comoda per noi, ma non necessariamente intuitiva per chi legge. Per esempio, se dobbiamo stampare un report col numero totale di ore lavorate, potremmo sommare tali ore nel codice che stampa il report o potremmo tentare di conservare il totale nel codice che accetta i dati orari.

Un modo per prendere questa decisione consiste nell’osservare il nome delle funzioni. Immaginiamo che il nostro modulo per i report abbia una funzione chiamata `getTotalHours`. Supponiamo anche che il modulo che accetta i dati orari abbia una funzione `saveTimeCard`. Quale di queste due funzioni, in base al nome, implica il calcolo del totale? La risposta dovrebbe essere ovvia.

Chiaramente, talvolta vi sono motivi prestazionali per i quali il totale dovrebbe essere calcolato mentre vengono inseriti i dati orari invece che quando viene stampato il report. Può andar bene, ma il nome delle funzioni deve rifletterlo. Per esempio, nel modulo del calcolo delle ore vi dovrebbe essere una funzione `computeRunningTotalOfHours`.

G18: Modificatore static inappropriato

`Math.max(double a, double b)` è un buon metodo statico. Non opera su un’unica istanza; in effetti, sarebbe sciocco dover dire `new Math().max(a,b)` o anche `a.max(b)`. Tutti i dati usati da `max` provengono dai suoi due argomenti e non da un oggetto “proprietario”. In buona sostanza, in nessun caso vogliamo che `Math.max` sia polimorfica.

Talvolta, tuttavia, scriviamo funzioni statiche che non dovrebbero essere statiche. Considerate il seguente esempio:

```
HourlyPayCalculator.calculatePay(employee, overtimeRate).
```

Di nuovo, sembra un funzione giustamente statica. Non opera su alcun oggetto specifico e trae tutti i suoi dati dai suoi argomenti. Tuttavia, è anche ragionevole che possiate volere che questa funzione sia polimorfica. Potremmo voler implementare più algoritmi per il calcolo della paga oraria, per esempio, `OvertimeHourlyPayCalculator` e `StraightTimeHourlyPayCalculator`. Pertanto, in questo caso la funzione non dovrebbe essere statica. Dovrebbe essere una funzione membro non statica di `Employee`.

In generale dovreste preferire i metodi non statici ai metodi statici. In caso di dubbi, rendete la funzione non statica. Se davvero volete che la funzione sia statica, assicuratevi che non vi sia alcuna possibilità che vogliate usarla in modo polimorfico.

G19: Usate variabili descrittive

Kent Beck ne ha parlato nel suo ottimo libro *Smalltalk Best Practice Patterns* ([Beck97], p. 108.) e di nuovo, recentemente, nell'altrettanto notevole libro *Implementation Patterns* [Beck07]. Uno dei modi più potenti per rendere leggibile un programma consiste nel suddividere i calcoli in valori intermedi contenuti in variabili dotate di nomi significativi.

Considerate questo esempio tratto da FitNesse:

```
Matcher match = headerPattern.matcher(line);
if(match.find())
{
    String key = match.group(1);
    String value = match.group(2);
    headers.put(key.toLowerCase(), value);
}
```

Il semplice uso di variabili descrittive chiarisce che il primo `match-group` è la *chiave* e il secondo `match-group` è il *valore*.

È difficile negarlo: più le variabili sono descrittive e meglio è. È spettacolare come un modulo oscuro possa diventare improvvisamente trasparente semplicemente suddividendo i calcoli in valori intermedi ben denominati.

G20: Il nome di una funzione deve essere comunicativo

Osservate il seguente codice:

```
Date newDate = date.add(5);
```

Vi aspettate che aggiunga cinque giorni alla data? O magari settimane o ore? L’istanza di `Date` viene modificata o la funzione restituisce semplicemente un nuovo `Date` senza modificare il vecchio? È impossibile capire dalla chiamata che cosa fa la funzione.

Se la funzione aggiungesse cinque giorni alla data e modificasse la data, dovrebbe chiamarsi `addDaysTo` o `increaseByDays`. Se invece, la funzione restituisce una nuova `Date` posta cinque giorni dopo, senza cambiare l’istanza, dovrebbe chiamarsi `daysLater` o `daysSince`.

Se dovete osservare l’implementazione (o la documentazione) della funzione per capire che cosa fa, dovrete scegliere un nome migliore o ridisporre la funzionalità in modo che possa essere collocata in funzioni dotate di nomi migliori.

G21: Comprendere l’algoritmo

Si è scritto del codice molto divertente perché qualcuno non si è preso la briga di comprendere l’algoritmo e ha messo insieme un po’ di istruzioni `if` e flag, senza darsi la pena di capire che cosa stava accadendo.

La programmazione è anche esplorazione. *Pensate* di conoscere l’algoritmo corretto per qualcosa, ma poi finite per modificarlo, alterarlo e ritoccarlo, fino a farlo “funzionare”. Ma come sapete che “funziona”? Se passa i casi di test potete anche pensarla.

Non c’è niente di sbagliato in questo approccio. In effetti, spesso è l’unico modo per convincere una funzione a fare quello che dovrebbe. Tuttavia, non è sufficiente lasciare gli apici attorno alla parola “funziona”.

Prima di pensare di aver terminato di lavorare con una funzione, assicuratevi di aver *capito* come funziona. Non è sufficiente che passi tutti i test. Dovete *assicurarvi* che la soluzione sia corretta. (C’è una

bella differenza tra sapere come funziona il tuo codice e sapere se l'algoritmo usato farà il lavoro richiesto. Essere sicuri che un algoritmo sia appropriato, spesso è un dato di fatto. Nutrire dubbi su ciò che fa il tuo codice è solo pigrizia.)

Spesso il modo migliore per acquisire tale conoscenza consiste nell'eseguire il refactoring di la funzione per ottenere qualcosa di pulito e talmente espressivo da rendere ovvio il fatto che funziona.

G22: Rendere fisiche le dipendenze logiche

Se un modulo dipende da un altro, tale dipendenza dovrebbe essere fisica, non solo logica. Il modulo dipendente non dovrebbe presumere nulla (in altre parole, avere dipendenze logiche) dal modulo da cui dipende. Piuttosto dovrebbe richiedere esplicitamente a tale modulo tutte le informazioni che gli servono.

Per esempio, immaginate di dover scrivere una funzione che stampa un report delle ore di lavoro dei dipendenti. Una classe chiamata `HourlyReporter` raccoglie tutti i dati in una forma appropriata e poi li passa a `HourlyReportFormatter` per la stampa (Listato 17.1)

Listato 17.1 HourlyReporter.java.

```
public class HourlyReporter {
    private HourlyReportFormatter formatter;
    private List<LineItem> page;
    private final int PAGE_SIZE = 55;
    public HourlyReporter(HourlyReportFormatter formatter) {
        this.formatter = formatter;
        page = new ArrayList<LineItem>();
    }
    public void generateReport(List<HourlyEmployee> employees) {
        for (HourlyEmployee e : employees) {
            addLineItemToPage(e);
            if (page.size() == PAGE_SIZE)
                printAndClearItemList();
        }
        if (page.size() > 0)
            printAndClearItemList();
    }
    private void printAndClearItemList() {
        formatter.format(page);
        page.clear();
    }
}
```

```

private void addLineItemToPage(HourlyEmployee e) {
    LineItem item = new LineItem();
    item.name = e.getName();
    item.hours = e.getTenthsWorked() / 10;
    item.tenths = e.getTenthsWorked() % 10;
    page.add(item);
}
public class LineItem {
    public String name;
    public int hours;
    public int tenths;
}
}

```

Questo codice ha una dipendenza logica che non è stata resa fisica. Riuscite a individuarla? È la costante `PAGE_SIZE`. Perché `HourlyReporter` dovrebbe conoscere le dimensioni della pagina? Ciò dovrebbe rientrare nelle responsabilità di `HourlyReportFormatter`.

Il fatto che `PAGE_SIZE` sia dichiarata in `HourlyReporter` rappresenta un'errata collocazione delle responsabilità [G17] la quale fa in modo che `HourlyReporter` debba conoscere le dimensioni della pagina. Questa è una dipendenza logica. `HourlyReporter` dipende dal fatto che `HourlyReportFormatter` possa gestire un `PAGE_SIZE` di 55. Se qualche implementazione di `HourlyReportFormatter` non riuscisse a gestire tali dimensioni, si avrebbe un errore.

Possiamo rendere fisica questa dipendenza creando in `HourlyReportFormatter` un nuovo metodo chiamato `getMaxPageSize()`. `HourlyReporter` richiamerà tale funzione invece di usare la costante `PAGE_SIZE`.

G23: Meglio il polimorfismo di un if/else o di uno switch/case

Questo può sembrare uno strano suggerimento dopo quanto detto nel Capitolo 6. Dopotutto, in tale capitolo ho detto che le istruzioni `switch` sono probabilmente appropriate in quelle parti del sistema in cui l'aggiunta di nuove funzioni è più probabile dell'aggiunta di nuovi tipi.

Innanzitutto, molti usano le istruzioni `switch` perché è la classica soluzione a forza bruta, non perché esse siano la soluzione più adatta alla situazione. Pertanto questa euristica è qui con lo scopo di ricordarci di considerare il polimorfismo prima di usare uno `switch`.

In secondo luogo, i casi in cui le funzioni sono più volatili dei tipi sono relativamente rare. Pertanto ogni istruzione `switch` dovrebbe essere guardata con sospetto.

Io uso la regola *One Switch*:

Non ci deve mai essere più di un'istruzione `switch` per un determinato tipo di selezione. I `case` in tale istruzione `switch` devono creare oggetti polimorfici che prendano il posto di altre istruzioni `switch` nel resto del sistema.

G24: Seguite convenzioni standard

Ogni team dovrebbe seguire uno standard di programmazione basato su norme convenzionali del settore. Questo standard di programmazione dovrebbe specificare cose come dove dichiarare le variabili di istanza; come denominare classi, metodi e variabili; dove collocare le parentesi e così via. Il team non dovrebbe aver bisogno di un documento per descrivere queste convenzioni, perché è già il codice a fornire gli esempi.

Tutti i membri del team dovrebbero poi seguire queste convenzioni. Ciò significa che ogni membro deve essere sufficientemente maturo per rendersi conto che non è importante l'esatta posizione in cui si collocano le parentesi, ma il fatto che tale posizione sia condivisa da tutti.

Se volete sapere quali convenzioni seguo io, ebbene le vedete nel codice rifattorizzato presentato nei Listati da B.7 a B.14, nell'Appendice B.

G25: Al posto dei “numeri magici”, usate costanti “parlanti”

Questa è probabilmente una delle regole più vecchie dello sviluppo software. Mi ricordo di averla letta alla fine degli anni Sessanta nei manuali introduttivi ai linguaggi COBOL, FORTRAN e PL/1. In generale non è una buona idea avere dei numeri grezzi dentro il codice. Dovreste celarli dietro costanti ben denominate.

Per esempio, il numero `86400` dovrebbe essere celato nella costante `SECONDS_PER_DAY`. Se stampate 55 righe per pagina, il valore `55` dovrebbe essere celato nella costante `LINES_PER_PAGE`.

Alcuni valori sono così facili da riconoscere che non sempre c’è bisogno di una costante entro la quale celarli sempre che vengano usate in codice molto auto-descrittivo. Un esempio:

```
double kmWalked = milesWalked * 0.621371;
int dailyPay = hourlyRate * 8;
double circumference = radius * Math.PI * 2;
```

Avremmo davvero bisogno delle costanti `KM_PER_MILE`, `WORK_HOURS_PER_DAY` e `TWO` negli esempi sopra riportati? Chiaramente, l’ultimo caso è assurdo. Vi sono alcune formule in cui è semplicemente meglio scrivere le costanti direttamente come numeri. Potreste avere dei dubbi sul caso `WORK_HOURS_PER_DAY`, perché le leggi o le convenzioni potrebbero cambiare. D’altra parte, tale formula si legge così bene con quell’8 che sarei riluttante ad aggiungere un carico di 17 caratteri per tutti coloro che leggono il codice. E nel caso `KM_PER_MILE`, il numero `0.621371` è piuttosto esplicito e così coloro che leggeranno il codice ne riconosceranno il significato anche senza altro contesto.

Costanti come `3.141592653589793` sono ben note e facilmente riconoscibili. Tuttavia, la probabilità di commettere un errore è troppo elevata per lasciarle così come sono. Ogni volta che qualcuno vede `3.1415927535890793`, sa che si tratta di Pi greco e così evita di esaminarla.

(Avete notato l'errore di una singola cifra?) D'altra parte non vogliamo usare un'abbreviazione eccessiva, come `3.14`, `3.14159`, `3.142` e così via.

Pertanto, è una buona cosa che la costante `Math.PI` sia già stata definita per noi.

Il termine “numero magico” non vale solo per i numeri. Vale per ogni elemento che abbia un valore che non è autodescrittivo. Per esempio:

```
assertEquals(7777, Employee.find("John Doe").employeeNumber());
```

Vi sono due numeri magici in questa asserzione. Il primo è ovviamente `7777`, e il suo significato non è ovvio. Il secondo numero magico è `"John Doe"` e il suo scopo non è chiaro.

Si scopre che `"John Doe"` è il nome del dipendente numero `7777` in un database di test creato dal team. Tutti nel team sanno che quando ci si connette a questo database, vi troveremo alcuni dipendenti con valori e attributi ben noti. E poi `"John Doe"` rappresenta l'unico dipendente pagato a ore in tale database di test. Pertanto questo test in realtà dovrebbe essere:

```
assertEquals(  
    HOURLY_EMPLOYEE_ID,  
    Employee.find(HOURLY_EMPLOYEE_NAME).employeeNumber()  
) ;
```

G26: Siate precisi

Aspettarsi che la prima corrispondenza di una query sia l'unica è probabilmente ingenuo. Usare numeri in virgola mobile per rappresentare delle valute rasenta la criminalità. Evitare i lock e/o la gestione delle transazioni per aver dimenticato l'aggiornamento concorrente è probabilmente solo pigrizia. Dichiarare una variabile come un `ArrayList` quando in realtà è una `List` è eccessivamente vincolante. Rendere tutte le variabili `protected` per default non è sufficientemente vincolante.

Quando prendete una decisione nel vostro codice, cercate di essere precisi. Preendetela con perizia e considerando anche eventuali eccezioni. Non siate pigri nella precisione delle vostre decisioni. Se decidete di richiamare una funzione che potrebbe restituire `null`, occupatevi di quel `null`. Se la vostra query vi restituisce un record del database, fate in modo che il vostro codice controlli anche che non ve ne siano altri. Se avete a che fare con le valute, usate gli interi (o, meglio ancora, una classe `Money`, che si basa su interi) e calcolate gli arrotondamenti in modo appropriato. Se vi è la possibilità di un aggiornamento concorrente, assicuratevi di implementare un qualche tipo di meccanismo di bloccaggio.

Le ambiguità e le imprecisioni nel codice sono conseguenza di una mancanza di accordo o di pura pigrizia. In ogni caso dovrebbero essere eliminati.

G27: Il principio “structure over convention”

Esplificate le decisioni progettuali con la struttura piuttosto che con una semplice convenzione. Le convenzioni di denominazione vanno bene, ma sono comunque inferiori alla scelta di una struttura progettuale che imponga la regola. Per esempio, degli `switch/case` con enumerazioni “parlanti” comunicano meno delle classi base con metodi astratti. Nessuno sarebbe costretto a implementare l’istruzione `switch/case` ogni volta allo stesso modo; al contrario le classi base obbligano le classi concrete a implementare tutti i metodi astratti.

G28: Incapsulate i costrutti condizionali

La logica booleana è difficile da comprendere senza entrare nel contesto di un’istruzione `if` o `while`. Estraete delle funzioni che spieghino lo scopo del costrutto condizionale.

Per esempio,

```
if (shouldBeDeleted(timer))  
    è preferibile a:  
  
if (timer.hasExpired() && !timer.isRecurrent())
```

G29: Evitate i negativi nei costrutti condizionali

I negativi sono più difficili da comprendere dei positivi. Pertanto, quando possibile, i costrutti condizionali dovrebbero essere espressi in termini positivi. Per esempio,

```
if (buffer.shouldCompact())  
    è preferibile a:  
  
if (!buffer.shouldNotCompact())
```

G30: Le funzioni dovrebbero fare una cosa sola

Spesso si è tentati di creare funzioni con più sezioni che svolgono una serie di operazioni. Le funzioni di questo tipo fanno più di una cosa e dovrebbero essere convertite in più funzioni più piccole, ognuna delle quali fa una cosa soltanto.

Per esempio:

```
public void pay() {  
    for (Employee e : employees) {  
        if (e.isPayday()) {  
            Money pay = e.calculatePay();  
            e.deliverPay(pay);  
        }  
    }  
}
```

Questo frammento di codice fa tre cose. Fa un ciclo su tutti i dipendenti, controlla se il dipendente in questione deve essere pagato, e poi, nel caso, lo paga. Questo codice dovrebbe essere scritto come:

```
public void pay() {  
    for (Employee e : employees)  
        payIfNecessary(e);  
    }  
private void payIfNecessary(Employee e) {
```

```

    if (e.isPayday())
        calculateAndDeliverPay(e);
    }
private void calculateAndDeliverPay(Employee e) {
    Money pay = e.calculatePay();
    e.deliverPay(pay);
}

```

Ognuna di queste funzioni fa una cosa sola (Vedi “Che faccia una cosa sola”, nel Capitolo 3).

G31: Accoppiamenti temporali nascosti

Gli accoppiamenti temporali possono essere necessari, ma non nascondeteli. Strutturate gli argomenti delle vostre funzioni in modo che l’ordine con cui vengono chiamate sia ovvio. Considerate la seguente:

```

public class MoogDiver {
    Gradient gradient;
    List<Spline> splines;
public void dive(String reason) {
    saturateGradient();
    reticulateSplines();
    diveForMoog(reason);
}
...
}

```

L’ordine delle tre funzioni è importante. Dovete saturare il gradiente prima di produrre il reticolo delle spline e solo allora potete occuparvi del moog. Sfortunatamente, il codice non impone questo accoppiamento temporale. Un altro programmatore potrebbe richiamare `reticulateSplines` prima di `saturateGradient`, col risultato di avere una

`UnsaturatedGradientException`. Una soluzione migliore sarebbe:

```

public class MoogDiver {
    Gradient gradient;
    List<Spline> splines;
public void dive(String reason) {
    Gradient gradient = saturateGradient();
    List<Spline> splines = reticulateSplines(gradient);
    diveForMoog(splines, reason);
}
...
}

```

Qui si evidenzia l'accoppiamento temporale, creando una “catena di secchi”. Ogni funzione produce un risultato che viene impiegato dalla funzione successiva, pertanto non vi è alcun modo ragionevole di richiamarle fuori ordine.

Potreste sollevare l'obiezione che così si aumenta la complessità delle funzioni; avete ragione. Ma questa complessità sintattica aggiuntiva non fa altro che chiarire la complessità temporale della situazione fisica. Notate che ho lasciato al loro posto le variabili di istanza. Presumo che siano richieste dai metodi privati della classe. Anche in questo caso, voglio che gli argomenti siano al loro posto, per esplicitare l'accoppiamento temporale.

G32: Non siate arbitrari

Abbate un motivo per scegliere la struttura del vostro codice e assicuratevi che tale motivo sia comunicato adeguatamente dalla struttura del codice. Se la sua struttura appare arbitraria, gli altri si sentiranno in diritto di cambiarla, di “migliorarla”. Se la struttura appare coerente in tutto il sistema, gli altri la useranno e adotteranno la convenzione che essa rappresenta. Per esempio, recentemente stavo apportando delle modifiche a FitNesse e ho scoperto che uno dei nostri committer faceva questo:

```
public class AliasLinkWidget extends ParentWidget
{
    public static class VariableExpandingWidgetRoot {
        ...
    }
}
```

Il problema era che `VariableExpandingWidgetRoot` non aveva alcuna necessità di trovarsi nel livello di visibilità (scope) di `AliasLinkWidget`. Inoltre, anche altre classi non correlate facevano uso di `AliasLinkWidget.VariableExpandingWidgetRoot`. Queste classi non avevano alcuna necessità di conoscere `AliasLinkWidget`.

Forse il programmatore aveva inserito `VariableExpandingWidgetRoot` in `AliasWidget` per comodità o forse pensava che dovesse essere visibile all'interno di `AliasWidget`. Qualsiasi sia il motivo, il risultato era arbitrario. Le classi pubbliche che non sono strumenti di servizio di qualche altra classe non dovrebbero essere visibili all'interno di un'altra classe. La convenzione è quella di renderle pubbliche al livello superiore del loro package.

G33: Incapsulate le condizioni di delimitazione

Le condizioni di delimitazione sono difficili da considerare. Collocate la loro elaborazione in un solo luogo. Non “sparpagliatele” in tutto il codice. Non vogliamo vedere nugoli di `+1` e `-1` sparsi qua e là. Considerate questo semplice esempio tratto da FIT:

```
if(level + 1 < tags.length)
{
    parts = new Parse(body, tags, level + 1, offset + endTag);
    body = null;
}
```

Notate che `level + 1` compare per due volte. Questa è una condizione di delimitazione che dovrebbe essere incapsulata all'interno di una variabile chiamata, per esempio, `nextLevel`.

```
int nextLevel = level + 1;
if(nextLevel < tags.length)
{
    parts = new Parse(body, tags, nextLevel, offset + endTag);
    body = null;
}
```

G34: Le funzioni devono discendere un solo livello di astrazione

Le istruzioni collocate all'interno di una funzione dovrebbero tutte trovarsi allo stesso livello di astrazione, il quale dovrebbe essere

esattamente un livello sotto l'operazione descritta dal nome della funzione. Questa può essere una delle euristiche più difficili da interpretare e seguire. Sebbene l'idea in sé sia semplice, gli esseri umani sono eccezionali nel mescolare più livelli di astrazione.

Considerate, per esempio, il seguente codice tratto da FitNesse:

```
public String render() throws Exception
{
    StringBuffer html = new StringBuffer("<hr");
    if(size > 0)
        html.append(" size=""").append(size + 1).append("")");
    html.append(">");
    return html.toString();
}
```

Bastano pochi istanti per vedere che cosa sta accadendo. Questa funzione costruisce il tag HTML che traccia una linea orizzontale nella pagina. L'altezza di tale linea è specificata nella variabile `size`.

Osservatela ancora una volta. Questo metodo mescola quanto meno due livelli di astrazione. Il primo è il concetto che una linea orizzontale abbia uno spessore, `size`. Il secondo è la sintassi del tag HR. Questo codice è tratto dal modulo `HruleWidget` di FitNesse. Tale modulo rileva la presenza di una fila di quattro o più trattini e la converte in un tag HR appropriato. Maggiore è il numero di trattini, maggiore è lo spessore.

Ho eseguito il refactoring di questo frammento di codice come segue. Notate che ho cambiato il nome del campo `size` in modo che riflettesse il suo reale scopo. Ora contiene il numero di trattini aggiuntivi.

```
public String render() throws Exception
{
    HtmlTag hr = new HtmlTag("hr");
    if (extraDashes > 0)
        hr.addAttribute("size", hrSize(extraDashes));
    return hr.html();
}
private String hrSize(int height)
{
    int hrSize = height + 1;
    return String.format("%d", hrSize);
}
```

Questa modifica separa i due livelli di astrazione. La funzione `render` costruisce semplicemente un tag HR, senza essere costretta a conoscere la sintassi HTML di tale tag. È il modulo `HtmlTag` a occuparsi di tutti i dettagli sintattici.

In effetti, apportando questa modifica mi sono accorto di un errore, subdolo. Il codice originale non inseriva lo slash di chiusura nel tag HR, come previsto dallo standard XHTML (in altre parole, emetteva un `<hr>` invece di un `<hr/>`). Il modulo `HtmlTag` era stato modificato per conformarsi allo standard XHTML molto tempo fa.

Separare i livelli di astrazione è uno dei compiti più importanti del refactoring ed è anche uno dei più difficili. Per esempio, osservate il codice sottostante. Questo è stato il mio primo tentativo di separare i livelli di astrazione nel metodo `HruleWidget.render`.

```
public String render() throws Exception
{
    HtmlTag hr = new HtmlTag("hr");
    if (size > 0) {
        hr.addAttribute("size", ""+(size+1));
    }
    return hr.html();
}
```

Il mio obiettivo, a questo punto, era di creare la necessaria separazione e di passare i test. Vi sono riuscito con facilità, ma il risultato è stata una funzione che continuava a contenere più livelli di astrazione. In questo caso tali livelli erano la costruzione del tag HR e l'interpretazione e formattazione della variabile `size`. Questo evidenzia che quando si spezza una funzione in base ai vari livelli di astrazione, spesso si scoprono nuovi livelli di astrazione che erano nascosti dalla struttura precedente.

G35: Mantenete ai livelli più elevati i dati configurabili

Se avete una costante, come un default o un valore di configurazione, la quale è nota e prevista a un livello di astrazione superiore, non seppellitela in una funzione di basso livello. Esponetela invece come argomento di tale funzione di basso livello quando viene richiamata dalla funzione di alto livello. Considerate il seguente codice, tratto da FitNesse:

```
public static void main(String[] args) throws Exception
{
    Arguments arguments = parseCommandLine(args);
    ...
}
public class Arguments
{
    public static final String DEFAULT_PATH = ".";
    public static final String DEFAULT_ROOT = "FitNesseRoot";
    public static final int DEFAULT_PORT = 80;
    public static final int DEFAULT_VERSION_DAYS = 14;
    ...
}
```

Gli argomenti inviati dalla riga di comando sono sottoposti a parsing nella primissima riga di codice eseguibile di FitNesse. I valori di default di questi argomenti sono specificati in cima alla classe `Argument`. Non dovete quindi andare a cercare nei livelli più bassi del sistema istruzioni come la seguente:

```
if (arguments.port == 0) // usa 80 come default
```

Le costanti di configurazione sono collocate al livello superiore e sono facili da modificare. Da lì vengono diffuse al resto dell'applicazione. I livelli più bassi dell'applicazione non saranno quindi proprietari del valore di queste costanti.

G36: Evitate la navigazione transitiva

In generale non vogliamo che un unico modulo sappia molto dei suoi collaboratori. Più precisamente, se A collabora con B e B collabora con C, non vogliamo che i moduli che usano A sappiano qualcosa di C.

Per esempio, non vogliamo avere qualcosa come

```
a.getB().getc().doSomething();
```

Questa è chiamata anche Legge di Demetra. In *Pragmatic Programmers* si parla di “scrivere codice timido” ([PRAG], p. 138). In ogni caso, significa assicurarsi che i moduli conoscano solo i loro collaboratori immediati e non siano a conoscenza della mappa di navigazione dell’intero sistema.

Se più moduli usassero qualche forma dell’istruzione `a.getB().getc()`, sarebbe difficile modificare la struttura e l’architettura per interporre un Q fra B e C. Sareste costretti a trovare ogni istanza di `a.getB().getc()` per convertirla in `a.getB().getQ().getC()`. Questo irrigidisce l’architettura.

Troppi moduli sono troppo a conoscenza della struttura dell’architettura.

Piuttosto vogliamo che i nostri immediati collaboratori ci offrano tutti i servizi di cui abbiamo bisogno. Non dovremmo vagare nel grafico degli oggetti del sistema, alla ricerca del metodo che vogliamo richiamare. Piuttosto dovremmo semplicemente poter dire:

```
myCollaborator.doSomething();
```

Java

J1: Evitate i lunghi elenchi di import; usate i caratteri jolly

Se usate due o più classi di un package, importate l’intero package con:

```
import package.*;
```

I lunghi elenchi di `import` sono una pena da leggere. Non vogliamo congestionare la cima dei nostri moduli con ottanta righe di `import`.

Piuttosto vogliamo che le `import` si trasformino in un’istruzione concisa che descriva i package dei quali ci serviamo.

Le `import` specifiche sono dipendenze rigide, mentre quelle a caratteri jolly no. Se importate una specifica classe, tale classe deve esistere. Ma se importate un package con un carattere jolly, nessuna classe è obbligata a esistere. L’istruzione `import` aggiunge semplicemente il package al percorso di ricerca usato nel momento in cui si cercano i nomi. Pertanto non viene creata alcuna vera dipendenza e ciò riduce gli accoppiamenti fra moduli.

Esistono situazioni in cui il lungo elenco di `import` può essere utile. Per esempio, se usate del codice “datato” e volete scoprire per quali classi avete bisogno di realizzare sistemi di collegamento, potete scorrere l’elenco delle `import` specifiche per scoprire il nome completo di tutte quelle classi e poi predisporre i collegamenti appropriati. Tuttavia, questo impiego di `import` specifiche è molto raro. Inoltre, la maggior parte dei nuovi ambienti IDE vi consentirà di convertire le `import` a caratteri jolly in un elenco di `import` specifiche con un unico comando. Pertanto anche nel caso del codice datato è meglio eseguire l’importazione tramite caratteri jolly.

Le importazioni con caratteri jolly talvolta possono provocare conflitti e ambiguità. Due classi con lo stesso nome, ma in package differenti, dovranno essere importate in modo più preciso o quanto meno dovranno essere qualificate completamente. Questo può essere fastidioso, ma è comunque raro: meglio quindi usare sempre importazioni a caratteri jolly.

J2: Non ereditate le costanti

L’ho visto parecchie volte e ogni volta mi fa sobbalzare. Un programmatore inserisce alcune costanti in un’interfaccia e poi ottiene

l'accesso a tali costanti ereditando tale interfaccia. Date un'occhiata al seguente codice:

```
public class HourlyEmployee extends Employee {  
    private int tenthsWorked;  
    private double hourlyRate;  
    public Money calculatePay() {  
        int straightTime = Math.min(tenthsWorked, TENTHS_PER_WEEK);  
        int overtime = tenthsWorked - straightTime;  
        return new Money(  
            hourlyRate * (tenthsWorked + OVERTIME_RATE * overtime)  
        );  
    }  
    ...  
}
```

Da dove arrivano le costanti `TENTHS_PER_WEEK` e `OVERTIME_RATE`? Potrebbero provenire dalla classe `Employee`; vediamo come è fatta:

```
public abstract class Employee implements PayrollConstants {  
    public abstract boolean isPayday();  
    public abstract Money calculatePay();  
    public abstract void deliverPay(Money pay);  
}
```

No, qui non c'è. Ma allora dov'è? Osserviamo attentamente la classe `Employee`. Essa implementa `PayrollConstants`.

```
public interface PayrollConstants {  
    public static final int TENTHS_PER_WEEK = 400;  
    public static final double OVERTIME_RATE = 1.5;  
}
```

Questa è una pratica perversa! Le costanti vengono nascoste proprio in cima alla gerarchia di ereditarietà. Ma accidenti! Non usate mai l'ereditarietà come un modo per aggirare le regole di visibilità del linguaggio. Usate piuttosto un'import statica.

```
import static PayrollConstants.*;  
public class HourlyEmployee extends Employee {  
    private int tenthsWorked;  
    private double hourlyRate;  
    public Money calculatePay() {  
        int straightTime = Math.min(tenthsWorked, TENTHS_PER_WEEK);  
        int overtime = tenthsWorked - straightTime;  
        return new Money(  
            hourlyRate * (tenthsWorked + OVERTIME_RATE * overtime)  
        );  
    }  
    ...  
}
```

```
}
```

J3: Costanti o enum?

Ora che le `enum` sono state aggiunte al linguaggio (Java 5), usatele!

Abbandonate il vecchio trucco dei `public static final int`. Il significato di quegli `int` si può perdere. Il significato delle `enum` no, perché esse appartengono a un'enumerazione con nome.

Ma fate di più: studiate con attenzione la sintassi delle `enum`. Esse possono avere metodi e campi. Questo le rende strumenti molto potenti e dotati di molta più espressività e flessibilità degli `int`. Considerate questa variante del codice delle paghe:

```
public class HourlyEmployee extends Employee {
    private int tenthsWorked;
    HourlyPayGrade grade;
    public Money calculatePay() {
        int straightTime = Math.min(tenthsWorked, TENTHS_PER_WEEK);
        int overtime = tenthsWorked - straightTime;
        return new Money(
            grade.rate() * (tenthsWorked + OVERTIME_RATE * overtime)
        );
    }
    ...
}
public enum HourlyPayGrade {
    APPRENTICE {
        public double rate() {
            return 1.0;
        }
    },
    LEUTENANT_JOURNEYMAN {
        public double rate() {
            return 1.2;
        }
    },
    JOURNEYMAN {
        public double rate() {
            return 1.5;
        }
    },
    MASTER {
        public double rate() {
            return 2.0;
        }
    }
}
public abstract double rate();
```

Nomi

N1: Scegliete nomi descrittivi

Non siate frettolosi nella scelta di un nome. Assicuratevi che sia descrittivo. Ricordate che il significato tende a sparire mentre il software evolve, pertanto ri-valutate frequentemente l'appropriatezza dei nomi che avevate scelto.

Questa non è una semplice raccomandazione da prendere alla “sì, sì, ok, lo so...”. Nel software, i nomi contribuiscono per il 90 percento alla leggibilità del codice. Pertanto sceglieteli sempre con grande cura e anche a mantenerne il significato nel tempo. I nomi sono troppo importanti per essere trattati con superficialità.

Considerate il codice sottostante. Che cosa fa? Se vi mostrassi il codice con dei nomi appropriati, il suo scopo vi risulterebbe perfettamente chiaro, ma così com'è sembra un ammasso di simboli e numeri magici.

```
public int x() {  
    int q = 0;  
    int z = 0;  
    for (int kk = 0; kk < 10; kk++) {  
        if (l[z] == 10)  
        {  
            q += 10 + (l[z + 1] + l[z + 2]);  
            z += 1;  
        }  
        else if (l[z] + l[z + 1] == 10)  
        {  
            q += 10 + l[z + 2];  
            z += 2;  
        } else {  
            q += l[z] + l[z + 1];  
            z += 2;  
        }  
    }  
    return q;  
}
```

Ecco il codice come dovrebbe essere scritto. Questo frammento di codice in realtà è meno completo di quello sopra. Tuttavia potete capire

immediatamente che cosa sta cercando di fare e molto probabilmente potreste scrivere con facilità le funzioni mancanti sulla base del significato che ormai avete afferrato. I numeri magici non ci sono più e la struttura dell'algoritmo è perfettamente descrittiva.

```
public int score() {  
    int score = 0;  
    int frame = 0;  
    for (int frameNumber = 0; frameNumber < 10; frameNumber++) {  
        if (isStrike(frame)) {  
            score += 10 + nextTwoBallsForStrike(frame);  
            frame += 1;  
        } else if (isSpare(frame)) {  
            score += 10 + nextBallForSpare(frame);  
            frame += 2;  
        } else {  
            score += twoBallsInFrame(frame);  
            frame += 2;  
        }  
    }  
    return score;  
}
```

La potenza di nomi ben selezionati consiste nel fatto che si aggancia la descrizione alla struttura stessa del codice. Coloro che leggeranno il codice capiranno subito che cosa fanno le altre funzioni del modulo. L'implementazione di `isStrike()` è immediata, semplicemente osservando il codice precedente. Ora che leggerete il metodo `isStrike`, potrete dire “Sì, fa esattamente quello che mi aspettavo” (come direbbe Ward Cunningham, vedi la sua citazione nel Capitolo 1).

```
private boolean isStrike(int frame) {  
    return rolls[frame] == 10;  
}
```

N2: Scegliete nomi collocati al corretto livello di astrazione

Non scegliete nomi che comunichino l'implementazione; scegliete dei nomi che riflettano il livello di astrazione della classe o della funzione nella quale state lavorando. Questo può essere difficile. Di nuovo, per noi esseri umani è facilissimo lavorare a più livelli di astrazione. Ogni

volta che date un'occhiata al vostro codice, probabilmente vi troverete qualche variabile la cui denominazione è a un livello troppo basso. Dovreste cogliere l'opportunità di cambiare tali nomi quando li troverete. La leggibilità del codice richiede una dedizione continua. Considerate l'interfaccia `Modem` seguente:

```
public interface Modem {  
    boolean dial(String phoneNumber);  
    boolean disconnect();  
    boolean send(char c);  
    char recv();  
    String getConnectedPhoneNumber();  
}
```

All'inizio tutto bene. Le funzioni sembrano appropriate. In effetti, per molte applicazioni lo sono. Ma ora considerate un'applicazione in cui alcuni modem non si connettono componendo il numero, ma sono connessi in modo permanente da cavi (come i modem che oggi forniscono la connettività a Internet). Magari qualcuno si connette inviando un numero di porta a uno switch su una connessione USB. Chiaramente il concetto di numero telefonico si trova al livello di astrazione errato. Una strategia di denominazione migliore potrebbe essere:

```
public interface Modem {  
    boolean connect(String connectionLocator);  
    boolean disconnect();  
    boolean send(char c);  
    char recv();  
    String getConnectedLocator();  
}
```

Ora i nomi non parlano di numeri telefonici e possono essere usati per i numeri telefonici o per qualsiasi altro tipo di connessione.

N3: Usate la nomenclatura standard, se possibile

I nomi sono più facili da comprendere se si basano su una convenzione o una prassi. Per esempio, se usate il pattern DECORATOR, dovreste impiegare la parola `Decorator` nel nome delle

classi di “decorazione”. Per esempio, `AutoHangupModemDecorator` potrebbe essere il nome di una classe che decora una classe `Modem` con la capacità di aggangiare automaticamente la linea alla fine di una sessione.

I pattern sono solo uno degli standard. In Java, per esempio, le funzioni che convertono degli oggetti in stringhe spesso si chiamano `toString`. È meglio seguire queste convenzioni invece di reinventarne una.

Spesso i team sviluppano un proprio standard di denominazione per un determinato progetto. Eric Evans lo chiama linguaggio ubiquitario per il progetto [DDD]. Il vostro codice dovrà impiegare in modo pervasivo i termini di questo “linguaggio”. In breve, più potete usare nomi che nel tempo si sono caricati di un significato preciso nel progetto, più facile sarà per coloro che leggono il codice capire qual è il suo significato.

N4: Nomi non ambigui

Scegliete nomi che non introducano ambiguità nel comportamento di una funzione o di una variabile. Considerate questo esempio tratto da FitNesse:

```
private String doRename() throws Exception
{
    if(refactorReferences)
        renameReferences();
    renamePage();
    pathToRename.removeNameFromEnd();
    pathToRename.addNameToEnd(newName);
    return PathParser.render(pathToRename);
}
```

Il nome di questa funzione non dice che cosa fa la funzione, se non in termini vaghi. Questo è enfatizzato dal fatto che vi è una funzione chiamata `renamePage` all'interno della funzione `doRename`! Che cosa vi dicono questi due nomi sulla differenza fra le due funzioni? Nulla!

Un nome migliore per tale funzione è

`renamePageAndOptionallyAllReferences`. Può sembrare lungo, e lo è, ma viene impiegato in un solo punto del modulo, e così il suo valore descrittivo ne compensa la lunghezza.

N5: Usate nomi lunghi per grandi campi di visibilità (scope)

La lunghezza di un nome dovrebbe essere correlata all'estensione del livello di visibilità (scope). Potete usare nomi molto corti per le variabili con scope molto limitato, ma se la variabile risulta ampiamente visibile, dovreste usare nomi lunghi.

Nomi di variabili come `i` e `j` vanno bene se la visibilità si estende solo per cinque righe. Considerate questo frammento di codice dal vecchio “Bowling Game”:

```
private void rollMany(int n, int pins)
{
    for (int i=0; i<n; i++)
        g.roll(pins);
}
```

Il significato è perfettamente chiaro e sarebbe solo peggiorato dal fatto che al posto della variabile `i` venisse impiegato qualcosa di più prolioso e prosaico come `rollCount`. D'altra parte, le variabili e le funzioni dal nome breve perdono significato all'aumentare della distanza. Pertanto, più esteso è il livello di visibilità (scope) del nome, più lungo e preciso deve essere il nome.

N6: Evitate le codifiche

I nomi non dovrebbero contenere una codifica del tipo o della visibilità. Prefissi come `m_` o `f` sono ormai inutili al giorno d'oggi. Anche le codifiche del progetto e/o del sottosistema, come `vis_` (per

Visual Imaging System) sono ridondanti e fonte di distrazione. Di nuovo, gli ambienti odierni forniscono tutte queste informazioni, senza deturpare i nomi. Proteggete i vostri nomi dall’“inquinamento ungherese”.

N7: I nomi dovrebbero descrivere anche gli effetti collaterali

I nomi dovrebbero descrivere tutto quello che una funzione, una variabile o una classe è o fa. Non nascondete gli effetti collaterali dal nome. Non usate un semplice verbo per descrivere una funzione che svolga più di un’unica azione. Per esempio, considerate questo codice tratto da TestNG:

```
public ObjectOutputStream getOos() throws IOException {
    if (m_oos == null) {
        m_oos = new ObjectOutputStream(m_socket.getOutputStream());
    }
    return m_oos;
}
```

Questa funzione fa qualcosa di più di quello che dice: crea anche gli “`oos`”, se non sono stati creati. Pertanto, un nome migliore potrebbe essere `createOrReturnOos`.

Test

T1: Test insufficienti

Quanti test dovrebbero trovarsi in un pacchetto di test? Sfortunatamente, la metrica di molti programmati sembra essere “QB, quanto basta”. Un pacchetto di test deve contenere tutto il necessario per sottoporre a test tutto quello che può non funzionare. I test sono

insufficienti ogni volta che ignorano anche solo un'unica condizione o che non convalidano tutti i calcoli.

T2: Usate uno strumento di copertura!

I report degli strumenti di copertura affiancano la vostra strategia di test. Aiutano a trovare con facilità i moduli, le classi e le funzioni che non sono sottoposti a sufficienti test. La maggior parte degli IDE fornisce un'indicazione visuale, contrassegnando in verde le righe “coperte” e in rosso quelle “scoperte”. Questo aiuta a trovare con facilità le istruzioni `if` o `catch` il cui contenuto non viene considerato.

T3: Non saltate i test che considerate banali

Sono facili da scrivere e il loro valore documentario è maggiore dell'impegno necessario per realizzarli.

T4: Un test ignorato è un problema di ambiguità

Talvolta non siamo certi di un dettaglio del comportamento, perché i requisiti non sono chiari. Possiamo esprimere la nostra perplessità sui requisiti come un test trasformato in commento o come un test annotato con `@ignore`. Quale scegliere dipende dal fatto che l'ambiguità riguardi qualcosa che passerebbe o meno la compilazione.

T5: Verificate le condizioni di delimitazione

Fate particolare attenzione ai test delle condizioni di delimitazione. Spesso il corpo dell'algoritmo è corretto, ma ne sottovalutiamo il comportamento in prossimità delle delimitazioni.

T6: Applicate test esaustivi in prossimità dei bug

I bug tendono a concentrarsi. Quando trovate un bug in una funzione, è saggio svolgere un test esaustivo di tale funzione. Probabilmente scoprirete che il bug “aveva compagnia”.

T7: Considerate quando i problemi sembrano far pensare a uno schema

Talvolta potete diagnosticare un problema ricercando un eventuale schema nel fallimento dei casi di test. Questo è un motivo in più per far sì che i casi di test siano il più possibile completi. Casi di test completi e ben ordinati evidenziano tali schemi.

Come semplice esempio, supponete di aver notato che tutti i test con un input di più di cinque caratteri falliscono? O magari possono fallire i test che passano un numero negativo nel secondo argomento di una funzione. Talvolta anche la sola sequenza di rossi e verdi nel report dei test basta a suggerire un “Aha!” che indirizza verso la soluzione. Nel Capitolo 16 trovate un interessante esempio con riferimento a `serialDate`.

T8: I risultati dei test di copertura possono essere rivelatori

Osservare il codice che viene o non viene eseguito fornisce indizi sul fallimento dei test.

T9: I test dovrebbero essere veloci

Un test lento è un test che non verrà eseguito volentieri. Quando si è alle strette, i primi a “cadere” sono i test lenti. Pertanto fate tutto il possibile per curare la velocità dei vostri test.

Conclusioni

Questo elenco di euristiche e avvertenze non è affatto esaustivo. In effetti, penso che non possa esistere un elenco davvero completo. Ma forse l'obiettivo non dovrebbe essere la completezza, perché quello che fa questo elenco è stabilire un sistema di valori.

In effetti, tale sistema di valori è esattamente l'obiettivo e l'argomento dell'intero libro. Non si scrive codice pulito seguendo un semplice insieme di regole. Non diventerete “artigiani sviluppatori” imparando un semplice elenco di euristiche. La professionalità e l'abilità derivano dai valori che guidano la disciplina.

Bibliografia

- [Refactoring]: Martin Fowler et al., Refactoring: Improving the Design of Existing Code, Addison-Wesley, Boston 1999.
- [PRAG]: Andrew Hunt, Dave Thomas, The Pragmatic Programmer, Addison-Wesley, Boston 2000. Edizione italiana, *Il Pragmatic Programmer*, Apogeo, Milano 2018.
- [GOF]: Gamma et al., Design Patterns: Elements of Reusable Object Oriented Software, Addison-Wesley, Boston 1996.
- [Beck97]: Kent Beck, Smalltalk Best Practice Patterns, Prentice Hall, Upper Saddle River, New Jersey 1997.
- [Beck07]: Kent Beck, Implementation Patterns, Addison-Wesley, Boston 2008.
- [PPP]: Robert C. Martin, Agile Software Development: Principles, Patterns, and Practices, Prentice Hall, Upper Saddle River, New Jersey 2002.
- [DDD]: Eric Evans, Domain Driven Design, Addison-Wesley, Boston 2003.

Concorrenza II

di Brett L. Schuchert

Questa appendice affianca ed estende il Capitolo 13, “Concorrenza”. È scritta come una serie di argomenti indipendenti, che potete leggere in qualsiasi ordine. Troverete un po’ di duplicazione fra le sezioni, proprio per consentire anche questo tipo di lettura frammentaria.

Un esempio client/server

Immaginate una semplice applicazione client/server. Un server resta in ascolto su un socket di una richiesta di connessione da parte di un client. Un client si connette e invia una richiesta.

Il server

Ecco una versione semplificata di un’applicazione server. Il codice sorgente completo di questo esempio si trova più avanti, nel Listato A.3.

```
ServerSocket serverSocket = new ServerSocket(8009);
while (keepProcessing) {
    try {
        Socket socket = serverSocket.accept();
        process(socket);
    } catch (Exception e) {
        handle(e);
    }
}
```

Questa semplice applicazione attende una connessione, elabora un messaggio in arrivo e poi torna in attesa della successiva richiesta da parte di un client. Ecco ora il codice client che si connette a questo server:

```
private void connectSendReceive(int i) {
    try {
        Socket socket = new Socket("localhost", PORT);
        MessageUtils.sendMessage(socket, Integer.toString(i));
        MessageUtils.getMessage(socket);
        socket.close();
    } catch (Exception e) {
        e.printStackTrace();
    }
}
```

Come si comporta questa coppia client/server? Come possiamo descrivere formalmente tali prestazioni? Ecco un test che valuta se le prestazioni sono “accettabili”:

```
@Test(timeout = 10000)
public void shouldRunInUnder10Seconds() throws Exception {
    Thread[] threads = createThreads();
    startAllThreads(threads);
    waitForAllThreadsToFinish(threads);
}
```

Il setup è stato ignorato per semplificare l’esempio (vedi “ClientTest.java”, nel Listato A.4). Questo test valuta che il tutto si svolga entro 10.000 millisecondi (10 secondi).

Questo è un classico esempio di verifica della produttività (throughput) di un sistema. Questo sistema dovrà completare una serie di richieste client in dieci secondi. Fintantoché il server si dimostra in grado di elaborare ogni singola richiesta client in questo tempo, il test passa.

Che cosa accade se il test fallisce? A parte sviluppare un qualche tipo di ciclo di polling degli eventi, non è possibile fare molto in un unico thread per rendere più veloce questo codice. L’impiego di più thread risolverebbe il problema? Forse, ma dobbiamo scoprire dove trascorrono questi dieci secondi. Vi sono due possibilità:

- I/O - uso di un socket, connessione a un database, attesa dello swapping della memoria virtuale e così via;
- processore - calcoli numerici, elaborazione delle espressioni regolari, *garbage collection* e così via.

In genere, i sistemi fanno un po' di tutto questo, ma in una determinata operazione ce n'è una che tende a dominare. Se il codice carica prevalentemente il processore, si può migliorare il throughput facendo evolvere l'hardware, finché il nostro test non passa. Ma dato che tutto si basa sui cicli della CPU, aggiungendo più thread a un problema che fa pesante uso del processore, non si ottiene più velocità.

D'altra parte, se il processo mette alla prova l'I/O, allora la concorrenza può effettivamente incrementarne l'efficienza. Quando una parte del sistema è in attesa di un'operazione di I/O, un'altra parte può usare quello stesso tempo per elaborare qualcos'altro, facendo un uso più efficiente della CPU.

Aggiunta del threading

Supponete per un istante che il test prestazionale fallisca. Come possiamo migliorare il throughput in modo che il test delle prestazioni venga passato? Se il metodo `process` del server ha prevalenza di I/O, ecco un modo per far sì che il server usi i thread (basta modificare `processMessage`):

```
void process(final Socket socket) {
    if (socket == null)
        return;
    Runnable clientHandler = new Runnable() {
        public void run() {
            try {
                String message = MessageUtils.getMessage(socket);
                MessageUtils.sendMessage(socket, "Processed: " + message);
                closeIgnoringException(socket);
            } catch (Exception e) {
                e.printStackTrace();
            }
        }
    };
}
```

```
        Thread clientConnection = new Thread(clientHandler);
        clientConnection.start();
    }
```

Supponete che questa modifica faccia passare il test. Potete verificarlo da soli provando il codice prima e dopo. Esamineate il codice senza thread a partire dal Listato A.3 e il codice a thread dopo il Listato A.5. Il codice è completo, giusto?

Osservazioni sul server

Il nuovo server completa il test con successo in poco più di un secondo. Sfortunatamente, questa soluzione è un po' ingenua e introduce alcuni problemi.

Quanti thread può creare il nostro server? Il codice non impone limiti, pertanto potremmo anche raggiungere il limite imposto dalla Java Virtual Machine (JVM). Per molti sistemi semplici può anche andare. Ma se il sistema deve poter supportare una grande quantità di utenti su una rete pubblica? Se troppi utenti si dovessero connettere contemporaneamente, il sistema potrebbe bloccarsi.

Ma per il momento lasciamo da parte il problema legato al comportamento. La soluzione presentata ha problemi di pulizia e struttura. Quante responsabilità ha il codice del server?

- Gestione della connessione con il socket.
- Elaborazione delle richieste del client.
- Gestione dei thread.
- Gestione della chiusura del server.

Sfortunatamente, tutte queste responsabilità si trovano nella funzione `process`. Inoltre, il codice attraversa molti livelli di astrazione differenti. Pertanto, per quanto sia piccola, la funzione `process` deve essere partizionata.

Il server ha molti motivi per cambiare; innanzitutto viola il principio SRP (Single Responsibility Principle) della singola responsabilità. Per mantenere puliti i sistemi concorrenti, la gestione dei thread dovrebbe essere mantenuta solo in alcune posizioni ben controllate. Ma, in più, tutto il codice che gestisce i thread dovrebbe occuparsi solo della gestione dei thread. Perché? Come minimo perché risolvere i problemi introdotti dalla concorrenza è già abbastanza difficile senza dover considerare anche altri tipi di problemi.

Se creiamo una classe distinta per ognuna delle responsabilità appena elencate, compresa la gestione dei thread, allora quand'anche dovessimo modificare la strategia di gestione dei thread, la modifica avrà un impatto inferiore sul codice in generale e non coinvolgerà le altre responsabilità. Questo inoltre rende più facile sottoporre a test tutte le altre responsabilità senza preoccuparsi del threading. Ecco una versione aggiornata che fa esattamente questo:

```
public void run() {
    while (keepProcessing) {
        try {
            ClientConnection clientConnection = connectionManager.awaitClient();
            ClientRequestProcessor requestProcessor
                = new ClientRequestProcessor(clientConnection);
            clientScheduler.schedule(requestProcessor);
        } catch (Exception e) {
            e.printStackTrace();
        }
    }
    connectionManager.shutdown();
}
```

Questa versione ha il pregio di concentrare tutto ciò che riguarda i thread in un solo posto, `clientScheduler`. Se avremo problemi con la concorrenza, avremo un solo posto in cui guardare:

```
public interface ClientScheduler {
    void schedule(ClientRequestProcessor requestProcessor);
}
```

L'attuale politica è facile da implementare:

```
public class ThreadPerRequestScheduler implements ClientScheduler {
    public void schedule(final ClientRequestProcessor requestProcessor) {
        Runnable runnable = new Runnable() {
```

```

        public void run() {
            requestProcessor.process();
        };
    }
    Thread thread = new Thread(runnable);
    thread.start();
}
}

```

Avendo isolato tutta la gestione dei thread in un'unica posizione, ora è molto più facile cambiare il tipo di funzionamento dei thread. Per esempio, il passaggio al framework Executor di Java 5 prevede la scrittura e connessione di una nuova classe (Listato A.1).

Listato A.1 ExecutorClientScheduler.java.

```

import java.util.concurrent.Executor;
import java.util.concurrent.Executors;
public class ExecutorClientScheduler implements ClientScheduler {
    Executor executor;
    public ExecutorClientScheduler(int availableThreads) {
        executor = Executors.newFixedThreadPool(availableThreads);
    }
    public void schedule(final ClientRequestProcessor requestProcessor) {
        Runnable runnable = new Runnable() {
            public void run() {
                requestProcessor.process();
            }
        };
        executor.execute(runnable);
    }
}

```

Conclusioni

L'introduzione della concorrenza in questo specifico esempio mostra un modo per migliorare il throughput di un sistema e anche un modo per convalidare tale throughput tramite un framework di test. Il fatto di concentrare tutto il codice di gestione della concorrenza in un piccolo numero di classi è un esempio di applicazione del principio SRP (*Single Responsibility Principle*). Nel caso della programmazione concorrente, questo è particolarmente importante a causa della sua complessità.

Possibili percorsi di esecuzione

Esaminiamo il metodo `incrementValue`, un metodo Java di una riga senza cicli o altro:

```
public class IdGenerator {  
    int lastIdUsed;  
    public int incrementValue() {  
        return ++lastIdUsed;  
    }  
}
```

Ignorate l'overflow degli interi e supponete che vi sia un solo thread che ha accesso a un'unica istanza di `IdGenerator`. In questo caso vi è un unico percorso di esecuzione e un unico risultato garantito:

- Il valore restituito è uguale al valore di `lastIdUsed` ed è uguale a un'unità in più rispetto a prima della chiamata del metodo.

Che cosa accade se usiamo due thread e non modifichiamo il metodo? Quali sono i risultati possibili se ogni thread richiama `incrementValue` una sola volta? Quanti possibili percorsi di esecuzione vi sono? Innanzitutto, i risultati (supponendo che `lastIdUsed` parta dal valore 93).

- Il thread 1 ottiene il valore 94, il thread 2 ottiene il valore 95 e alla fine `lastIdUsed` vale 95.
- Il thread 1 ottiene il valore 95, il thread 2 ottiene il valore 94 e alla fine `lastIdUsed` vale 95.
- Il thread 1 ottiene il valore 94, il thread 2 ottiene il valore 94 e alla fine `lastIdUsed` vale 94.

L'ultimo risultato, anche se può sorprendere, è possibile. Per comprendere come siano possibili tutti questi risultati, dobbiamo comprendere il numero di percorsi di esecuzione possibili e il modo in cui vengono gestiti dalla Java Virtual Machine.

Numero di percorsi

Per calcolare il numero di percorsi di esecuzione possibili, partiamo dal bytecode generato. L'unica riga di Java (`return ++lastIdUsed;`) viene tradotta in otto istruzioni bytecode. È possibile che i due thread capitino in mezzo a queste otto istruzioni, come farebbero due mazzi di otto carte da mescolare. (Questa è una semplificazione. Tuttavia, ai fini di questa discussione, possiamo usare questo tipo di semplificazione.) Anche con solo otto carte in ogni mano, vi sono molte possibili combinazioni di carte dei due mazzi.

Per questo semplice caso di N istruzioni in sequenza, senza cicli o costrutti condizionali, e T thread, il numero totale di percorsi di esecuzione possibili è uguale a

$$(NT)! / N!^T$$

Calcolo delle combinazioni possibili

Questo testo è tratto da un'email da Uncle Bob a Brett.

Con N passi e T thread vi sono $T * N$ passi totali. Prima di ogni passo vi è un cambio di contesto che sceglie fra i T thread. Ogni percorso può pertanto essere rappresentato come una stringa di cifre che denota i cambi di contesto. Dati i passi A e B e i thread 1 e 2, i sei percorsi possibili sono 1122, 1212, 1221, 2112, 2121 e 2211. O, in termini di passi, si tratta di A1B1A2B2, A1A2B1B2, A1A2B2B1, A2A1B1B2, A2A1B2B1 e A2B2A1B1. Per tre thread la sequenza è 112233, 112323, 113223, 113232, 112233, 121233, 121323, 121332, 123132, 123123, ...

Una caratteristica di queste stringhe è che vi devono sempre essere N istanze di ogni T . Pertanto, la stringa 111111 non è valida perché ha sei istanze di 1 e zero istanze di 2 e 3.

Pertanto vogliamo le permutazioni di N 1, N 2, ... e N T . In pratica si tratta delle permutazioni di $N * T$ elementi presi $N * T$ alla volta, ovvero $(N * T)!$, ma eliminando tutti i duplicati. Pertanto il trucco consiste nel contare i duplicati e sottrarli da $(N * T)!$.

Dati due passi e due thread, quanti duplicati ci sono? Ogni stringa di quattro cifre ha due 1 e due 2. Ognuna di queste coppie potrebbe essere scambiata senza modificare il senso della stringa. Potete scambiare gli 1 o i 2, entrambi o nessuno. Pertanto vi sono quattro isomorfi per ogni stringa, il che significa che vi sono tre duplicati. Pertanto tre opzioni su quattro sono duplicate; alternativamente si può dire

che una permutazione su quattro NON è duplicata. $4! * .25 = 6$. Pertanto questo ragionamento sembra funzionare.

Quanti duplicati vi sono? Nel caso in cui $N = 2$ e $T = 2$, si possono scambiare gli 1, i 2 o entrambi. Nel caso in cui $N = 2$ e $T = 3$, si possono scambiare gli 1, i 2, i 3, gli 1 e i 2, gli 1 e i 3 o i 2 e i 3. Gli scambi sono le permutazioni di N . Immaginiamo che vi siano P permutazioni di N . Il numero di modi differenti per disporre tali permutazioni è $P^{**}T$.

Pertanto il numero di possibili isomorfi è $N!^{**}T$. E così il numero di percorsi è $(T^N)!/(N!^{**}T)$. Di nuovo, nel nostro caso, con $T = 2$ e $N = 2$ abbiamo 6, ovvero $24/4$.

Per $N = 2$ e $T = 3$ otteniamo $720/8 = 90$.

Per $N = 3$ e $T = 3$ otteniamo $9!/6A3 = 1680$.

Per il nostro semplice caso di una sola riga di codice Java, che produce otto righe di bytecode, e due thread, il numero totale di possibili percorsi di esecuzione è pari a 12.870. Se il tipo di `lastIdUsed` è un `long`, allora ogni operazione di lettura e scrittura genera due operazioni anziché una e il numero di possibili ordinamenti diviene 2.704.156.

Che cosa accade se modifichiamo questo metodo?

```
public synchronized void incrementValue() {  
    ++lastIdUsed;  
}
```

Il numero di possibili percorsi di esecuzione diviene 2 per due thread e $N!$ nel caso generale.

Approfondimento

Che cosa dire del sorprendente risultato di due thread, che richiamano entrambi il metodo una sola volta (prima di aggiungere `synchronized`) e che ottengono lo stesso risultato numerico? Come è possibile? Partiamo dall'inizio.

Che cos'è un'operazione atomica? Possiamo definire atomica un'operazione non interrompibile. Per esempio, nel seguente codice, la riga 5, dove a `lastId` viene assegnato il valore 0, è atomica, perché

secondo il modello di memoria Java, l’assegnamento a un valore a 32 bit non è interrompibile.

```
01: public class Example {  
02:     int lastId;  
03:  
04:     public void resetId() {  
05:         value = 0;  
06:     }  
07:  
08:     public int getNextId() {  
09:         ++value;  
10:    }  
  
11: }
```

Che cosa accade se modifichiamo il tipo di `lastId` da `int` a `long`? La riga 5 rimane atomica? No, secondo le specifiche della JVM. Potrebbe essere atomica su un determinato processore, ma secondo le specifiche della JVM, un assegnamento a un valore a 64 bit richiede due assegnamenti a 32 bit. Ciò significa che fra il primo e il secondo assegnamento a 32 bit, qualche altro thread potrebbe “imbucarsi” e modificare uno dei valori.

E che dire dell’operatore di pre-incremento, `++`, alla riga 9? L’operatore di pre-incremento può essere interrotto, pertanto non è atomico. Per comprendere il perché, riesaminiamo in dettaglio il bytecode di entrambi questi metodi.

Prima di procedere, ecco tre definizioni che saranno importanti.

- *Frame* - Ogni chiamata a metodo richiede un frame. Il frame include l’indirizzo di rientro, eventuali parametri passati al metodo e le variabili locali definite nel metodo. Questa è una tecnica standard usata per definire lo stack delle chiamate, usata dai linguaggi di oggi per il meccanismo di chiamata a funzione/metodo e per consentire il funzionamento delle chiamate ricorsive.
- *Variabile locale* - Ogni variabile definita nel livello di visibilità (scope) del metodo. Tutti i metodi non statici hanno quanto meno

una variabile, `this`, che rappresenta l'oggetto corrente, l'oggetto che riceve il messaggio più recente (nel thread corrente), che ha causato la chiamata a metodo.

- *Stack degli operandi* - Molte delle istruzioni della Java Virtual Machine richiedono parametri. Lo stack degli operandi è il luogo in cui vengono collocati tali parametri. Lo stack opera in modo LIFO (*last-in, first-out*).

Ecco di seguito il byte-code generato per `resetId()`.

Codice mnemonico	Descrizione	Stato degli operandi (dopo)
<code>ALOAD 0</code>	Carica la 0-esima variabile nello stack degli operandi. Che cosa si intende per 0-esima variabile? È <code>this</code> , l'oggetto corrente. Quando è stato richiamato il metodo, nell'array delle variabili locali del frame creato per la chiamata al metodo è stato inserito il destinatario del messaggio, un'istanza di <code>Example</code> . Questa è sempre la prima delle variabili inserite in ogni metodo di istanza.	<code>this</code>
<code>ICONST_0</code>	Inserisce il valore costante 0 nello stack degli operandi.	<code>this, 0</code>
<code>PUTFIELD lastId</code>	Memorizza il valore superiore presente nello stack (in questo caso 0) nel valore del campo dell'oggetto indicato dal riferimento all'oggetto che si trova a una posizione di distanza dalla cima dello stack, ovvero <code>this</code> .	<vuoto>

Queste tre istruzioni hanno la garanzia di atomicità, perché sebbene il thread che le esegue possa essere interrotto dopo ognuna di esse, le informazioni per l'istruzione `PUTFIELD` (il valore costante 0 in cima allo stack e il riferimento a `this` appena sotto la cima, insieme al valore del campo) non possono essere toccate da nessun altro thread. Pertanto quando si verifica l'assegnamento, abbiamo la garanzia che il valore 0 sarà memorizzato nel valore del campo. L'operazione è atomica. Gli operandi operano sulle informazioni locali del metodo, pertanto non vi è alcuna interferenza fra più thread.

Pertanto se queste tre istruzioni vengono eseguite da dieci thread, vi sono $4,38679733629^{24}$ possibili ordinamenti. Tuttavia vi è un solo risultato possibile, pertanto i numerosi ordinamenti sono irrilevanti. Inoltre, lo stesso risultato è garantito anche per i `long`, in questo caso. Perché? Tutti i dieci thread assegnano un valore costante. Anche se dovessero arrivare casualmente, il risultato finale sarebbe sempre lo stesso.

Con l'operazione `++` del metodo `getNextId`, invece, avremmo dei problemi. Supponete che `lastId` contenga il valore `42` all'inizio di questo metodo. Ecco di seguito il byte-code di questo nuovo metodo.

Codice mnemonico	Descrizione	Stato degli operandi (dopo)
ALOAD_0	Carica <code>this</code> nello stack degli operandi.	<code>this</code>
DUP	Copia ciò che si trova alla cima dello stack. Ora nello stack degli operandi abbiamo due copie di <code>this</code> .	<code>this, this</code>
GETFIELD <code>lastId</code>	Recupera il valore del campo <code>lastId</code> dall'oggetto puntato da ciò che si trova in cima allo stack (<code>this</code>) e memorizza tale valore di nuovo nello stack.	<code>this, 42</code>
ICONST_1	Inserisce nello stack la costante intera <code>1</code> .	<code>this, 42, 1</code>
IADD	Somma come interi i due valori superiori dello stack degli operandi e memorizza il risultato di nuovo nello stack degli operandi.	<code>this, 43</code>
DUP_X1	Duplica il valore <code>43</code> e lo pone prima di <code>this</code> .	<code>43, this, 43</code>
PUTFIELD <code>valore</code>	Memorizza il valore in cima allo stack degli operandi, <code>43</code> , nel valore del campo dell'oggetto corrente, rappresentato dal secondo valore (dalla cima dello stack) nello stack degli operandi, ovvero da <code>this</code> .	<code>43</code>
IRETURN	Restituisce il valore che si trova in cima allo stack, che ormai è anche l'unico.	<vuoto>

Immaginate il caso in cui il primo thread completi le prime tre istruzioni, fino a `GETFIELD`, compresa, e che poi venga interrotto. Un secondo thread arriva e svolge l'intero metodo, incrementando `lastid` di

₁; esso riceverà il risultato ₄₃. A questo punto il primo thread riprende da dove era rimasto; il ₄₂ è ancora nello stack degli operandi, perché quello era il valore di `lastID` quando aveva eseguito `GETFIELD`. Somma ₁ per ottenere ancora ₄₃ e memorizza il risultato. Il valore ₄₃ viene quindi restituito anche al primo thread. Il risultato è che uno degli incrementi si perde, perché il primo thread ha ceduto il passo al secondo, e poi ha continuato la sua esecuzione.

Il fatto di rendere `synchronized` il metodo `getNextID()` risolve questo problema.

Conclusioni

Non è necessario conoscere il byte-code per comprendere in quale modo più thread possono “mescolarsi”. Se siete riusciti a seguire questo semplice esempio, ora sapete che c’è la possibilità che l’esecuzione di più thread si accavalli, e questa è già una conoscenza sufficiente.

Detto questo, questo semplice esempio dimostra che è bene conoscere qualche dettaglio del funzionamento del modello di memoria per capire che cosa è o non è sicuro. Si ritiene erroneamente che l’operatore `++` (pre- o post-incremento) sia atomico, mentre non è così. Ciò significa che occorre sapere:

- dove vi sono oggetti/valori condivisi;
- quale codice può provocare problemi nelle operazioni di lettura/scrittura concorrenti della memoria;
- come evitare che si verifichino questi problemi legati alla concorrenza.

Studiare la libreria

Il framework Executor

Come abbiamo visto nel Listato A.1 (`ExecutorClientScheduler.java`), il framework `Executor`, introdotto in Java 5, offre dei meccanismi di esecuzione sofisticati grazie ai thread pool. Tutto si basa su una classe del package `java.util.concurrent`.

Se state creando dei thread e non usate un thread pool o ne avete realizzato uno voi stessi, dovreste considerare l'impiego di `Executor`: il vostro codice sarà più pulito, più facile da leggere e più compatto.

Il framework `Executor` crea dei pool di thread, ridimensionandoli e ricreandoli automaticamente se necessario. Supporta anche i `future`, un costrutto molto comune nella programmazione concorrente. Il framework `Executor` opera con classi che implementano l'interfaccia `Runnable` e anche con quelle che implementano l'interfaccia `Callable`. Una `Callable` ha l'aspetto di una `Runnable`, ma può restituire un risultato, una necessità molto comune nelle soluzioni multi-thread.

Un `future` è comodo quando il codice deve eseguire più operazioni indipendenti e deve attendere che esse terminino:

```
public String processRequest(String message) throws Exception {
    Callable<String> makeExternalCall = new Callable<String>() {
        public String call() throws Exception {
            String result = "";
            // richiesta esterna
            return result;
        }
    };
    Future<String> result = executorService.submit(makeExternalCall);
    String partialResult = doSomeLocalProcessing();
    return result.get() + partialResult;
}
```

In questo esempio, il metodo parte eseguendo l'oggetto `makeExternalCall`. Il metodo continua con altre elaborazioni. L'ultima riga richiama `result.get()`, che si blocca fino al completamento del `future`.

Soluzioni senza blocchi

La VM di Java 5 sfrutta l'architettura dei nuovi processori, che supporta aggiornamenti affidabili, senza bloccaggio. Considerate, per esempio, una classe che usi la sincronizzazione (e pertanto il bloccaggio) per fornire l'aggiornamento di un valore sicuro in termini di thread:

```
public class ObjectWithValue {  
    private int value;  
    public void synchronized incrementValue() { ++value; }  
    public int getValue() { return value; }  
}
```

Java 5 ha una serie di nuove classi per questo genere di situazioni: `AtomicBoolean`, `AtomicInteger` e `AtomicReference` sono solo tre degli esempi, ma ve ne sono molti altri. Possiamo riscrivere il codice precedente in modo che impieghi un approccio senza bloccaggio:

```
public class ObjectWithValue {  
    private AtomicInteger value = new AtomicInteger(0);  
    public void incrementValue() {  
        value.incrementAndGet();  
    }  
    public int getValue() {  
        return value.get();  
    }  
}
```

Anche se questa usa un oggetto invece di un valore primitivo e invii messaggi come `incrementAndGet()` invece di `++`, le prestazioni di questa classe superano quasi sempre la versione precedente. In alcuni casi solo di poco, ma praticamente mai la seconda versione è più lenta della prima.

Come è possibile? I nuovi processori offrono un'operazione, tipicamente chiamata *Compare and Swap* (CAS). Questa operazione è analoga al bloccaggio ottimistico dei database, mentre la versione sincronizzata è analoga al bloccaggio pessimistico.

La parola chiave `synchronized` acquisisce sempre un *lock*, anche quando non vi è un secondo thread a tentare di aggiornare lo stesso

valore. Anche se le prestazioni dei lock intrinseci sono migliorate di versione in versione, presentano comunque dei costi.

La versione senza bloccaggio parte presumendo che generalmente più thread non modifichino lo stesso valore con una frequenza tale da rappresentare un problema. Al contrario, rileva efficientemente se si è verificata tale situazione e riprova finché l'aggiornamento non si verifica con successo. Questo rilevamento è quasi sempre meno costoso dell'acquisizione di un lock, anche in situazioni di contesa moderate o elevate.

Come si comporta la JVM? L'operazione CAS è atomica. In termini logici, l'operazione CAS avrà all'incirca il seguente aspetto:

```
int variableBeingSet;
void simulateNonBlockingSet(int newValue) {
    int currentValue;
    do {
        currentValue = variableBeingSet
    } while(currentValue != compareAndSwap(currentValue, newValue));
}
int synchronized compareAndSwap(int currentValue, int newValue) {
    if(variableBeingSet == currentValue) {
        variableBeingSet = newValue;
        return currentValue;
    }
    return variableBeingSet;
}
```

Quando un metodo tenta di aggiornare una variabile condivisa, l'operazione CAS verifica che la variabile da impostare abbia ancora l'ultimo valore noto. In caso affermativo, la variabile viene modificata. In caso contrario, la variabile non viene impostata, perché nel frattempo è intervenuto un altro thread. Il metodo che effettua il tentativo (usando l'operazione CAS) vede che la modifica non è stata eseguita, e riprova.

Classi problematiche quanto ai thread

Vi sono determinate classi che, intrinsecamente, non sono *thread-safe*. Ecco alcuni esempi:

- `SimpleDateFormat`;
- connessioni a database;
- container in `java.util`;
- servlet.

Notate che alcune classi contengono singoli metodi *thread-safe*.

Tuttavia, ogni operazione che prevede la chiamata di più di un metodo non lo è. Per esempio, se non volete sostituire qualcosa in una `HashTable` perché è già presente, potreste scrivere il seguente codice:

```
if(!hashTable.containsKey(someKey)) {
    hashTable.put(someKey, new SomeValue());
}
```

Ogni singolo metodo è *thread-safe*. Tuttavia, un altro thread potrebbe aggiungere un valore fra le chiamate a `containsKey` e a `put`. Vi sono alcune possibilità per correggere questo problema.

- Bloccare prima la `HashTable` e assicurarsi che tutti gli altri utenti della `HashTable` facciano lo stesso (*client-based locking*):

```
synchronized(map) {
    if(!map.containsKey(key))
        map.put(key,value);
}
```

- Incorporare la `HashTable` in un proprio oggetto e usare un'altra API (*server-based locking* con un `ADAPTER`):

```
public class WrappedHashtable<K, V> {
    private Map<K, V> map = new Hashtable<K, V>();
    public synchronized void putIfAbsent(K key, V value) {
        if (map.containsKey(key)) map.put(key, value);
    }
}
```

- Usare le collection *thread-safe*:

```
ConcurrentHashMap<Integer, String> map = new ConcurrentHashMap<Integer, String>();
map.putIfAbsent(key, value);
```

Le collection in `java.util.concurrent` offrono operazioni come `putIfAbsent()` per consentire tali operazioni.

Le dipendenze fra i metodi possono pregiudicare il funzionamento del codice concorrente

Ecco un banale esempio di un modo per introdurre le dipendenze fra i metodi:

```
public class IntegerIterator implements Iterator<Integer>
{
    private Integer nextValue = 0;
    public synchronized boolean hasNext() {
        return nextValue < 100000;
    }
    public synchronized Integer next() {
        if (nextValue == 100000)
            throw new IteratorPastEndException();
        return nextValue++;
    }
    public synchronized Integer getNextValue() {
        return nextValue;
    }
}
```

Ecco ora del codice che impiega questo `IntegerIterator`:

```
IntegerIterator iterator = new IntegerIterator();
while(iterator.hasNext()) {
    int nextValue = iterator.next();
    //opera su nextValue
}
```

Se un thread esegue questo codice, non vi sarà alcun problema. Ma che cosa accade se due thread cercano di condividere un'unica istanza di `IntegerIterator` con lo scopo di fare in modo che ogni thread elabori i valori che riceve, ma che ogni elemento della lista venga elaborato una sola volta? La maggior parte delle volte, non si verifica niente di male; i thread condividono tranquillamente la lista, elaborando gli elementi ricevuti dall'iteratore e fermandosi quando l'iteratore è completato. Tuttavia, vi è una piccola probabilità che, alla fine dell'iterazione, i due thread interferiranno fra loro e facciano sì che un thread vada oltre la fine dell'iteratore e lanci un'eccezione.

Ecco il problema: il thread 1 pone la domanda `hasNext()`, che restituisce `true`. Il thread 1 subisce una prelazione e il thread 2 pone la stessa domanda, che è ancora `true`. Il thread 2 allora richiama `next()`, che restituisce un valore come previsto, ma ha l'effetto collaterale di far restituire `false` a `hasNext()`. Il thread 1 riprende, pensando che `hasNext()` sia ancora `true` e richiama `next()`. Anche se i singoli metodi sono sincronizzati, il client usa *due metodi*.

Questo è un vero problema e un esempio dei tipi di problemi che possono verificarsi nel codice concorrente. In questa specifica situazione, questo problema è particolarmente subdolo, perché l'unico caso in cui provoca guai è quando si verifica durante l'ultima iterazione. Se i thread hanno problemi proprio lì, uno di essi potrebbe andare oltre la fine dell'iteratore. Questo è un tipo di bug che si verifica molto dopo che il sistema è andato in produzione ed è difficile da individuare.

Avete tre possibilità.

- Tollerarlo.
- Risolvere il problema modificando il client (*bloccaggio basato sul client*).
- Risolvere il problema modificando il server, con eventuali modifiche al client (*bloccaggio basato sul server*).

Tollerare i problemi

Talvolta potete impostare le cose in modo che il problema non provochi guai. Per esempio, il client precedente è in grado di rilevare l'eccezione e di gestirla. Francamente, questo è un po' poco. È un po' come eseguire un reset della memoria con un reboot a mezzanotte.

Bloccaggio basato sul client

Per far funzionare correttamente `IntegerIterator` con più thread, potete modificare questo client (e ogni altro client) come segue:

```
IntegerIterator iterator = new IntegerIterator();
while (true) {
    int nextValue;
    synchronized (iterator) {
        if (!iterator.hasNext())
            break;
        nextValue = iterator.next();
    }
    doSometingWith(nextValue);
}
```

Ogni client introduce un lock tramite la parola chiave `synchronized`. Questa duplicazione viola il principio DRY, ma ciò può essere necessario se il codice impiega strumenti esterni non *thread-safe*.

Questa strategia è pericolosa, perché tutti i programmatore che usano il server devono ricordarsi di bloccarlo prima dell'uso e sbloccarlo al termine. Molti (molti!) anni fa lavorai su un sistema che impiegava il bloccaggio basato sul client su una risorsa condivisa. La risorsa era impiegata in centinaia di punti, in tutto il codice. Un povero programmatore si dimenticò di bloccare la risorsa in uno di quei punti.

Si trattava di un sistema multi terminale in time-sharing che ospitava del software di contabilità per il sistema di spedizioni Local 705. Il computer era in un locale condizionato al primo piano a 50 miglia a nord del quartier generale di Local 705. Al quartier generale avevano decine di addetti al data-entry che inserivano ordini al proprio terminale. I terminali erano connessi al computer tramite linee telefoniche dedicate e modem half-duplex a 600bps (sì, è stato molto, molto tempo fa).

Circa una volta al giorno, uno dei terminali si “impiantava”. Senza apparentemente alcun motivo. Il fenomeno non prediligeva una particolare ora, né un particolare terminale. Sembrava come se qualcuno lanciasse un dado ed estraesse l’ora e il terminale da

“impiantare”. Talvolta però si impiantava più di un terminale. Talvolta però passavano giorni senza alcun problema.

All’inizio l’unica soluzione è sembrata il reboot. Ma il reboot non era semplice da coordinare. Dovevamo chiamare il quartier generale e chiedere a tutti di terminare quello che stavano facendo, su tutti i terminali. Poi potevamo chiudere e riavviare il sistema. Se anche uno solo dei dipendenti stava facendo qualcosa di importante che richiedeva un’ora o due, il terminale bloccato doveva, semplicemente, rimanere bloccato.

Dopo alcune settimane di debugging, abbiamo scoperto che la causa era il contatore di un buffer circolare che perdeva la sincronizzazione col suo puntatore. Questo buffer controllava l’output verso il terminale. Il valore del puntatore indicava che il buffer era vuoto, ma il contatore diceva che era pieno. Poiché era vuoto, non vi era nulla da visualizzare; ma poiché era anche pieno, non si poteva aggiungere al buffer nulla da visualizzare sullo schermo.

Pertanto sapevamo perché i terminali si bloccavano, ma non sapevamo perché il buffer circolare perdeva la sincronizzazione. Così aggiungemmo un piccolo stratagemma per poter risalire al problema. Era possibile leggere la posizione degli interruttori del pannello di controllo del computer (ve lo avevo detto che questo è accaduto molto, molto, molto tempo fa). Abbiamo scritto una piccola funzione di intercettazione che rilevava quando uno di questi interruttori veniva azionato e poi osservava se un buffer circolare era sia vuoto sia pieno. Quando capitava, svuotava il buffer. E voilà! I terminali bloccati tornavano in vita.

Così, ora non dovevamo più riavviare il sistema ogni volta che un terminale si impiantava. Semplicemente ci chiamavano e ci dicevano che c’era stato un bloccaggio e noi andavamo al computer e azionavamo uno switch.

Naturalmente talvolta capitava anche nei weekend e non potevamo farlo. Pertanto aggiungemmo allo scheduler una funzione che ogni minuto controllava tutti i buffer circolari ed eseguiva il reset di quelli che trovava vuoti e anche pieni. In tal modo il terminale tornava online prima ancora che qualcuno potesse prendere in mano la cornetta.

Ci sono volute molte settimane di minuzioso esame di pagine e pagine di monolitico codice Assembly prima di scoprire il colpevole. Abbiamo calcolato che la frequenza dei bloccaggi era coerente con un unico uso non protetto del buffer circolare. Pertanto tutto ciò che dovevamo fare era trovare quella specifica situazione. Sfortunatamente, la cosa si è verificata talmente tanti anni fa che non avevamo strumenti di ricerca o riferimenti incrociati o un qualsiasi altro tipo di supporto automatico. Non potevamo fare altro che scorrere i listati.

Ho appreso un’importante lezione in quel freddo inverno del 1971 a Chicago. Il bloccaggio basato sul client è una maledettissima “gatta da pelare”.

Bloccaggio basato sul server

La duplicazione può essere eliminata apportando le seguenti modifiche a `IntegerIterator`:

```
public class IntegerIteratorServerLocked {  
    private Integer nextValue = 0;  
    public synchronized Integer getNextOrNull() {  
        if (nextValue < 100000)  
            return nextValue++;  
        else  
            return null;  
    }  
}
```

E cambia anche il codice client:

```
while (true) {  
    Integer nextValue = iterator.getNextOrNull();  
    if (next == null)  
        break;  
    // fa qualcosa con nextValue  
}
```

In questo caso, in realtà modifichiamo l'API della nostra classe per renderla multi-thread. Il client deve eseguire il controllo di un `null` invece del controllo `hasNext()`.

In generale dovreste preferire il bloccaggio basato sul server per questi motivi.

- Riduce le ripetizioni di codice - Il bloccaggio basato sul client costringe ogni client a bloccare correttamente il server.
Collocando il codice di bloccaggio nel server, i client sono liberi di usare l'oggetto, senza alcun codice aggiuntivo di bloccaggio.
- Migliora le prestazioni - Potete passare da un server thread-safe a uno non thread-safe nel caso di soluzioni mono-thread, eliminando tutto il sovraccarico di codice.
- Riduce la possibilità di errori - È sufficiente che un programmatore si dimentichi di eseguire correttamente il bloccaggio.
- Impone un'unica politica - La politica si trova in un solo luogo, il server, e non in più punti, tutti i client.
- Riduce il livello di visibilità (scope) delle variabili condivise - Il client non sa più nulla del sistema di bloccaggio. Tutto viene gestito nel server. Quando si verifica un problema, il numero di punti in cui cercarlo si riduce drasticamente.
E che cosa fare se non potete gestire il codice del server?
- Usate un ADAPTER per cambiare l'API e aggiungere il bloccaggio:

```
public class ThreadSafeIntegerIterator {  
    private IntegerIterator iterator = new IntegerIterator();  
    public synchronized Integer getNextOrNull() {  
        if(iterator.hasNext())  
            return iterator.next();  
        return null;  
    }  
}
```

- Oppure, ancora meglio, usate le collection thread-safe con le interfacce estese.

Migliorare la produttività (throughput)

Supponiamo di voler accedere a Internet e di voler leggere il contenuto di un insieme di pagine da un elenco di URL. Mentre ogni pagina viene letta, la analizziamo per accumulare determinate statistiche. Dopo aver letto tutte le pagine, stampiamo un report di riepilogo.

La seguente classe restituisce il contenuto di una pagina, dato un URL.

```
public class PageReader {
    //...
    public String getPageFor(String url) {
        HttpMethod method = new GetMethod(url);
        try {
            httpClient.executeMethod(method);
            String response = method.getResponseBodyAsString();
            return response;
        } catch (Exception e) {
            handle(e);
        } finally {
            method.releaseConnection();
        }
    }
}
```

La classe successiva è l'iteratore che fornisce il contenuto delle pagine sulla base di un iteratore di URL:

```
public class PageIterator {
    private PageReader reader;
    private URLIterator urls;
    public PageIterator(PageReader reader, URLIterator urls) {
        this.urls = urls;
        this.reader = reader;
    }
    public synchronized String getNextPageOrNull() {
        if (urls.hasNext())
            getPageFor(urls.next());
        else
            return null;
    }
    public String getPageFor(String url) {
        return reader.getPageFor(url);
    }
}
```

}

Un’istanza di `PageIterator` può essere condivisa fra più thread, ognuno dei quali usa la propria istanza del `PageReader` per leggere e analizzare le pagine che ottiene dall’iteratore.

Notate che abbiamo mantenuto molto piccolo il blocco `synchronized`. Contiene solo la sezione critica all’interno del `PageIterator`. È sempre meglio sincronizzare il meno possibile.

Calcolo del throughput mono-thread

Ora facciamo qualche semplice calcolo. Per i nostri scopi, immaginate che le prestazioni siano le seguenti.

- Tempo (medio) di I/O per ottenere una pagina: 1 secondo.
- Tempo (medio) di elaborazione per analizzare la pagina: 0,5 secondi.
- L’I/O richiede lo 0 per cento del tempo di CPU; l’elaborazione richiede il 100 per cento.

Per N pagine elaborate da un unico thread, il tempo di esecuzione totale è di $1,5 \text{ secondi} * N$. La Figura A.1 presenta un’istantanea di 13 pagine, per circa 19,5 secondi.

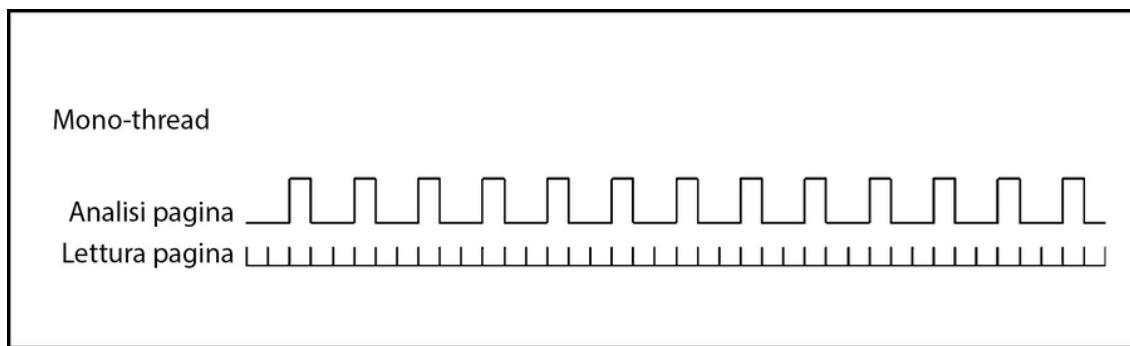


Figura A.1 Mono-thread.

Calcolo del throughput multi-thread

Se è possibile ottenere le pagine in qualsiasi ordine e poi elaborarle in modo indipendente, allora è possibile impiegare più thread per aumentare il throughput. Che cosa accade se usiamo tre thread? Quante pagine possiamo acquisire nello stesso tempo?

Come potete vedere nella Figura A.2, la soluzione multi-thread consente alla fase di elaborazione di sovrapporsi alla fase di I/O delle pagine. In un mondo ideale, significa che il processore viene impiegato appieno. Ogni secondo di lettura di una pagina viene impiegato per due analisi. Pertanto, possiamo elaborare due pagine per secondo, che rappresenta una produttività (throughput) tripla rispetto alla soluzione mono-thread.

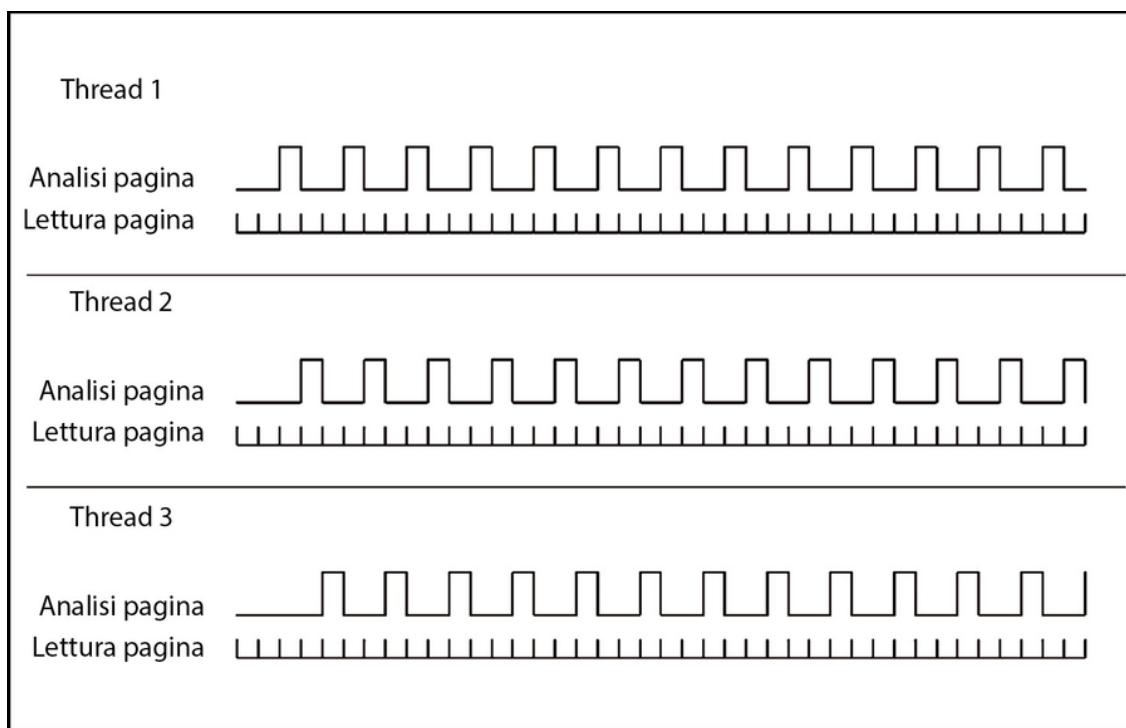


Figura A.2 Tre thread concorrenti.

Deadlock

Immaginate un'applicazione web con due pool di risorse condivise di dimensione finita:

- un pool di connessioni a database per l'elaborazione locale;
- un pool di connessioni MQ con un archivio master.

Supponete che vi siano due operazioni in questa applicazione, *Create* e *Update*.

- *Create* - Acquisisce la connessione con l'archivio master e con il database. Comunica con l'archivio master e poi memorizza il lavoro nel database locale per l'elaborazione.
- *Update* - Acquisisce la connessione con il database e con l'archivio master. Legge il contenuto del database locale e lo invia all'archivio master.

Che cosa accade quando vi sono più utenti che pool? Immaginate che entrambi i pool accettino dieci connessioni.

- Dieci utenti cercano di usare Create, pertanto tutte e dieci le connessioni al database sono impegnate e ogni thread si ferma dopo aver acquisito una connessione al database ma prima di acquisire una connessione all'archivio master.
- Dieci utenti cercano di usare Update, pertanto tutte e dieci le connessioni all'archivio master sono impegnate e ogni thread si ferma dopo aver acquisito l'archivio master ma prima di acquisire una connessione al database.
- Ora i dieci thread “Create” devono attendere di acquisire una connessione con l'archivio master, ma i dieci thread “Update” devono attendere di acquisire una connessione al database.
- Deadlock. Il sistema è bloccato in attesa.

Può sembrare una situazione improbabile, ma a chi piacerebbe un sistema che si blocca completamente una settimana sì e una no? E a chi

piacerebbe eseguire il debug di un sistema che presenta sintomi così difficili da replicare? Questo è il tipo di problema che si verifica sul campo, e la cui soluzione richiede settimane.

Una “soluzione” tipica consiste nell’introdurre delle istruzioni di debug per scoprire che cosa sta accadendo. Naturalmente, le istruzioni di debug modificano il codice in modo che il deadlock si verifichi in situazioni leggermente differenti, magari dopo mesi. (Per esempio, qualcuno potrebbe aggiungere un output di debug che, magicamente, fa “sparire” il problema. Il problema in realtà rimane, ma il codice di debug sembra “risolverlo”).

Per risolvere davvero il problema del deadlock, dobbiamo capire che cosa lo provoca. Vi sono quattro condizioni per il verificarsi di un deadlock:

- esclusione reciproca (*mutual exclusion*);
- blocco e attesa (*lock & wait*);
- nessuna prelazione (*no preemption*);
- attesa circolare (*circular wait*).

Esclusione reciproca (*mutual exclusion*)

L’esclusione reciproca si verifica quando più thread devono usare le stesse risorse, e tali risorse:

- non possono essere usate da più thread contemporaneamente;
- sono in numero limitato.

Un esempio tipico di tale risorsa è una connessione al database, un file aperto in scrittura, un record bloccato o un semaforo.

Blocco e attesa (*lock & wait*)

Quando un thread acquisisce una risorsa, non la rilascia finché non ha acquisito tutte le altre risorse che richiede e finché non ha completato il suo lavoro.

Nessuna prelazione (no preemption)

Un thread non può togliere risorse a un altro thread. Quando un thread ha una risorsa, l'unico modo in cui un altro thread può ottenerla è che il primo thread la rilasci.

Attesa circolare (circular wait)

Questo è chiamato anche *deadly embrace*, “abbraccio mortale”.

Immaginate due thread, T1 e T2, e due risorse, R1 e R2. T1 ha R1, T2 ha R2. T1 ha bisogno di R2 e anche T2 ha bisogno di R1. La situazione è rappresentata nella Figura A.3.

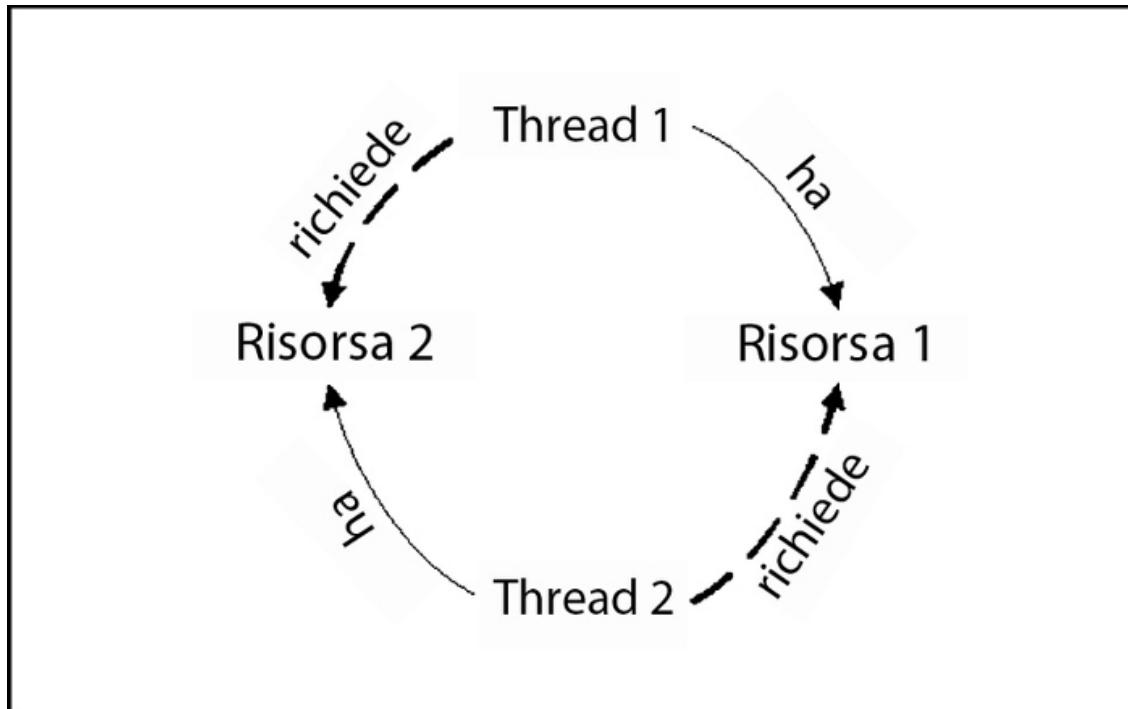


Figura A.3

Se si hanno queste quattro condizioni, può verificarsi un deadlock. Intervenite su queste condizioni e il deadlock diventa impossibile.

Risolvere l'esclusione reciproca

Una strategia per evitare il *deadlock* consiste nel disinnesicare la condizione dell'esclusione reciproca. Potete farlo in vari modi.

- Usando risorse che consentono utilizzi simultanei, per esempio `AtomicInteger`.
- Incrementando il numero di risorse in modo che raggiunga o superi il numero di thread in competizione.
- Controllando che tutte le risorse siano disponibili prima di richiederle.

Sfortunatamente, la maggior parte delle risorse è disponibile in numero limitato e non consente utilizzi simultanei. E inoltre capita frequentemente che l'identità della seconda risorsa sia nota solo dopo aver acquisito la prima. Ma non scoraggiatevi; vi sono le altre tre condizioni.

Risolvere il lock & wait

Potete eliminare il deadlock anche se vi rifiutate di attendere. Controllate ogni risorsa prima di acquisirla e rilasciate tutte le risorse e ricominciate se ne trovate anche solo una impegnata.

Questo approccio introduce però numerosi potenziali problemi.

- *Starvation* - Un thread continua inutilmente a tentare di acquisire le risorse di cui ha bisogno (magari ha una particolare combinazione di risorse che sono disponibili, tutte insieme, solo raramente).

- *Livelock* - Più thread possono tentare di acquisire una risorsa e poi la rilasciano, più e più volte. Questo è più probabile con gli algoritmi di scheduling delle CPU più semplici (come nei dispositivi embedded o con algoritmi troppo semplici di bilanciamento dei thread).

Entrambe queste situazioni possono provocare un calo del throughput. La prima ha come risultato un basso utilizzo della CPU, mentre la seconda innalza l'utilizzo infruttuoso della CPU.

Per quanto sembri inefficiente questa strategia, è meglio di niente. Presenta il vantaggio di poter essere (quasi) sempre implementata nel caso in cui le altre fallissero.

Risolvere il problema del no preemption

Un'altra strategia per evitare i deadlock consiste nel consentire ai thread di sottrarre risorse ad altri thread. Questo viene solitamente svolto tramite un semplice meccanismo di richiesta. Quando un thread scopre che una risorsa è impegnata, domanda al proprietario di rilasciarla. Se il proprietario è in attesa di qualche altra risorsa, potrà acconsentire e ritentare più tardi.

Questo è simile all'approccio precedente, ma presenta il vantaggio di consentire a un thread di attendere una risorsa. Questo riduce il numero di doppi tentativi. Ma attenzione: gestire tutte queste richieste può essere problematico.

Risolvere il problema circular wait

Questo è l'approccio più comune per prevenire i deadlock. Nella maggior parte dei sistemi richiede solo che una semplice convenzione venga accettata da tutte le parti in causa.

Nell'esempio precedente in cui il Thread 1 voleva la Risorsa 1 e la Risorsa 2 e il Thread 2 voleva la Risorsa 2 e poi la Risorsa 1, il semplice fatto di costringere entrambi i thread ad allocare le risorse nello stesso ordine rende impossibile l'attesa circolare.

Più in generale, se tutti i thread si accordano su un ordinamento globale delle risorse e se tutti allocano le risorse in tale ordine, il deadlock diventa impossibile. Come tutte le altre strategie, anche questa può provocare problemi.

- L'ordine di acquisizione potrebbe non corrispondere all'ordine d'uso; pertanto una risorsa acquisita all'inizio potrebbe essere effettivamente utilizzata solo alla fine. Questo può fare in modo che le risorse rimangano bloccate più del necessario.
- Talvolta non è possibile imporre un ordine nell'acquisizione delle risorse. Se l'ID della seconda risorsa è dato da un'operazione svolta sulla prima, l'ordinamento è impossibile.

In buona sostanza esistono molti modi per evitare i deadlock. Alcuni conducono alla *starvation*, mentre altri caricano eccessivamente la CPU e riducono la reattività. Non si può avere tutto dalla vita...

In ogni caso, il fatto di isolare la parte a thread della soluzione per potervi applicare ottimizzazioni e sperimentazioni è un ottimo modo per acquisire le conoscenze necessarie per determinare la migliore strategia.

Collaudo del codice multi-thread

Come possiamo scrivere un test per dimostrare che il seguente codice è problematico?

```
01: public class ClassWithThreadingProblem {  
02:     int nextId;  
03:  
04:     public int takeNextId() {  
05:         return nextId++;  
06:     }  
07: }
```

```
07: }
```

Ecco una descrizione di un test che dimostra che il codice ha problemi.

- Ricordate l'attuale valore di `nextId`.
- Create due thread, entrambi i quali richiamano `takeNextId()` una volta.
- Verificate che `nextId` è pari a 2 unità più di quelle dell'inizio.
- Eseguitelo finché `nextId` non viene incrementato solo di 1 e non di 2.

Il Listato A.2 presenta tale test.

Listato A.2 ClassWithThreadingProblemTest.java.

```
01: package example;
02:
03: import static org.junit.Assert.fail;
04:
05: import org.junit.Test;
06:
07: public class ClassWithThreadingProblemTest {
08:     @Test
09:     public void twoThreadsShouldFailEventually() throws Exception {
10:         final ClassWithThreadingProblem classWithThreadingProblem
11:             = new ClassWithThreadingProblem();
12:
13:         Runnable runnable = new Runnable() {
14:             public void run() {
15:                 classWithThreadingProblem.takeNextId();
16:             }
17:         };
18:
19:         for (int i = 0; i < 50000; ++i) {
20:             int startingId = classWithThreadingProblem.lastId;
21:             int expectedResult = 2 + startingId;
22:
23:             Thread t1 = new Thread(runnable);
24:             Thread t2 = new Thread(runnable);
25:             t1.start();
26:             t2.start();
27:             t1.join();
28:             t2.join();
29:
30:             int endingId = classWithThreadingProblem.lastId;
31:
32:             if (endingId != expectedResult)
33:                 return;
34:         }
35:     }
36: }
```

```

35:     fail("Should have exposed a threading issue ma it did not.");
36: }
37: }
```

Riga	Descrizione
10	Crea un'unica istanza di <code>ClassWithThreadingProblem</code> . Nota: dobbiamo usare la parola chiave <code>final</code> , perché la usiamo più sotto in una classe interna anonima.
12-16	Crea una classe interna anonima che impiega la singola istanza di <code>ClassWithThreadingProblem</code> .
18	Esegui questo codice un numero di volte sufficiente a dimostrare che il codice ha problemi, ma non troppo a lungo. Questo è un tentativo di equilibrio; non vogliamo attendere troppo per dimostrare che vi è un problema. Scegliere questo numero è difficile, ma più avanti vedremo che possiamo ridurre notevolmente questo numero.
19	Memorizza il valore iniziale. Questo test sta cercando di dimostrare che il codice contenuto in <code>ClassWithThreadingProblem</code> ha problemi. Se questo test passa, ha dimostrato che il codice è problematico. Se questo test fallisce, il test non è stato in grado di dimostrare che il codice è problematico.
20	Ci aspettiamo che il valore finale sia 2 di più rispetto all'attuale valore.
22-23	Crea due thread, entrambi i quali usano l'oggetto che abbiamo creato nelle righe 12-16. Ciò ci dà il potenziale di due thread che tentano di usare la nostra unica istanza di <code>ClassWithThreadingProblem</code> e che possono interferire fra loro.
24-25	Consenti l'avvio dei due thread.
26-27	Attendi che entrambi i thread terminino prima di controllare i risultati.
29	Memorizza il valore finale.
31-32	Il nostro <code>endingId</code> differisce da quanto previsto? In caso affermativo, <code>return</code> termina il test: abbiamo dimostrato che il codice ha problemi. In caso contrario, riprova.
35	Se siamo arrivati qui, il nostro test non è stato in grado di dimostrare che il codice era problematico in un tempo "ragionevole"; il nostro codice ha fallito. O il codice non ha problemi o le iterazioni sono state insufficienti per far sorgere la condizione d'errore.

Questo test imposta le condizioni per un problema di aggiornamento concorrente. Tuttavia, il problema si verifica così raramente che la maggior parte delle volte questo test non lo rileva.

In effetti, per rilevare davvero il problema dovremmo impostare il numero di iterazioni a oltre un milione. Ma anche così, in dieci esecuzioni con un ciclo che arriva fino a `1000000`, il problema si è

verificato una sola volta. Questo significa che probabilmente dovremmo impostare l’iterazione a ben oltre un centinaio di milioni per ottenere sicuramente un fallimento. Ma quanto dovremmo aspettare?

Anche se abbiamo ottimizzato il test per ottenere fallimenti affidabili su una macchina, probabilmente dovremo ri-ottimizzarlo con valori differenti per dimostrare il fallimento su un’altra macchina, un altro sistema operativo o un’altra versione della JVM.

E questo è un problema semplice. Se non possiamo dimostrare con facilità la problematicità di questo semplice problema, come potremo farlo con problemi più complessi?

Quali approcci possiamo seguire per dimostrare questo semplice fallimento? E, ancora più importante, come possiamo scrivere dei test in grado di rilevare i problemi presenti in codice più complesso? Come potremo scoprire se il nostro codice è difettoso se non sappiamo dove cercare?

Ecco alcune idee.

- *Monte Carlo Testing.* Create test flessibili, in modo che possano essere regolati. Poi eseguite il test più e più volte (su un server di test) variando casualmente le sue impostazioni. Se i test falliscono, il codice ha problemi. Assicuratevi di iniziare a scrivere tali test anticipatamente, in modo che un server delle integrazioni “in parallelo” inizi a eseguirli il più presto possibile. A proposito, assicuratevi di registrare con cura le condizioni di fallimento del test.
- Eseguite il test su ognuna delle piattaforme di impiego. Ripetutamente. Continuamente. Più a lungo i test funzionano senza rilevare problemi, più è probabile che:
 - il codice sia corretto o
 - i test non sono adeguati a rilevare i problemi.

- Eseguite i test su una macchina con carichi di lavoro variabili. Se potete simulare dei carichi analoghi a quelli di un ambiente di produzione, fatelo.

In ogni caso, anche se fate tutto questo, le probabilità di trovare dei problemi coi thread nel vostro codice sono molto limitate. I problemi più insidiosi sono quelli che hanno un “impronta” talmente sottile da presentarsi solo in un miliardo di tentativi. Tali problemi rappresentano l’incubo di tutti i sistemi complessi.

Strumenti per il test del codice multi-thread

IBM ha creato uno strumento, chiamato ConTest

(<http://www.haifa.ibm.com/projects/verification/contest/index.html>), che mette alla prova le classi che hanno maggiori probabilità di contenere codice che risulterebbe problematico con i thread.

Non abbiamo alcuna relazione diretta con IBM o con il team di sviluppo di ConTest. È stato un nostro collega a segnalarcelo. Abbiamo notato un grande miglioramento nella nostra capacità di trovare problemi nei thread già dopo pochi minuti d’uso.

Ecco come si usa ConTest.

- Scrivete dei test e del codice di produzione, assicurandovi di avere dei test progettati appositamente per simulare più utenti a carichi di lavoro variabili, come abbiamo detto.
- Sottponete i test e il codice di produzione a ConTest.
- Eseguite i test.

Quando abbiamo sottoposto il codice a ConTest, il nostro tasso di successo è passato da circa un fallimento ogni dieci milioni di iterazioni a circa un fallimento ogni trenta iterazioni. Ecco i valori dei

loop per varie esecuzioni del test con ConTest: 13, 23, 0, 54, 16, 14, 6, 69, 107, 49, 2. Pertanto, chiaramente, le classi con ConTest fallivano molto prima e con un'affidabilità molto superiore.

Conclusioni

Questo capitolo ha rappresentato un brevissimo viaggio nel territorio, vasto e irtto di insidie, della programmazione concorrente. Abbiamo solo avuto modo di scalfire la superficie di questa problematica. Abbiamo posto l'accento principalmente sulla disciplina, per aiutarvi a curare la pulizia del codice concorrente, ma vi è molto altro da imparare prima di poter scrivere dei sistemi concorrenti. Vi consiglio di iniziare dal meraviglioso libro di Doug Lea *Concurrent Programming in Java: Design Principles and Patterns* (vedi [Lea99], p. 191).

In questo capitolo abbiamo parlato di aggiornamento concorrente e della disciplina impiegabile per curare la pulizia dei meccanismi di sincronizzazione e dei sistemi di bloccaggio disponibili. Abbiamo visto come i thread possano migliorare il throughput di un sistema con molte operazioni di I/O e abbiamo mostrato alcune tecniche sicure per rendere possibili tali miglioramenti. Abbiamo parlato dei deadlock e di come prevenirli. Infine, abbiamo parlato delle strategie per rilevare i problemi legati alla concorrenza, mettendo alla prova il vostro codice.

Tutorial: esempi di codice

Client/server senza threading

Listato A.3 Server.java.

```
package com.objectmentor.clientserver.nonthreaded;
import java.io.IOException;
```

```

import java.net.ServerSocket;
import java.net.Socket;
import java.net.SocketException;
import common.MessageUtils;
public class Server implements Runnable {
    ServerSocket serverSocket;
    volatile boolean keepProcessing = true;
    public Server(int port, int millisecondsTimeout) throws IOException {
        serverSocket = new ServerSocket(port);
        serverSocket.setSoTimeout(millisecondsTimeout);
    }
    public void run() {
        System.out.printf("Server Starting\n");
        while (keepProcessing) {
            try {
                System.out.printf("accepting client\n");
                Socket socket = serverSocket.accept();
                System.out.printf("got client\n");
                process(socket);
            } catch (Exception e) {
                handle(e);
            }
        }
    }
    private void handle(Exception e) {
        if (!(e instanceof SocketException)) {
            e.printStackTrace();
        }
    }
    public void stopProcessing() {
        keepProcessing = false;
        closeIgnoringException(serverSocket);
    }
    void process(Socket socket) {
        if (socket == null)
            return;
        try {
            System.out.printf("Server: getting message\n");
            String message = MessageUtils.getMessage(socket);
            System.out.printf("Server: got message: %s\n", message);
            Thread.sleep(1000);
            System.out.printf("Server: sending reply: %s\n", message);
            MessageUtils.sendMessage(socket, "Processed: " + message);
            System.out.printf("Server: sent\n");
            closeIgnoringException(socket);
        } catch (Exception e) {
            e.printStackTrace();
        }
    }
    private void closeIgnoringException(Socket socket) {
        if (socket != null) try {
            socket.close();
        } catch (IOException ignore) {
        }
    }
    private void closeIgnoringException(ServerSocket serverSocket) {
        if (serverSocket != null)
            try {
                serverSocket.close();
            } catch (IOException ignore) {

```

```
        }
    }
}
```

Listato A4 ClientTest.java.

```
package com.objectmentor.clientserver.nonthreaded;
import java.io.IOException;
import java.net.Socket;
import org.junit.After;
import org.junit.Before;
import org.junit.Test;
import common.MessageUtils;
public class ClientTest {
    private static final int PORT = 8009;
    private static final int TIMEOUT = 2000;
    Server server;
    Thread serverThread;
    @Before
    public void createServer() throws Exception {
        try {
            server = new Server(PORT, TIMEOUT);
            serverThread = new Thread(server);
            serverThread.start();
        } catch (Exception e) {
            e.printStackTrace(System.err);
            throw e;
        }
    }
    @After
    public void shutdownServer() throws InterruptedException {
        if (server != null) {
            server.stopProcessing();
            serverThread.join();
        }
    }
    class TrivialClient implements Runnable {
        int clientNumber;
        TrivialClient(int clientNumber) {
            this.clientNumber = clientNumber;
        }
        public void run() {
            try {
                connectSendReceive(clientNumber);
            } catch (IOException e) {
                e.printStackTrace();
            }
        }
    }
    @Test(timeout = 10000)
    public void shouldRunInUnder10Seconds() throws Exception {
        Thread[] threads = new Thread[10];
        for (int i = 0; i < threads.length; ++i) {
            threads[i] = new Thread(new TrivialClient(i));
            threads[i].start();
        }
        for (int i = 0; i < threads.length; ++i) {
            threads[i].join();
        }
    }
}
```

```

        private void connectSendReceive(int i) throws IOException {
            System.out.printf("Client %d: connecting\n", i);
            Socket socket = new Socket("localhost", PORT);
            System.out.printf("Client %d: sending message\n", i);
            MessageUtils.sendMessage(socket, Integer.toString(i));
            System.out.printf("Client %d: getting reply\n", i);
            MessageUtils.getMessage(socket);
            System.out.printf("Client %d: finished\n", i);
            socket.close();
        }
    }
}

```

Listato A.5 MessageUtils.java.

```

package common;
import java.io.IOException;
import java.io.InputStream;
import java.io.ObjectInputStream;
import java.io.ObjectOutputStream;
import java.io.OutputStream;
import java.net.Socket;
public class MessageUtils {
    public static void sendMessage(Socket socket, String message) throws IOException {
    }
        OutputStream stream = socket.getOutputStream();
        ObjectOutputStream oos = new ObjectOutputStream(stream);
        oos.writeUTF(message);
        oos.flush();
    }
    public static String getMessage(Socket socket) throws IOException {
        InputStream stream = socket.getInputStream();
        ObjectInputStream ois = new ObjectInputStream(stream);
        return ois.readUTF();
    }
}

```

Client/server con threading

Per modificare il server in modo da impiegare i thread basta modificare il metodo `process` (le nuove righe sono indicate in grassetto):

```

void process(final Socket socket) {
    if (socket == null)
        return;
Runnable clientHandler = new Runnable() {
    public void run() {
        try {
            System.out.printf("Server: getting message\n");
            String message = MessageUtils.getMessage(socket);
            System.out.printf("Server: got message: %s\n", message);
            Thread.sleep(1000);
            System.out.printf("Server: sending reply: %s\n", message);
            MessageUtils.sendMessage(socket, "Processed: " + message);
            System.out.printf("Server: sent\n");
            closeIgnoringException(socket);
        } catch (Exception e) {

```

```
        e.printStackTrace();
    }
}
Thread clientConnection = new Thread(clientHandler);
clientConnection.start();
}
```

Appendice B

org.jfree.date.SerialDate

Listato B.1 SerialDate.Java.

```
/*
=====
* JCommon : a free general purpose class library for the Java(tm) platform
* =====
*
* (C) Copyright 2000-2005, by Object Refinery Limited and Contributors.
*
* Project Info: http://www.jfree.org/jcommon/index.html
*
* This library is free software; you can redistribute it and/or modify it
* under the terms of the GNU Lesser General Public License as published by
* the Free Software Foundation; either version 2.1 of the License, or
* (at your option) any later version.
*
* This library is distributed in the hope that it will be useful, but
* WITHOUT ANY WARRANTY; without even the implied warranty of MERCHANTABILITY
* or FITNESS FOR A PARTICULAR PURPOSE. See the GNU Lesser General Public
* License for more details.
*
* You should have received a copy of the GNU Lesser General Public
* License along with this library; if not, write to the Free Software
* Foundation, Inc., 51 Franklin Street, Fifth Floor, Boston, MA 02110-1301,
* USA.
*
* [Java is a trademark or registered trademark of Sun Microsystems, Inc.
* in the United States and other countries.]
*
-----
* SerialDate.java
-----
* (C) Copyright 2001-2005, by Object Refinery Limited.
*
* Original Author: David Gilbert (for Object Refinery Limited);
* Contributor(s): -;
*
* $Id: SerialDate.java,v 1.7 2005/11/03 09:25:17 mungady Exp $
*
* Changes (from 11-Oct-2001)
*
-----
* 11-Oct-2001 : Re-organised the class and moved it to new package
* com.jrefinery.date (DG);
* 05-Nov-2001 : Added a getDescription() method, and eliminated NotableDate
* class (DG);
* 12-Nov-2001 : IBD requires setDescription() method, now that NotableDate
```

```

* class is gone (DG); Changed getPreviousDayOfWeek(),
* getFollowingDayOfWeek() and getNearestDayOfWeek() to correct
* bugs (DG);
* 05-Dec-2001 : Fixed bug in SpreadsheetDate class (DG);
* 29-May-2002 : Moved the month constants into a separate interface
* (MonthConstants) (DG);
* 27-Aug-2002 : Fixed bug in addMonths() method, thanks to N???levka Petr (DG);
* 03-Oct-2002 : Fixed errors reported by Checkstyle (DG);
* 13-Mar-2003 : Implemented Serializable (DG);
* 29-May-2003 : Fixed bug in addMonths method (DG);
* 04-Sep-2003 : Implemented Comparable. Updated the isInRange javadocs (DG);
* 05-Jan-2005 : Fixed bug in addYears() method (1096282) (DG);
*
*/
package org.jfree.date;
import java.io.Serializable;
import java.text.DateFormatSymbols;
import java.text.SimpleDateFormat;
import java.util.Calendar;
import java.util.GregorianCalendar;
/**
* An abstract class that defines our requirements for manipulating dates,
* without tying down a particular implementation.
* <P>
* Requirement 1 : match at least what Excel does for dates;
* Requirement 2 : the date represented by the class is immutable;
* <P>
* Why not just use java.util.Date? We will, when it makes sense. At times,
* java.util.Date can be *too* precise - it represents an instant in time,
* accurate to 1/1000th of a second (with the date itself depending on the
* time-zone). Sometimes we just want to represent a particular day (e.g. 21
* January 2015) without concerning ourselves about the time of day, or the
* time-zone, or anything else. That's what we've defined SerialDate for.
* <P>
* You can call getInstance() to get a concrete subclass of SerialDate,
* without worrying about the exact implementation.
*
* @author David Gilbert
*/
public abstract class SerialDate implements Comparable,
Serializable,
MonthConstants {
/** For serialization. */
private static final long serialVersionUID = -293716040467423637L;

/** Date format symbols. */
public static final DateFormatSymbols
DATE_FORMAT_SYMBOLS = new SimpleDateFormat().getDateFormatSymbols();
/** The serial number for 1 January 1900. */
public static final int SERIAL_LOWER_BOUND = 2;
/** The serial number for 31 December 9999. */
public static final int SERIAL_UPPER_BOUND = 2958465;
/** The lowest year value supported by this date format. */
public static final int MINIMUM_YEAR_SUPPORTED = 1900;
/** The highest year value supported by this date format. */
public static final int MAXIMUM_YEAR_SUPPORTED = 9999;
/** Useful constant for Monday. Equivalent to java.util.Calendar.MONDAY. */
public static final int MONDAY = Calendar.MONDAY;
/**
* Useful constant for Tuesday. Equivalent to java.util.Calendar.TUESDAY.
*/

```

```

public static final int TUESDAY = Calendar.TUESDAY;
/**
 * Useful constant for Wednesday. Equivalent to
 * java.util.Calendar.WEDNESDAY.
 */
public static final int WEDNESDAY = Calendar.WEDNESDAY;
/**
 * Useful constant for Thursday. Equivalent to java.util.Calendar.THURSDAY.
 */
public static final int THURSDAY = Calendar.THURSDAY;
/** Useful constant for Friday. Equivalent to java.util.Calendar.FRIDAY. */
public static final int FRIDAY = Calendar.FRIDAY;
/**
 * Useful constant for Saturday. Equivalent to java.util.Calendar.SATURDAY.
 */
public static final int SATURDAY = Calendar.SATURDAY;
/** Useful constant for Sunday. Equivalent to java.util.Calendar.SUNDAY. */
public static final int SUNDAY = Calendar.SUNDAY;
/** The number of days in each month in non leap years. */
static final int[] LAST_DAY_OF_MONTH =
{0, 31, 28, 31, 30, 31, 30, 31, 30, 31, 30, 31};
/** The number of days in a (non-leap) year up to the end of each month. */
static final int[] AGGREGATE_DAYS_TO_END_OF_MONTH =
{0, 31, 59, 90, 120, 151, 181, 212, 243, 273, 304, 334, 365};
/** The number of days in a year up to the end of the preceding month. */
static final int[] AGGREGATE_DAYS_TO_END_OF_PRECEDING_MONTH =
{0, 0, 31, 59, 90, 120, 151, 181, 212, 243, 273, 304, 334, 365};
/** The number of days in a leap year up to the end of each month. */
static final int[] LEAP_YEAR_AGGREGATE_DAYS_TO_END_OF_MONTH =
{0, 31, 60, 91, 121, 152, 182, 213, 244, 274, 305, 335, 366};
/**
 * The number of days in a leap year up to the end of the preceding month.
 */
static final int[]
LEAP_YEAR_AGGREGATE_DAYS_TO_END_OF_PRECEDING_MONTH =
{0, 0, 31, 60, 91, 121, 152, 182, 213, 244, 274, 305, 335, 366};
/** A useful constant for referring to the first week in a month. */
public static final int FIRST_WEEK_IN_MONTH = 1;
/** A useful constant for referring to the second week in a month. */
public static final int SECOND_WEEK_IN_MONTH = 2;
/** A useful constant for referring to the third week in a month. */
public static final int THIRD_WEEK_IN_MONTH = 3;
/** A useful constant for referring to the fourth week in a month. */
public static final int FOURTH_WEEK_IN_MONTH = 4;
/** A useful constant for referring to the last week in a month. */
public static final int LAST_WEEK_IN_MONTH = 0;
/** Useful range constant. */
public static final int INCLUDE_NONE = 0;
/** Useful range constant. */
public static final int INCLUDE_FIRST = 1;
/** Useful range constant. */
public static final int INCLUDE_SECOND = 2;
/** Useful range constant. */
public static final int INCLUDE_BOTH = 3;
/**
 * Useful constant for specifying a day of the week relative to a fixed
 * date.
 */
public static final int PRECEDING = -1;
/**
 * Useful constant for specifying a day of the week relative to a fixed

```

```

* date.
*/
public static final int NEAREST = 0;
/**
* Useful constant for specifying a day of the week relative to a fixed
* date.
*/
public static final int FOLLOWING = 1;
/** A description for the date. */
private String description;
/**
* Default constructor.
*/
protected SerialDate() {
}
/**
* Returns <code>true</code> if the supplied integer code represents a
* valid day-of-the-week, and <code>false</code> otherwise.
*
* @param code the code being checked for validity.
*
* @return <code>true</code> if the supplied integer code represents a
* valid day-of-the-week, and <code>false</code> otherwise.
*/
public static boolean isValidWeekdayCode(final int code) {
switch(code) {
case SUNDAY:
case MONDAY:
case TUESDAY:
case WEDNESDAY:
case THURSDAY:
case FRIDAY:
case SATURDAY:
return true;
default:
return false;
}
}
/**
* Converts the supplied string to a day of the week.
*
* @param s a string representing the day of the week.
*
* @return <code>-1</code> if the string is not convertable, the day of
* the week otherwise.
*/
public static int stringToWeekdayCode(String s) {
final String[] shortWeekdayNames
= DATE_FORMAT_SYMBOLS.getShortWeekdays();
final String[] weekDayNames = DATE_FORMAT_SYMBOLS.getWeekdays();
int result = -1;
s = s.trim();
for (int i = 0; i < weekDayNames.length; i++) {
if (s.equals(shortWeekdayNames[i])) {
result = i;
break;
}
if (s.equals(weekDayNames[i])) {
result = i;
break;
}
}
}

```

```
        }
        return result;
    }
    /**
     * Returns a string representing the supplied day-of-the-week.
     * <P>
     * Need to find a better approach.
     *
     * @param weekday the day of the week.
     *
     * @return a string representing the supplied day-of-the-week.
     */
    public static String weekdayCodeToString(final int weekday) {
        final String[] weekdays = DATE_FORMAT_SYMBOLS.getWeekdays();
        return weekdays[weekday];
    }
    /**
     * Returns an array of month names.
     *
     * @return an array of month names.
     */
    public static String[] getMonths() {
        return getMonths(false);
    }
    /**
     * Returns an array of month names.
     *
     * @param shortened a flag indicating that shortened month names should
     * be returned.
     *
     * @return an array of month names.
     */
    public static String[] getMonths(final boolean shortened) {
        if (shortened) {
            return DATE_FORMAT_SYMBOLS.getShortMonths();
        }
        else {
            return DATE_FORMAT_SYMBOLS.getMonths();
        }
    }
    /**
     * Returns true if the supplied integer code represents a valid month.
     *
     * @param code the code being checked for validity.
     *
     * @return <code>true</code> if the supplied integer code represents a
     * valid month.
     */
    public static boolean isValidMonthCode(final int code) {
        switch(code) {
        case JANUARY:
        case FEBRUARY:
        case MARCH:
        case APRIL:
        case MAY:
        case JUNE:
        case JULY:
        case AUGUST:
        case SEPTEMBER:
        case OCTOBER:
        case NOVEMBER:
```

```

        case DECEMBER:
            return true;
        default:
            return false;
        }
    }
}

/**
 * Returns the quarter for the specified month.
 *
 * @param code the month code (1-12).
 *
 * @return the quarter that the month belongs to.
 * @throws java.lang.IllegalArgumentException
 */
public static int monthCodeToQuarter(final int code) {
    switch(code) {
        case JANUARY:
        case FEBRUARY:
        case MARCH: return 1;
        case APRIL:
        case MAY:
        case JUNE: return 2;
        case JULY:
        case AUGUST:
        case SEPTEMBER: return 3;
        case OCTOBER:
        case NOVEMBER:
        case DECEMBER: return 4;
        default: throw new IllegalArgumentException(
            "SerialDate.monthCodeToQuarter: invalid month code.");
    }
}

/**
 * Returns a string representing the supplied month.
 * <P>
 * The string returned is the long form of the month name taken from the
 * default locale.
 *
 * @param month the month.
 *
 * @return a string representing the supplied month.
 */
public static String monthCodeToString(final int month) {
    return monthCodeToString(month, false);
}

/**
 * Returns a string representing the supplied month.
 * <P>
 * The string returned is the long or short form of the month name taken
 * from the default locale.
 *
 * @param month the month.
 * @param shortened if <code>true</code> return the abbreviation of the
 * month.
 *
 * @return a string representing the supplied month.
 * @throws java.lang.IllegalArgumentException
 */
public static String monthCodeToString(final int month,
    final boolean shortened) {
    // check s...
}

```

```

if (!isValidMonthCode(month)) {
    throw new IllegalArgumentException(
        "SerialDate.monthCodeToString: month outside valid range.");
}
final String[] months;
if (shortened) {
    months = DATE_FORMAT_SYMBOLS.getShortMonths();
}
else {
    months = DATE_FORMAT_SYMBOLS.getMonths();
}
return months[month - 1];
}
<*/
* Converts a string to a month code.
* <P>
* This method will return one of the constants JANUARY, FEBRUARY, ...,
* DECEMBER that corresponds to the string. If the string is not
* recognised, this method returns -1.
*
* @param s the string to parse.
*
* @return <code>-1</code> if the string is not parseable, the month of the
* year otherwise.
*/
public static int stringToMonthCode(String s) {
final String[] shortMonthNames = DATE_FORMAT_SYMBOLS.getShortMonths();
final String[] monthNames = DATE_FORMAT_SYMBOLS.getMonths();
int result = -1;
s = s.trim();
// first try parsing the string as an integer (1-12)...
try {
    result = Integer.parseInt(s);
}
catch (NumberFormatException e) {
    // suppress
}
// now search through the month names...
if ((result < 1) || (result > 12)) {
    for (int i = 0; i < monthNames.length; i++) {
        if (s.equals(shortMonthNames[i])) {
            result = i + 1;
            break;
        }
        if (s.equals(monthNames[i])) {
            result = i + 1;
            break;
        }
    }
}
return result;
}
<*/
* Returns true if the supplied integer code represents a valid
* week-in-the-month, and false otherwise.
*
* @param code the code being checked for validity.
* @return <code>true</code> if the supplied integer code represents a
* valid week-in-the-month.
*/
public static boolean isValidWeekInMonthCode(final int code) {

```

```

switch(code) {
    case FIRST_WEEK_IN_MONTH:
    case SECOND_WEEK_IN_MONTH:
    case THIRD_WEEK_IN_MONTH:
    case FOURTH_WEEK_IN_MONTH:
    case LAST_WEEK_IN_MONTH: return true;
    default: return false;
}
}

/***
 * Determines whether or not the specified year is a leap year.
 *
 * @param yyyy the year (in the range 1900 to 9999).
 *
 * @return <code>true</code> if the specified year is a leap year.
 */
public static boolean isLeapYear(final int yyyy) {
    if ((yyyy % 4) != 0) {
        return false;
    }
    else if ((yyyy % 400) == 0) {
        return true;
    }
    else if ((yyyy % 100) == 0) {
        return false;
    }
    else {
        return true;
    }
}

/***
 * Returns the number of leap years from 1900 to the specified year
 * INCLUSIVE.
 * <P>
 * Note that 1900 is not a leap year.
 *
 * @param yyyy the year (in the range 1900 to 9999).
 *
 * @return the number of leap years from 1900 to the specified year.
 */
public static int leapYearCount(final int yyyy) {
    final int leap4 = (yyyy - 1896) / 4;
    final int leap100 = (yyyy - 1800) / 100;
    final int leap400 = (yyyy - 1600) / 400;
    return leap4 - leap100 + leap400;
}

/***
 * Returns the number of the last day of the month, taking into account
 * leap years.
 *
 * @param month the month.
 * @param yyyy the year (in the range 1900 to 9999).
 *
 * @return the number of the last day of the month.
 */
public static int lastDayOfMonth(final int month, final int yyyy) {
    final int result = LAST_DAY_OF_MONTH[month];
    if (month != FEBRUARY) {
        return result;
    }
    else if (isLeapYear(yyyy)) {

```

```

        return result + 1;
    }
    else {
        return result;
    }
}
/***
 * Creates a new date by adding the specified number of days to the base
 * date.
 *
 * @param days the number of days to add (can be negative).
 * @param base the base date.
 *
 * @return a new date.
 */
public static SerialDate addDays(final int days, final SerialDate base) {
    final int serialDayNumber = base.toSerial() + days;
    return SerialDate.createInstance(serialDayNumber);
}
/***
 * Creates a new date by adding the specified number of months to the base
 * date.
 * <P>
 * If the base date is close to the end of the month, the day on the result
 * may be adjusted slightly: 31 May + 1 month = 30 June.
 *
 * @param months the number of months to add (can be negative).
 * @param base the base date.
 *
 * @return a new date.
 */
public static SerialDate addMonths(final int months,
final SerialDate base) {
    final int yy = (12 * base.getYYYY() + base.getMonth() + months - 1)
    / 12;
    final int mm = (12 * base.getYYYY() + base.getMonth() + months - 1)
    % 12 + 1;
    final int dd = Math.min(
    base.getDayOfMonth(), SerialDate.lastDayOfMonth(mm, yy)
    );
    return SerialDate.createInstance(dd, mm, yy);
}
/***
 * Creates a new date by adding the specified number of years to the base
 * date.
 *
 * @param years the number of years to add (can be negative).
 * @param base the base date.
 *
 * @return A new date.
 */
public static SerialDate addYears(final int years, final SerialDate base) {
    final int baseY = base.getYYYY();
    final int baseM = base.getMonth();
    final int baseD = base.getDayOfMonth();
    final int targetY = baseY + years;
    final int targetD = Math.min(
    baseD, SerialDate.lastDayOfMonth(baseM, targetY)
    );
    return SerialDate.createInstance(targetD, baseM, targetY);
}

```

```

    /**
     * Returns the latest date that falls on the specified day-of-the-week and
     * is BEFORE the base date.
     *
     * @param targetWeekday a code for the target day-of-the-week.
     * @param base the base date.
     *
     * @return the latest date that falls on the specified day-of-the-week and
     * is BEFORE the base date.
     */
    public static SerialDate getPreviousDayOfWeek(final int targetWeekday,
final SerialDate base) {
    // check arguments...
    if (!SerialDate.isValidWeekdayCode(targetWeekday)) {
        throw new IllegalArgumentException(
            "Invalid day-of-the-week code.");
    }
    // find the date...
    final int adjust;
    final int baseDOW = base.getDayOfWeek();
    if (baseDOW > targetWeekday) {
        adjust = Math.min(0, targetWeekday - baseDOW);
    }
    else {
        adjust = -7 + Math.max(0, targetWeekday - baseDOW);
    }
    return SerialDate.addDays(adjust, base);
}
/**
 * Returns the earliest date that falls on the specified day-of-the-week
 * and is AFTER the base date.
 *
 * @param targetWeekday a code for the target day-of-the-week.
 * @param base the base date.
 *
 * @return the earliest date that falls on the specified day-of-the-week
 * and is AFTER the base date.
 */
public static SerialDate getFollowingDayOfWeek(final int targetWeekday,
final SerialDate base) {
    // check arguments...
    if (!SerialDate.isValidWeekdayCode(targetWeekday)) {
        throw new IllegalArgumentException(
            "Invalid day-of-the-week code.");
    }
    // find the date...
    final int adjust;
    final int baseDOW = base.getDayOfWeek();
    if (baseDOW > targetWeekday) {
        adjust = 7 + Math.min(0, targetWeekday - baseDOW);
    }
    else {
        adjust = Math.max(0, targetWeekday - baseDOW);
    }
    return SerialDate.addDays(adjust, base);
}
/**
 * Returns the date that falls on the specified day-of-the-week and is
 * CLOSEST to the base date.

```

```

*
* @param targetDOW a code for the target day-of-the-week.
* @param base the base date.
*
* @return the date that falls on the specified day-of-the-week and is
* CLOSEST to the base date.
*/
public static SerialDate getNearestDayOfWeek(final int targetDOW,
final SerialDate base) {
// check arguments...
if (!SerialDate.isValidWeekdayCode(targetDOW)) {
throw new IllegalArgumentException(
"Invalid day-of-the-week code."
);
}
// find the date...
final int baseDOW = base.getDayOfWeek();
int adjust = -Math.abs(targetDOW - baseDOW);
if (adjust >= 4) {
adjust = 7 - adjust;
}
if (adjust <= -4) {
adjust = 7 + adjust;
}
return SerialDate.addDays(adjust, base);
}
/***
* Rolls the date forward to the last day of the month.
*
* @param base the base date.
*
* @return a new serial date.
*/
public SerialDate getEndOfCurrentMonth(final SerialDate base) {
final int last = SerialDate.lastDayOfMonth(
base.getMonth(), base.getYYYY()
);
return SerialDate.createInstance(last, base.getMonth(), base.getYYYY());
}
/***
* Returns a string corresponding to the week-in-the-month code.
* <P>
* Need to find a better approach.
*
* @param count an integer code representing the week-in-the-month.
*
* @return a string corresponding to the week-in-the-month code.
*/
public static String weekInMonthToString(final int count) {
switch (count) {
case SerialDate.FIRST_WEEK_IN_MONTH : return "First";
case SerialDate.SECOND_WEEK_IN_MONTH : return "Second";
case SerialDate.THIRD_WEEK_IN_MONTH : return "Third";
case SerialDate.FOURTH_WEEK_IN_MONTH : return "Fourth";
case SerialDate.LAST_WEEK_IN_MONTH : return "Last";
default :
return "SerialDate.weekInMonthToString(): invalid code.";
}
}
}
/***
* Returns a string representing the supplied 'relative'.

```

```

* <P>
* Need to find a better approach.
*
* @param relative a constant representing the 'relative'.
*
* @return a string representing the supplied 'relative'.
*/
public static String relativeToString(final int relative) {
    switch (relative) {
        case SerialDate.PRECEDING : return "Preceding";
        case SerialDate.NEAREST : return "Nearest";
        case SerialDate.FOLLOWING : return "Following";
        default : return "ERROR : Relative To String";
    }
}
/**
* Factory method that returns an instance of some concrete subclass of
* {@link SerialDate}.
*
* @param day the day (1-31).
* @param month the month (1-12).
* @param yyyy the year (in the range 1900 to 9999).
*
* @return An instance of {@link SerialDate}.
*/
public static SerialDate createInstance(final int day, final int month,
final int yyyy) {
    return new SpreadsheetDate(day, month, yyyy);
}
/**
* Factory method that returns an instance of some concrete subclass of
* {@link SerialDate}.
*
* @param serial the serial number for the day (1 January 1900 = 2).
*
* @return a instance of SerialDate.
*/
public static SerialDate createInstance(final int serial) {
    return new SpreadsheetDate(serial);
}
/**
* Factory method that returns an instance of a subclass of SerialDate.
*
* @param date A Java date object.
*
* @return a instance of SerialDate.
*/
public static SerialDate createInstance(final java.util.Date date) {
    final GregorianCalendar calendar = new GregorianCalendar();
    calendar.setTime(date);
    return new SpreadsheetDate(calendar.get(Calendar.DATE),
        calendar.get(Calendar.MONTH) + 1,
        calendar.get(Calendar.YEAR));
}
/**
* Returns the serial number for the date, where 1 January 1900 = 2 (this
* corresponds, almost, to the numbering system used in Microsoft Excel for
* Windows and Lotus 1-2-3).
*
* @return the serial number for the date.
*/

```

```
public abstract int toSerial();
/**
 * Returns a java.util.Date. Since java.util.Date has more precision than
 * SerialDate, we need to define a convention for the 'time of day'.
 *
 * @return this as <code>java.util.Date</code>.
 */
public abstract java.util.Date toDate();
/**
 * Returns a description of the date.
 *
 * @return a description of the date.
 */
public String getDescription() {
    return this.description;
}
/**
 * Sets the description for the date.
 *
 * @param description the new description for the date.
 */
public void setDescription(final String description) {
    this.description = description;
}
/**
 * Converts the date to a string.
 *
 * @return a string representation of the date.
 */
public String toString() {
    return getDayOfMonth() + "-" + SerialDate.monthCodeToString(getMonth())
        + "-" + getYYYY();
}
/**
 * Returns the year (assume a valid range of 1900 to 9999).
 *
 * @return the year.
 */
public abstract int getYYYY();
/**
 * Returns the month (January = 1, February = 2, March = 3).
 *
 * @return the month of the year.
 */
public abstract int getMonth();
/**
 * Returns the day of the month.
 *
 * @return the day of the month.
 */
public abstract int getDayOfMonth();
/**
 * Returns the day of the week.
 *
 * @return the day of the week.
 */
public abstract int getDayOfWeek();
/**
 * Returns the difference (in days) between this date and the specified
 * 'other' date.
 *
 * <P>
```

```

* The result is positive if this date is after the 'other' date and
* negative if it is before the 'other' date.
*
* @param other the date being compared to.
*
* @return the difference between this and the other date.
*/
public abstract int compare(SerialDate other);
/***
* Returns true if this SerialDate represents the same date as the
* specified SerialDate.
*
* @param other the date being compared to.
*
* @return <code>true</code> if this SerialDate represents the same date as
* the specified SerialDate.
*/
public abstract boolean isOn(SerialDate other);
/***
* Returns true if this SerialDate represents an earlier date compared to
* the specified SerialDate.
*
* @param other The date being compared to.
*
* @return <code>true</code> if this SerialDate represents an earlier date
* compared to the specified SerialDate.
*/
public abstract boolean isBefore(SerialDate other);
/***
* Returns true if this SerialDate represents the same date as the
* specified SerialDate.
*
* @param other the date being compared to.
*
* @return <code>true</code> if this SerialDate represents the same date
* as the specified SerialDate.
*/
public abstract boolean isOnOrBefore(SerialDate other);
/***
* Returns true if this SerialDate represents the same date as the
* specified SerialDate.
*
* @param other the date being compared to.
*
* @return <code>true</code> if this SerialDate represents the same date
* as the specified SerialDate.
*/
public abstract boolean isAfter(SerialDate other);
/***
* Returns true if this SerialDate represents the same date as the
* specified SerialDate.
*
* @param other the date being compared to.
*
* @return <code>true</code> if this SerialDate represents the same date
* as the specified SerialDate.
*/
public abstract boolean isOnOrAfter(SerialDate other);
/***
* Returns <code>true</code> if this {@link SerialDate} is within the
* specified range (INCLUSIVE). The date order of d1 and d2 is not

```

```

* important.
*
* @param d1 a boundary date for the range.
* @param d2 the other boundary date for the range.
*
* @return A boolean.
*/
public abstract boolean isInRange(SerialDate d1, SerialDate d2);
/**
 * Returns <code>true</code> if this {@link SerialDate} is within the
 * specified range (caller specifies whether or not the end-points are
 * included). The date order of d1 and d2 is not important.
*
* @param d1 a boundary date for the range.
* @param d2 the other boundary date for the range.
* @param include a code that controls whether or not the start and end
* dates are included in the range.
*
* @return A boolean.
*/
public abstract boolean isInRange(SerialDate d1, SerialDate d2,
int include);
/**
* Returns the latest date that falls on the specified day-of-the-week and
* is BEFORE this date.
*
* @param targetDOW a code for the target day-of-the-week.
*
* @return the latest date that falls on the specified day-of-the-week and
* is BEFORE this date.
*/
public SerialDate getPreviousDayOfWeek(final int targetDOW) {
    return getPreviousDayOfWeek(targetDOW, this);
}
/**
* Returns the earliest date that falls on the specified day-of-the-week
* and is AFTER this date.
*
* @param targetDOW a code for the target day-of-the-week.
*
* @return the earliest date that falls on the specified day-of-the-week
* and is AFTER this date.
*/
public SerialDate getFollowingDayOfWeek(final int targetDOW) {
    return getFollowingDayOfWeek(targetDOW, this);
}
/**
* Returns the nearest date that falls on the specified day-of-the-week.
*
* @param targetDOW a code for the target day-of-the-week.
*
* @return the nearest date that falls on the specified day-of-the-week.
*/
public SerialDate getNearestDayOfWeek(final int targetDOW) {
    return getNearestDayOfWeek(targetDOW, this);
}
}

```

Listato B.2 SerialDateTest.java.

```
/* =====
 * JCommon : a free general purpose class library for the Java(tm) platform
 * =====
 *
 * (C) Copyright 2000-2005, by Object Refinery Limited and Contributors..
 *
 * Project Info: http://www.jfree.org/jcommon/index.html
 *
 * This library is free software; you can redistribute it and/or modify it
 * under the terms of the GNU Lesser General Public License as published by
 * the Free Software Foundation; either version 2.1 of the License, or
 * (at your option) any later version.
 *
 * This library is distributed in the hope that it will be useful, but
 * WITHOUT ANY WARRANTY; without even the implied warranty of MERCHANTABILITY
 * or FITNESS FOR A PARTICULAR PURPOSE. See the GNU Lesser General Public
 * License for more details.
 *
 * You should have received a copy of the GNU Lesser General Public
 * License along with this library; if not, write to the Free Software
 * Foundation, Inc., 51 Franklin Street, Fifth Floor, Boston, MA 02110-1301,
 * USA.
 *
 * [Java is a trademark or registered trademark of Sun Microsystems, Inc.
 * in the United States and other countries.]
 *
 * -----
 * SerialDateTest.java
 * -----
 * (C) Copyright 2001-2005, by Object Refinery Limited.
 *
 * Original Author: David Gilbert (for Object Refinery Limited);
 * Contributor(s): -;
 *
 * $Id: SerialDateTests.java,v 1.6 2005/11/16 15:58:40 taqua Exp $
 *
 * Changes
 * -----
 * 15-Nov-2001 : Version 1 (DG);
 * 25-Jun-2002 : Removed unnecessary import (DG);
 * 24-Oct-2002 : Fixed errors reported by Checkstyle (DG);
 * 13-Mar-2003 : Added serialization test (DG);
 * 05-Jan-2005 : Added test for bug report 1096282 (DG);
 *
 */
package org.jfree.date;
import java.io.ByteArrayInputStream;
import java.io.ByteArrayOutputStream;
import java.io.ObjectInput;
import java.io.ObjectInputStream;
import java.io.ObjectOutput;
import java.io.ObjectOutputStream;
import junit.framework.Test;
import junit.framework.TestCase;
import junit.framework.TestSuite;
import org.jfree.date.MonthConstants;
import org.jfree.date.SerialDate;
/**
 * Some JUnit tests for the {@link SerialDate} class.
 */
public class SerialDateTest extends TestCase {
```

```

/** Date representing November 9. */
private SerialDate nov9Y2001;
/**
 * Creates a new test case.
 *
 * @param name the name.
 */
public SerialDateTest(final String name) {
    super(name);
}
/**
 * Returns a test suite for the JUnit test runner.
 *
 * @return The test suite.
 */
public static Test suite() {
    return new TestSuite(SerialDateTest.class);
}
/**
 * Problem set up.
 */
protected void setUp() {
    this.nov9Y2001 = SerialDate.createInstance(9, MonthConstants.NOVEMBER, 2001);
}
/**
 * 9 Nov 2001 plus two months should be 9 Jan 2002.
 */
public void testAddMonthsTo9Nov2001() {
    final SerialDate jan9Y2002 = SerialDate.addMonths(2, this.nov9Y2001);
    final SerialDate answer = SerialDate.createInstance(9, 1, 2002);
    assertEquals(answer, jan9Y2002);
}
/**
 * A test case for a reported bug, now fixed.
 */
public void testAddMonthsTo5Oct2003() {
    final SerialDate d1 = SerialDate.createInstance(5, MonthConstants.OCTOBER, 2003);
    final SerialDate d2 = SerialDate.addMonths(2, d1);
    assertEquals(d2, SerialDate.createInstance(5, MonthConstants.DECEMBER, 2003));
}
/**
 * A test case for a reported bug, now fixed.
 */
public void testAddMonthsTo1Jan2003() {
    final SerialDate d1 = SerialDate.createInstance(1, MonthConstants.JANUARY, 2003);
    final SerialDate d2 = SerialDate.addMonths(0, d1);
    assertEquals(d2, d1);
}
/**
 * Monday preceding Friday 9 November 2001 should be 5 November.
 */
public void testMondayPrecedingFriday9Nov2001() {
    SerialDate mondayBefore = SerialDate.getPreviousDayOfWeek(
        SerialDate.MONDAY, this.nov9Y2001
    );
    assertEquals(5, mondayBefore.getDayOfMonth());
}
/**
 * Monday following Friday 9 November 2001 should be 12 November.
 */
public void testMondayFollowingFriday9Nov2001() {

```

```

    SerialDate mondayAfter = SerialDate.getFollowingDayOfWeek(
        SerialDate.MONDAY, this.nov9Y2001
    );
    assertEquals(12, mondayAfter.getDayOfMonth());
}
/**
 * Monday nearest Friday 9 November 2001 should be 12 November.
 */
public void testMondayNearestFriday9Nov2001() {
    SerialDate mondayNearest = SerialDate.getNearestDayOfWeek(
        SerialDate.MONDAY, this.nov9Y2001
    );
    assertEquals(12, mondayNearest.getDayOfMonth());
}
/**
 * The Monday nearest to 22nd January 1970 falls on the 19th.
 */
public void testMondayNearest22Jan1970() {
    SerialDate jan22Y1970 = SerialDate.createInstance(22, MonthConstants.JANUARY,
        1970);
    SerialDate mondayNearest = SerialDate.getNearestDayOfWeek(SerialDate.MONDAY,
        jan22Y1970);
    assertEquals(19, mondayNearest.getDayOfMonth());
}
/**
 * Problem that the conversion of days to strings returns the right result.
 * Actually, this
 * result depends on the Locale so this test needs to be modified.
 */
public void testWeekdayCodeToString() {
    final String test = SerialDate.weekdayCodeToString(SerialDate.SATURDAY);
    assertEquals("Saturday", test);
}
/**
 * Test the conversion of a string to a weekday. Note that this test will fail if
 * the
 * default locale doesn't use English weekday names...devise a better test!
 */
public void testStringToWeekday() {
    int weekday = SerialDate.stringToWeekdayCode("Wednesday");
    assertEquals(SerialDate.WEDNESDAY, weekday);
    weekday = SerialDate.stringToWeekdayCode(" Wednesday ");
    assertEquals(SerialDate.WEDNESDAY, weekday);
    weekday = SerialDate.stringToWeekdayCode("Wed");
    assertEquals(SerialDate.WEDNESDAY, weekday);
}
/**
 * Test the conversion of a string to a month. Note that this test will fail if the
 * default locale doesn't use English month names...devise a better test!
 */
public void testStringToMonthCode() {
    int m = SerialDate.stringToMonthCode("January");
    assertEquals(MonthConstants.JANUARY, m);
    m = SerialDate.stringToMonthCode(" January ");
    assertEquals(MonthConstants.JANUARY, m);
    m = SerialDate.stringToMonthCode("Jan");
    assertEquals(MonthConstants.JANUARY, m);
}
/**
 * Tests the conversion of a month code to a string.
 */

```

```

public void testMonthCodeToStringCode() {
    final String test = SerialDate.monthCodeToString(MonthConstants.DECEMBER);
    assertEquals("December", test);
}
/**
 * 1900 is not a leap year.
 */
public void testIsNotLeapYear1900() {
    assertTrue(!SerialDate.isLeapYear(1900));
}
/**
 * 2000 is a leap year.
 */
public void testIsLeapYear2000() {
    assertTrue(SerialDate.isLeapYear(2000));
}
/**
 * The number of leap years from 1900 up-to-and-including 1899 is 0.
 */
public void testLeapYearCount1899() {
    assertEquals(SerialDate.leapYearCount(1899), 0);
}
/**
 * The number of leap years from 1900 up-to-and-including 1903 is 0.
 */
public void testLeapYearCount1903() {
    assertEquals(SerialDate.leapYearCount(1903), 0);
}
/**
 * The number of leap years from 1900 up-to-and-including 1904 is 1.
 */
public void testLeapYearCount1904() {
    assertEquals(SerialDate.leapYearCount(1904), 1);
}
/**
 * The number of leap years from 1900 up-to-and-including 1999 is 24.
 */
public void testLeapYearCount1999() {
    assertEquals(SerialDate.leapYearCount(1999), 24);
}
/**
 * The number of leap years from 1900 up-to-and-including 2000 is 25.
 */
public void testLeapYearCount2000() {
    assertEquals(SerialDate.leapYearCount(2000), 25);
}
/**
 * Serialize an instance, restore it, and check for equality.
 */
public void testSerialization() {
    SerialDate d1 = SerialDate.createInstance(15, 4, 2000);
    SerialDate d2 = null;
    try {
        ByteArrayOutputStream buffer = new ByteArrayOutputStream();
        ObjectOutputStream out = new ObjectOutputStream(buffer);
        out.writeObject(d1);
        out.close();
        ObjectInputStream in = new ObjectInputStream(new
            ByteArrayInputStream(buffer.toByteArray()));
        d2 = (SerialDate) in.readObject();
        in.close();
    }
}

```

```

        }
        catch (Exception e) {
            System.out.println(e.toString());
        }
        assertEquals(d1, d2);
    }

    /**
     * A test for bug report 1096282 (now fixed).
     */
    public void test1096282() {
        SerialDate d = SerialDate.createInstance(29, 2, 2004);
        d = SerialDate.addYears(1, d);
        SerialDate expected = SerialDate.createInstance(28, 2, 2005);
        assertTrue(d.isOn(expected));
    }

    /**
     * Miscellaneous tests for the addMonths() method.
     */
    public void testAddMonths() {
        SerialDate d1 = SerialDate.createInstance(31, 5, 2004);

        SerialDate d2 = SerialDate.addMonths(1, d1);
        assertEquals(30, d2.getDayOfMonth());
        assertEquals(6, d2.getMonth());
        assertEquals(2004, d2.getYYYY());

        SerialDate d3 = SerialDate.addMonths(2, d1);
        assertEquals(31, d3.getDayOfMonth());
        assertEquals(7, d3.getMonth());
        assertEquals(2004, d3.getYYYY());

        SerialDate d4 = SerialDate.addMonths(1, SerialDate.addMonths(1, d1));
        assertEquals(30, d4.getDayOfMonth());
        assertEquals(7, d4.getMonth());
        assertEquals(2004, d4.getYYYY());
    }
}

```

Listato B.3 MonthConstants.java.

```

/*
=====
* JCommon : a free general purpose class library for the Java(tm) platform
* =====
*
* (C) Copyright 2000-2005, by Object Refinery Limited and Contributors.
*
* Project Info: http://www.jfree.org/jcommon/index.html
*
* This library is free software; you can redistribute it and/or modify it
* under the terms of the GNU Lesser General Public License as published by
* the Free Software Foundation; either version 2.1 of the License, or
* (at your option) any later version.
*
* This library is distributed in the hope that it will be useful, but
* WITHOUT ANY WARRANTY; without even the implied warranty of MERCHANTABILITY
* or FITNESS FOR A PARTICULAR PURPOSE. See the GNU Lesser General Public
* License for more details.
*
* You should have received a copy of the GNU Lesser General Public
* License along with this library; if not, write to the Free Software

```

```

* Foundation, Inc., 51 Franklin Street, Fifth Floor, Boston, MA 02110-1301,
* USA.
*
* [Java is a trademark or registered trademark of Sun Microsystems, Inc.
* in the United States and other countries.]
*
* -----
* MonthConstants.java
* -----
* (C) Copyright 2002, 2003, by Object Refinery Limited.
*
* Original Author: David Gilbert (for Object Refinery Limited);
* Contributor(s): -;
*
* $Id: MonthConstants.java,v 1.4 2005/11/16 15:58:40 taqua Exp $
*
* Changes
* -----
* 29-May-2002 : Version 1 (code moved from SerialDate class) (DG);
*
*/
package org.jfree.date;
/**
* Useful constants for months. Note that these are NOT equivalent to the
* constants defined by java.util.Calendar (where JANUARY=0 and DECEMBER=11).
* <P>
* Used by the SerialDate and RegularTimePeriod classes.
*
* @author David Gilbert
*/
public interface MonthConstants {
    /** Constant for January. */
    public static final int JANUARY = 1;
    /** Constant for February. */
    public static final int FEBRUARY = 2;
    /** Constant for March. */
    public static final int MARCH = 3;
    /** Constant for April. */
    public static final int APRIL = 4;
    /** Constant for May. */
    public static final int MAY = 5;
    /** Constant for June. */
    public static final int JUNE = 6;
    /** Constant for July. */
    public static final int JULY = 7;
    /** Constant for August. */
    public static final int AUGUST = 8;
    /** Constant for September. */
    public static final int SEPTEMBER = 9;
    /** Constant for October. */
    public static final int OCTOBER = 10;
    /** Constant for November. */
    public static final int NOVEMBER = 11;
    /** Constant for December. */
    public static final int DECEMBER = 12;
}

```

Listato B.4 BobsSerialDateTest.java.

```

package org.jfree.date.junit;
import junit.framework.TestCase;

```

```

import org.jfree.date.*;
import static org.jfree.date.SerialDate.*;
import java.util.*;
public class BobbsSerialDateTest extends TestCase {
    public void testIsValidWeekdayCode() throws Exception {
        for (int day = 1; day <= 7; day++)
            assertTrue(isValidWeekdayCode(day));
        assertFalse(isValidWeekdayCode(0));
        assertFalse(isValidWeekdayCode(8));
    }
    public void testStringToWeekdayCode() throws Exception {
        assertEquals(-1, stringToWeekdayCode("Hello"));
        assertEquals(MONDAY, stringToWeekdayCode("Monday"));
        assertEquals(MONDAY, stringToWeekdayCode("Mon"));
        //todo assertEquals(MONDAY, stringToWeekdayCode("monday"));
        // assertEquals(MONDAY, stringToWeekdayCode("MONDAY"));
        // assertEquals(MONDAY, stringToWeekdayCode("mon"));
        assertEquals(TUESDAY, stringToWeekdayCode("Tuesday"));
        assertEquals(TUESDAY, stringToWeekdayCode("Tue"));
        // assertEquals(TUESDAY, stringToWeekdayCode("tuesday"));
        // assertEquals(TUESDAY, stringToWeekdayCode("TUESDAY"));
        // assertEquals(TUESDAY, stringToWeekdayCode("tue"));
        // assertEquals(TUESDAY, stringToWeekdayCode("tues"));
        assertEquals(WEDNESDAY, stringToWeekdayCode("Wednesday"));
        assertEquals(WEDNESDAY, stringToWeekdayCode("Wed"));
        // assertEquals(WEDNESDAY, stringToWeekdayCode("wednesday"));
        // assertEquals(WEDNESDAY, stringToWeekdayCode("WEDNESDAY"));
        // assertEquals(WEDNESDAY, stringToWeekdayCode("wed"));
        assertEquals(THURSDAY, stringToWeekdayCode("Thursday"));
        assertEquals(THURSDAY, stringToWeekdayCode("Thu"));
        // assertEquals(THURSDAY, stringToWeekdayCode("thursday"));
        // assertEquals(THURSDAY, stringToWeekdayCode("THURSDAY"));
        // assertEquals(THURSDAY, stringToWeekdayCode("thu"));
        // assertEquals(THURSDAY, stringToWeekdayCode("thurs"));
        assertEquals(FRIDAY, stringToWeekdayCode("Friday"));
        assertEquals(FRIDAY, stringToWeekdayCode("Fri"));
        // assertEquals(FRIDAY, stringToWeekdayCode("friday"));
        // assertEquals(FRIDAY, stringToWeekdayCode("FRIDAY"));
        // assertEquals(FRIDAY, stringToWeekdayCode("fri"));
        assertEquals(SATURDAY, stringToWeekdayCode("Saturday"));
        assertEquals(SATURDAY, stringToWeekdayCode("Sat"));
        // assertEquals(SATURDAY, stringToWeekdayCode("saturday"));
        // assertEquals(SATURDAY, stringToWeekdayCode("SATURDAY"));
        // assertEquals(SATURDAY, stringToWeekdayCode("sat"));
        assertEquals(SUNDAY, stringToWeekdayCode("Sunday"));
        assertEquals(SUNDAY, stringToWeekdayCode("Sun"));
        // assertEquals(SUNDAY, stringToWeekdayCode("sunday"));
        // assertEquals(SUNDAY, stringToWeekdayCode("SUNDAY"));
        // assertEquals(SUNDAY, stringToWeekdayCode("sun"));
    }
    public void testWeekdayCodeToString() throws Exception {
        assertEquals("Sunday", weekdayCodeToString(SUNDAY));
        assertEquals("Monday", weekdayCodeToString(MONDAY));
        assertEquals("Tuesday", weekdayCodeToString(TUESDAY));
        assertEquals("Wednesday", weekdayCodeToString(WEDNESDAY));
        assertEquals("Thursday", weekdayCodeToString(THURSDAY));
        assertEquals("Friday", weekdayCodeToString(FRIDAY));
        assertEquals("Saturday", weekdayCodeToString(SATURDAY));
    }
    public void testIsValidMonthCode() throws Exception {
        for (int i = 1; i <= 12; i++)

```

```

assertTrue(isValidMonthCode(i));
assertFalse(isValidMonthCode(0));
assertFalse(isValidMonthCode(13));
}
public void testMonthToQuarter() throws Exception {
assertEquals(1, monthCodeToQuarter(JANUARY));
assertEquals(1, monthCodeToQuarter(FEBRUARY));
assertEquals(1, monthCodeToQuarter(MARCH));
assertEquals(2, monthCodeToQuarter(APRIL));
assertEquals(2, monthCodeToQuarter(MAY));
assertEquals(2, monthCodeToQuarter(JUNE));
assertEquals(3, monthCodeToQuarter(JULY));
assertEquals(3, monthCodeToQuarter(AUGUST));
assertEquals(3, monthCodeToQuarter(SEPTEMBER));
assertEquals(4, monthCodeToQuarter(OCTOBER));
assertEquals(4, monthCodeToQuarter(NOVEMBER));
assertEquals(4, monthCodeToQuarter(DECEMBER));
try {
monthCodeToQuarter(-1);
fail("Invalid Month Code should throw exception");
} catch (IllegalArgumentException e) {
}
}
public void testMonthCodeToString() throws Exception {
assertEquals("January", monthCodeToString(JANUARY));
assertEquals("February", monthCodeToString(FEBRUARY));
assertEquals("March", monthCodeToString(MARCH));
assertEquals("April", monthCodeToString(APRIL));
assertEquals("May", monthCodeToString(MAY));
assertEquals("June", monthCodeToString(JUNE));
assertEquals("July", monthCodeToString(JULY));
assertEquals("August", monthCodeToString(AUGUST));
assertEquals("September", monthCodeToString(SEPTEMBER));
assertEquals("October", monthCodeToString(OCTOBER));
assertEquals("November", monthCodeToString(NOVEMBER));
assertEquals("December", monthCodeToString(DECEMBER));
assertEquals("Jan", monthCodeToString(JANUARY, true));
assertEquals("Feb", monthCodeToString(FEBRUARY, true));
assertEquals("Mar", monthCodeToString(MARCH, true));
assertEquals("Apr", monthCodeToString(APRIL, true));
assertEquals("May", monthCodeToString(MAY, true));
assertEquals("Jun", monthCodeToString(JUNE, true));
assertEquals("Jul", monthCodeToString(JULY, true));
assertEquals("Aug", monthCodeToString(AUGUST, true));
assertEquals("Sep", monthCodeToString(SEPTEMBER, true));
assertEquals("Oct", monthCodeToString(OCTOBER, true));
assertEquals("Nov", monthCodeToString(NOVEMBER, true));
assertEquals("Dec", monthCodeToString(DECEMBER, true));
try {
monthCodeToString(-1);
fail("Invalid month code should throw exception");
} catch (IllegalArgumentException e) {
}
}
public void testStringToMonthCode() throws Exception {
assertEquals(JANUARY, stringToMonthCode("1"));
assertEquals(FEBRUARY, stringToMonthCode("2"));
assertEquals(MARCH, stringToMonthCode("3"));
assertEquals(APRIL, stringToMonthCode("4"));
assertEquals(MAY, stringToMonthCode("5"));
assertEquals(JUNE, stringToMonthCode("6"));
}

```

```

assertEquals(JULY, stringToMonthCode("7"));
assertEquals(AUGUST, stringToMonthCode("8"));
assertEquals(SEPTEMBER, stringToMonthCode("9"));
assertEquals(OCTOBER, stringToMonthCode("10"));
assertEquals(NOVEMBER, stringToMonthCode("11"));
assertEquals(DECEMBER, stringToMonthCode("12"));
//todo assertEquals(-1, stringToMonthCode("0"));
// assertEquals(-1, stringToMonthCode("13"));
assertEquals(-1, stringToMonthCode("Hello"));
for (int m = 1; m <= 12; m++) {
    assertEquals(m, stringToMonthCode(monthCodeToString(m, false)));
    assertEquals(m, stringToMonthCode(monthCodeToString(m, true)));
}
// assertEquals(1, stringToMonthCode("jan"));
// assertEquals(2, stringToMonthCode("feb"));
// assertEquals(3, stringToMonthCode("mar"));
// assertEquals(4, stringToMonthCode("apr"));
// assertEquals(5, stringToMonthCode("may"));
// assertEquals(6, stringToMonthCode("jun"));
// assertEquals(7, stringToMonthCode("jul"));
// assertEquals(8, stringToMonthCode("aug"));
// assertEquals(9, stringToMonthCode("sep"));
// assertEquals(10, stringToMonthCode("oct"));
// assertEquals(11, stringToMonthCode("nov"));
// assertEquals(12, stringToMonthCode("dec"));
// assertEquals(1, stringToMonthCode("JAN"));
// assertEquals(2, stringToMonthCode("FEB"));
// assertEquals(3, stringToMonthCode("MAR"));
// assertEquals(4, stringToMonthCode("APR"));
// assertEquals(5, stringToMonthCode("MAY"));
// assertEquals(6, stringToMonthCode("JUN"));
// assertEquals(7, stringToMonthCode("JUL"));
// assertEquals(8, stringToMonthCode("AUG"));
// assertEquals(9, stringToMonthCode("SEP"));
// assertEquals(10, stringToMonthCode("OCT"));
// assertEquals(11, stringToMonthCode("NOV"));
// assertEquals(12, stringToMonthCode("DEC"));
// assertEquals(1, stringToMonthCode("january"));
// assertEquals(2, stringToMonthCode("february"));
// assertEquals(3, stringToMonthCode("march"));
// assertEquals(4, stringToMonthCode("april"));
// assertEquals(5, stringToMonthCode("may"));
// assertEquals(6, stringToMonthCode("june"));
// assertEquals(7, stringToMonthCode("july"));
// assertEquals(8, stringToMonthCode("august"));
// assertEquals(9, stringToMonthCode("september"));
// assertEquals(10, stringToMonthCode("october"));
// assertEquals(11, stringToMonthCode("november"));
// assertEquals(12, stringToMonthCode("december"));
// assertEquals(1, stringToMonthCode("JANUARY"));
// assertEquals(2, stringToMonthCode("FEBRUARY"));
// assertEquals(3, stringToMonthCode("MAR"));
// assertEquals(4, stringToMonthCode("APRIL"));
// assertEquals(5, stringToMonthCode("MAY"));
// assertEquals(6, stringToMonthCode("JUNE"));
// assertEquals(7, stringToMonthCode("JULY"));
// assertEquals(8, stringToMonthCode("AUGUST"));
// assertEquals(9, stringToMonthCode("SEPTEMBER"));
// assertEquals(10, stringToMonthCode("OCTOBER"));
// assertEquals(11, stringToMonthCode("NOVEMBER"));
// assertEquals(12, stringToMonthCode("DECEMBER"));

```

```

}
public void testIsValidWeekInMonthCode() throws Exception {
for (int w = 0; w <= 4; w++) {
assertTrue(isValidWeekInMonthCode(w));
}
assertFalse(isValidWeekInMonthCode(5));
}
public void testIsLeapYear() throws Exception {
assertFalse(isLeapYear(1900));
assertFalse(isLeapYear(1901));
assertFalse(isLeapYear(1902));
assertFalse(isLeapYear(1903));
assertTrue(isLeapYear(1904));
assertTrue(isLeapYear(1908));
assertFalse(isLeapYear(1955));
assertTrue(isLeapYear(1964));
assertTrue(isLeapYear(1980));
assertTrue(isLeapYear(2000));
assertFalse(isLeapYear(2001));
assertFalse(isLeapYear(2100));
}
public void testLeapYearCount() throws Exception {
assertEquals(0, leapYearCount(1900));
assertEquals(0, leapYearCount(1901));
assertEquals(0, leapYearCount(1902));
assertEquals(0, leapYearCount(1903));
assertEquals(1, leapYearCount(1904));
assertEquals(1, leapYearCount(1905));
assertEquals(1, leapYearCount(1906));
assertEquals(1, leapYearCount(1907));
assertEquals(2, leapYearCount(1908));
assertEquals(24, leapYearCount(1999));
assertEquals(25, leapYearCount(2001));
assertEquals(49, leapYearCount(2101));
assertEquals(73, leapYearCount(2201));
assertEquals(97, leapYearCount(2301));
assertEquals(122, leapYearCount(2401));
}
public void testLastDayOfMonth() throws Exception {
assertEquals(31, lastDayOfMonth(JANUARY, 1901));
assertEquals(28, lastDayOfMonth(FEBRUARY, 1901));
assertEquals(31, lastDayOfMonth(MARCH, 1901));
assertEquals(30, lastDayOfMonth(APRIL, 1901));
assertEquals(31, lastDayOfMonth(MAY, 1901));
assertEquals(30, lastDayOfMonth(JUNE, 1901));
assertEquals(31, lastDayOfMonth(JULY, 1901));
assertEquals(31, lastDayOfMonth(AUGUST, 1901));
assertEquals(30, lastDayOfMonth(SEPTMBER, 1901));
assertEquals(31, lastDayOfMonth(OCTOBER, 1901));
assertEquals(30, lastDayOfMonth(NOVEMBER, 1901));
assertEquals(31, lastDayOfMonth(DECEMBER, 1901));
assertEquals(29, lastDayOfMonth(FEBRUARY, 1904));
}
public void testAddDays() throws Exception {
SerialDate newYears = d(1, JANUARY, 1900);
assertEquals(d(2, JANUARY, 1900), addDays(1, newYears));
assertEquals(d(1, FEBRUARY, 1900), addDays(31, newYears));
assertEquals(d(1, JANUARY, 1901), addDays(365, newYears));
assertEquals(d(31, DECEMBER, 1904), addDays(5 * 365, newYears));
}
private static SpreadsheetDate d(int day, int month, int year) { return new

```

```

SpreadsheetDate(day, month, year);}
public void testAddMonths() throws Exception {
assertEquals(d(1, FEBRUARY, 1900), addMonths(1, d(1, JANUARY, 1900)));
assertEquals(d(28, FEBRUARY, 1900), addMonths(1, d(31, JANUARY, 1900)));
assertEquals(d(28, FEBRUARY, 1900), addMonths(1, d(30, JANUARY, 1900)));
assertEquals(d(28, FEBRUARY, 1900), addMonths(1, d(29, JANUARY, 1900)));
assertEquals(d(28, FEBRUARY, 1900), addMonths(1, d(28, JANUARY, 1900)));
assertEquals(d(27, FEBRUARY, 1900), addMonths(1, d(27, JANUARY, 1900)));
assertEquals(d(30, JUNE, 1900), addMonths(5, d(31, JANUARY, 1900)));
assertEquals(d(30, JUNE, 1901), addMonths(17, d(31, JANUARY, 1900)));
assertEquals(d(29, FEBRUARY, 1904), addMonths(49, d(31, JANUARY, 1900)));
}
public void testAddYears() throws Exception {
assertEquals(d(1, JANUARY, 1901), addYears(1, d(1, JANUARY, 1900)));
assertEquals(d(28, FEBRUARY, 1905), addYears(1, d(29, FEBRUARY, 1904)));
assertEquals(d(28, FEBRUARY, 1905), addYears(1, d(28, FEBRUARY, 1904)));
assertEquals(d(28, FEBRUARY, 1904), addYears(1, d(28, FEBRUARY, 1903)));
}
public void testGetPreviousDayOfWeek() throws Exception {
assertEquals(d(24, FEBRUARY, 2006), getPreviousDayOfWeek(FRIDAY, d(1, MARCH,
2006)));
assertEquals(d(22, FEBRUARY, 2006), getPreviousDayOfWeek(WEDNESDAY, d(1, MARCH,
2006)));
assertEquals(d(29, FEBRUARY, 2004), getPreviousDayOfWeek(SUNDAY, d(3, MARCH,
2004)));
assertEquals(d(29, DECEMBER, 2004), getPreviousDayOfWeek(WEDNESDAY, d(5, JANUARY,
2005)));
try {
getPreviousDayOfWeek(-1, d(1, JANUARY, 2006));
fail("Invalid day of week code should throw exception");
} catch (IllegalArgumentException e) {
}
}
public void testGetFollowingDayOfWeek() throws Exception {
// assertEquals(d(1, JANUARY, 2005), getFollowingDayOfWeek(SATURDAY, d(25,
DECEMBER, 2004)));
assertEquals(d(1, JANUARY, 2005), getFollowingDayOfWeek(SATURDAY, d(26, DECEMBER,
2004)));
assertEquals(d(3, MARCH, 2004), getFollowingDayOfWeek(WEDNESDAY, d(28, FEBRUARY,
2004)));
try {
getFollowingDayOfWeek(-1, d(1, JANUARY, 2006));
fail("Invalid day of week code should throw exception");
} catch (IllegalArgumentException e) {
}
}
public void testGetNearestDayOfWeek() throws Exception {
assertEquals(d(16, APRIL, 2006), getNearestDayOfWeek(SUNDAY, d(16, APRIL, 2006)));
assertEquals(d(16, APRIL, 2006), getNearestDayOfWeek(SUNDAY, d(17, APRIL, 2006)));
assertEquals(d(16, APRIL, 2006), getNearestDayOfWeek(SUNDAY, d(18, APRIL, 2006)));
assertEquals(d(16, APRIL, 2006), getNearestDayOfWeek(SUNDAY, d(19, APRIL, 2006)));
assertEquals(d(23, APRIL, 2006), getNearestDayOfWeek(SUNDAY, d(20, APRIL, 2006)));
assertEquals(d(23, APRIL, 2006), getNearestDayOfWeek(SUNDAY, d(21, APRIL, 2006)));
assertEquals(d(23, APRIL, 2006), getNearestDayOfWeek(SUNDAY, d(22, APRIL, 2006)));

//todo assertEquals(d(17, APRIL, 2006), getNearestDayOfWeek(MONDAY, d(16, APRIL,
2006)));
assertEquals(d(17, APRIL, 2006), getNearestDayOfWeek(MONDAY, d(17, APRIL, 2006)));
assertEquals(d(17, APRIL, 2006), getNearestDayOfWeek(MONDAY, d(18, APRIL, 2006)));
assertEquals(d(17, APRIL, 2006), getNearestDayOfWeek(MONDAY, d(19, APRIL, 2006)));
assertEquals(d(17, APRIL, 2006), getNearestDayOfWeek(MONDAY, d(20, APRIL, 2006)));
}

```

```

assertEquals(d(24, APRIL, 2006), getNearestDayOfWeek(MONDAY, d(21, APRIL, 2006)));
assertEquals(d(24, APRIL, 2006), getNearestDayOfWeek(MONDAY, d(22, APRIL, 2006)));

// assertEquals(d(18, APRIL, 2006), getNearestDayOfWeek(TUESDAY, d(16, APRIL,
2006)));
// assertEquals(d(18, APRIL, 2006), getNearestDayOfWeek(TUESDAY, d(17, APRIL,
2006)));
assertEquals(d(18, APRIL, 2006), getNearestDayOfWeek(TUESDAY, d(18, APRIL,
2006)));
assertEquals(d(18, APRIL, 2006), getNearestDayOfWeek(TUESDAY, d(19, APRIL,
2006)));
assertEquals(d(18, APRIL, 2006), getNearestDayOfWeek(TUESDAY, d(20, APRIL,
2006)));
assertEquals(d(18, APRIL, 2006), getNearestDayOfWeek(TUESDAY, d(21, APRIL,
2006)));
assertEquals(d(25, APRIL, 2006), getNearestDayOfWeek(TUESDAY, d(22, APRIL,
2006)));
// assertEquals(d(19, APRIL, 2006), getNearestDayOfWeek(WEDNESDAY, d(16, APRIL,
2006)));
// assertEquals(d(19, APRIL, 2006), getNearestDayOfWeek(WEDNESDAY, d(17, APRIL,
2006)));
// assertEquals(d(19, APRIL, 2006), getNearestDayOfWeek(WEDNESDAY, d(18, APRIL,
2006)));
assertEquals(d(19, APRIL, 2006), getNearestDayOfWeek(WEDNESDAY, d(19, APRIL,
2006)));
assertEquals(d(19, APRIL, 2006), getNearestDayOfWeek(WEDNESDAY, d(20, APRIL,
2006)));
assertEquals(d(19, APRIL, 2006), getNearestDayOfWeek(WEDNESDAY, d(21, APRIL,
2006)));
assertEquals(d(19, APRIL, 2006), getNearestDayOfWeek(WEDNESDAY, d(22, APRIL,
2006)));
// assertEquals(d(13, APRIL, 2006), getNearestDayOfWeek(THURSDAY, d(16, APRIL,
2006)));
// assertEquals(d(20, APRIL, 2006), getNearestDayOfWeek(THURSDAY, d(17, APRIL,
2006)));
// assertEquals(d(20, APRIL, 2006), getNearestDayOfWeek(THURSDAY, d(18, APRIL,
2006)));
// assertEquals(d(20, APRIL, 2006), getNearestDayOfWeek(THURSDAY, d(19, APRIL,
2006)));
assertEquals(d(20, APRIL, 2006), getNearestDayOfWeek(THURSDAY, d(20, APRIL,
2006)));
assertEquals(d(20, APRIL, 2006), getNearestDayOfWeek(THURSDAY, d(21, APRIL,
2006)));
assertEquals(d(20, APRIL, 2006), getNearestDayOfWeek(THURSDAY, d(22, APRIL,
2006)));
// assertEquals(d(14, APRIL, 2006), getNearestDayOfWeek(FRIDAY, d(16, APRIL,
2006)));
// assertEquals(d(14, APRIL, 2006), getNearestDayOfWeek(FRIDAY, d(17, APRIL,
2006)));
// assertEquals(d(21, APRIL, 2006), getNearestDayOfWeek(FRIDAY, d(18, APRIL,
2006)));
// assertEquals(d(21, APRIL, 2006), getNearestDayOfWeek(FRIDAY, d(19, APRIL,
2006)));
// assertEquals(d(21, APRIL, 2006), getNearestDayOfWeek(FRIDAY, d(20, APRIL,
2006)));
assertEquals(d(21, APRIL, 2006), getNearestDayOfWeek(FRIDAY, d(21, APRIL, 2006)));
assertEquals(d(21, APRIL, 2006), getNearestDayOfWeek(FRIDAY, d(22, APRIL, 2006)));
// assertEquals(d(15, APRIL, 2006), getNearestDayOfWeek(SATURDAY, d(16, APRIL,
2006)));
// assertEquals(d(15, APRIL, 2006), getNearestDayOfWeek(SATURDAY, d(17, APRIL,
2006)));

```

```

// assertEquals(d(15, APRIL, 2006), getNearestDayOfWeek(SATURDAY, d(18, APRIL,
2006)));
// assertEquals(d(22, APRIL, 2006), getNearestDayOfWeek(SATURDAY, d(19, APRIL,
2006)));
// assertEquals(d(22, APRIL, 2006), getNearestDayOfWeek(SATURDAY, d(20, APRIL,
2006)));
// assertEquals(d(22, APRIL, 2006), getNearestDayOfWeek(SATURDAY, d(21, APRIL,
2006)));
assertEquals(d(22, APRIL, 2006), getNearestDayOfWeek(SATURDAY, d(22, APRIL,
2006)));
try {
getNearestDayOfWeek(-1, d(1, JANUARY, 2006));
fail("Invalid day of week code should throw exception");
} catch (IllegalArgumentException e) {
}
}
public void testEndOfCurrentMonth() throws Exception {
SerialDate d = SerialDate.createInstance(2);
assertEquals(d(31, JANUARY, 2006), d.getEndOfCurrentMonth(d(1, JANUARY, 2006)));
assertEquals(d(28, FEBRUARY, 2006), d.getEndOfCurrentMonth(d(1, FEBRUARY, 2006)));
assertEquals(d(31, MARCH, 2006), d.getEndOfCurrentMonth(d(1, MARCH, 2006)));
assertEquals(d(30, APRIL, 2006), d.getEndOfCurrentMonth(d(1, APRIL, 2006)));
assertEquals(d(31, MAY, 2006), d.getEndOfCurrentMonth(d(1, MAY, 2006)));
assertEquals(d(30, JUNE, 2006), d.getEndOfCurrentMonth(d(1, JUNE, 2006)));
assertEquals(d(31, JULY, 2006), d.getEndOfCurrentMonth(d(1, JULY, 2006)));
assertEquals(d(31, AUGUST, 2006), d.getEndOfCurrentMonth(d(1, AUGUST, 2006)));
assertEquals(d(30, SEPTEMBER, 2006), d.getEndOfCurrentMonth(d(1, SEPTEMBER,
2006)));
assertEquals(d(31, OCTOBER, 2006), d.getEndOfCurrentMonth(d(1, OCTOBER, 2006)));
assertEquals(d(30, NOVEMBER, 2006), d.getEndOfCurrentMonth(d(1, NOVEMBER, 2006)));
assertEquals(d(31, DECEMBER, 2006), d.getEndOfCurrentMonth(d(1, DECEMBER, 2006)));
assertEquals(d(29, FEBRUARY, 2008), d.getEndOfCurrentMonth(d(1, FEBRUARY, 2008)));
}
public void testWeekInMonthToString() throws Exception {
assertEquals("First", weekInMonthToString(FIRST_WEEK_IN_MONTH));
assertEquals("Second", weekInMonthToString(SECOND_WEEK_IN_MONTH));
assertEquals("Third", weekInMonthToString(THIRD_WEEK_IN_MONTH));
assertEquals("Fourth", weekInMonthToString(FOURTH_WEEK_IN_MONTH));
assertEquals("Last", weekInMonthToString(LAST_WEEK_IN_MONTH));
//todo try {
// weekInMonthToString(-1);
// fail("Invalid week code should throw exception");
// } catch (IllegalArgumentException e) {
// }
}
public void testRelativeToString() throws Exception {
assertEquals("Preceding", relativeToString(PRECEDING));
assertEquals("Nearest", relativeToString(NEAREST));
assertEquals("Following", relativeToString(FOLLOWING));
//todo try {
// relativeToString(-1000);
// fail("Invalid relative code should throw exception");
// } catch (IllegalArgumentException e) {
// }
}
public void testCreateInstanceFromDDMMYYYY() throws Exception {
SerialDate date = createInstance(1, JANUARY, 1900);
assertEquals(1, date.getDayOfMonth());
assertEquals(JANUARY, date.getMonth());
assertEquals(1900, date.getYYYY());
assertEquals(2, date.toSerial());
}

```

```

}
public void testCreateInstanceFromSerial() throws Exception {
assertEquals(d(1, JANUARY, 1900),createInstance(2));
assertEquals(d(1, JANUARY, 1901), createInstance(367));
}
public void testCreateInstanceFromJavaDate() throws Exception {
assertEquals(d(1, JANUARY, 1900), createInstance(new
GregorianCalendar(1900,0,1).getTime()));
assertEquals(d(1, JANUARY, 2006), createInstance(new
GregorianCalendar(2006,0,1).getTime()));
}
public static void main(String[] args) {
junit.textui.TestRunner.run(BobsSerialDateTest.class);
}
}

```

Listato B.5 SpreadsheetDate.java.

```

/*
=====
* JCommon : a free general purpose class library for the Java(tm) platform
* =====
*
* (C) Copyright 2000-2005, by Object Refinery Limited and Contributors.
*
* Project Info: http://www.jfree.org/jcommon/index.html
*
* This library is free software; you can redistribute it and/or modify it
* under the terms of the GNU Lesser General Public License as published by
* the Free Software Foundation; either version 2.1 of the License, or
* (at your option) any later version.
*
* This library is distributed in the hope that it will be useful, but
* WITHOUT ANY WARRANTY; without even the implied warranty of MERCHANTABILITY
* or FITNESS FOR A PARTICULAR PURPOSE. See the GNU Lesser General Public
* License for more details.
*
* You should have received a copy of the GNU Lesser General Public
* License along with this library; if not, write to the Free Software
* Foundation, Inc., 51 Franklin Street, Fifth Floor, Boston, MA 02110-1301,
* USA.
*
* [Java is a trademark or registered trademark of Sun Microsystems, Inc.
* in the United States and other countries.]
*
-----
* SpreadsheetDate.java
* -----
*
* (C) Copyright 2000-2005, by Object Refinery Limited and Contributors.
*
* Original Author: David Gilbert (for Object Refinery Limited);
* Contributor(s): -;
*
* $Id: SpreadsheetDate.java,v 1.8 2005/11/03 09:25:39 mungady Exp $
*
* Changes
* -----
* 11-Oct-2001 : Version 1 (DG);
* 05-Nov-2001 : Added getDescription() and setDescription() methods (DG);
* 12-Nov-2001 : Changed name from ExcelDate.java to SpreadsheetDate.java (DG);
* Fixed a bug in calculating day, month and year from serial
* number (DG);

```

```

* 24-Jan-2002 : Fixed a bug in calculating the serial number from the day,
* month and year. Thanks to Trevor Hills for the report (DG);
* 29-May-2002 : Added equals(Object) method (SourceForge ID 558850) (DG);
* 03-Oct-2002 : Fixed errors reported by Checkstyle (DG);
* 13-Mar-2003 : Implemented Serializable (DG);
* 04-Sep-2003 : Completed isInRange() methods (DG);
* 05-Sep-2003 : Implemented Comparable (DG);
* 21-Oct-2003 : Added hashCode() method (DG);
*
*/
package org.jfree.date;
import java.util.Calendar;
import java.util.Date;
/**
* Represents a date using an integer, in a similar fashion to the
* implementation in Microsoft Excel. The range of dates supported is
* 1-Jan-1900 to 31-Dec-9999.
* <P>
* Be aware that there is a deliberate bug in Excel that recognises the year
* 1900 as a leap year when in fact it is not a leap year. You can find more
* information on the Microsoft website in article Q181370:
* <P>
* http://support.microsoft.com/support/kb/articles/Q181/3/70.asp
* <P>
* Excel uses the convention that 1-Jan-1900 = 1. This class uses the
* convention 1-Jan-1900 = 2.
* The result is that the day number in this class will be different to the
* Excel figure for January and February 1900...but then Excel adds in an extra
* day (29-Feb-1900 which does not actually exist!) and from that point forward
* the day numbers will match.
*
* @author David Gilbert
*/
public class SpreadsheetDate extends SerialDate {
/** For serialization. */
private static final long serialVersionUID = -2039586705374454461L;
/**
* The day number (1-Jan-1900 = 2, 2-Jan-1900 = 3, ..., 31-Dec-9999 =
* 2958465).
*/
private int serial;
/** The day of the month (1 to 28, 29, 30 or 31 depending on the month). */
private int day;
/** The month of the year (1 to 12). */
private int month;
/** The year (1900 to 9999). */
private int year;
/** An optional description for the date. */
private String description;
/**
* Creates a new date instance.
*
* @param day the day (in the range 1 to 28/29/30/31).
* @param month the month (in the range 1 to 12).
* @param year the year (in the range 1900 to 9999).
*/
public SpreadsheetDate(final int day, final int month, final int year) {
if ((year >= 1900) && (year <= 9999)) {
this.year = year;
}
else {

```

```

throw new IllegalArgumentException(
"The 'year' argument must be in range 1900 to 9999."
);
}
if ((month >= MonthConstants.JANUARY)
&& (month <= MonthConstants.DECEMBER)) {
this.month = month;
}
else {
throw new IllegalArgumentException(
"The 'month' argument must be in the range 1 to 12."
);
}
if ((day >= 1) && (day <= SerialDate.lastDayOfMonth(month, year))) {
this.day = day;
}
else {
throw new IllegalArgumentException("Invalid 'day' argument.");
}
// the serial number needs to be synchronised with the day-month-year...
this.serial = calcSerial(day, month, year);
this.description = null;
}
/**
* Standard constructor - creates a new date object representing the
* specified day number (which should be in the range 2 to 2958465.
*
* @param serial the serial number for the day (range: 2 to 2958465).
*/
public SpreadsheetDate(final int serial) {
if ((serial >= SERIAL_LOWER_BOUND) && (serial <= SERIAL_UPPER_BOUND)) {
this.serial = serial;
}
else {
throw new IllegalArgumentException(
"SpreadsheetDate: Serial must be in range 2 to 2958465.");
}
// the day-month-year needs to be synchronised with the serial number...
calcDayMonthYear();
}
/**
* Returns the description that is attached to the date. It is not
* required that a date have a description, but for some applications it
* is useful.
*
* @return The description that is attached to the date.
*/
public String getDescription() {
return this.description;
}
/**
* Sets the description for the date.
*
* @param description the description for this date (<code>null</code>
* permitted).
*/
public void setDescription(final String description) {
this.description = description;
}
/**
* Returns the serial number for the date, where 1 January 1900 = 2

```

```

* (this corresponds, almost, to the numbering system used in Microsoft
* Excel for Windows and Lotus 1-2-3).
*
* @return The serial number of this date.
*/
public int toSerial() {
    return this.serial;
}
/***
* Returns a <code>java.util.Date</code> equivalent to this date.
*
* @return The date.
*/
public Date toDate() {
    final Calendar calendar = Calendar.getInstance();
    calendar.set(getYYYY(), getMonth() - 1, getDayOfMonth(), 0, 0, 0);
    return calendar.getTime();
}
/***
* Returns the year (assume a valid range of 1900 to 9999).
*
* @return The year.
*/
public int getYYYY() {
    return this.year;
}
/***
* Returns the month (January = 1, February = 2, March = 3).
*
* @return The month of the year.
*/
public int getMonth() {
    return this.month;
}
/***
* Returns the day of the month.
*
* @return The day of the month.
*/
public int getDayOfMonth() {
    return this.day;
}
/***
* Returns a code representing the day of the week.
* <P>
* The codes are defined in the {@link SerialDate} class as:
* <code>SUNDAY</code>, <code>MONDAY</code>, <code>TUESDAY</code>,
* <code>WEDNESDAY</code>, <code>THURSDAY</code>, <code>FRIDAY</code>, and
* <code>SATURDAY</code>.
*
* @return A code representing the day of the week.
*/
public int getDayOfWeek() {
    return (this.serial + 6) % 7 + 1;
}
/***
* Tests the equality of this date with an arbitrary object.
* <P>
* This method will return true ONLY if the object is an instance of the
* {@link SerialDate} base class, and it represents the same day as this
* {@link SpreadsheetDate}.

```

```

*
* @param object the object to compare (<code>null</code> permitted).
*
* @return A boolean.
*/
public boolean equals(final Object object) {
if (object instanceof SerialDate) {
final SerialDate s = (SerialDate) object;
return (s.toSerial() == this.toSerial());
}
else {
return false;
}
}
/***
* Returns a hash code for this object instance.
*
* @return A hash code.
*/
public int hashCode() {
return toSerial();
}
/***
* Returns the difference (in days) between this date and the specified
* 'other' date.
*
* @param other the date being compared to.
*
* @return The difference (in days) between this date and the specified
* 'other' date.
*/
public int compare(final SerialDate other) {
return this.serial - other.toSerial();
}
/***
* Implements the method required by the Comparable interface.
*
* @param other the other object (usually another SerialDate).
*
* @return A negative integer, zero, or a positive integer as this object
* is less than, equal to, or greater than the specified object.
*/
public int compareTo(final Object other) {
return compare((SerialDate) other);
}
/***
* Returns true if this SerialDate represents the same date as the
* specified SerialDate.
*
* @param other the date being compared to.
*
* @return <code>true</code> if this SerialDate represents the same date as
* the specified SerialDate.
*/
public boolean isOn(final SerialDate other) {
return (this.serial == other.toSerial());
}
/***
* Returns true if this SerialDate represents an earlier date compared to
* the specified SerialDate.
*

```

```

* @param other the date being compared to.
*
* @return <code>true</code> if this SerialDate represents an earlier date
* compared to the specified SerialDate.
*/
public boolean isBefore(final SerialDate other) {
    return (this.serial < other.toSerial());
}
/***
* Returns true if this SerialDate represents the same date as the
* specified SerialDate.
*
* @param other the date being compared to.
*
* @return <code>true</code> if this SerialDate represents the same date
* as the specified SerialDate.
*/
public boolean isOnOrBefore(final SerialDate other) {
    return (this.serial <= other.toSerial());
}
/***
* Returns true if this SerialDate represents the same date as the
* specified SerialDate.
*
* @param other the date being compared to.
*
* @return <code>true</code> if this SerialDate represents the same date
* as the specified SerialDate.
*/
public boolean isAfter(final SerialDate other) {
    return (this.serial > other.toSerial());
}
/***
* Returns true if this SerialDate represents the same date as the
* specified SerialDate.
*
* @param other the date being compared to.
*
* @return <code>true</code> if this SerialDate represents the same date as
* the specified SerialDate.
*/
public boolean isOnOrAfter(final SerialDate other) {
    return (this.serial >= other.toSerial());
}
/***
* Returns <code>true</code> if this {@link SerialDate} is within the
* specified range (INCLUSIVE). The date order of d1 and d2 is not
* important.
*
* @param d1 a boundary date for the range.
* @param d2 the other boundary date for the range.
*
* @return A boolean.
*/
public boolean isInRange(final SerialDate d1, final SerialDate d2) {
    return isInRange(d1, d2, SerialDate.INCLUDE_BOTH);
}
/***
* Returns true if this SerialDate is within the specified range (caller
* specifies whether or not the end-points are included). The order of d1
* and d2 is not important.

```

```

*
* @param d1 one boundary date for the range.
* @param d2 a second boundary date for the range.
* @param include a code that controls whether or not the start and end
* dates are included in the range.
*
* @return <code>true</code> if this SerialDate is within the specified
* range.
*/
public boolean isInRange(final SerialDate d1, final SerialDate d2,
final int include) {
final int s1 = d1.toSerial();
final int s2 = d2.toSerial();
final int start = Math.min(s1, s2);
final int end = Math.max(s1, s2);
final int s = toSerial();
if (include == SerialDate.INCLUDE_BOTH) {
return (s >= start && s <= end);
}
else if (include == SerialDate.INCLUDE_FIRST) {
return (s >= start && s < end);
}
else if (include == SerialDate.INCLUDE_SECOND) {
return (s > start && s <= end);
}
else {
return (s > start && s < end);
}
}
/***
* Calculate the serial number from the day, month and year.
* <P>
* 1-Jan-1900 = 2.
*
* @param d the day.
* @param m the month.
* @param y the year.
*
* @return the serial number from the day, month and year.
*/
private int calcSerial(final int d, final int m, final int y) {
final int yy = ((y - 1900) * 365) + SerialDate.leapYearCount(y - 1);
int mm = SerialDate.AGGREGATE_DAYS_TO_END_OF_PRECEDING_MONTH[m];
if (m > MonthConstants.FEBRUARY) {
if (SerialDate.isLeapYear(y)) {
mm = mm + 1;
}
}
final int dd = d;
return yy + mm + dd + 1;
}
/***
* Calculate the day, month and year from the serial number.
*/
private void calcDayMonthYear() {
// get the year from the serial date
final int days = this.serial - SERIAL_LOWER_BOUND;
// overestimated because we ignored leap days
final int overestimatedYYYY = 1900 + (days / 365);
final int leaps = SerialDate.leapYearCount(overestimatedYYYY);
final int nonleapdays = days - leaps;
}

```

```

// underestimated because we overestimated years
int underestimatedYYYY = 1900 + (nonleapdays / 365);
if (underestimatedYYYY == overestimatedYYYY) {
    this.year = underestimatedYYYY;
}
else {
    int ss1 = calcSerial(1, 1, underestimatedYYYY);
    while (ss1 <= this.serial) {
        underestimatedYYYY = underestimatedYYYY + 1;
        ss1 = calcSerial(1, 1, underestimatedYYYY);
    }
    this.year = underestimatedYYYY - 1;
}
final int ss2 = calcSerial(1, 1, this.year);
int[] daysToEndOfPrecedingMonth
= AGGREGATE_DAYS_TO_END_OF_PRECEDING_MONTH;
if (isLeapYear(this.year)) {
    daysToEndOfPrecedingMonth
= LEAP_YEAR_AGGREGATE_DAYS_TO_END_OF_PRECEDING_MONTH;
}
// get the month from the serial date
int mm = 1;
int sss = ss2 + daysToEndOfPrecedingMonth[mm] - 1;
while (sss < this.serial) {
    mm = mm + 1;
    sss = ss2 + daysToEndOfPrecedingMonth[mm] - 1;
}
this.month = mm - 1;
// what's left is d(+1);
this.day = this.serial - ss2
- daysToEndOfPrecedingMonth[this.month] + 1;
}
}

```

Listato B.6 RelativeDayOfWeekRule.java.

```

/*
=====
* JCommon : a free general purpose class library for the Java(tm) platform
* =====
*
* (C) Copyright 2000-2005, by Object Refinery Limited and Contributors.
*
* Project Info: http://www.jfree.org/jcommon/index.html
*
* This library is free software; you can redistribute it and/or modify it
* under the terms of the GNU Lesser General Public License as published by
* the Free Software Foundation; either version 2.1 of the License, or
* (at your option) any later version.
*
* This library is distributed in the hope that it will be useful, but
* WITHOUT ANY WARRANTY; without even the implied warranty of MERCHANTABILITY
* or FITNESS FOR A PARTICULAR PURPOSE. See the GNU Lesser General Public
* License for more details.
*
* You should have received a copy of the GNU Lesser General Public
* License along with this library; if not, write to the Free Software
* Foundation, Inc., 51 Franklin Street, Fifth Floor, Boston, MA 02110-1301,
* USA.
*
* [Java is a trademark or registered trademark of Sun Microsystems, Inc.
* in the United States and other countries.]

```

```

*
* -----
* RelativeDayOfWeekRule.java
* -----
* (C) Copyright 2000-2003, by Object Refinery Limited and Contributors.
*
* Original Author: David Gilbert (for Object Refinery Limited);
* Contributor(s): -;
*
* $Id: RelativeDayOfWeekRule.java,v 1.6 2005/11/16 15:58:40 taqua Exp $
*
* Changes (from 26-Oct-2001)
* -----
* 26-Oct-2001 : Changed package to com.jrefinery.date.*;
* 03-Oct-2002 : Fixed errors reported by Checkstyle (DG);
*
*/
package org.jfree.date;
/**
* An annual date rule that returns a date for each year based on (a) a
* reference rule; (b) a day of the week; and (c) a selection parameter
* (SerialDate.PRECEDING, SerialDate.NEAREST, SerialDate.FOLLOWING).
* <P>
* For example, Good Friday can be specified as 'the Friday PRECEDING Easter
* Sunday'.
*
* @author David Gilbert
*/
public class RelativeDayOfWeekRule extends AnnualDateRule {
    /**
     * A reference to the annual date rule on which this rule is based.
     */
    private AnnualDateRule subrule;
    /**
     * The day of the week (SerialDate.MONDAY, SerialDate.TUESDAY, and so on).
     */
    private int dayOfWeek;
    /**
     * Specifies which day of the week (PRECEDING, NEAREST or FOLLOWING).
     */
    private int relative;
    /**
     * Default constructor - builds a rule for the Monday following 1 January.
     */
    public RelativeDayOfWeekRule() {
        this(new DayAndMonthRule(), SerialDate.MONDAY, SerialDate.FOLLOWING);
    }
    /**
     * Standard constructor - builds rule based on the supplied sub-rule.
     *
     * @param subrule the rule that determines the reference date.
     * @param dayOfWeek the day-of-the-week relative to the reference date.
     * @param relative indicates *which* day-of-the-week (preceding, nearest
     * or following).
     */
    public RelativeDayOfWeekRule(final AnnualDateRule subrule,
        final int dayOfWeek, final int relative) {
        this.subrule = subrule;
        this.dayOfWeek = dayOfWeek;
        this.relative = relative;
    }
    /**
     * Returns the sub-rule (also called the reference rule).
     *
     * @return The annual date rule that determines the reference date for this

```

```

* rule.
*/
public AnnualDateRule getSubrule() {
    return this.subrule;
}
/**
 * Sets the sub-rule.
 *
 * @param subrule the annual date rule that determines the reference date
 * for this rule.
 */
public void setSubrule(final AnnualDateRule subrule) {
    this.subrule = subrule;
}
/**
 * Returns the day-of-the-week for this rule.
 *
 * @return the day-of-the-week for this rule.
 */
public int getDayOfWeek() {
    return this.dayOfWeek;
}
/**
 * Sets the day-of-the-week for this rule.
 *
 * @param dayOfWeek the day-of-the-week (SerialDate.MONDAY,
 * SerialDate.TUESDAY, and so on).
 */
public void setDayOfWeek(final int dayOfWeek) {
    this.dayOfWeek = dayOfWeek;
}
/**
 * Returns the 'relative' attribute, that determines *which*
 * day-of-the-week we are interested in (SerialDate.PRECEDING,
 * SerialDate.NEAREST or SerialDate.FOLLOWING).
 *
 * @return The 'relative' attribute.
 */
public int getRelative() {
    return this.relative;
}
/**
 * Sets the 'relative' attribute (SerialDate.PRECEDING, SerialDate.NEAREST,
 * SerialDate.FOLLOWING).
 *
 * @param relative determines *which* day-of-the-week is selected by this
 * rule.
 */
public void setRelative(final int relative) {
    this.relative = relative;
}
/**
 * Creates a clone of this rule.
 *
 * @return a clone of this rule.
 *
 * @throws CloneNotSupportedException this should never happen.
 */
public Object clone() throws CloneNotSupportedException {
    final RelativeDayOfWeekRule duplicate
        = (RelativeDayOfWeekRule) super.clone();
}

```

```

duplicate.subrule = (AnnualDateRule) duplicate.getSubrule().clone();
return duplicate;
}
/**
 * Returns the date generated by this rule, for the specified year.
 *
 * @param year the year (1900 <= year <= 9999).
 *
 * @return The date generated by the rule for the given year (possibly
 * <code>null</code>).
 */
public SerialDate getDate(final int year) {
// check argument...
if ((year < SerialDate.MINIMUM_YEAR_SUPPORTED)
|| (year > SerialDate.MAXIMUM_YEAR_SUPPORTED)) {
throw new IllegalArgumentException(
"RelativeDayOfWeekRule.getDate(): year outside valid range.");
}
// calculate the date...
SerialDate result = null;
final SerialDate base = this.subrule.getDate(year);
if (base != null) {
switch (this.relative) {
case(SerialDate.PRECEDING):
result = SerialDate.getPreviousDayOfWeek(this.dayOfWeek,
base);
break;
case(SerialDate.NEAREST):
result = SerialDate.getNearestDayOfWeek(this.dayOfWeek,
base);
break;
case(SerialDate.FOLLOWING):
result = SerialDate.getFollowingDayOfWeek(this.dayOfWeek,
base);
break;
default:
break;
}
}
return result;
}
}

```

Listato B.7 DayDate.java (finale).

```

/*
=====
* JCommon : a free general purpose class library for the Java(tm) platform
* =====
*
* (C) Copyright 2000-2005, by Object Refinery Limited and Contributors.
*
...
*/
package org.jfree.date;
import java.io.Serializable;
import java.util.*;
/**
 * An abstract class that represents immutable dates with a precision of
 * one day. The implementation will map each date to an integer that
 * represents an ordinal number of days from some fixed origin.

```

```

*
* Why not just use java.util.Date? We will, when it makes sense. At times,
* java.util.Date can be *too* precise - it represents an instant in time,
* accurate to 1/1000th of a second (with the date itself depending on the
* time-zone). Sometimes we just want to represent a particular day (e.g. 21
* January 2015) without concerning ourselves about the time of day, or the
* time-zone, or anything else. That's what we've defined DayDate for.
*
* Use DayDateFactory.makeDate to create an instance.
*
* @author David Gilbert
* @author Robert C. Martin did a lot of refactoring.
*/
public abstract class DayDate implements Comparable, Serializable {
    public abstract int getOrdinalDay();
    public abstract int getYear();
    public abstract Month getMonth();
    public abstract int getDayOfMonth();
    protected abstract Day getDayOfWeekForOrdinalZero();
    public DayDate plusDays(int days) {
        return DayDateFactory.makeDate(getOrdinalDay() + days);
    }
    public DayDate plusMonths(int months) {
        int thisMonthAsOrdinal = getMonth().toInt() - Month.JANUARY.toInt();
        int thisMonthAndYearAsOrdinal = 12 * getYear() + thisMonthAsOrdinal;
        int resultMonthAndYearAsOrdinal = thisMonthAndYearAsOrdinal + months;
        int resultYear = resultMonthAndYearAsOrdinal / 12;
        int resultMonthAsOrdinal = resultMonthAndYearAsOrdinal % 12 +
            Month.JANUARY.toInt();
        Month resultMonth = Month.fromInt(resultMonthAsOrdinal);
        int resultDay = correctLastDayOfMonth(getDayOfMonth(), resultMonth, resultYear);
        return DayDateFactory.makeDate(resultDay, resultMonth, resultYear);
    }
    public DayDate plusYears(int years) {
        int resultYear = getYear() + years;
        int resultDay = correctLastDayOfMonth(getDayOfMonth(), getMonth(), resultYear);
        return DayDateFactory.makeDate(resultDay, getMonth(), resultYear);
    }
    private int correctLastDayOfMonth(int day, Month month, int year) {
        int lastDayOfMonth = DateUtil.lastDayOfMonth(month, year);
        if (day > lastDayOfMonth)
            day = lastDayOfMonth;
        return day;
    }
    public DayDate getPreviousDayOfWeek(Day targetDayOfWeek) {
        int offsetToTarget = targetDayOfWeek.toInt() - getDayOfWeek().toInt();
        if (offsetToTarget >= 0)
            offsetToTarget -= 7;
        return plusDays(offsetToTarget);
    }
    public DayDate getFollowingDayOfWeek(Day targetDayOfWeek) {
        int offsetToTarget = targetDayOfWeek.toInt() - getDayOfWeek().toInt();
        if (offsetToTarget <= 0)
            offsetToTarget += 7;
        return plusDays(offsetToTarget);
    }
    public DayDate getNearestDayOfWeek(Day targetDayOfWeek) {
        int offsetToThisWeeksTarget = targetDayOfWeek.toInt() - getDayOfWeek().toInt();
        int offsetToFutureTarget = (offsetToThisWeeksTarget + 7) % 7;
        int offsetToPreviousTarget = offsetToFutureTarget - 7;
        if (offsetToFutureTarget > 3)

```

```

        return plusDays(offsetToPreviousTarget);
    else
        return plusDays(offsetToFutureTarget);
    }
    public DayDate getEndOfMonth() {
Month month = getMonth();
int year = getYear();
int lastDay = DateUtil.lastDayOfMonth(month, year);
return DayDateFactory.makeDate(lastDay, month, year);
}
public Date toDate() {
final Calendar calendar = Calendar.getInstance();
int ordinalMonth = getMonth().toInt() - Month.JANUARY.toInt();
calendar.set(getYear(), ordinalMonth, getDayOfMonth(), 0, 0, 0);
return calendar.getTime();
}
public String toString() {
return String.format("%02d-%s-%d", getDayOfMonth(), getMonth(), getYear());
}
public Day getDayOfWeek() {
Day startingDay = getDayOfWeekForOrdinalZero();
int startingOffset = startingDay.toInt() - Day.SUNDAY.toInt();
int ordinalOfDayOfWeek = (getOrdinalDay() + startingOffset) % 7;
return Day.fromInt(ordinalOfDayOfWeek + Day.SUNDAY.toInt());
}
public int daysSince(DayDate date) {
return getOrdinalDay() - date.getOrdinalDay();
}
public boolean isOn(DayDate other) {
return getOrdinalDay() == other.getOrdinalDay();
}
public boolean isBefore(DayDate other) {
return getOrdinalDay() < other.getOrdinalDay();
}
public boolean isOnOrBefore(DayDate other) {
return getOrdinalDay() <= other.getOrdinalDay();
}
public boolean isAfter(DayDate other) {
return getOrdinalDay() > other.getOrdinalDay();
}
public boolean isOnOrAfter(DayDate other) {
return getOrdinalDay() >= other.getOrdinalDay();
}
public boolean isInRange(DayDate d1, DayDate d2) {
return isInRange(d1, d2, DateInterval.CLOSED);
}
public boolean isInRange(DayDate d1, DayDate d2, DateInterval interval) {
int left = Math.min(d1.getOrdinalDay(), d2.getOrdinalDay());
int right = Math.max(d1.getOrdinalDay(), d2.getOrdinalDay());
return interval.isIn(getOrdinalDay(), left, right);
}
}

```

Listato B.8 Month.java (finale).

```

package org.jfree.date;
import java.text.DateFormatSymbols;
public enum Month {
JANUARY(1), FEBRUARY(2), MARCH(3),
APRIL(4), MAY(5), JUNE(6),
JULY(7), AUGUST(8), SEPTEMBER(9),

```

```

OCTOBER(10), NOVEMBER(11), DECEMBER(12);
private static DateFormatSymbols dateFormatSymbols = new DateFormatSymbols();
private static final int[] LAST_DAY_OF_MONTH =
{0, 31, 28, 31, 30, 31, 30, 31, 31, 30, 31, 30, 31};
private int index;
Month(int index) {
this.index = index;
}
public static Month fromInt(int monthIndex) {
for (Month m : Month.values()) {
if (m.index == monthIndex)
return m;
}
throw new IllegalArgumentException("Invalid month index " + monthIndex);
}
public int lastDay() {
return LAST_DAY_OF_MONTH[index];
}
public int quarter() {
return 1 + (index - 1) / 3;
}
public String toString() {
return dateFormatSymbols.getMonths()[index - 1];
}
public String toShortString() {
return dateFormatSymbols.getShortMonths()[index - 1];
}
public static Month parse(String s) {
s = s.trim();
for (Month m : Month.values())
if (m.matches(s))
return m;
try {
return fromInt(Integer.parseInt(s));
}
catch (NumberFormatException e) {}
throw new IllegalArgumentException("Invalid month " + s);
}
private boolean matches(String s) {
return s.equalsIgnoreCase(toString()) ||
s.equalsIgnoreCase(toShortString());
}
public int toInt() {
return index;
}
}

```

Listato B.9 Day.java (finale).

```

package org.jfree.date;
import java.util.Calendar;
import java.text.DateFormatSymbols;
public enum Day {
MONDAY(Calendar.MONDAY),
TUESDAY(Calendar.TUESDAY),
WEDNESDAY(Calendar.WEDNESDAY),
THURSDAY(Calendar.THURSDAY),
FRIDAY(Calendar.FRIDAY),
SATURDAY(Calendar.SATURDAY),
SUNDAY(Calendar.SUNDAY);
private final int index;

```

```

private static DateFormatSymbols dateSymbols = new DateFormatSymbols();
Day(int day) {
    index = day;
}
public static Day fromInt(int index) throws IllegalArgumentException {
    for (Day d : Day.values())
        if (d.index == index)
            return d;
    throw new IllegalArgumentException(
        String.format("Illegal day index: %d.", index));
}
public static Day parse(String s) throws IllegalArgumentException {
    String[] shortWeekdayNames =
        dateSymbols.getShortWeekdays();
    String[] weekDayNames =
        dateSymbols.getWeekdays();
    s = s.trim();
    for (Day day : Day.values()) {
        if (s.equalsIgnoreCase(shortWeekdayNames[day.index]) ||
            s.equalsIgnoreCase(weekDayNames[day.index])) {
            return day;
        }
    }
    throw new IllegalArgumentException(
        String.format("%s is not a valid weekday string", s));
}
public String toString() {
    return dateSymbols.getWeekdays()[index];
}
public intToInt() {
    return index;
}
}

```

Listato B.10 DateInterval.java (finale).

```

package org.jfree.date;
public enum DateInterval {
    OPEN {
        public boolean isIn(int d, int left, int right) {
            return d > left && d < right;
        }
    },
    CLOSED_LEFT {
        public boolean isIn(int d, int left, int right) {
            return d >= left && d < right;
        }
    },
    CLOSED_RIGHT {
        public boolean isIn(int d, int left, int right) {
            return d > left && d <= right;
        }
    },
    CLOSED {
        public boolean isIn(int d, int left, int right) {
            return d >= left && d <= right;
        }
    };
    public abstract boolean isIn(int d, int left, int right);
}

```

Listato B.11 WeekInMonth.java (finale).

```
package org.jfree.date;
public enum WeekInMonth {
    FIRST(1), SECOND(2), THIRD(3), FOURTH(4), LAST(0);
    private final int index;
    WeekInMonth(int index) {
        this.index = index;
    }
    public int toInt() {
        return index;
    }
}
```

Listato B.12 WeekdayRange.java (finale).

```
package org.jfree.date;
public enum WeekdayRange {
    LAST, NEAREST, NEXT
}
```

Listato B.13 DateUtil.java (finale).

```
package org.jfree.date;
import java.text.DateFormatSymbols;
public class DateUtil {
    private static DateFormatSymbols dateFormatSymbols = new DateFormatSymbols();
    public static String[] getMonthNames() {
        return dateFormatSymbols.getMonths();
    }
    public static boolean isLeapYear(int year) {
        boolean fourth = year % 4 == 0;
        boolean hundredth = year % 100 == 0;
        boolean fourHundredth = year % 400 == 0;
        return fourth && (!hundredth || fourHundredth);
    }
    public static int lastDayOfMonth(Month month, int year) {
        if (month == Month.FEBRUARY && isLeapYear(year))
            return month.lastDay() + 1;
        else
            return month.lastDay();
    }
    public static int leapYearCount(int year) {
        int leap4 = (year - 1896) / 4;
        int leap100 = (year - 1800) / 100;
        int leap400 = (year - 1600) / 400;
        return leap4 - leap100 + leap400;
    }
}
```

Listato B.14 DayDateFactory.java (finale).

```
package org.jfree.date;
public abstract class DayDateFactory {
    private static DayDateFactory factory = new SpreadsheetDateFactory();
    public static void setInstance(DayDateFactory factory) {
        DayDateFactory.factory = factory;
    }
}
```

```

protected abstract DayDate _makeDate(int ordinal);
protected abstract DayDate _makeDate(int day, Month month, int year);
protected abstract DayDate _makeDate(int day, int month, int year);
protected abstract DayDate _makeDate(java.util.Date date);
protected abstract int _getMinimumYear();
protected abstract int _getMaximumYear();
public static DayDate makeDate(int ordinal) {
    return factory._makeDate(ordinal);
}
public static DayDate makeDate(int day, Month month, int year) {
    return factory._makeDate(day, month, year);
}
public static DayDate makeDate(int day, int month, int year) {
    return factory._makeDate(day, month, year);
}
public static DayDate makeDate(java.util.Date date) {
    return factory._makeDate(date);
}
public static int getMinimumYear() {
    return factory._getMinimumYear();
}
public static int getMaximumYear() {
    return factory._getMaximumYear();
}
}

```

Listato B.15 SpreadsheetDateFactory.java (finale).

```

package org.jfree.date;
import java.util.*;
public class SpreadsheetDateFactory extends DayDateFactory {
    public DayDate _makeDate(int ordinal) {
        return new SpreadsheetDate(ordinal);
    }
    public DayDate _makeDate(int day, Month month, int year) {
        return new SpreadsheetDate(day, month, year);
    }
    public DayDate _makeDate(int day, int month, int year) {
        return new SpreadsheetDate(day, month, year);
    }
    public DayDate _makeDate(Date date) {
        final GregorianCalendar calendar = new GregorianCalendar();
        calendar.setTime(date);
        return new SpreadsheetDate(
            calendar.get(Calendar.DATE),
            Month.fromInt(calendar.get(Calendar.MONTH) + 1),
            calendar.get(Calendar.YEAR));
    }
    protected int _getMinimumYear() {
        return SpreadsheetDate.MINIMUM_YEAR_SUPPORTED;
    }
    protected int _getMaximumYear() {
        return SpreadsheetDate.MAXIMUM_YEAR_SUPPORTED;
    }
}

```

Listato B.16 SpreadsheetDate.java (finale).

```

/*
=====
* JCommon : a free general purpose class library for the Java(tm) platform

```

```

* =====
*
* (C) Copyright 2000-2005, by Object Refinery Limited and Contributors.
*
...
*
*/
package org.jfree.date;
import static org.jfree.date.Month.FEBRUARY;
import java.util.*;
/** 
 * Represents a date using an integer, in a similar fashion to the
 * implementation in Microsoft Excel. The range of dates supported is
 * 1-Jan-1900 to 31-Dec-9999.
 * <p>
 * Be aware that there is a deliberate bug in Excel that recognises the year
 * 1900 as a leap year when in fact it is not a leap year. You can find more
 * information on the Microsoft website in article Q181370:
 * <p>
 * http://support.microsoft.com/support/kb/articles/Q181/3/70.asp
 * <p>
 * Excel uses the convention that 1-Jan-1900 = 1. This class uses the
 * convention 1-Jan-1900 = 2.
 * The result is that the day number in this class will be different to the
 * Excel figure for January and February 1900...but then Excel adds in an extra
 * day (29-Feb-1900 which does not actually exist!) and from that point forward
 * the day numbers will match.
 *
 * @author David Gilbert
 */
public class SpreadsheetDate extends DayDate {
    public static final int EARLIEST_DATE_ORDINAL = 2; // 1/1/1900
    public static final int LATEST_DATE_ORDINAL = 2958465; // 12/31/9999
    public static final int MINIMUM_YEAR_SUPPORTED = 1900;
    public static final int MAXIMUM_YEAR_SUPPORTED = 9999;
    static final int[] AGGREGATE_DAYS_TO_END_OF_PRECEDING_MONTH =
    {0, 0, 31, 59, 90, 120, 151, 181, 212, 243, 273, 304, 334, 365};
    static final int[] LEAP_YEAR_AGGREGATE_DAYS_TO_END_OF_PRECEDING_MONTH =
    {0, 0, 31, 60, 91, 121, 152, 182, 213, 244, 274, 305, 335, 366};
    private int ordinalDay;
    private int day;
    private Month month;
    private int year;
    public SpreadsheetDate(int day, Month month, int year) {
        if (year < MINIMUM_YEAR_SUPPORTED || year > MAXIMUM_YEAR_SUPPORTED)
            throw new IllegalArgumentException(
                "The 'year' argument must be in range " +
                MINIMUM_YEAR_SUPPORTED + " to " + MAXIMUM_YEAR_SUPPORTED + ".");
        if (day < 1 || day > DateUtil.lastDayOfMonth(month, year))
            throw new IllegalArgumentException("Invalid 'day' argument.");
        this.year = year;
        this.month = month;
        this.day = day;
        ordinalDay = calcOrdinal(day, month, year);
    }
    public SpreadsheetDate(int day, int month, int year) {
        this(day, Month.fromInt(month), year);
    }
    public SpreadsheetDate(int serial) {
        if (serial < EARLIEST_DATE_ORDINAL || serial > LATEST_DATE_ORDINAL)

```

```

throw new IllegalArgumentException(
    "SpreadsheetDate: Serial must be in range 2 to 2958465.");
ordinalDay = serial;
calcDayMonthYear();
}
public int getOrdinalDay() {
    return ordinalDay;
}
public int getYear() {
    return year;
}
public Month getMonth() {
    return month;
}
public int getDayOfMonth() {
    return day;
}
protected Day getDayOfWeekForOrdinalZero() {return Day.SATURDAY;}
public boolean equals(Object object) {
    if (!(object instanceof DayDate))
        return false;
    DayDate date = (DayDate) object;
    return date.getOrdinalDay() == getOrdinalDay();
}

public int hashCode() {
    return getOrdinalDay();
}
public int compareTo(Object other) {
    return daysSince((DayDate) other);
}
private int calcOrdinal(int day, Month month, int year) {
    int leapDaysForYear = DateUtil.leapYearCount(year - 1);
    int daysUpToYear = (year - MINIMUM_YEAR_SUPPORTED) * 365 + leapDaysForYear;
    int daysUpToMonth = AGGREGATE_DAYS_TO_END_OF_PRECEDING_MONTH[month.toInt()];
    if (DateUtil.isLeapYear(year) && month.toInt() > FEBRUARY.toInt())
        daysUpToMonth++;
    int daysInMonth = day - 1;
    return daysUpToYear + daysUpToMonth + daysInMonth + EARLIEST_DATE_ORDINAL;
}
private void calcDayMonthYear() {
    int days = ordinalDay - EARLIEST_DATE_ORDINAL;
    int overestimatedYear = MINIMUM_YEAR_SUPPORTED + days / 365;
    int nonleapdays = days - DateUtil.leapYearCount(overestimatedYear);
    int underestimatedYear = MINIMUM_YEAR_SUPPORTED + nonleapdays / 365;
    year = huntForYearContaining(ordinalDay, underestimatedYear);
    int firstOrdinalOfYear = firstOrdinalOfYear(year);
    month = huntForMonthContaining(ordinalDay, firstOrdinalOfYear);
    day = ordinalDay - firstOrdinalOfYear - daysBeforeThisMonth(month.toInt());
}
private Month huntForMonthContaining(int anOrdinal, int firstOrdinalOfYear) {
    int daysIntoThisYear = anOrdinal - firstOrdinalOfYear;
    int aMonth = 1;
    while (daysBeforeThisMonth(aMonth) < daysIntoThisYear)
        aMonth++;
    return Month.fromInt(aMonth - 1);
}
private int daysBeforeThisMonth(int aMonth) {
    if (DateUtil.isLeapYear(year))
        return LEAP_YEAR_AGGREGATE_DAYS_TO_END_OF_PRECEDING_MONTH[aMonth] - 1;
    else

```

```
    return AGGREGATE_DAYS_TO_END_OF_PRECEDING_MONTH[aMonth] - 1;
}
private int huntForYearContaining(int anOrdinalDay, int startingYear) {
    int aYear = startingYear;
    while (firstOrdinalOfYear(aYear) <= anOrdinalDay)
        aYear++;

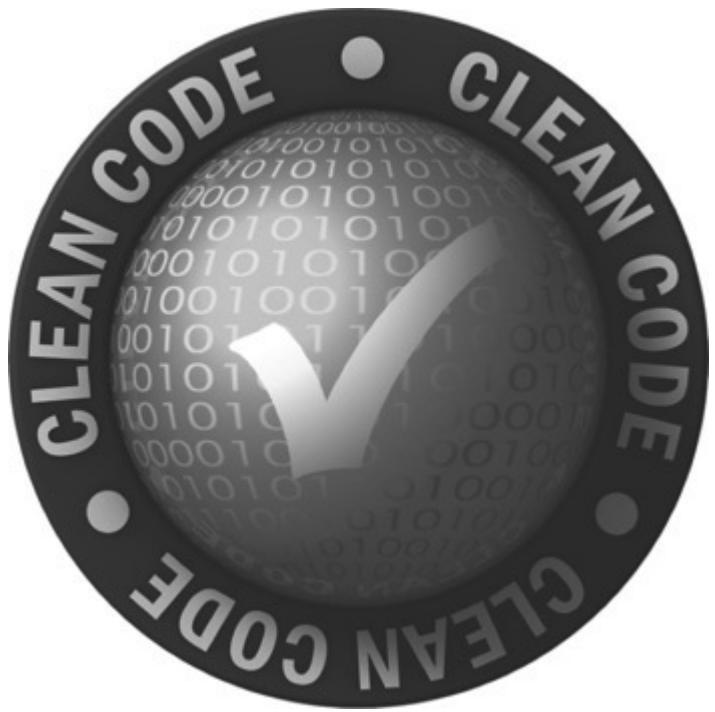
    return aYear - 1;
}
private int firstOrdinalOfYear(int year) {
    return calcOrdinal(1, Month.JANUARY, year);
}
public static DayDate createInstance(Date date) {
    GregorianCalendar calendar = new GregorianCalendar();
    calendar.setTime(date);
    return new SpreadsheetDate(calendar.get(Calendar.DATE),
        Month.fromInt(calendar.get(Calendar.MONTH) + 1),
        calendar.get(Calendar.YEAR));
}
```

Epilogo

Nel 2005, mentre partecipavo a una conferenza su Agile a Denver, Elisabeth Hedrickson mi passò un braccialetto verde, simile a quello reso così popolare da Lance Armstrong. Sopra c'era scritto *Test Obsessed*. L'ho indossato con grande orgoglio. Da quando ho conosciuto lo sviluppo TDD di Kent Beck, nel 1999, in effetti sono ossessionato dallo sviluppo guidato dai test.

Poi accadde qualcosa di strano. Non riuscivo più a sfilarmi il braccialetto. Non perché vi fosse un impedimento fisico, ma perché io mi sentivo moralmente bloccato. Ormai era diventato un'affermazione esplicita della mia etica professionale. Era un'indicazione visibile del mio impegno a scrivere il miglior codice che mi fosse possibile. Toglierlo mi sarebbe sembrato come tradire tale etica e tale impegno.

Pertanto l'ho ancora al polso. Quando scrivo del codice, accompagna sempre la visione periferica dei miei occhi. Mi ricorda costantemente la promessa che ho fatto a me stesso di scrivere solo codice pulito.



Indice

Prefazione

Introduzione

Ringraziamenti

Capitolo 1 - Codice pulito

Che il codice sia!

Cattivo codice

Il costo totale di possedere un vero groviglio di codice

Scuole di pensiero

Gli autori

La regola dei boy-scout

Il “prequel” e i principi

Conclusioni

Bibliografia

Capitolo 2 - Nomi significativi

Introduzione

Usate nomi “parlanti”

Evitate la disinformazione

Adottate distinzioni sensate

Usate nomi pronunciabili

Usate nomi ricercabili

Evitate le codifiche

Evitate le mappe mentali

- Nomi di classi
- Nomi di metodi
- Non fate i “simpatici”
- Una parola, un concetto
- Non siate fuorvianti
- Usate nomi tratti dal dominio della soluzione
- Usate nomi tratti dal dominio del problema
- Aggiungete un contesto significativo
- Non aggiungete contesti inesistenti
- Conclusioni

Capitolo 3 - Funzioni

- Che sia piccola!
- Che faccia una cosa sola
- Un livello di astrazione per funzione
- Istruzioni switch
- Usate nomi descrittivi
- Argomenti di funzione
- Niente effetti collaterali
- Separate i comandi dalle richieste
- Scegliete le eccezioni invece di restituire codici di errore
- Non ripetetevi (il principio DRY)
- Programmazione strutturata
- Come scrivere le funzioni in questo modo?
- Conclusioni
- SetupTeardownIncluder
- Bibliografia

Capitolo 4 - Commenti

- I commenti non bastano a migliorare il codice cattivo
- Spiegatevi nel codice
- Buoni commenti

Cattivi commenti

Bibliografia

Capitolo 5 - Formattazione

Lo scopo della formattazione

Formattazione verticale

Formattazione orizzontale

Le regole del team

Le regole di formattazione di Uncle Bob

Capitolo 6 - Oggetti e strutture

Astrazione dei dati

Asimmetria dei dati/oggetti

La Legge di Demetra

Data Transfer Object

Conclusioni

Bibliografia

Capitolo 7 - Gestione degli errori

Usate eccezioni al posto dei codici di return

Scrivere prima l'istruzione try-catch-finally

Usate eccezioni non controllate

Fornite un contesto con le eccezioni

Definite le classi per le eccezioni in termini di esigenze del chiamante

Definite il flusso “normale”

Non restituite null

Non passate null

Conclusioni

Bibliografia

Capitolo 8 - Delimitazioni

Usare codice esterno

Esplorazione delle delimitazioni

Imparare a usare log4j

I learning test sono più che gratis

Usare codice che non esiste ancora

Delimitazioni chiare

Bibliografia

Capitolo 9 - Unit test

Le tre leggi dello sviluppo TDD (Test-Driven Development)

Curate la pulizia dei test

Test “puliti”

Una sola richiesta per test

F.I.R.S.T.

Conclusioni

Bibliografia

Capitolo 10 - Classi

Organizzazione delle classi

Le classi dovrebbero essere piccole!

Organizzare gli interventi di modifica

Bibliografia

Capitolo 11 - Sistemi

Come edifichereste una città?

Separate la realizzazione dall'uso di un sistema

Estensione di scala

Proxy Java

Framework AOP puri Java

AspectJ

Sottoporre a test l'architettura del sistema

Ottimizzazione delle decisioni

Usate gli standard con cura, solo se dimostrano il proprio valore

I sistemi richiedono l'impiego di linguaggi specifici del dominio

Conclusioni

Bibliografia

Capitolo 12 - Simple Design

Come far emergere un codice pulito

Regola 1 di Simple Design – Passa tutti i test

Regole 2-4 di Simple Design – Refactoring

Niente duplicazione

Espressività

Minimizzare classi e metodi

Conclusioni

Bibliografia

Capitolo 13 - Concorrenza

A che cosa serve la concorrenza?

Sfide

Concorrenza: principi di difesa

Studiate la vostra libreria

Studiate i modelli di esecuzione

Attenzione alle dipendenze fra metodi sincronizzati

Riducete al minimo le sezioni sincronizzate

Scrivere il codice di chiusura corretto è difficile

Collaudo del codice di un thread

Conclusioni

Bibliografia

Capitolo 14 - Raffinamento progressivo

Implementazione di Args

Args: “la brutta”

Argomenti String

Conclusioni

Capitolo 15 - JUnit

Il framework JUnit
Conclusioni

Capitolo 16 - Refactoring di SerialDate

Innanzitutto, facciamola funzionare
Ora raffiniamola
Conclusioni
Bibliografia

Capitolo 17 - Avvertenze ed euristiche

Commenti
Ambiente
Funzioni
Generali
Java
Nomi
Test
Conclusioni
Bibliografia

Appendice A - Concorrenza II

Un esempio client/server
Possibili percorsi di esecuzione
Studiare la libreria
Migliorare la produttività (throughput)
Deadlock
Collaudo del codice multi-thread
Strumenti per il test del codice multi-thread
Conclusioni
Tutorial: esempi di codice

Appendice B - org.jfree.date.SerialDate

Epilogo