ANDREW HUNT, DAVID THOMAS

il Pragmatic Programmer

Guida per manovali del software che vogliono diventare maestri

APOGEO

Il Pragmatic Programmer

Guida per manovali del software che vogliono diventare maestri

Andrew Hunt
David Thomas

APOGEO

ISBN edizione cartacea: 9788850332540

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

In quanto revisore, ho avuto la possibilità di leggere in anteprima il libro che avete fra le mani, ed era grande, già in bozza. Dave Thomas e Andy Hunt hanno qualcosa da dire, e sanno come dirlo. Ho visto quello che stavano facendo e sapevo che avrebbe funzionato. Così ho chiesto di scrivere questa prefazione, per poter spiegare perché.

Detto semplicemente, questo libro racconta come programmare in un modo che si può seguire. Forse non penserete che sia una cosa difficile da fare, ma lo è. Perché? Da un lato, non tutti i libri di programmazione sono scritti da programmatori. Molti sono compilati dai progettisti dei linguaggi, o dai giornalisti che lavorano con loro per promuovere le loro creazioni. Quei libri vi dicono come *parlare* in un linguaggio di programmazione, il che è certamente importante ma è solo una piccola parte di quel che fa un programmatore.

Che cosa fa un programmatore oltre a parlare in un linguaggio di programmazione? Beh, questa è una faccenda più profonda. La maggior parte dei programmatori avrebbe qualche difficoltà a spiegare che cosa fa. La programmazione è un lavoro pieno di dettagli, e tener traccia di tutti quei dettagli richiede grande attenzione. Le ore passano e il codice compare. Lo guardate e ci sono tutti quegli enunciati. Se non pensate con grande attenzione, potreste credere che la programmazione sia solo scrivere enunciati in un linguaggio di programmazione. Avreste torto, ovviamente, ma non riuscireste a stabilirlo guardandovi intorno nella sezione “programmazione” della libreria.

In questo libro, Dave e Andy ci dicono come programmare, e lo dicono in un modo che possiamo seguire. Come sono riusciti a essere così abili? Non sono concentrati sui dettagli come gli altri programmatori? La risposta è che hanno fatto attenzione a quello che facevano mentre lo facevano, e poi hanno cercato di farlo meglio.

Immaginatevi di partecipare a una riunione. Magari state pensando che la riunione potrebbe andare avanti all’infinito e che preferireste programmare. Dave e Andy penserebbero perché stanno partecipando a

una riunione, e si chiederebbero se ci sia qualche altra cosa che potrebbero fare che prenda il posto della riunione, e cercherebbero di decidere se quel qualcosa potrebbe essere automatizzato in modo che il lavoro della riunione "si faccia da solo" in futuro. Poi lo farebbero.

Questo è il modo in cui pensano Dave e Andy. Quella riunione non era una cosa che impedisse loro di programmare: era programmazione. Ed era programmazione che poteva essere migliorata. So che pensano in questo modo, perché è il loro secondo suggerimento: "Pensate al vostro lavoro".

Hanno pensato in questo modo per anni: come potete immaginare, presto hanno avuto una bella raccolta di soluzioni. Ora immaginateveli che usano quelle soluzioni nel loro lavoro per qualche anno ancora e buttano via quelle troppo pesanti o che non danno sempre buoni risultati. Beh, questo modo di procedere spiega sostanzialmente che cosa voglia dire *pragmatico*. Ora immaginateli che si prendono ancora un anno o due per scrivere le loro soluzioni. Penserete: *quelle informazioni sarebbero una miniera d'oro*. E avreste ragione.

Gli autori ci dicono come programmano e ce lo dicono in un modo che possiamo seguire. Ma in questa affermazione c'è più di quello che forse pensate. Mi spiego.

Gli autori sono stati ben attenti a evitare di proporre una teoria dello sviluppo di software. È una fortuna, perché altrimenti sarebbero stati costretti a trasformare ogni capitolo per difendere la loro teoria. Il che sarebbe nella tradizione, per esempio, delle scienze fisiche, dove le teorie alla fine diventano leggi o vengono scartate senza clamore. La programmazione invece ha poche leggi (o forse nessuna). Perciò un consiglio di programmazione formulato in base a qualche presunta legge potrà suonare bene scritto, ma non soddisfa mai nella pratica. È quello che non va in tanti libri che parlano di metodologia.

Ho studiato il problema per una dozzina d'anni e ho trovato che la cosa più promettente era un dispositivo chiamato *linguaggio a pattern*. In breve, un *pattern* è una soluzione, e un linguaggio a pattern è un sistema di soluzioni che si rafforzano a vicenda. Si è formata un'intera comunità attorno alla ricerca di questi sistemi.

Questo libro è più di una raccolta di suggerimenti. È un linguaggio a pattern travestito. Lo dico perché ogni suggerimento è ricavato dall'esperienza, formulato come consiglio concreto e messo in rapporto

con gli altri a formare un sistema. Sono le caratteristiche che ci permettono di apprendere e di seguire un linguaggio a pattern, e funzionano nello stesso modo qui.

Potete seguire i consigli di questo libro perché sono concreti. Non troverete astrazioni vaghe. Dave e Andy scrivono direttamente per voi, come se ogni suggerimento fosse una strategia vitale per infondere energia nella vostra carriera di programmatori. Restano semplici, raccontano una storia, usano un po' di umorismo e poi propongono le risposte alle domande che vi sorgono mentre provate.

E non è finita qui. Dopo aver letto dieci o quindici suggerimenti comincerete a vedere un'ulteriore dimensione di questo lavoro. A volte la chiamiamo QWAN, che sta per *quality without a name*, qualità senza nome. Il libro ha una filosofia che filtrerà e si mescolerà con la vostra coscienza. Non tiene sermoni; vi dice semplicemente quello che funziona. Ma nel dirlo passa anche molto altro. È la bellezza del libro: incarna la propria filosofia, e lo fa senza presunzione.

Perciò, eccolo qui: un libro completo sulla programmazione, facile da leggere e da mettere in pratica. Ho speso una gran quantità di tempo per spiegare il motivo che lo rende così speciale, ma a voi probabilmente interesserà solo che funzioni. Ed è così. Provare per credere.

Ward Cunningham

Per Ellie e Juliet,
Elizabeth e Zachary,
Stuart ed Henry

Questo libro vi aiuterà a diventare programmatori migliori.

Non è importante se lavorate da soli, se fate parte di un team di progetto molto ampio o se siete consulenti che lavorano con molti clienti contemporaneamente: questo libro vi aiuterà a lavorare meglio, a livello individuale. Non è un libro teorico: ci concentriamo su argomenti pratici, su come usare l'esperienza per prendere decisioni più informate. L'aggettivo *pragmatico* viene dal latino, *pragmaticus*, ovvero "competente nel proprio lavoro", che a sua volta deriva dal greco πραττειν, che significa "fare". Questo è un libro sul fare.

Programmare è un lavoro artigianale. Nel caso più semplice, significa convincere un computer a fare quello che volete che faccia (o che i vostri utenti vogliono che faccia). Da programmatori, siete in parte ascoltatori, in parte consiglieri, in parte interpreti e in parte dittatori. Cercate di cogliere requisiti che un po' vi sfuggono e di trovare un modo per esprimerli, così che una semplice macchina possa render loro giustizia. Cercate di documentare il vostro lavoro in modo che altri possano capirlo e cercate di ingegnerizzarlo, in modo che altri possano usarlo per costruzioni ancora più complesse. Cosa ancora più importante, provate a fare tutte queste cose mentre l'orologio del progetto continua a ticchettare senza mai fermarsi. Fate ogni giorno tanti piccoli miracoli.

È un lavoro difficile.

Molti vi offrono aiuto. Produttori di *tool* vantano le meraviglie che i loro prodotti possono fare. Guru di metodologia promettono che con le loro tecniche i risultati sono garantiti. Ognuno dice che il suo linguaggio di programmazione è il migliore, e ogni sistema operativo è la risposta a tutti i mali possibili e immaginabili.

Ovviamente, è tutto falso. Non ci sono risposte facili. Non esiste *la* soluzione migliore, che sia uno strumento, un linguaggio o un sistema operativo. Ci sono solo sistemi che possono essere più o meno adatti a un certo insieme di circostanze.

Qui entra in campo il pragmatismo. Non dovete sposare una particolare tecnologia, ma dovete avere un background e una base di esperienze sufficientemente ampi da consentirvi di scegliere buone soluzioni in situazioni particolari. Il background viene da una comprensione dei principi di fondo dell'informatica, e l'esperienza arriva da un'ampia gamma di progetti pratici. Teoria e pratica si combinano per rendervi forti.

Regolate la vostra impostazione in modo che si adatti alle circostanze e all'ambiente. Valutate l'importanza relativa di tutti i fattori che influenzano un progetto e usate la vostra esperienza per produrre soluzioni appropriate. E fate tutto questo continuamente, al procedere del lavoro. I "programmatori pragmatici" portano a termine il lavoro, e lo fanno bene.

Per chi è questo libro?

Questo libro si rivolge a persone che vogliono diventare programmatori più efficaci e più produttivi. Forse vi sentite frustrati perché non vi sembra di esprimere tutto il vostro potenziale; magari invidiate colleghi che sembra usino strumenti che li rendono più produttivi di voi; o forse state usando tecnologie datate, e volete sapere se potete applicare idee più recenti a quel che fate.

Non pretendiamo di avere tutte le risposte (e nemmeno la maggior parte), e non tutte le nostre idee sono applicabili a tutte le situazioni. Quel che possiamo dire è che, se seguite la nostra impostazione, accumulerete esperienza rapidamente, la vostra produttività aumenterà, e avrete una comprensione migliore di tutto il processo di sviluppo. E scriverete software migliore.

Che cosa fa di un programmatore un pragmatic programmer?

Ogni sviluppatore è unico, con i suoi punti di forza e le sue debolezze, le sue preferenze e le cose che non gli piacciono. Con il tempo, ciascuno si costruirà il proprio ambiente personale, che rispecchierà la sua

individualità non meno dei suoi hobby, del modo di vestire o del taglio di capelli. Se siete programmatori pragmatici, avrete molte di queste caratteristiche.

- *Early adopter/fast adapter.* Avete un istinto per le tecnologie e le tecniche, e vi piace sperimentare. Quando vi imbattete in qualcosa di nuovo, lo afferrate in fretta e lo integrate al resto della vostra conoscenza. La vostra fiducia deriva dall'esperienza.
- *Curiosi.* Fate tante domande. *Bello - come hai fatto? Hai avuto problemi con quella libreria? Che cos'è quel BeOS di cui ho sentito parlare? Come sono implementati i collegamenti simbolici?* Accumulate piccoli fatti, ciascuno dei quali potrebbe influenzare qualche decisione magari a distanza di anni.
- *Pensatori critici.* Raramente date qualcosa per scontato senza conoscere a fondo i fatti. Quando i colleghi dicono "perché si fa così" o un produttore vi promette la soluzione di tutti i vostri problemi, percepite una sfida.
- *Realisti.* Cercate di capire la natura di fondo di ogni problema che incontrate. Questo realismo vi dà una buona percezione di quanto le cose siano difficili e di quanto tempo richiederanno. Capire da soli che un processo *deve* essere difficile o che *richiederà* del tempo per poter essere completato vi darà la forza di perseverare.
- *Tuttofare.* Cercate in tutti i modi di avere familiarità con molte tecnologie e molti ambienti, e cercate di rimanere aggiornati sui nuovi sviluppi. Magari la vostra attuale mansione vi costringe a essere degli specialisti, ma sarete sempre in grado di passare a nuovi campi e nuove sfide.

Abbiamo lasciato per ultime le caratteristiche più basilari, che tutti i programmatori pragmatici hanno in comune. Sono abbastanza fondamentali da formularle come suggerimenti:

SUGGERIMENTO 1
Abbiate cura del vostro mestiere.

Pensiamo che non abbia senso sviluppare software, se non si ha intenzione di farlo bene.

SUGGERIMENTO 2
Pensate al vostro lavoro.

Per essere programmatori pragmatici, dovete pensare a quello che fate mentre lo fate. Non si tratta di una valutazione *una tantum* delle pratiche correnti, ma di una valutazione critica di ogni decisione che prendete, ogni giorno, per ogni progetto. Mai mettere il pilota automatico. Dovete sempre pensare e criticare il vostro lavoro in tempo reale. Il vecchio motto aziendale della IBM, THINK!, è il mantra del programmatore pragmatico.

Se tutto questo vi sembra difficile, allora siete dei *realisti*. Ci vorrà un po' di tempo prezioso, tempo che probabilmente già scarseggia. La ricompensa però è un coinvolgimento più attivo in un lavoro che amate, una sensazione di padronanza su una serie crescente di argomenti, e il piacere di un miglioramento continuo. Sul lungo termine, l'investimento di tempo sarà ripagato: voi e la vostra squadra sarete più efficienti, scriverete codice più facile da manutenere e passerete meno tempo in riunione.

Pragmatici singoli, grandi squadre

Qualcuno pensa che non ci sia spazio per l'individualità nelle grandi squadre o nei progetti complessi. "La costruzione di software è una disciplina ingegneristica", dicono, "e tutto crolla se i singoli membri della squadra prendono decisioni per i fatti loro".

Non siamo d'accordo.

La costruzione di software *deve* essere una disciplina ingegneristica, ma questo non esclude le capacità individuali. Pensate alle grandi cattedrali costruite in Europa nel medioevo. Ciascuna ha richiesto migliaia di anni/persona di lavoro, spalmati nell'arco di decenni. Quel che veniva appreso era trasferito al gruppo successivo di costruttori, che facevano progredire con i loro risultati lo stato dell'ingegneria strutturale. Ma carpentieri, tagliatori, scalpellini e lavoratori del vetro erano tutti artigiani, che interpretavano i requisiti ingegneristici per produrre un tutto che andava al di là della pura meccanica della costruzione. Era la loro convinzione nei loro contributi individuali che sosteneva i progetti:

> Noi che tagliamo solamente pietre dobbiamo sempre avere in mente le cattedrali.
> - *Credo del cavatore*

Entro la struttura generale di un progetto c'è sempre spazio per l'individualità e le capacità artigianali, il che è particolarmente vero, visto lo stato attuale dell'ingegneria del software. Fra cento anni, forse, la nostra ingegneria sembrerà tanto arcaica quanto le tecniche degli antichi costruttori di cattedrali paiono oggi ai nostri ingegneri civili, ma le nostre capacità artigianali saranno ancora onorate.

È un processo continuo

> Un turista in visita all'Eton College, in Inghilterra, chiede al giardiniere come ha fatto a ottenere prati così perfetti. "Facile", risponde quello. "Basta spazzar via la rugiada ogni mattina, tosare l'erba un giorno sì e uno no, e rivoltare la terra una volta alla settimana".
>
> "Tutto qui?"; chiede il turista.
>
> "Assolutamente", risponde il giardiniere. "Faccia così per 500 anni e anche lei avrà un bel prato."

Bei prati hanno bisogno di poca cura quotidiana, e lo stesso vale per i grandi programmatori. I consulenti di management amano lasciar cadere nella conversazione il termine *kaizen*. È una parola giapponese che coglie l'idea di apportare continuamente tanti piccoli miglioramenti, ed è stata considerata una delle ragioni principali dei grandi incrementi di produttività e qualità della manifattura giapponese, ampiamente copiata in tutto il mondo. *Kaizen* vale anche per gli individui. Ogni giorno, cercate di perfezionare le vostre competenze e di aggiungere nuovi strumenti al vostro repertorio. A differenza dei prati di Eton, comincerete a vedere i risultati entro pochi giorni. Con gli anni, sarete stupiti da come la vostra esperienza sia fiorita e le vostre competenze siano cresciute.

Come è organizzato il libro

Questo libro è scritto sotto forma di una serie di brevi sezioni, ciascuna delle quali è indipendente e affronta un argomento particolare. Troverete molti rimandi incrociati, che vi aiuteranno a collocare ogni argomento nel suo contesto. Potete leggere le sezioni in qualsiasi ordine: questo non è un libro da leggere in fila, dalla prima pagina all'ultima.

Ogni tanto incontrerete qualche riga evidenziata dal titolo *Suggerimento* (come “Abbiate cura del vostro mestiere”). Oltre a evidenziare quanto viene detto nel testo, ci sembra che questi suggerimenti abbiano una vita propria: viviamo seguendoli ogni giorno.

L’Appendice A contiene una serie di risorse: la bibliografia del libro, un elenco di URL di risorse web e un elenco di riviste, libri e organizzazioni professionali che vi consigliamo. In tutto il libro troverete rimandi alla bibliografia e all’elenco degli URL, nella forma [KP99] e [URL 18], rispettivamente.

Dove opportuno abbiamo inserito esercizi e progetti “sfida”. Gli esercizi in genere hanno risposte relativamente immediate, mentre le sfide sono più aperte. Per darvi un’idea del nostro modo di pensare, nell’Appendice B trovate le nostre risposte agli esercizi, ma pochissimi hanno un’unica soluzione giusta. Le sfide possono costituire la base di discussioni di gruppo o di saggi in corsi di programmazione avanzata.

Che cosa c’è in un nome?

> “Quando io uso una parola”, Humpty Dumpty disse in tono piuttosto sdegnato, “essa significa esattamente quello che voglio - né più né meno”.
>
> - Lewis Carroll, *Alice attraverso lo specchio*

Nel corso del libro troverete qua e là un po’ di gergo: parole o espressioni italiane o inglesi che sono state piegate a indicare qualche concetto tecnico, o orrende parole inventate a cui informatici con qualche conto in sospeso con la lingua hanno assegnato un significato. La prima volta che usiamo questi termini, cerchiamo di definirli, o almeno di dare un’indicazione del loro significato. Siamo sicuri che qualcuno ci sarà sfuggito, mentre altri, come *oggetto* e *database relazionale*, sono di uso così comune che l’aggiunta di una definizione sarebbe stata noiosa. Se vi capitasse di incontrare un termine che non avete mai visto prima, non saltatelo: prendetevi il tempo per cercarlo, magari nel Web o in un manuale di informatica; e, nel caso, mandateci una email e lamentatevi, così potremo aggiungere una definizione nella prossima edizione.

Detto questo, abbiamo deciso di vendicarci degli informatici. A volte, esistono termini di gergo tecnico, che abbiamo deciso di ignorare. Perché? Perché il gergo esistente di solito è limitato a un particolare

ambito di problemi, o a una particolare fase dello sviluppo, mentre una delle filosofie fondamentali di questo libro è che la maggior parte delle tecniche che consigliamo sono universali: la modularità vale per il codice, i progetti, la documentazione e l'organizzazione del team, per esempio. Usare il termine convenzionale in un contesto più ampio sarebbe fonte di confusione: non è facile trascurare il bagaglio di significati che il termine originale porta con sé. Quando ci siamo trovati in situazioni simili, abbiamo dato il nostro contributo al declino della lingua inventando i nostri nuovi termini.

Codice sorgente e altre risorse

La maggior parte del codice che troverete in queste pagine è disponibile all'indirizzo:

https://pragprog.com/titles/tpp/source_code

Lì troverete anche collegamenti a risorse che ci sono sembrate utili, con aggiornamenti al libro e notizie di altri sviluppi da programmatori pragmatici.

Mandateci un feedback

Ci farebbe piacere sentirvi. Commenti, suggerimenti, segnalazioni di errori nel testo e di problemi negli esempi sono tutti benvenuti. Scriveteci all'indirizzo di posta elettronica

ppbook@pragmaticprogrammer.com

Ringraziamenti

Quando abbiamo iniziato a scrivere questo libro, non avevamo idea che avrebbe finito per diventare una grande impresa di squadra.

La Addison-Wesley è stata brillante: ha preso un paio di hacker senza esperienza editoriale e li ha portati lungo tutto il processo di produzione, dall'idea al materiale pronto per la stampa. Grazie mille a John Wait e Meera Ravindiran per il loro sostegno iniziale, a Mike Hendrickson, il nostro editor entusiasta (e grande progettista di copertine), a Lorraine

Ferrie e John Fuller per l'aiuto in fase di produzione e all'instancabile Julie DeBaggis per aver tenuto insieme il tutto.

Poi ci sono stati i revisori: Greg Andress, Mark Cheers, Chris Cleeland, Alistair Cockburn, Ward Cunningham, Martin Fowler, Thanh T. Giang, Robert L. Glass, Scott Henninger, Michael Hunter, Brian Kirby, John Lakos, Pete McBreen, Carey P. Morris, Jared Richardson, Kevin Ruland, Eric Starr, Eric Vought, Chris Van Wyk e Deborra Zukowski. Senza i loro commenti e le loro idee preziose questo libro sarebbe stato meno leggibile, meno preciso e lungo il doppio. Grazie a tutti per il vostro tempo e la vostra saggezza.

La seconda edizione di questo libro ha tratto grande vantaggio dagli occhi di falco dei nostri lettori. Grazie mille a Brian Blank, Paul Boal, Tom Ekberg, Brent Fulgham, Louis Paul Hebert, Henk-Jan Olde Loohuis, Alan Lund, Gareth McCaughan, Yoshiki Shibata e Volker Wurst, per aver trovato gli errori e per aver avuto la cortesia di farceli notare.

Nel corso degli anni, abbiamo lavorato con un gran numero di clienti, accumulando e perfezionando l'esperienza di cui scriviamo qui. Recentemente, abbiamo avuto la fortuna di lavorare su vari grandi progetti con Peter Gehrke: abbiamo apprezzato molto il suo sostegno e il suo entusiasmo per le nostre tecniche.

Questo libro è stato prodotto con LATEX, pic, Perl, dvips, ghostview, ispell, GNU make, CVS, Emacs, XEmacs, EGCS, GCC, Java, iContract, SmallEiffel e le shell Bash e zsh sotto Linux. La cosa incredibile è che tutto questo software è liberamente disponibile. Dobbiamo un enorme "grazie" alle migliaia di programmatori pragmatici di tutto il mondo che hanno creato queste cose per tutti noi. Vogliamo ringraziare in particolare Reto Kramer per l'aiuto che ci ha dato con iContract.

Ultimo, ma certo non per importanza, abbiamo un debito enorme on le nostre famiglie. Non solo hanno sopportato la scrittura a tarda notte, le enormi bollette telefoniche e la nostra costante aria distratta, ma hanno anche avuto la cortesia, ogni tanto, di leggere quello che avevamo scritto. Grazie per averci lasciato sognare.

Andy Hunt
Dave Thomas

Una filosofia pragmatica

Che cosa distingue i programmatori pragmatici? Un atteggiamento, uno stile, una filosofia con cui si accostano ai problemi e alle loro soluzioni. Pensano oltre il problema immediato, cercano sempre di collocarlo nel suo contesto più ampio, di capire il quadro più generale. In fin dei conti, senza questo contesto più ampio, come si potrebbe essere pragmatici? Come si potrebbe scendere a compromessi intelligenti e prendere decisioni informate?

Un altro elemento fondamentale per il loro successo è che si assumono la responsabilità di tutto quello che fanno, cosa di cui parleremo nel paragrafo *Il gatto mi ha mangiato il codice sorgente*. Essendo responsabili, i programmatori pragmatici non se ne staranno lì seduti a far niente mentre i loro progetti vanno a pezzi per trascuratezza. In *Entropia del software*, vi diremo come mantenere immacolati i vostri progetti.

Per molti è difficile accettare il cambiamento, a volte per buoni motivi, a volte per pura inerzia. In *Zuppa di pietre e rane bollite*, esamineremo una strategia per favorire il cambiamento e (per essere equilibrati) presenteremo come ammonimento la storia di un anfibio che ha ignorato i pericoli del cambiamento graduale.

Uno dei vantaggi di comprendere il contesto in cui si lavora è che diventa più facile sapere quanto deve essere buono il vostro software. A volte l'unica possibilità è la quasi-perfezione, ma spesso ci sono anche altri compromessi da fare. Lo vedremo in *Software abbastanza buono*.

Ovviamente, dovete avere un'ampia base di conoscenze e di esperienze, ma l'apprendimento è un processo continuo e sempre in corso. In *Il portafoglio delle conoscenze*, analizzeremo alcune strategie per essere sempre al passo.

Infine, nessuno di noi lavora in un vuoto pneumatico. Tutti passiamo molto tempo a interagire con gli altri. *Comunicate!* elenca i modi in cui possiamo farlo meglio.

La programmazione pragmatica nasce da una filosofia pragmatica, e questo capitolo pone le basi di quella filosofia.

Il gatto mi ha mangiato il codice sorgente

> La maggiore di tutte le debolezze è la paura di apparire deboli.
> - J. P. Bossuet, *Politics from Holy Writ*, 1709.

Una delle pietre angolari della filosofia pragmatica è l'idea di assumersi la responsabilità per se sessi e le proprie azioni - per quanto riguarda la carriera, il progetto specifico, il lavoro quotidiano. Un programmatore pragmatico si fa carico della propria carriera e non ha paura di ammettere quando non sa o quando sbaglia. Non è l'aspetto più piacevole della programmazione, certo, ma succede, anche nei progetti migliori. Nonostante test esaurienti, buona documentazione e robusta automazione, le cose vanno storte. La consegna è in ritardo. Spuntano problemi tecnici imprevisti.

Sono cose che succedono, e cerchiamo di affrontarle il più professionalmente possibile. Questo significa essere onesti e diretti: possiamo essere orgogliosi delle nostre capacità, ma dobbiamo essere onesti sulle nostre mancanze, la nostra ignoranza e i nostri errori.

Prendersi la responsabilità

La responsabilità è qualcosa che ci si accolla attivamente. Ci si impegna a che qualcosa venga fatto bene, ma non si ha necessariamente un controllo diretto su ogni aspetto. Oltre a fare del vostro meglio a livello personale, dovete analizzare la situazione e individuare rischi che siano al di fuori del vostro controllo. Avete il diritto di non assumervi la responsabilità di una situazione impossibile o in cui i rischi sono troppo grandi. Dovrete decidere in base alla vostra etica e secondo il vostro giudizio.

Se accettate la responsabilità di un esito, dovete aspettarvi di essere chiamati a renderne conto. Se commettete uno sbaglio (tutti ne facciamo) o un errore di giudizio, ammettetelo onestamente e cercate di offrire delle opzioni.

Non date la colpa a qualcuno o qualcosa d'altro, non inventate scuse. Non scaricate tutti i problemi su un produttore, un linguaggio di programmazione, il management o i vostri colleghi. Tutti possono avere avuto un ruolo, ma voi dovete fornire soluzioni, non scuse.

Se c'era il rischio che il produttore non vi venisse in aiuto, dovevate avere un piano di riserva. Se il disco si rovina, portandosi via tutto il vostro codice sorgente, e non avete una copia di riserva, è colpa vostra. Dire al vostro capo "il gatto mi ha mangiato il codice sorgente" non funzionerà.

SUGGERIMENTO 3
Fornite opzioni, non inventate scuse patetiche.

Prima di avvicinare qualcuno e dirgli perché qualcosa non si può fare, è in ritardo o è rotto, fermatevi e ascoltatevi. Parlate alla papera di gomma sul monitor o al gatto. La vostra scusa suona ragionevole o stupida? Come sembrerà al vostro capo?

Provate a condurre la conversazione nella vostra testa. Che cosa è probabile che vi dica l'altra persona? Vi chiederà, "Hai provato a …" o "Non hai tenuto conto di questo?" Come risponderete? Prima di andare a dare la brutta notizia, potete provare ancora qualche altra cosa? A volte, *sapete* esattamente che cosa vi diranno, perciò risparmiate loro la fatica.

Invece di scuse, presentate opzioni. Non dite che non si può fare; spiegate che cosa si *può* fare per salvare la situazione. Bisogna buttar via del codice? Convinceteli del valore del refactoring (vedi *Refactoring* nel Capitolo 6). Dovete passare del tempo a costruire prototipi per stabilire il modo migliore di procedere (vedi *Prototipi e Post-it* nel Capitolo 2)? Dovete introdurre test migliori (vedi *Codice facile da sottoporre a test*, nel Capitolo 6, e *Test senza pietà*, nel Capitolo 8) o una migliore automazione (vedi *Automazione onnipresente*, nel Capitolo 8), per evitare che succeda di nuovo? Forse avete bisogno di altre risorse. Non abbiate paura di chiedere, o di ammettere che avete bisogno di aiuto.

Provate a escludere le scuse patetiche prima di dirle ad alta voce. Se è il caso, raccontatele prima al vostro gatto. In fin dei conti, se proprio deve prendersi la colpa…

Vedi anche

- *Prototipi e Post-it*, Capitolo 2
- *Refactoring*, Capitolo 6
- *Codice facile da sottoporre a test*, Capitolo 6
- *Automazione onnipresente*, Capitolo 8
- *Test senza pietà*, Capitolo 8

Sfide

- Come reagite se qualcuno, per esempio un cassiere di banca, un meccanico d'auto o un impiegato, arriva da voi con una scusa patetica? Che cosa pensate di quella persona e della sua azienda?

Entropia del software

Lo sviluppo del software non risponde a quasi nessuna delle leggi fisiche, ma l'*entropia* ci tormenta con forza. *Entropia* è un termine della fisica, che indica il grado di "disordine" di un sistema. Purtroppo, le leggi della termodinamica dicono che l'entropia nell'universo tende a crescere. Quando il disordine aumenta nel software, i programmatori parlano di *software rot*, decomposizione del software.

Molti fattori contribuiscono al *software rot*, ma il più importante sembra sia la psicologia, o la cultura, che entra in gioco in un progetto. Anche se lavorate da soli, la vostra psicologia di progetto può essere una cosa molto delicata. Nonostante i piani meglio tracciati e le persone migliori, un progetto può comunque andare a gambe all'aria e soffrire di decadimento o decomposizione. Esistono invece altri progetti che, nonostante difficoltà enormi e intoppi continui, riescono a combattere con successo la tendenza naturale al disordine e riescono a finire nel migliore dei modi.

Che cosa fa la differenza?

Nelle città, certi edifici sono belli e ben tenuti, altri sono un disastro. Perché? I ricercatori che si occupano di criminalità e decadimento urbano hanno scoperto un meccanismo affascinante, in grado di trasformare

rapidamente un edificio pulito, integro e abitato in un rudere abbandonato [WK82].

Una finestra rotta.

Una finestra rotta, non riparata per un tempo abbastanza lungo, instilla in chi abita nell'edifico un senso di abbandono, la percezione che chi ne avrebbe il potere non ha alcun interesse per quel palazzo. Così viene rotta un'altra finestra. Le persone lasciano spazzatura ovunque. Compaiono i graffiti. Iniziano i danni strutturali seri. In un arco di tempo relativamente breve, l'edificio si danneggia al punto che il proprietario non ha più voglia di pensare a risistemarlo e il senso di abbandono diventa realtà.

La "teoria della finestra rotta" ha convinto la polizia di New York e di altre grandi città a essere molto rigida sulle piccole cose, per evitare quelle più grandi. E funziona: l'attenzione alle finestre rotte, ai graffiti e altre piccole infrazioni ha ridotto il livello dei crimini più gravi.

SUGGERIMENTO 4
Non vivete con le finestre rotte.

Non lasciate che le "finestre rotte" (progetti mal fatti, decisioni sbagliate o codice di scarsa qualità) non vengano riparate. Sistematele non appena le scoprite. Se non c'è tempo sufficiente per sistemarle bene, almeno *tappatele con un asse di legno*. Forse potete trasformare in commento il codice che non va, o presentare un messaggio "non implementato" o sostituire con dati fittizi. Fate qualcosa, per evitare danni ulteriori e per far vedere che avete la situazione sotto controllo.

Abbiamo visto sistemi puliti e funzionali deteriorarsi molto rapidamente, non appena qualche finestra cominciava a rompersi. Altri fattori contribuiscono al *software rot*, e ne vedremo ancora qualcuno più avanti, ma la trascuratezza *lo accelera* molto più di qualsiasi altro fattore.

Forse penserete che nessuno ha il tempo di andare in giro e sistemare tutti i vetri rotti di un progetto. Se continuate a pensarla così, meglio pensare all'acquisto di un cassonetto della spazzatura, o a trasferirvi in un altro quartiere. Non lasciate che l'entropia vinca.

Estinguere l'incendio

Andy aveva un conoscente incredibilmente ricco: la sua casa era immacolata, bellissima, piena di antichità senza prezzo, oggetti d'arte e

così via. Un giorno, un arazzo appeso troppo vicino al camino del soggiorno ha preso fuoco. I pompieri sono subito accorsi, ma prima di portare in casa le loro grosse e sporche manichette antincendio, si sono fermati, con l'incendio che divampava, per stendere un materassino fra la porta e la fonte dell'incendio.

Non volevano rovinare il tappeto.

Un caso estremo, certo, ma è così che deve essere con il software. Una finestra rotta (un pezzo di codice mal progettato, una cattiva decisione gestionale con cui il team deve convivere per tutto il progetto) è l'unica cosa che serve perché il declino inizi. Se vi ritrovate a lavorare su un progetto con parecchie finestre rotte, è facilissimo scivolare nella mentalità "Tutto il resto di questo codice è spazzatura. Seguirò l'esempio". Non importa se il progetto è andato benissimo fino a quel punto. Nell'esperimento originale che ha portato alla "teoria della finestra rotta", un'automobile abbandonata è rimasta intatta per una settimana, ma non appena qualcuno ha rotto un finestrino, l'auto è stata completamente vandalizzata nell'arco di *ore*.

Analogamente, se vi ritrovate in una squadra e in un progetto dove il codice è splendido, pulito, ben progettato, elegante, con tutta probabilità starete più che attenti a non fare danni, come i vigili del fuoco. Anche se c'è un incendio (scadenza, data di rilascio, demo per una fiera ecc.), non vorrete essere i primi a fare un pasticcio.

Vedi anche

- *Zuppa di pietre e rane bollite*, in questo Capitolo
- *Refactoring*, Capitolo 6
- *Team pragmatici,* Capitolo 8

Sfide

- Rafforzate la vostra squadra facendo un'indagine sul vostro "quartiere" informatico. Scegliete due o tre "finestre rotte" e analizzate con i vostri colleghi quali siano i problemi e che cosa si potrebbe fare per risolverli.
- Potete stabilire quando viene rotta una finestra? Qual è la vostra reazione? Se è il risultato della decisione di qualcun altro, o un'imposizione del management, che cosa potete farci?

Zuppa di pietre e rane bollite

Tre soldati stanno tornando a casa dalla guerra, e sono affamati. Quando vedono un villaggio davanti a loro, il loro morale si risolleva: certo gli abitanti daranno loro qualcosa da mangiare. Quando arrivano, però, trovano le porte sbarrate e le finestre chiuse. Dopo molti anni di guerra, gli abitanti del villaggio erano a corto di cibo e accumulavano il poco che avevano.

Senza scompaginarsi, i soldati mettono a bollire una pentola d'acqua e ci mettono dentro tre pietre. Gli abitanti del villaggio, stupiti, escono a guardarli.

"Questa è zuppa di pietre", spiegano i soldati. "E ci mettete solo quello?", chiedono gli abitanti. "Assolutamente - anche se qualcuno dice che ha un sapore ancora migliore con qualche carota…" Uno degli abitanti del villaggio corre via e in un attimo ritorna con un cesto di carote del suo tesoro.

Un paio di minuti dopo, gli abitanti del villaggio chiedono ancora: "Tutto qui?"

"Beh", dicono i soldati, "un paio di patate gli danno più sostanza." E un altro abitante del villaggio corre via.

Nel corso dell'ora successiva, i soldati elencano altri ingredienti che potrebbero migliorare la zuppa: manzo, porri, sale, erbe aromatiche. Ogni volta un diverso abitante del villaggio corre via per pescare qualcosa dalle sue riserve personali.

Alla fine, hanno cucinato una grossa pentola di zuppa fumante. I soldati tolgono le pietre e si mettono a tavola con tutto il villaggio, per godersi il primo vero pasto che tutti loro hanno fatto da mesi.

Ci sono un paio di morali da ricavare dalla storia della zuppa di pietre. Gli abitanti del villaggio sono presi in giro dai soldati, che usano la loro curiosità per farsi dare del cibo. Cosa più importante, però, i soldati fungono da catalizzatori: radunano il villaggio e così producono insieme qualcosa che nessuno di loro isolatamente avrebbe potuto produrre: un risultato sinergico. Alla fine, vincono tutti.

Ogni tanto, potreste voler emulare i soldati.

Vi trovate magari in una situazione in cui sapete esattamente che cosa si debba fare e come farlo. Tutto il sistema è davanti ai vostri occhi, sapete che è tutto giusto. Ma chiedete il permesso di ribaltare il tutto e vi guarderanno a occhi sbarrati e non avrete altro che rinvii. Si formeranno commissioni, sarà necessario far approvare un budget, tutto diventerà complicato e tutti proteggeranno le loro risorse. Qualche volta la si definisce "la fatica dell'inizio".

È il momento di tirar fuori le pietre. Capite che cosa *potete* chiedere ragionevolmente. Sviluppatelo bene. Non appena l'avete fatto, mostratelo alla gente, perché si meravigli. Poi potete dire "certo, sarebbe meglio se aggiungessimo…". Fate finta che non sia importante. Mettetevi comodi e aspettate che siano loro a chiedervi di aggiungere le funzionalità che volevate sin dall'inizio. È più facile aggregarsi a un successo in corso. Mostrate un barlume di futuro e tutti vi saranno intorno. (Forse vi sarà di conforto una frase attribuita al viceammiraglio Grace Hopper: "È più facile chiedere perdono che ottenere il permesso".)

SUGGERIMENTO 5
Siate catalizzatori del cambiamento.

Dalla parte degli abitanti del villaggio

La storiella della zuppa di pietre è anche un caso di inganno, delicato e graduale, e dice qualcosa sul focalizzarsi in modo ristretto. Gli abitanti del villaggio pensano alle pietre e si dimenticano del resto del mondo. Ci caschiamo tutti, ogni giorno. Le cose ci colgono impreparati.

Abbiamo visto tutti i sintomi. I progetti, lentamente ma inesorabilmente, finiscono per sfuggire di mano. Quasi tutti i disastri del software iniziano troppo piccoli perché qualcuno li noti, e la maggior parte degli sforamenti avvengono un giorno alla volta. Il sistema si allontana dalle specifiche una caratteristica alla volta, mentre si inserisce nel codice una pezza dopo l'altra, finché non resta nulla dell'originale. Spesso è l'accumulo di piccole cose che spezza il morale e distrugge le squadre.

SUGGERIMENTO 6
Ricordate il quadro generale.

Questo non l'abbiamo mai sperimentato, ma dicono che, se si prende una rana e la si butta nell'acqua bollente, salterà fuori subito. Se si mette invece la rana in una pentola piena d'acqua fredda e la si scalda gradualmente, la rana non si accorgerà del lento aumento della temperatura e rimarrà tranquilla finché non sarà cotta.

Notate che il problema della rana è diverso da quello della teoria della finestra rotta: in questo secondo caso, le persone perdono la volontà di

combattere l'entropia perché hanno la sensazione che non interessi a nessun altro. La rana semplicemente non nota invece il cambiamento.

Non fate coma la rana. Tenete d'occhio il quadro generale. Esaminate costantemente quel che succede intorno a voi, non solo quello che state facendo personalmente.

Vedi anche

- *Entropia del software*, in questo Capitolo
- *Programmazione per coincidenza*, Capitolo 6
- *Refactoring*, Capitolo 6
- *La fossa dei requisiti*, Capitolo 7
- *Team pragmatici*, Capitolo 8

Sfide

- Nel riguardare una bozza di questo libro, John Lakos ha sollevato questo problema. I soldati ingannano progressivamente gli abitanti del villaggio, ma il cambiamento che catalizzano è un bene per tutti. Ingannando la rana, invece, con il riscaldare l'acqua lentamente, le si fa del male. Potete stabilire se preparate una zuppa di pietre o una zuppa di rane, quando tentate di catalizzare un cambiamento? La decisione è soggettiva o oggettiva?

Software abbastanza buono

Cercando il meglio, spesso guastiamo il bene.
- Shakespeare, *Re Lear*, Atto I, scena IV

Una vecchia barzelletta parla di una azienda americana che ordina 100.000 circuiti integrati a un produttore giapponese. Fra le specifiche, c'è anche il tasso tollerabile di difetti: un chip su 10.000. Qualche settimana dopo arriva l'ordine: una grande cassa contenente migliaia di circuiti integrati e una scatolina che ne contiene solo dieci. Sulla scatolina un'etichetta che dice "Questi sono quelli che non funzionano".

Sarebbe bello se avessimo un simile controllo sulla qualità, ma il mondo reale non ci permette di produrre molte cose davvero perfette, in particolare non software privo di "bachi". Tempo, tecnologia e temperamento sono tutti contro di noi.

Questo non deve essere per forza frustrante. Come descriveva Ed Yourdon in un articolo per *IEEE Software* [You95], ci si può abituare a scrivere software che sia abbastanza buono - abbastanza buono per gli utenti, per chi dovrà manutenerlo in futuro, per la vostra tranquillità mentale. Scoprirete che siete più produttivi e i vostri utenti sono più felici. Potreste anche scoprire che i vostri programmi sono effettivamente migliori, nonostante la più breve incubazione.

Prima di procedere oltre, dobbiamo precisare quel che stiamo dicendo. L'espressione "abbastanza buono" non vuole alludere a codice approssimativo o prodotto in modo scadente. Tutti i sistemi devono soddisfare i requisiti degli utenti, per avere successo. Stiamo semplicemente difendendo l'idea che agli utenti deve essere data la possibilità di partecipare alla decisione, se quel che avete prodotto è abbastanza buono.

Coinvolgete gli utenti nel compromesso

Normalmente, scriverete software per altri. Spesso vi ricorderete di ottenere da loro i requisiti che vogliono. (Era una battuta!) Ma quanto spesso chiedete loro *quanto buono* vogliono che sia il loro software? A volte non c'è scelta. Se lavorate a pacemaker, allo space shuttle o a una libreria di basso livello che verrà ampiamente diffusa, i requisiti saranno molto più rigidi e il vostro spazio di manovra più limitato. Se lavorate invece a un nuovo gadget, avrete vincoli diversi. Il marketing avrà promesse da mantenere, gli utenti finali avranno fatto dei piani sulla base di un tempo di consegna e la vostra azienda sicuramente avrà dei vincoli legati ai flussi di cassa. Non sarebbe professionale ignorare questi requisiti degli utenti semplicemente per aggiungere al programma nuove funzionalità o per ripulire il codice ancora una volta. Non stiamo neanche propugnando il terrore: è altrettanto poco professionale promettere tempistiche impossibili e prendere scorciatoie per rispettare una scadenza.

La portata e la qualità devono essere specificate fra i requisiti del sistema.

SUGGERIMENTO 7
Fate della qualità un aspetto delle specifiche.

Spesso vi troverete in situazioni in cui sono necessari dei compromessi. A sorpresa, molti utenti preferiranno usare del software con qualche fronzolo in meno *oggi* che aspettare un anno per la versione multimediale. Molti reparti IT con budget limitati saranno d'accordo. Un buon software oggi è spesso preferibile a un software perfetto domani. Se date ai vostri utenti presto qualcosa con cui giocare, il loro feedback spesso vi porterà a una soluzione migliore finale (vedi *Proiettili traccianti*, nel Capitolo 2).

Sapere quando fermarsi

Per certi aspetti, programmare è come dipingere. Si parte da una tela bianca e da certe materie prime fondamentali. Si usa una combinazione di scienza, arte e capacità artigianali per decidere che cosa farne. Si schizza una forma complessiva, si dipinge l'ambiente intorno, si completano i dettagli. Ci si allontana costantemente con occhio critico per guardare quel che si è fatto. Ogni tanto si butta via la tela e si ricomincia da capo.

Gli artisti vi diranno però che tutto il lavoro fatto è fatica sprecata, se non si sa quando smettere. Se si aggiunge uno strato sopra l'altro, un particolare sopra l'altro, *il quadro si perde nella pittura*.

Non rovinate un programma perfettamente valido con abbellimenti eccessivi e un eccesso di perfezionamento. Andate avanti, e lasciate che il codice se la cavi da solo per un po'. Magari non sarà perfetto. Non preoccupatevi: non sarà mai perfetto. (Nel Capitolo 5 analizzeremo le filosofie per sviluppare codice in un mondo imperfetto.)

Vedi anche

- *Proiettili traccianti*, Capitolo 2
- *La fossa dei requisiti*, Capitolo 7
- *Team pragmatici*, Capitolo 8
- *Grandi speranze*, Capitolo 8

Sfide

- Pensate alle aziende che producono gli strumenti software e i sistemi operativi che usate. Potete trovare qualche prova che queste aziende siano a loro agio nel pubblicare software che sanno non essere perfetto? Da utente, preferireste

(1) aspettare che abbiano eliminato tutti i bachi, (2) avere software complesso e accettare qualche baco o (3) optare per software più semplice ma con meno difetti?

- Considerate l'effetto della modularità sulla realizzazione del software. Ci vorrà meno o più tempo per portare un blocco monolitico di software alla qualità richiesta, rispetto a un sistema progettato a moduli? Potete trovare esempi commerciali?

Il portafoglio delle conoscenze

Un investimento in conoscenza paga sempre l'interesse migliore.
- Benjamin Franklin

Il buon vecchio Ben Franklin, mai a corto di belle frasi istruttive. In questo caso, però, ci prende in pieno. Le vostre conoscenze e la vostra esperienza sono il patrimonio professionale più importante. Si tratta però, purtroppo, di *beni con una scadenza*, il cui valore diminuisce con il tempo (come un magazzino pieno di banane e il biglietto per un evento sportivo). Le conoscenze diventano obsolete, perché si sviluppano nuove tecniche, nuovi linguaggi e nuovi ambienti. Le mutevoli forze del mercato possono rendere la vostra esperienza obsoleta o irrilevante. Data la velocità a cui trascorrono gli anni-web, può succedere molto in fretta.

Se diminuisce il valore della vostra conoscenza, diminuisce il vostro valore per la vostra azienda o per i clienti. E non vogliamo che succeda mai.

Il vostro portafoglio di conoscenze

Tutto ciò che i programmatori sanno di informatica e dei campi applicativi in cui lavorano e tutta la loro esperienza costituiscono quello che chiamiamo il loro *portafoglio di conoscenze*. Gestire un portafoglio di conoscenze è molto simile a gestire un portafoglio finanziario.

1. Gli investitori seri investono regolarmente, d'abitudine.
2. La diversificazione è la chiave del successo sul lungo termine.
3. Gli investitori furbi bilanciano i loro portafogli tra investimenti prudenti e investimenti ad alto rischio e alto guadagno.

4. Gli investitori cercano di acquistare a basso prezzo e di vendere ad alto prezzo per massimizzare il loro ritorno.
5. Il portafoglio deve essere passato in rassegna e riequilibrato periodicamente.

Per avere successo nella vostra carriera, dovete gestire il vostro portafoglio di conoscenze seguendo le stesse direttive.

Costruire il proprio portafoglio

- *Investite regolarmente*. Come in campo finanziario, dovete investire nel vostro portafoglio di conoscenze *con regolarità*. Anche se si tratta solo di poca cosa, l'abitudine in sé è importante quanto le somme investite. Qualche obiettivo campione è elencato nella prossima sezione.
- *Diversificate*. Quante più cose *diverse* sapete, tanto più sarete preziosi. Come linea di base, dovete conoscere in tutti i suoi aspetti la particolare tecnologia con cui state lavorando al momento, ma non dovete fermarvi lì. L'aspetto dell'informatica cambia rapidamente, la tecnologia calda oggi magari sarà quasi inutile (o almeno poco richiesta) domani. Se siete a vostro agio con più tecnologie, sarete più in grado di adattarvi ai cambiamenti.
- *Gestite il rischio.* Le tecnologie si dispongono lungo un ampio spettro, da quelle rischiose ma potenzialmente ad alto guadagno fino agli standard a basso rischio e scarso guadagno. Non è una buona idea investire tutto il proprio denaro in azioni ad alto rischio che potrebbero crollare all'improvviso, né investire tutto in modo prudente e perdersi qualche possibile occasione. Non mettete tutte le vostre uova tecniche in un unico paniere.
- *Comprate basso, vendete alto*. Imparare una tecnologia emergente prima che diventi popolare può essere difficile quanto scovare un'azione sottovalutata, ma il compenso può essere altrettanto gratificante. Imparare Java quando è venuto per la prima volta alla ribalta è stato forse rischioso, ma ha pagato molto bene per gli "early adopter" che ora sono in vetta in questo campo.

- *Rivedete e riequilibrate*. Questo è un settore molto dinamico. Quella tecnologia calda che avete cominciato a esplorare il mese scorso potrebbe essere già morta. Magari dovete dare una spolverata a quella tecnologia di database che non usate da un po', o magari sareste in una posizione migliore per un'occasione lavorativa se provaste quell'altro linguaggio…

Di tutte queste indicazioni, la più importante è anche la più semplice:

SUGGERIMENTO 8
Investite regolarmente nel vostro portafoglio di conoscenze.

Obiettivi

Ora che avete qualche indicazione su che cosa e quando aggiungere al vostro portafoglio di conoscenze, qual è il modo migliore per acquisire capitale intellettuale con cui finanziare il vostro portafoglio? Ecco qualche suggerimento.

- *Imparate almeno un nuovo linguaggio ogni anno*. Linguaggi diversi risolvono gli stessi problemi in modi diversi. Imparando molti approcci diversi, potete allargare il vostro modo di pensare ed evitare di finire impantanati. Inoltre, imparare molti linguaggi è molto più facile oggi, grazie a tutto il software liberamente disponibile in Internet (vedete l'Appendice A).
- *Leggete un libro tecnico ogni trimestre*. Le librerie sono piene di libri tecnici su temi interessanti legati al vostro progetto corrente. Una volta presa l'abitudine, leggete un libro al mese. Quando avrete padroneggiato le tecnologie che state usando, andate oltre e studiatene qualcuna che non sia legata al progetto che state seguendo.
- *Leggete anche libri non tecnici*. È importante ricordare che i computer sono usati da *persone*, persone di cui cercate di soddisfare i bisogni. Non dimenticate il lato umano dell'equazione.
- *Seguite dei corsi*. Cercate corsi interessanti all'università più vicina, oppure alle manifestazioni fieristiche e ai convegni che si svolgono nella vostra città.

- *Partecipate ai gruppi di utenti locali*. Non limitatevi ad ascoltare, ma partecipate attivamente. L'isolamento può essere esiziale per la carriera: scoprite a che cosa lavorano le persone fuori dalla vostra azienda.
- *Sperimentate con ambienti diversi*. Se avete lavorato solo con Windows, giocate con Unix a casa (Linux, che è disponibile gratuitamente, è perfetto allo scopo). Se avete usato solo `makefile` e un editor, provate un IDE (un ambiente di sviluppo integrato, di tipo grafico) e viceversa.
- *Rimanete aggiornati*. Abbonatevi a riviste di settore e ad altre riviste (vedete l'Appendice A per qualche consiglio). Sceglietene qualcuna che tratti di tecnologie diverse da quelle utilizzate nel progetto che state seguendo.
- *Restate connessi*. Volete sapere tutto di un nuovo linguaggio o di qualche altra tecnologia? I newsgroup sono un ottimo modo per scoprire quali esperienze stanno facendo altre persone, il particolare gergo che usano e così via. Navigate nel Web alla ricerca di articoli, siti commerciali e di qualsiasi altra fonte di informazione.

È importante continuare a investire. Quando vi sentite a vostro agio con un nuovo linguaggio o una certa tecnologia, andate oltre e imparate qualcosa di nuovo.

Non importa se userete mai una di queste tecnologie in un progetto, o nemmeno se le inserirete nel vostro curriculum. Il processo di apprendimento amplierà la vostra capacità di pensare, vi aprirà a nuove possibilità e nuovi modi di fare le cose. La fecondazione incrociata delle idee è importante; provate ad applicare al progetto che state seguendo quello che avete imparato. Anche se il progetto non usa quella tecnologia, magari potete mutuarne qualche buona idea. Se acquisite familiarità con gli oggetti, per esempio, scriverete in modo diverso anche i programmi in C puro e semplice.

Occasioni di apprendimento

Dunque, leggete voracemente, siete aggiornati su tutti gli ultimi sviluppi nel vostro campo (e non è una cosa facile), e qualcuno vi fa una

domanda. Voi non avete la minima idea di quale sia la risposta e lo ammettete tranquillamente.

Non fermatevi lì. Prendete come una sfida personale trovare la risposta. Chiedete a un guru. (Se non avete un guru in ufficio, dovreste riuscire a trovarne uno in Internet: leggete il riquadro seguente.) Cercate nel Web. Andate in biblioteca. (Nell'era del Web, molti sembra si siano dimenticati che esistono biblioteche reali, concrete, piene di materiali di ricerca e di personale.)

Se non riuscite a trovare la risposta voi, trovate chi *sia in grado di trovarla*. Non fermatevi. Parlare con altri vi aiuterà a costruire la vostra rete personale, e magari lungo la strada scoprirete delle soluzioni ad altri problemi che non c'entrano nulla. E quel vecchio portafoglio intanto continua a diventare sempre più grande.

Tutto questo leggere e fare ricerche richiede tempo, e il tempo è sempre scarso. Perciò dovete pianificare in anticipo. Dovete avere sempre qualcosa da leggere, nel caso di un momento morto. Il tempo nella sala d'attesa del medico o del dentista può essere una grande opportunità per mettervi in pari con la lettura, ma ricordatevi di portare con voi la vostra rivista, altrimenti vi toccherà sfogliare un vecchio articolo del 1973 sulla Nuova Guinea.

Cura e coltivazione dei guru

Con l'adozione globale di Internet, i guru sono a distanza di un tasto. Allora, come trovarne uno e come far sì che parli con voi?
Ci sono alcuni trucchi molto semplici.

- Dovete sapere esattamente che cosa volete chiedere, e dovete essere il più specifici possibile.
- Formulate la vostra domanda con cura e con cortesia. Ricordate che state chiedendo un favore: non dovete dare l'impressione che consideriate dovuta una risposta.
- Una volta formulata la domanda, fermatevi e guardatevi ancora in giro in cerca della risposta. Scegliete qualche parola chiave adeguata e cercate nel Web. Esaminate le opportune FAQ (elenchi di domande frequenti, o *Frequently Asked Questions*, con relativa risposta).
- Decidete se volete chiedere in pubblico o in privato. I newsgroup di Usenet sono meravigliosi luoghi di incontro per esperti di quasi qualsiasi argomento, ma qualcuno non ama la natura pubblica di questi gruppi. In alternativa, potete sempre scrivere un'email al vostro guru direttamente. In ogni caso, indicate una cosa sensata nell'oggetto. ("Aiuto!!!" non funziona.)
- Mettetevi seduti e aspettate con pazienza. Le persone hanno molto da fare, ci potrebbero volere giorni prima di ricevere una risposta specifica.

- Infine, non dimenticate di ringraziare chiunque vi risponda. E se vedete che qualcuno fa una domanda a cui *voi* sapete dare una risposta, fate la vostra parte e partecipate.

Pensiero critico

L'ultimo punto importante è pensare *criticamente* a quello che leggete e sentite. Dovete essere sicuri che le conoscenze nel vostro portafoglio siano accurate e non viziate dagli interessi dei produttori o dalle chiacchiere dei media. State attenti ai fanatici per i quali c'è un dogma che dà l'*unica* risposta: magari va bene anche a voi e al vostro progetto, e magari no.

Non sottovalutate la forma commerciale. Solo perché un motore di ricerca mette al primo posto un certo risultato non vuol dire che sia la cosa migliore: qualcuno può aver pagato per arrivare in prima posizione. Solo perché una libreria mette in vetrina un certo libro non vuol dire che sia un buon libro, e nemmeno che sia un bestseller: magari il libraio è stato pagato per collocarlo bene in vista.

SUGGERIMENTO 9
Analizzate criticamente quello che leggete e ascoltate.

Purtroppo, le risposte semplici sono sempre più rare, ma con un portafoglio molto ampio e utilizzando un po' di analisi critica sul fiume di pubblicazioni tecniche che leggerete, potrete capire le risposte *complesse*.

Sfide

- Cominciate a imparare un nuovo linguaggio questa settimana. Avete sempre programmato in C++? Provate Smalltalk [URL 13] o Squeak [URL 14]. Lavorate con Java? Provate Eiffel [URL 10] o TOM [URL 15]. Vedete nell'Appendice A dove trovare altri compilatori e ambienti gratuiti.
- Iniziate a leggere un nuovo libro (ma prima finite questo!). Se siete impegnati in implementazione e codifica a livello molto dettagliato, leggete un libro su progettazione e architettura. Se state facendo progetti ad alto livello, leggete un libro sulle tecniche di codifica.
- Uscite e parlate di tecnologia con persone che non sono coinvolte nel vostro attuale progetto, o che non lavorano per la stessa azienda. Chiacchierate con qualcuno nella mensa aziendale, oppure cercate qualche collega entusiasta alle riunioni di un gruppo di utenti locali.

Comunicate!

> Penso sia meglio essere spogliata con gli occhi che essere trascurata.
> - Mae West, *Belle of the Nineties*, 1934

Forse possiamo imparare qualcosa dalla signora West. Non importa solo quello che si è fatto, ma anche come lo si è confezionato. Le idee migliori, il codice più fine o il pensiero più pragmatico alla fine sono sterili se non si sa comunicare con gli altri. Senza una comunicazione efficace una buona idea è orfana.

Da sviluppatori, dobbiamo comunicare a molti livelli. Passiamo ore in riunioni ad ascoltare e parlare. Lavoriamo con gli utenti finali, e cerchiamo di capire di che cosa hanno bisogno. Scriviamo codice, che comunica le nostre intenzioni a una macchina e documenta quello che pensiamo per generazioni future di sviluppatori. Scriviamo proposte e memorandum per chiedere e giustificare risorse, per riferire sullo stato di avanzamento dei lavori e per suggerire nuovi approcci. Lavoriamo quotidianamente con la nostra squadra per sostenere le nostre idee, modificare le pratiche esistenti e suggerirne di nuove. Passiamo gran parte della nostra giornata comunicando, perciò dobbiamo farlo bene.

Abbiamo assemblato una lista di idee che troviamo utili.

Dovete aver chiaro che cosa volete dire

Probabilmente la parte più difficile degli stili di comunicazione più formali utilizzati nel mondo del lavoro è stabilire esattamente che cosa si vuole dire. Gli scrittori elaborano la trama dei loro romanzi prima di iniziare, ma chi scrive documenti tecnici spesso si siede alla tastiera, scrive “1- Introduzione”, e continua con quello che gli viene in testa.

Pianificate che cosa volete dire. Scrivete una scaletta. Poi chiedetevi: “Così riesco a far capire quel che voglio dire?”. Perfezionatela finché non potete rispondere “sì”.

Questo metodo non si applica solo alla scrittura di documenti. Se dovete affrontare una riunione importante, o una telefonata con un cliente di primo piano, buttate già le idee che volete comunicare e pianificate un paio di strategie per farle arrivare a destinazione.

Dovete conoscere il vostro pubblico

Si comunica solo se si veicolano informazioni. Per questo, bisogna capire i bisogni, gli interessi e le capacità del pubblico che si ha di fronte. È capitato a tutti di assistere a una riunione in cui un mago della tecnologia fa gelare lo sguardo del responsabile marketing con un lungo monologo sui pregi di qualche strano strumento. Questo non è comunicare: è solo parlare, ed è anche seccante e noioso. Fatevi una buona immagine mentale del vostro pubblico. L'acrostico WISDOM (Figura 1.1) può essere utile.

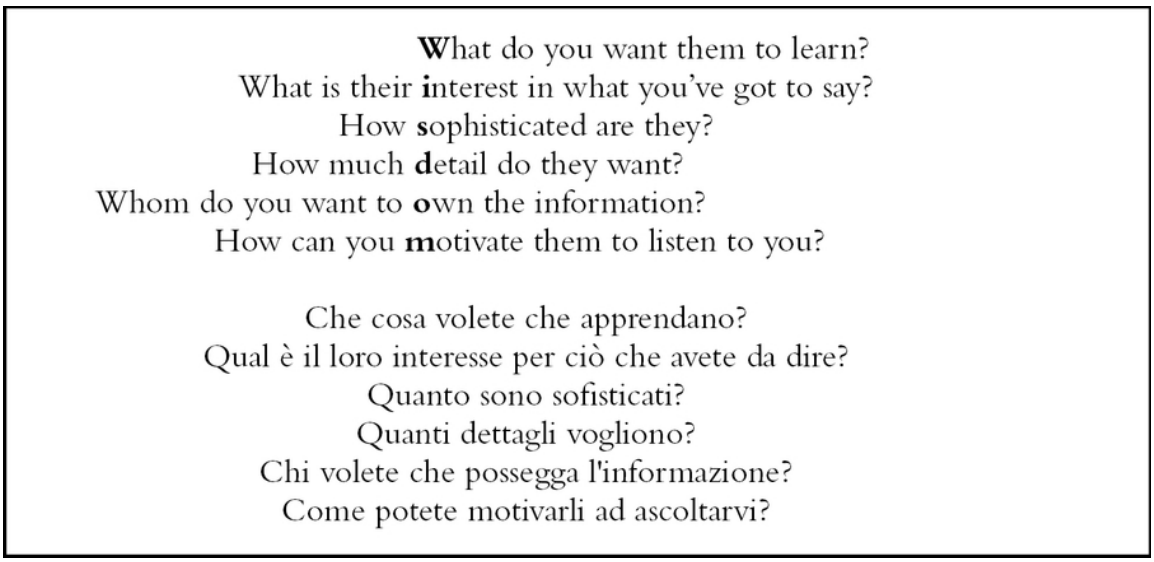


Figura 1.1 L'acrostico WISDOM (e una traduzione, che purtroppo non permette di conservare la mnemonica): comprendere il pubblico.

Supponiamo che vogliate proporre un sistema basato sul Web che consenta ai vostri utenti finali di inviare segnalazioni di errori. Potete presentare questo sistema in molti modi diversi, a seconda di chi vi sta di fronte. Gli utenti finali apprezzeranno il fatto di poter inoltrare segnalazioni di errore in qualsiasi ora del giorno o della notte senza attendere al telefono. Il marketing potrà utilizzare questo fatto per incrementare le vendite. I responsabili dell'assistenza avranno due motivi per essere felici: avranno bisogno di meno personale e la segnalazione dei problemi sarà automatizzata. Infine, gli sviluppatori saranno magari felici di fare esperienza con le tecnologie client-server del Web e con un nuovo motore di database. Esponendo la vostra idea nel modo più adatto per

ciascun gruppo, potrete contare sull'appoggio di tutti per il vostro progetto.

Scegliete il momento giusto

Sono le sei di venerdì pomeriggio. Per tutta la settimana sono stati in ufficio i revisori dei conti. Il figlio più piccolo del vostro capo è in ospedale. Piove a catinelle, e il viaggio di ritorno a casa sarà un incubo. Probabilmente non è il momento migliore per chiederle un aumento della memoria del vostro PC.

Quando si dice che dovete capire che cosa vuol sentire il vostro pubblico, si vuol dire che dovete anche stabilire quali sono le sue priorità. Se trovate un manager che ha appena ricevuto una lavata di testa dal suo superiore perché è andato perso del codice sorgente, presterà ascolto più attentamente alle vostre idee su un respository del codice sorgente. Fate in modo che quello che dite sia rilevante per il suo contenuto ma anche per il momento in cui lo dite. A volte basta una domanda semplice: "È un buon momento per parlare di …?".

Scegliete uno stile

Adattate lo stile della vostra presentazione al vostro pubblico. Qualcuno vuole una presentazione formale "solo i fatti"; altri preferiscono lunghe chiacchierate a ruota libera prima di entrare nel merito. Nel caso di documenti scritti, qualcuno vuole ricevere lunghe relazioni rilegate, altri si aspettano un semplice appunto o una email. Se avete dubbi, chiedete.

Ricordate, però, che rappresentate metà della transazione di comunicazione. Se qualcuno vi dice che gli basta una frase che descriva qualcosa, e pensate che non ci sia modo di farlo in meno di qualche pagina, diteglielo. Ricordate: anche questo tipo di feedback è una forma di comunicazione.

Che siano di bell'aspetto

Le vostre idee sono importanti e meritano un veicolo di bell'aspetto che le porti al vostro pubblico.

Molti sviluppatori (e i loro manager), quando devono produrre documenti scritti, si concentrano esclusivamente sul contenuto, ma è un errore. Qualsiasi chef vi dirà che affannarsi per ore in cucina non serve a nulla, se poi tutti gli sforzi sono vanificati da una brutta presentazione.

Oggi chi produce documenti a stampa di brutto aspetto non ha scusanti: con i word processor moderni (e con sistemi di impaginazione come LATEX e troff) si possono ottenere risultati molto eleganti. Bisogna imparare solo qualche comando di base, Se il word processor prevede i fogli stile, usateli. (La vostra azienda potrebbe aver già definito dei fogli stile che potete usare.) Imparate come definire testatine e piè di pagina. Esaminate i documenti campione inclusi nel pacchetto per ricavarne qualche idea su stile e impaginazione. *Controllate l'ortografia*, prima in automatico e poi manualmente. Ci sono errori che il controllore ortografico non rileva.

Coinvolgete il vostro pubblico

Spesso a noi succede che i documenti che produciamo finiscano per essere meno importanti del processo che seguiamo per produrli. Se possibile, coinvolgete i vostri lettori con le prime bozze del vostro documento. Sollecitate i loro feedback, solleticate il loro cervello. Ne nascerà una buona relazione di lavoro e probabilmente alla fine avrete anche un documento migliore.

Imparate ad ascoltare

C'è una tecnica che dovete usare se volete che le persone vi ascoltino: *ascoltarle*. Anche se è una situazione in cui voi avete tutte le informazioni, anche se è una riunione formale e vi ritrovate in piedi davanti a venti dirigenti: se voi non ascoltate loro, loro non ascolteranno voi.

Incoraggiate le persone a parlare facendo domande, o facendo loro riassumere quello che dite. Trasformate la riunione in un dialogo e quello

che dovete dire risulterà più efficace. Chissà, magari potreste anche finire per imparare qualcosa.

Comunicazioni via email
Tutto quel che abbiamo detto sulla comunicazione scritta vale anche per la posta elettronica. L'email si è ormai evoluta al punto da essere un cardine delle comunicazioni all'interno di ogni azienda e fra un'azienda e l'altra: la si usa per discutere contratti, risolvere dispute e come prova in tribunale. Per qualche motivo, però, persone che non manderebbero mai in giro un documento cartaceo malfatto non si preoccupano di lasciar circolare per il mondo email orribili.
I nostri consigli per le email sono semplici.

- Rileggete quel che avete scritto prima di premere Invia.
- Controllate l'ortografia.
- Mantenete semplice il formato. Qualcuno legge le email usando caratteri proporzionali, perciò le immagini artistiche ASCII che avete faticosamente costruito a loro sembreranno solo zampe di gallina.
- Usate il formato rich-text o HTML solo se sapete che tutti i vostri destinatari sono in grado di leggerlo. Il testo semplice va bene per tutti.
- Cercate di ridurre al minimo la riproduzione dei contenuti: a nessuno fa piacere vedersi tornare indietro la propria mail di cento righe con un semplice "sono d'accordo" aggiunto.
- Se citate la mail di altri, indicate chiaramente l'attribuzione e citate all'interno del messaggio (e non come allegato).
- Non iniziate dei *flame*, se non volete che poi vi si ritorcano contro.
- Controllate l'elenco dei destinatari, prima di spedire il messaggio. Recentemente un articolo del *Wall Street Journal* raccontava di un impiegato che aveva criticato il suo superiore usando l'email dell'ufficio, senza rendersi conto che il suo superiore era incluso nella lista di distribuzione.
- Archiviate e organizzate i messaggi di posta elettronica, le cose importanti che ricevete e i messaggi che inviate.

Come vari dipendenti di Microsoft e Netscape hanno scoperto durante l'indagine condotta nel 1999 dal Dipartimento della giustizia degli USA, l'email è per sempre. Prestate alle email la stessa attenzione e la stessa cura che prestereste a qualsiasi relazione scritta.

Rispondete alle persone

Se fate una domanda a qualcuno, se non vi risponde penserete che sia una persona maleducata. Ma quanto spesso non rispondete, se qualcuno vi manda una email o vi lascia un appunto chiedendovi informazioni o qualche azione? Nella convulsione della vita quotidiana, si fa presto a dimenticare. Rispondete sempre alle email e ai messaggi vocali, anche se la risposta fosse semplicemente "Ti chiamo più tardi". Se tenete

informate le persone, saranno molto più inclini a perdonarvi quando ogni tanto sbagliate, e avranno la sensazione che non le avete dimenticate.

SUGGERIMENTO 10

È importante quello che dite, ma anche il modo in cui lo dite.

A meno che lavoriate in un vuoto pneumatico, dovete saper comunicare. Quanto più efficace è la comunicazione, tanto più potrete esercitare la vostra influenza.

Riepilogo

- Decidete che cosa volete dire.
- Identificate il vostro pubblico.
- Scegliete il momento giusto.
- Scegliete uno stile.
- Date alla comunicazione un bell'aspetto.
- Coinvolgete il vostro pubblico.
- Imparate ad ascoltare.
- Rispondete alle persone.

Vedi anche

- *Prototipi e Post-it*, Capitolo 2
- *Team pragmatici*, Capitolo 8

Sfide

- Esistono molti buoni libri che contengono parti dedicate alla comunicazione nei team di sviluppo [Bro95, McC95, DL99]. Imponetevi di cercare di leggerli tutti e tre entro i prossimi 18 mesi. Inoltre, *Dinosaur Brains* [Ber96] analizza il bagaglio di emozioni che tutti portiamo nell'ambiente di lavoro.
- La prossima volta che dovrete fare una presentazione, o scrivere un memorandum in difesa di qualche posizione, provate ad applicare l'acrostico WISDOM (p. 14) prima di iniziare. Vedete se vi aiuta a comprende come posizionare quello che dite. Se non è sconveniente, parlate poi con chi vi ha ascoltato e cercate di capire quanto sia stata precisa la vostra valutazione dei loro bisogni.

Un approccio pragmatico

Vi sono suggerimenti e trucchi che valgono per tutti i livelli dello sviluppo di software, idee che sono quasi assiomatiche e processi praticamente universali. Raramente però sono documentati come tali: per lo più li troverete scritti come frasi un po' strane mentre si parla di design, gestione di progetto o codifica.

In questo capitolo vogliamo riunire queste idee e questi processi. Le prime due sezioni, *I mali della duplicazione* e *Ortogonalità*, sono strettamente legate fra loro. La prima ammonisce a non duplicare le conoscenze nei vostri sistemi, la seconda a non suddividere un pezzo di conoscenza fra più componenti del sistema.

Il ritmo del cambiamento è sempre più rapido, e diventa sempre più difficile mantenere rilevanti le nostre applicazioni. In *Reversibilità*, vedremo qualche tecnica utile a isolare i nostri progetti dai loro ambienti in mutamento.

Le successive due sezioni sono correlate. In *Proiettili traccianti*, parliamo di uno stile di sviluppo che permette di raccogliere requisiti, mettere alla prova progetti e implementare codice allo stesso tempo. Se vi sembra troppo bello per essere vero, avete ragione: lo sviluppo con i proiettili traccianti non è sempre applicabile. Quando non lo è, *Prototipi e Post-it* vi mostra come usare la prototipazione per mettere alla prova architetture, algoritmi, interfacce e idee.

Con la progressiva maturazione dell'informatica, vengono realizzati linguaggi di livello sempre più alto. Il compilatore che accetta in input "fai così" non è stato ancora inventato, ma in *Linguaggi di settore* presentiamo qualche suggerimento più modesto, che potete realizzare voi stessi.

Infine, lavoriamo tutti in un mondo in cui tempo e risorse sono limitati. Potete sopravvivere meglio a questa scarsità (e fare più felici i vostri capi) se siete capaci di valutare meglio quanto tempo richiederanno le diverse attività, cosa di cui parliamo in *Stime*.

Ricordando questi principi fondamentali nel corso dello sviluppo, potrete scrivere codice migliore, più veloce e più robusto. Potete persino farlo sembrare facile.

I mali della duplicazione

Dare a un computer due pezzi di informazione contraddittori era il modo preferito dal Capitano James T. Kirk per rendere inoffensiva un'intelligenza artificiale finita fuori strada. Purtroppo, lo stesso principio può essere efficace per mettere al tappeto il *vostro* codice.

Da programmatori raccogliamo, organizziamo, manuteniamo e imbrigliamo conoscenza. Documentiamo conoscenza nelle specifiche, le diamo vita con l'esecuzione del codice e la usiamo per le verifiche necessarie durante i test.

Purtroppo, però, la conoscenza non è stabile. Cambia, e spesso anche rapidamente. La vostra comprensione di un requisito può mutare dopo una riunione con il cliente. Il governo modifica una norma e una certa logica aziendale diventa obsoleta. I test possono dire che l'algoritmo scelto non funziona. Tutta questa instabilità significa che dobbiamo passare gran parte del nostro tempo in modalità "manutenzione", riorganizzando e riformulando la conoscenza nei nostri sistemi.

Molti pensano che la manutenzione inizi quando un'applicazione viene pubblicata, che significhi risolvere gli errori e migliorare le caratteristiche. Secondo noi sbagliano. I programmatori sono sempre in modalità manutenzione. La nostra comprensione cambia quotidianamente. Arrivano nuovi requisiti mentre stiamo progettando o codificando: magari cambia l'ambiente. Quale che sia la ragione, la manutenzione non è un'attività discreta, ma una parte di routine di tutto il processo di sviluppo.

Quando svolgiamo della manutenzione, dobbiamo trovare e modificare le rappresentazioni delle cose, quelle capsule di conoscenza immerse nell'applicazione. Il problema è che è facile duplicare conoscenza nelle

specifiche, nei processi e nei programmi che sviluppiamo e, quando accade, ci tiriamo addosso un incubo manutentivo - un incubo che inizia molto prima che l'applicazione venga rilasciata.

Secondo noi l'unico modo per sviluppare software in modo affidabile e per rendere ciò che sviluppiamo più facile da comprendere e da manutenere è seguire quello che chiamiamo il principio DRY: ogni elemento di conoscenza in un sistema deve avere un'unica rappresentazione, priva di ambiguità e autorevole.

Perché lo chiamiamo DRY?

SUGGERIMENTO 11
DRY: *Don't Repeat Yourself*. Non ripeterti.

L'alternativa è avere la stessa cosa espressa in due o più luoghi: se si cambia da una parte, bisogna ricordarsi di cambiarla anche dalle altre parti, altrimenti, come i computer degli alieni, il vostro programma sarà messo in ginocchio da una contraddizione. Non è questione di se vi ricorderete, ma di quando vi dimenticherete.

Vedrete spuntare il principio DRY ogni tanto in questo libro, spesso in contesti che non hanno nulla a che fare con la codifica. Secondo noi è uno degli utensili più importanti nella casetta degli attrezzi del programmatore pragmatico.

In questa sezione vedremo i problemi della duplicazione e consiglieremo qualche strategia generale per affrontarli.

Come nasce la duplicazione?

Quasi tutti i casi di duplicazione che abbiamo visto ricadevano in una di queste categorie.

- *Duplicazione imposta*. Gli sviluppatori pensano di non avere scelta: l'ambiente sembra richiedere una duplicazione.
- *Duplicazione inavvertita*. Gli sviluppatori non si rendono conto che stanno duplicando informazioni.
- *Duplicazione per impazienza*. Gli sviluppatori impigriscono e duplicano perché sembra più facile.
- *Duplicazione fra sviluppatori*. Persone diverse in un team (o in team diversi) duplicano un pezzo di informazione.

Vediamo i quattro casi più in dettaglio.

Duplicazione imposta

A volte sembra che la duplicazione ci sia imposta. Standard di progetto possono richiedere documenti che contengono informazioni duplicate, o documenti che contengono informazioni duplicate nel codice. Quando si punta a più piattaforme, ciascuna richiede linguaggi di programmazione, librerie e ambienti di sviluppo propri, il che ci fa duplicare definizioni e procedure condivise. Gli stessi linguaggi di programmazione richiedono certe strutture che duplicano informazioni. Tutti abbiamo lavorato in situazioni in cui ci sentivamo impossibilitati a evitare la duplicazione. Eppure spesso esistono modi per mantenere ciascun pezzo di informazione in un solo posto, rispettando il principio DRY e allo stesso tempo rendendoci più facile la vita. Ecco qualche tecnica.

Più rappresentazioni della stessa informazione. A livello di codifica, spesso abbiamo bisogno della stessa informazione rappresentata in forme diverse. Magari scriviamo un'applicazione client-server utilizzando linguaggi diversi su client e server, e dobbiamo rappresentare su entrambi qualche struttura condivisa. Forse ci serve una classe i cui attributi rispecchiano lo schema di una tabella di database. Magari state scrivendo un libro e volete includervi frammenti di programmi che dovrete anche compilare e sottoporre a test.

Con un po' di ingegnosità normalmente si può eliminare il bisogno di duplicazione. Spesso la risposta sta nello scrivere un semplice filtro o un generatore di codice. Strutture in più linguaggi si possono costruire a partire da una rappresentazione comune in metadati, utilizzando un semplice generatore di codice ogni volta che viene costruito il codice (un esempio è visibile nella Figura 3.4). I brani di codice in questo libro sono inseriti da un preprocessore ogni volta che formattiamo il testo. Il trucco è rendere attivo il processo: non può trattarsi di una conversione *una tantum*, altrimenti ci ritroviamo ancora a duplicare dati.

Documentazione nel codice. Ai programmatori si insegna a commentare il codice: buon codice ha molti commenti. Purtroppo, non si insegna mai *perché* il codice abbia bisogno di commenti: il codice cattivo *richiede* una gran quantità di commenti.

Il principio DRY ci dice di mantenere la conoscenza di basso livello nel codice, che è il suo posto, e lasciare i commenti per altre spiegazioni, di livello più alto. Altrimenti duplichiamo conoscenza e ogni cambiamento imporrà di cambiare sia il codice sia i commenti. I commenti inevitabilmente finiranno per diventare obsoleti, e commenti di cui non ci si può fidare sono peggio dell'assenza di ogni commento. (Vedi *È tutta scrittura*, nel Capitolo 8, per maggiori informazioni sui commenti.)

Documentazione e codice. Scrivete documentazione, poi scrivete codice. Qualcosa cambia, sistemate la documentazione e aggiornate il codice. Tanto la documentazione quanto il codice contengono rappresentazioni della stessa conoscenza, e tutti sappiamo che sotto pressione, con le scadenze che incombono e clienti importanti che sbraitano, si tende a rimandare l'aggiornamento della comunicazione.

Dave una volta ha lavorato a una centralina di commutazione internazionale. Comprensibilmente, il cliente pretendeva una specifica esaustiva dei test e voleva che il software superasse tutti i testi a ogni consegna. Per essere sicuro che i test rispecchiassero accuratamente le specifiche, il team li generava mediante un programma dal documento stesso. Quando il cliente modificava le sue specifiche, la suite di test si modificava automaticamente. Quando il team è riuscito a convincere il cliente che il procedimento era corretto, la generazione di test di accettazione normalmente richiedeva solo pochi secondi.

Problemi di linguaggio. Molti linguaggi impongono duplicazioni notevoli nel sorgente. Spesso accade quando il linguaggio separa l'interfaccia di un modulo dalla sua implementazione. C e C++ hanno file di intestazione che duplicano le informazioni su nome e tipo delle variabili, delle funzioni e (per il C++) delle classi esportate. L'Object Pascal duplica queste informazioni addirittura nello stesso file. Se usate chiamate di procedura remota o CORBA [URL 29], duplicherete le informazioni di interfaccia fra la specifica dell'interfaccia e il codice che la implementa.

Non esiste una tecnica facile per aggirare i requisiti di un linguaggio. Alcuni ambienti di sviluppo nascondono la necessità di file di intestazione generandoli automaticamente, e l'Object Pascal consente di abbreviare le dichiarazioni di funzione ripetute, ma in generale si è vincolati a quel che è dato. Almeno, per molti problemi dipendenti dal

linguaggio, un file di intestazione che non concorda con l'implementazione genererà qualche forma di errore di compilazione o di *linkage*. Potete ancora sbagliare, ma se non altro vi viene detto abbastanza presto.

Pensate anche ai commenti nei file di intestazione e di implementazione. Non ha assolutamente senso duplicare un commento a una funzione o a una intestazione di classe fra i due file. Usate i file di intestazione per documentare problemi di interfaccia, e i file di implementazione per documentare i dettagli interni che gli utenti del vostro codice non hanno bisogno di conoscere.

Duplicazione inavvertita

A volte, la duplicazione nasce come risultato di errori di progettazione.

Vediamo un esempio tratto dal settore della distribuzione. La vostra analisi, poniamo, vi dice che, fra i tanti attributi, un camion ha un tipo, un numero di targa e un autista. Analogamente, un percorso di consegna è una combinazione di un percorso, un camion e un autista. Codifichiamo un po' di classi sulla base di queste informazioni.

Che cosa succede quando Sally telefona che è malato e dobbiamo cambiare gli autisti? Sia `Truck` che `DeliveryRoute` contengono un autista. Quale cambiamo? Chiaramente questa duplicazione non va bene. Normalizziamola in base al sottostante modello di business: un camion deve proprio avere un autista come parte del suo insieme sottostante di attributi? Deve avercelo un percorso? O magari c'è bisogno di un terzo oggetto che metta in relazione un autista, un camion e un percorso. Quale che sia la soluzione finale, evitate questo tipo di dati non normalizzati.

Vi è un caso meno ovvio di dati non normalizzati, che si presenta quando più elementi di dati sono fra loro dipendenti. Prendiamo una classe che rappresenta una linea:

```
class Line {
 public:
  Point  start;
  Point  end;
  double length;
};
```

A prima vista, questa classe sembra ragionevole. Una linea chiaramente ha un estremo iniziale e un estremo finale e avrà sempre una

lunghezza (al limite, zero). Ma abbiamo dei duplicati. La lunghezza è definita dai due estremi: cambiate uno dei due punti e la lunghezza cambia. Meglio trasformare la lunghezza in un campo calcolato:

```
class Line {
 public:
  Point start;
  Point end;
  double length() { return start.distanceTo(end); }
};
```

Più avanti, nel processo di sviluppo, potreste scegliere di violare il principio DRY per ragioni di prestazioni. Spesso succede se si ha bisogno di trasferire dei dati in cache per evitare di ripetere operazioni costose. Il trucco sta nel localizzare l'impatto. La violazione non è esposta al mondo esterno: solo i metodi all'interno della classe devono preoccuparsi di mantenere le cose corrette.

```
class Line {
 private:
  bool changed;
  double length;
  Point start;
  Point end;
 public:
  void setStart(Point p) { start = p; changed = true; }
  void setEnd(Point p) { end = p; changed = true; }
  Point getStart(void) { return start; }
  Point getEnd(void) { return end; }
  double getLength() {
    if (changed) {
      length = start.distanceTo(end);
      changed = false;
    }
    return length;
  }
};
```

Questo esempio illustra anche un problema importante per linguaggi a oggetti come Java e C++. Laddove è possibile, usate funzioni di accesso per leggere e scrivere gli attributi degli oggetti. Sarà più facile aggiungere funzionalità, come il *caching*, in futuro.

NOTA

L'uso di funzioni di accesso si accorda con il *principio di accesso uniforme* di Meyer [Mey97b], che dice: "Tutti i servizi offerti da un modulo devono essere disponibili attraverso una notazione uniforme, che non lascia trasparire se sono implementati attraverso la memoria o attraverso il calcolo".

Duplicazione per impazienza

Ogni progetto è soggetto a sollecitazioni temporali - forze che possono spingere anche i migliori a prendere delle scorciatoie. Vi serve una routine simile a una che avete già scritto? Sarete tentati di copiare l'originale e di fare qualche cambiamento. Vi serve un valore per rappresentare il numero massimo di punti? Se cambio il file di intestazione, tutto il progetto verrà ricostruito. Magari dovrei solo usare un numerino qui; e qui; e qui. Serve una classe come quella là nel runtime di Java? Il sorgente è disponibile, perciò perché non limitarsi a copiarla e apportare le modifiche che servono (nonostante quanto preveda la licenza)?

Se sentite questa tentazione, ricordate l'aforisma: "le scorciatoie creano lunghi ritardi". Magari risparmierete qualche secondo adesso, ma rischiate di perdere ore più tardi. Pensate ai problemi legati all'anno 2000: molti sono stati conseguenza della pigrizia di sviluppatori che non hanno parametrato la dimensione dei campi data o non hanno implementato librerie centralizzate di servizi per le date.

La duplicazione per impazienza è facile da identificare e gestire, ma ci vuole disciplina e la disponibilità a perdere un po' di tempo prima per risparmiarsi guai dopo.

NOTA

Per chi nel 2000 era troppo giovane per ricordarselo, con il nome di "Millennium bug", o "baco del millennio" (o anche "Y2K bug") si è indicato, al volgere del secolo, l'insieme dei problemi che avrebbero potuto venire a galla a causa di una poco lungimirante impostazione di molti sistemi tradizionali, in cui i programmatori avevano riservato, per la rappresentazione delle date, due sole cifre. Con l'inizio del nuovo millennio, ovviamente, sarebbe stato impossibile distinguere il 2000 dal 1900, e così via. In sistemi creati anche molti anni prima (spesso poco "ortogonali"), si temeva potessero verificarsi ripercussioni anche gravi, e il timore del baco del millennio, verso la fine degli anni Novanta ha prodotto un gran fervore di attività per la correzione dei sistemi (ma è stato anche lo stimolo in molti casi per una loro radicale sostituzione).

Duplicazione fra sviluppatori

Il tipo di duplicazione probabilmente più difficile da identificare e gestire è quello che si verifica quando più sviluppatori lavorano a uno stesso progetto. È possibile che interi gruppi di funzionalità vengano

duplicati inavvertitamente, e che nessuno se ne accorga per anni, con conseguenti problemi di manutenzione. Abbiamo saputo di prima mano di uno degli stati degli USA, i cui sistemi informatici governativi sono stati analizzati per il problema dell'anno 2000: l'esame ha rilevato oltre 10.000 programmi, ciascuno dei quali conteneva una propria versione del controllo del numero di sicurezza sociale.

Ad alto livello, affrontate il problema con un progetto chiaro, un forte leader tecnico del progetto (vedi *Team pragmatici* nel Capitolo 8) e una divisione chiara delle responsabilità all'interno del progetto. A livello di modulo, il problema è più insidioso. Funzionalità o dati di uso comune che non ricadono chiaramente in un'area di responsabilità vengono implementati molte volte.

Ci sembra che il modo migliore di affrontare la cosa sia favorire comunicazioni attive e frequenti fra gli sviluppatori. Create dei forum per discutere problemi comuni. (In passato, abbiamo impostato dei newsgroup di Usenet per consentire agli sviluppatori di scambiarsi idee e fare domande. Questo è un modo non intrusivo di comunicare, anche fra più siti, conservando una traccia permanente di tutto quello che è stato detto.) Attribuite a qualcuno l'incarico di bibliotecario di progetto, il cui compito sia facilitare lo scambio di conoscenza. Nell'albero del sorgente trovate un punto centrale in cui possano essere depositati routine e script. Formulate il proposito di leggere il codice sorgente e la documentazione degli altri, o informalmente o durante le revisioni del codice. Non si tratta di spiare, ma di imparare dagli altri. Ricordate, poi, che l'accesso è reciproco: anche voi, non siate suscettibili perché gli altri esaminano il *vostro* codice.

SUGGERIMENTO 12
Fate in modo che sia facile riusarlo.

Quello che cercate di fare è favorire la creazione di un ambiente in cui sia più facile trovare e riusare materiali esistenti che riscriverli in prima persona. *Se non è facile, nessuno lo farà*. E se non c'è riuso, si rischia di duplicare conoscenza.

Vedi anche

- *Ortogonalità*, in questo Capitolo
- *Manipolazione del testo*, Capitolo 3
- *Generatori di codice*, Capitolo 3
- *Refactoring*, Capitolo 6

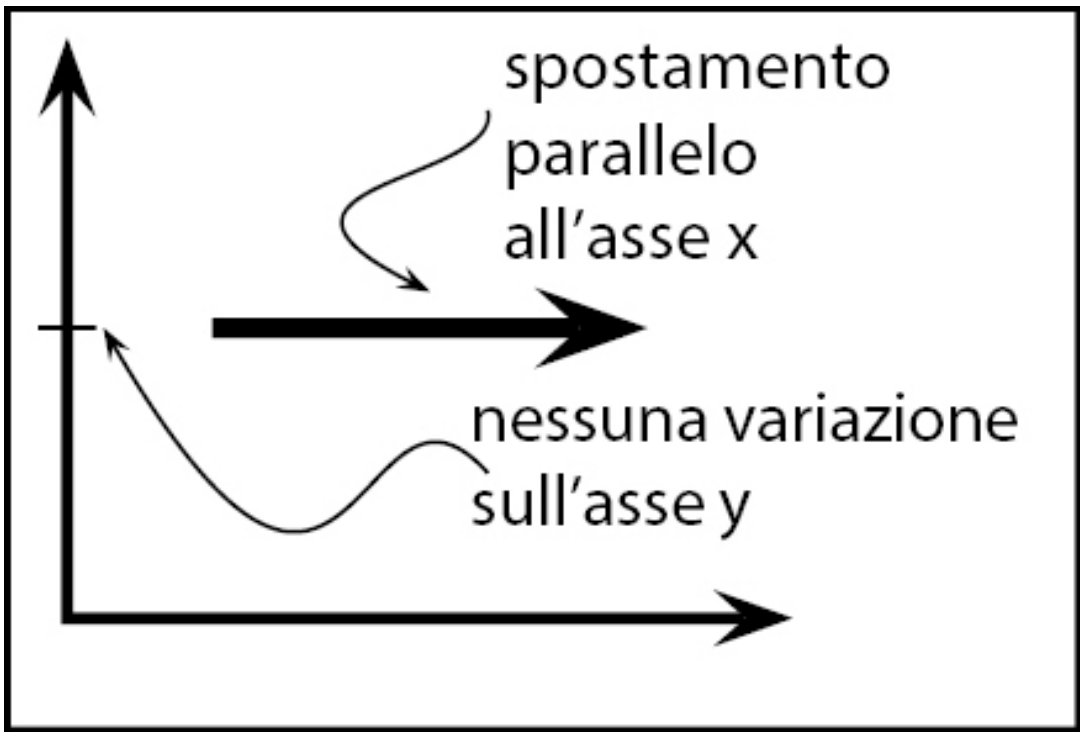
- *Team pragmatici*, Capitolo 8
- *Automazione onnipresente*, Capitolo 8
- *È tutta scrittura*, Capitolo 8

Ortogonalità

Quello di ortogonalità è un concetto fondamentale, se volete produrre sistemi facili da progettare, costruire, collaudare ed estendere. Raramente però viene insegnato direttamente; spesso è una caratteristica implicita di altri metodi e di altre tecniche. Ma è un errore: se si impara ad applicare direttamente il principio di ortogonalità, si nota un miglioramento immediato nella qualità dei sistemi che si producono.

Che cos'è l'ortogonalità?

"Ortogonalità" è un termine mutuato dalla geometria. Due rette sono ortogonali se incontrandosi formano angoli retti, come gli assi di un sistema di riferimento cartesiano. In termini vettoriali, le due rette sono *indipendenti*. Spostatevi parallelamente a una delle due rette, e la posizione proiettata sull'altra non cambia.



In informatica, il termine ha preso il significato di una forma di indipendenza o di disaccoppiamento. Due o più cose sono ortogonali se le variazioni nell'una non influenzano le altre. In un sistema ben progettato, il codice del database sarà ortogonale all'interfaccia utente: potete cambiare l'interfaccia senza intervenire sul database e sostituire un database con un altro senza modificare l'interfaccia.

Prima di esaminare i benefici dei sistemi ortogonali, vediamo prima un sistema che ortogonale non è.

Un sistema non ortogonale

State facendo un tour del Grand Canyon in elicottero quando il pilota, che ha commesso l'ovvio errore di mangiare pesce a pranzo, all'improvviso grugnisce e sviene. Per fortuna, vi state librando a una trentina di metri da terra. Riflettete: la leva del collettivo controlla la portanza generale, perciò abbassandola leggermente dovreste iniziare una discesa dolce verso terra. Quando ci provate, però, scoprite che la vita non è così semplice. Il muso dell'elicottero si inclina verso il basso e cominciate a scendere a spirale verso sinistra. All'improvviso scoprite che state volando su un sistema in cui ogni input di controllo ha effetti

secondari. Abbassate la leva di sinistra e dovete aggiungere un movimento di compensazione verso il retro alla leva di destra e dovete schiacciare il pedale destro. Ma poi ciascuno di questi cambiamenti influenza di nuovo tutti gli altri controlli. All'improvviso state cercando di dominare un sistema di incredibile complessità, dove ogni cambiamento ha conseguenze su tutti gli altri input. Il carico di lavoro è fenomenale: mani e piedi si muovono costantemente, nel tentativo di trovare un equilibrio fra tutte le forze che interagiscono. I controlli dell'elicottero sono decisamente non ortogonali.

Benefici dell'ortogonalità

Come illustra l'esempio dell'elicottero, i sistemi non ortogonali sono intrinsecamente più complessi da modificare e controllare. Quando i componenti di un sistema sono fortemente interdipendenti, non esistono soluzioni locali.

SUGGERIMENTO 13
Eliminate gli effetti fra cose che non sono in rapporto fra loro.

Vogliamo progettare componenti che siano autocontenuti: indipendenti e con un singolo scopo ben definito (quella che Yourdon e Constantine chiamano *coesione* [YC86]). Quando i componenti sono isolati uno dall'altro, sappiamo di poterne modificare uno senza doverci preoccupare del resto. Se non si cambiano le interfacce esterne di quel componente, si può stare tranquilli che non si causeranno problemi che possano propagarsi a tutto il sistema.

Se si scrivono sistemi ortogonali si ottengono due benefici importanti: si aumenta la produttività e si riduce il rischio.

Aumento di produttività

- I cambiamenti sono localizzati, perciò tempo di sviluppo e tempo di collaudo si riducono. È più facile scrivere componenti relativamente piccoli, chiusi in se stessi, che non un unico grande blocco di codice. Componenti semplici si possono progettare, codificare, collaudare e

poi dimenticare: non c'è bisogno di continuare a cambiare il codice esistente quando se ne aggiunge di nuovo.

- Un approccio ortogonale inoltre promuove il riuso. Se i componenti hanno responsabilità specifiche, ben definite, possono essere combinati con nuovi componenti in modi che chi li aveva implementati non aveva assolutamente in mente. Quanto più lasco è l'accoppiamento fra i sistemi, tanto più facile risulta riconfigurarli e reingegnerizzarli.
- Vi è un guadagno molto sottile di produttività quando si combinano componenti ortogonali. Supponiamo che un componente faccia *M* cose diverse e un altro ne faccia *N*. Se sono ortogonali e li si combina, il risultato fa $M \times N$ cose. Se i due componenti però non sono ortogonali, ci saranno delle sovrapposizioni, e il risultato farà di meno. Si ottengono più funzionalità per unità di sforzo combinando componenti ortogonali.

Riduzione del rischio

Un approccio ortogonale riduce i rischi connaturati a qualsiasi sviluppo.

- Le sezioni "malate" di codice sono isolate. Se un modulo ha qualcosa che non va, è meno probabile che diffonda i sintomi in giro per il resto del sistema. È anche più facile sezionarlo e trapiantarvi qualcosa di nuovo e sano.
- Il sistema risultante è meno fragile. Apportate piccoli cambiamenti e aggiustamenti a un'area particolare, e qualsiasi problema sorga resterà confinato in quell'area.
- Un sistema ortogonale probabilmente verrà collaudato meglio, perché sarà più facile progettare ed eseguire test sui suoi componenti.
- Non sarete legati troppo strettamente a un particolare produttore, a un certo prodotto o a una determinata piattaforma, perché le interfacce verso questi componenti di terzi saranno isolate in parti più piccole dello sviluppo complessivo.

Vediamo ora qualcuno dei modi in cui potete applicare il principio di ortogonalità al vostro lavoro.

Team di progetto

Avete mai notato come certi team di progetto siano efficienti, con tutti che sanno che cosa fare e danno il loro contributo a pieno, mentre i membri di altri team sono sempre lì a litigare e sembra che non riescano a fare a meno di intralciarsi a vicenda?

Spesso è un problema di ortogonalità. Se i team sono organizzati con molte sovrapposizioni, le responsabilità sono confuse. Ogni cambiamento richiede una riunione di tutto il team, perché chiunque *potrebbe* esserne influenzato.

Come si organizzano i team in gruppi con responsabilità ben definite e sovrapposizioni ridotte al minimo? Non esiste una risposta semplice. Dipende in parte dal progetto e dall'analisi delle aree di potenziale cambiamento. Dipende anche dalle persone che avete a disposizione. Noi preferiamo iniziare separando l'infrastruttura dall'applicazione. Ciascuno dei principali componenti infrastrutturali (database, interfaccia di comunicazione, middleware e così via) ha il suo sotto-team. Lo stesso vale per ogni divisione delle funzionalità dell'applicazione. Poi consideriamo le persone che abbiamo (o che contiamo di avere) e sistemiamo i raggruppamenti di conseguenza.

Potete avere una misura informale dell'ortogonalità della struttura di un team di progetto. Basta che consideriate quante persone *devono* essere coinvolte nell'analisi di ogni cambiamento richiesto. Quanto più grande il loro numero, tanto meno ortogonale è il gruppo. Chiaramente, un team ortogonale è più efficiente. (Detto questo, però, incoraggiamo i sotto-team a comunicare costantemente gli uni con gli altri.)

Progetto

La maggior parte degli sviluppatori conosce bene la necessità di progettare sistemi ortogonali, anche se magari usa termini diversi, come *modulare*, *basato su componenti* e *stratificato*. I sistemi devono essere composti da un insieme di moduli che cooperano, ciascuno dei quali

implementa funzionalità indipendenti dagli altri. A volte questi componenti sono organizzati a strati, ciascuno dei quali corrisponde a un livello di astrazione. Questo approccio stratificato è un modo eccellente per progettare sistemi ortogonali: poiché ciascuno strato usa solo le astrazioni messe a disposizione dagli strati inferiori, si ha una grande flessibilità nel modificare le implementazioni sottostanti senza influire sul codice. La stratificazione inoltre riduce il rischio di dipendenze a spasso fra i moduli. Spesso vedrete la stratificazione rappresentata da diagrammi come quello della Figura 2.1.

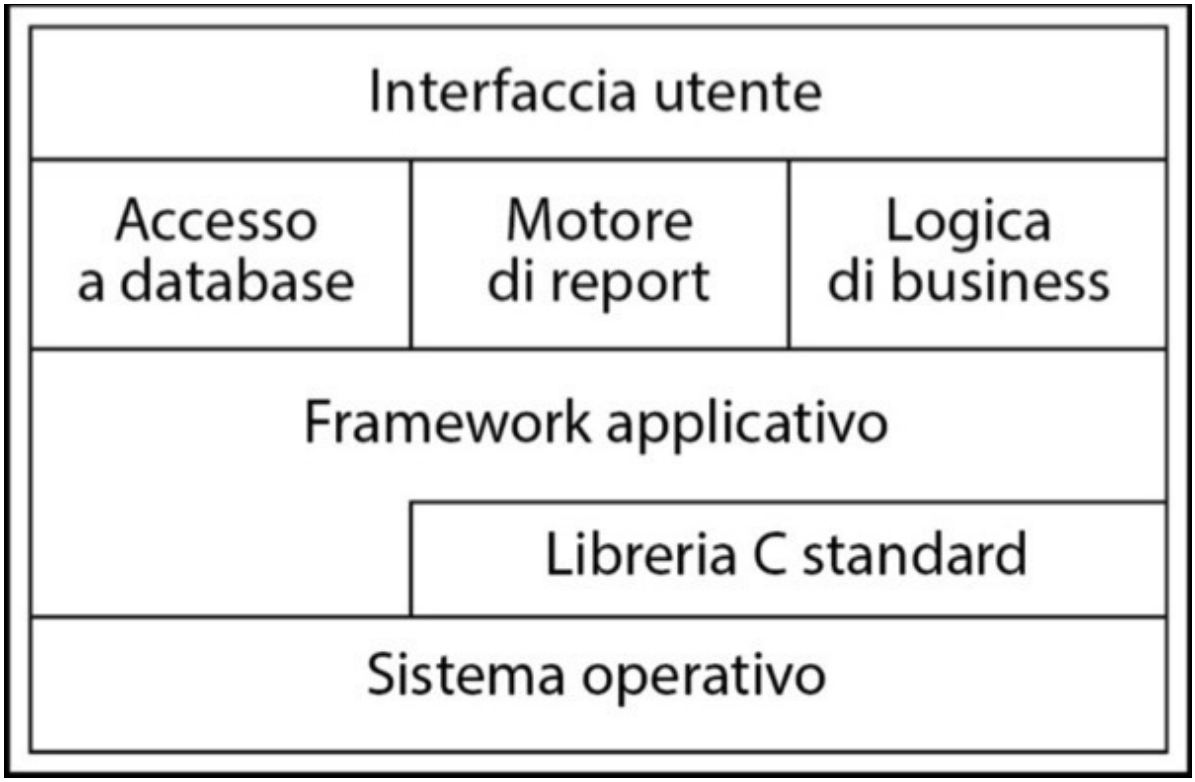


Figura 2.1 Un tipico diagramma a strati (o livelli).

C'è un test molto semplice per stabilire l'ortogonalità di un progetto. Una volta rappresentati tutti i componenti, chiedetevi: *se dovessi modificare drasticamente i requisiti dietro una particolare funzione, quanti moduli ne verrebbero colpiti?* In un sistema ortogonale, la risposta deve essere "uno". Lo spostamento di un pulsante in una schermata di GUI non deve comportare un cambiamento nello schema del database. L'aggiunta di un sistema di aiuto sensibile al contesto non deve costringere a modificare il sottosistema della fatturazione.

NOTA
In realtà, dire che deve essere influenzato un solo modulo è un po' ingenuo. A meno che non si sia molto fortunati, la maggior parte dei cambiamenti di requisiti nel mondo reale andrà a influire su molte funzioni nel sistema. Se si analizza il cambiamento in termini di funzioni, però, ogni cambiamento funzionale deve ancora idealmente influire su un modulo solamente.

Prendiamo un sistema complesso di monitoraggio e controllo di un impianto di riscaldamento. Il requisito di partenza chiedeva un'interfaccia utente grafica, ma poi i requisiti sono stati modificati con l'aggiunta di un sistema di riposta vocale con un controllo via telefono dell'impianto. In un sistema progettato ortogonalmente, dovreste modificare solo i moduli associati all'interfaccia utente: la logica di fondo di controllo dell'impianto rimarrebbe immutata. In effetti, se strutturate attentamente il sistema, dovreste poter supportare entrambe le interfacce con la stessa base di codice. *È solo una vista*, nel Capitolo 5, parla di scrittura di codice disaccoppiato seguendo il paradigma MVC (Model-View-Controller), che funziona bene in situazioni di questo tipo.

Chiedetevi anche se il vostro progetto è disaccoppiato dai cambiamenti nel mondo reale. Usate un numero di telefono come identificatore di un cliente? E se la compagnia telefonica riassegna i suoi codici? *Non basatevi sulle caratteristiche di cose che non potete controllare*.

Toolkit e librerie

Fate attenzione a mantenere l'ortogonalità del vostro sistema quando introducete toolkit e librerie di terze parti. Scegliete le vostre tecnologie con cura.

Una volta abbiamo lavorato a un progetto che richiedeva l'esecuzione di un certo corpus di codice Java sia localmente sul server sia in remoto su una macchia client. Le alternative per distribuire le classi in quel modo erano RMI e CORBA. Se una classe veniva resa accessibile in remoto con RMI, ogni chiamata a un metodo remoto in quella classe poteva lanciare un'eccezione, il che significa che un'implementazione ingenua ci avrebbe costretti a gestire l'eccezione ogni volta che venivano usate le nostre classi remote. L'uso di RMI in quel caso chiaramente non era ortogonale: codice che chiamava le classi remote non avrebbe dovuto essere "consapevole" della loro posizione. L'alternativa, cioè usare

CORBA, non imponeva quella limitazione: avremmo potuto scrivere codice che non doveva sapere dove si trovassero le nostre classi.

Se introducete un toolkit (o anche una libreria da altri membri del vostro team) chiedetevi se impone al vostro codice cambiamenti che non dovrebbero aver ragione di esistere. Se uno schema di persistenza di un oggetto è trasparente, allora è ortogonale. Se vi impone di creare oggetti o di accedervi in un modo speciale, non lo è. Mantenere questi dettagli isolati dal vostro codice ha l'ulteriore vantaggio di rendere più facile il passaggio ad altri produttori in futuro.

Il sistema *Enterprise Java Beans* (EJB) è un esempio interessante di ortogonalità. Nella maggior parte dei sistemi orientati alla transazione, il codice applicativo deve definire l'inizio e la fine di ogni transazione. Con EJB, questa informazione è espressa in forma dichiarativa come metadato, all'esterno di ogni codice. Lo stesso codice applicativo può girare in ambienti di transazione EJB diversi senza alcun cambiamento. Questo è probabile sia un modello per molti ambienti futuri.

Un altro risvolto interessante dell'ortogonalità è la programmazione orientata all'aspetto (AOP, *Aspect-Oriented Programming*), un progetto di ricerca dello Xerox Parc ([KLM97] e [URL49]). AOP permette di esprimere in un punto preciso un comportamento che altrimenti sarebbe distribuito in tutto il codice sorgente. Per esempio, i messaggi di log vengono generati normalmente spargendo in tutto il sorgente chiamate esplicite a qualche funzione di log; con AOP, si implementa il logging ortogonalmente alle cose di cui si vuole tenere traccia. Utilizzando la versione Java di AOP, si potrebbe scrivere un messaggio di log quando si inserisce qualsiasi metodo di classe `Fred` codificando l'*aspetto*:

```
aspect Trace {
  advise * Fred.*(..) {
    static before {
      Log.write("-> Entering " + thisJoinPoint.methodName);
    }
  }
}
```

Se si *intreccia* questo aspetto nel codice, verranno generati messaggi di traccia. Altrimenti, non vedrete messaggi. In un modo o nell'altro, il sorgente originale è immutato.

Codifica

Ogni volta che si scrive codice si corre il rischio di ridurre l'ortogonalità dell'applicazione. A meno che non si tenga sott'occhio costantemente non solo quello che si fa ma anche il contesto più generale dell'applicazione, si può finire per duplicare senza volerlo le funzionalità in qualche altro modulo, o per esprimere due volte la stessa conoscenza.

Esistono varie tecniche per mantenere l'ortogonalità.

- *Mantenete disaccoppiato il codice*. Scrivete codice "timido", ovvero moduli che non rivelano nulla di non necessario ad altri moduli e che non si basano sulle implementazioni di altri moduli. Provate la legge di Demetra [LH89], che analizziamo in *Disaccoppiamento e legge di Demetra*, nel Capitolo 5. Se dovete cambiare lo stato di un oggetto, fatelo fare all'oggetto al posto vostro. In questo modo il codice resta isolato dall'implementazione di altro codice e migliorano le probabilità di rimanere ortogonali.
- *Evitate dati globali*. Ogni volta che il vostro codice fa riferimento a dati globali, si lega agli altri componenti che condividono quei dati. Anche dati globali che devono essere solo letti possono creare guai (per esempio, se all'improvviso dovete cambiare il vostro codice perché sia multithreaded). In generale, il codice è più facile da comprendere e da manutenere se si passa ogni contesto necessario nei moduli. Nelle applicazioni a oggetti, spesso il contesto viene passato sotto forma di parametri ai costruttori degli oggetti. In altro codice, potete creare strutture che contengono il contesto e passare riferimenti a quelle strutture. Lo schema Singleton in *Design Patterns* [GHJV95] è un modo per essere sicuri che esista una sola istanza di un oggetto di una particolare classe. Molti usano questi oggetti "singoletto" come una sorta di variabile globale (in particolare in linguaggi come Java, che altrimenti non ammettono il concetto di dati globali). Fate attenzione comunque ai singoletti: possono anche portare a legami non necessari.
- *Evitate funzioni simili.* Spesso vi imbatterete in un insieme di funzioni di aspetto simile: magari condividono del codice comune all'inizio e alla fine, ma hanno poi al centro un algoritmo diverso. Codice duplicato è un sintomo di problemi strutturali. Guardate lo schema Strategy in *Design Patterns* per un'implementazione migliore.

Abituatevi a essere costantemente critici rispetto al vostro codice. Cercate ogni opportunità per riorganizzarlo e migliorarne struttura e ortogonalità. Questo processo è chiamato *refactoring*, ed è tanto importante che vi abbiamo dedicato una sezione specifica (vedi *Refactoring*, nel Capitolo 6).

Test

Un sistema progettato e implementato in modo ortogonale è più facile da sottoporre a test. Poiché le interazioni fra i componenti del sistema sono formalizzate e limitate, una parte maggiore del test di sistema può essere svolta a livello dei singoli moduli. È una buona notizia, perché il test a livello dei moduli (*unit testing*) è molto più facile da specificare ed eseguire che non un test di integrazione. In effetti, consigliamo che ogni modulo abbia il proprio unit test incorporato nel proprio codice, e che questi test vengano eseguiti automaticamente nell'ambito del regolare processo di build (vedi *Codice facile da sottoporre a test*, nel Capitolo 6).

La costruzione di unit test è a sua volta un interessante test di ortogonalità. Che cosa ci vuole per costruire e collegare uno unit test? Dovete chiamare in causa una grande percentuale del resto del sistema solo per compilare o collegare un test? Se è così, avete trovato un modulo che non è ben disaccoppiato dal resto del sistema.

Il *bug fixing*, la risoluzione degli errori, è un altro buon momento per valutare l'ortogonalità del sistema nel suo complesso. Se incontrate un problema, valutate quanto sia localizzata la soluzione. Modificate solo un modulo, o i cambiamenti sono sparsi per tutto il sistema? Quando apportate un cambiamento, risolve tutto o spuntano misteriosamente altri problemi? È una buona occasione per l'automazione. Se usate un sistema di controllo del sorgente (e lo farete, dopo aver letto *Controllo del codice sorgente*, nel Capitolo 3), etichettate i punti in cui avete inserito una soluzione a un bug quando verificate il codice di nuovo dopo il test. Potete poi avere report mensili di analisi delle tendenze nel numero di file sorgente influenzati da ogni risoluzione di un bug.

Documentazione

Forse vi sorprenderà, ma l'ortogonalità vale anche per la documentazione. Gli assi sono contenuto e presentazione. Con una documentazione davvero ortogonale, dovreste poter modificare drasticamente l'aspetto senza dover modificare il contenuto. I word processor moderni mettono a disposizione fogli stile e macro, a questo scopo (vedi *È tutta scrittura*, nel Capitolo 8).

Vivere con l'ortogonalità

L'ortogonalità è in stretto rapporto con il principio DRY introdotto nel Capitolo 1. Con DRY, si punta a ridurre al minimo la duplicazione in un sistema, mentre con l'ortogonalità si riduce l'interdipendenza fra i componenti del sistema. Può essere una parola un po' ingombrante, ma se usate il principio di ortogonalità, insieme con il principio DRY, vedrete che svilupperete sistemi più flessibili, più comprensibili e più facili da sottoporre a debug, test e manutenzione.

Se vi capita di entrare in un progetto in cui le persone lottano disperatamente per apportare cambiamenti, e dove ogni cambiamento sembra far emergere altre quattro cose che vanno storte, ricordate l'incubo dell'elicottero. Il progetto probabilmente non è stato disegnato e codificato ortogonalmente. È tempo di refactoring.

E, se siete piloti di elicottero, non mangiate pesce.

Vedi anche

- *I mali della duplicazione*, in questo Capitolo
- *Controllo del codice sorgente*, Capitolo 3
- *Progettare per contratto*, Capitolo 4
- *Disaccoppiamento e legge di Demetra*, Capitolo 5
- *Metaprogrammazione*, Capitolo 5
- *È solo una vista*, Capitolo 5
- *Refactoring*, Capitolo 6
- *Codice facile da sottoporre a test*, Capitolo 6
- *Maghi cattivi*, Capitolo 6
- *Team pragmatici*, Capitolo 8
- *È tutta scrittura*, Capitolo 8

Sfide

- Considerate la differenza fra i grandi strumenti orientati alle GUI che si trovano in genere nei sistemi Windows e le utility da riga di comando, piccole ma combinabili, usate dal prompt della shell. Quale dei due insiemi è più ortogonale, e perché? Quale è più facile da usare proprio allo scopo per cui è stato creato? Quale insieme è più facile da combinare con altri strumenti per affrontare nuove sfide?
- Il C++ ammette l'eredità multipla e Java permette che una classe implementi più interfacce. Quali conseguenze ha l'uso di queste possibilità per l'ortogonalità? L'uso dell'eredità multipla e quello di più interfacce hanno un impatto diverso? Esiste una differenza fra usare la delega e usare l'eredità?

Esercizi

1. State scrivendo una classe che si chiama Split, che suddivide le righe di input in campi. Quale delle seguenti due firme di classe Java ha una progettazione più ortogonale?

```
class Split1 {
  public Split1(InputStreamReader rdr) { ...
  public void readNextLine() throws IOException { ...
  public int numFields() { ...
  public String getField(int fieldNo) { ...
}


class Split2 {
  public Split2(String line) { ... public int numFields() { ...
  public String getField(int fieldNo) { ...
}
```

2. Che cosa porta a una progettazione più ortogonale: le finestre di dialogo senza modalità o quelle modali?
3. Che cosa dire dei linguaggi procedurali rispetto alla tecnologia a oggetti? Che cosa dà un sistema più ortogonale?

Reversibilità

Nulla è più pericoloso di un'idea, se è l'unica che avete.
- Emil-Auguste Chartier, *Propos sur la religion*, 1938

Gli ingegneri preferiscono le soluzioni semplici e uniche ai problemi. I test di matematica che permettono di affermare con grande fiducia che $x = 2$ ci mettono molto più a nostro agio dei saggi, assai più sfumati, sulle molte cause della Rivoluzione francese. Il management è tendenzialmente d'accordo con gli ingegneri: risposte singole e facili si inseriscono bene nei fogli di calcolo e nei piani progettuali.

Se solo il mondo reale fosse più collaborativo! Purtroppo, se oggi *x* è 2, magari domani deve essere 5, e 3 la prossima settimana. Nulla è per sempre e, se vi basate molto su qualche fatto, potete quasi star certi che *cambierà*.

C'è sempre più di un modo per implementare qualcosa, e di solito ci sono più *vendor* disponibili a fornire un prodotto di terze parti. Se affrontate un progetto portando con voi l'idea miope che ci sia solo un modo di portarlo a termine, andrete incontro a una sorpresa spiacevole. Molti team di progetto sono costretti con la forza ad aprire gli occhi, quando il futuro si realizza:

> "Ma avevi detto che avremmo usato il database XYZ! E adesso che abbiamo codificato il progetto all'85 per cento dovremmo cambiare?", protestava il programmatore. "Mi dispiace, ma la nostra azienda ha deciso di fissare invece come standard il database PDQ, per tutti i progetti. Non dipende da me. Dovrete ricodificare. Fino a nuovo avviso, dovrete lavorare anche tutti i fine settimana."

I cambiamenti non sono sempre così draconiani, e nemmeno altrettanto immediati, ma con il trascorrere del tempo e il procedere del progetto è possibile che vi troviate bloccati in una posizione insostenibile. A ogni decisione critica, il team si concentra su un bersaglio più piccolo, una versione più ristretta della realtà con meno opzioni.

Quando sono state prese molte decisioni critiche, il bersaglio diventa così piccolo che, se si sposta, se il vento cambia direzione o una farfalla a Tokyo sbatte le ali, non lo si centra più e lo si manca di molto.

NOTA

Prendete un sistema non lineare o caotico e modificate anche di poco uno dei suoi input: è possibile che otteniate un risultato enorme e imprevedibile. Il battito delle ali di una farfalla a Tokyo può essere l'inizio di una catena di eventi che finisce per generare un tornado in Texas. Vi ricorda un po' qualche progetto che conoscete?

Il problema è che dalle decisioni critiche non si torna facilmente indietro. Una volta deciso di usare un certo database, o un certo modello (client-server anziché standalone, per esempio), vi siete infilati in una strada da cui è difficile tornare indietro, se non con grandi costi.

Reversibilità

Molti dei temi di questo libro sono orientati alla produzione di software flessibile e adattabile. Se ci atteniamo ad alcune idee di fondo, il principio DRY (Capitolo 1), il disaccoppiamento (Capitolo 5) e l'uso dei

metadati (Capitolo 5), non dobbiamo prendere tante decisioni critiche irreversibili. È una buona cosa, perché non sempre si prendono le decisioni migliori al primo tentativo. Ci affidiamo a una data tecnologia solo per scoprire che non possiamo assumere abbastanza persone con le competenze necessarie. Ci fissiamo su un certo produttore e poco dopo viene assorbito dal suo concorrente. Requisiti, utenti e hardware cambiano a una velocità maggiore di quella a cui si sviluppa il software.

Supponiamo che, agli inizi del progetto, decidiate di usare un database relazionale del produttore A. Molto più avanti, durante i test di prestazioni, scoprite che quel database è troppo lento, mentre il database a oggetti del produttore B è più veloce. Nella maggior parte dei progetti convenzionali, sareste nei guai. Per lo più, le chiamate a prodotti di terze parti sono intrecciate in tutto il codice, ma se avete *realmente* astratto l'idea di un database, al punto in cui fornisce semplicemente la persistenza come servizio, allora avrete la flessibilità di cambiare cavallo in corsa.

Analogamente, supponiamo che il progetto inizi con un modello client-server, ma poi, più avanti, il marketing decida che i server sono troppo costosi per certi clienti, e che quindi sia meglio una versione standalone. Quanto lavoro in più vi comporterà? Poiché si tratta solo di un problema di *deployment*, *non dovrebbe richiedervi più di qualche giorno*. Se vi richiede di più, non avete pensato alla reversibilità. L'altra direzione è anche più interessante. Che cosa succederebbe se il prodotto standalone che state realizzando dovesse essere messo in opera in una versione client-server o *n*-tier? *Neanche questo dovrebbe essere troppo difficile*.

L'errore sta nel dare per scontato che ogni decisione sia incisa nella pietra - e nel non essere pronti per le eventualità che possono sorgere. Invece di incidere le decisioni nel marmo, pensatele più come scritte sulla sabbia in riva al mare: può arrivare una bella ondata in qualsiasi momento e spazzarle via.

SUGGERIMENTO 14
Non esistono decisioni irrevocabili.

Architettura flessibile

Molti cercano di mantenere flessibile il loro *codice*, ma bisogno pensare a mantenere la flessibilità anche sul fronte dell'architettura, del *deployment* e dell'integrazione dei vendor.

Tecnologie come CORBA possono aiutare a isolare parti di un progetto da cambiamenti del linguaggio di sviluppo o della piattaforma. Le prestazioni di Java su quella piattaforma non sono all'altezza delle aspettative? Ricodificate il client in C++ e nient'altro deve cambiare. Il motore delle regole in C++ non è abbastanza flessibile? Passate a una versione in Smalltalk. Con un'architettura CORBA dovrete accusare il colpo solo per il componente che dovete sostituire: gli altri componenti non dovrebbero essere influenzati.

State sviluppando per Unix? Quale? Avete affrontato tutte le questioni di portabilità? State sviluppando per una particolare versione di Windows? Quale? Quanto sarebbe difficile supportare altre versioni? Se prendete decisioni morbide, senza rigidità, non sarà affatto difficile, Se avete scarso incapsulamento, elevato accoppiamento e logica codificata in modo rigido o parametri nel codice, forse è addirittura impossibile.

Non siete sicuri di come il marketing voglia posizionare il sistema? Pensateci prima e potrete supportare una versione standalone, client-server o *n*-tier semplicemente cambiando un file di configurazione. Abbiamo scritto programmi che si comportano proprio così.

Normalmente, potete semplicemente nascondere un prodotto di terzi dietro un'interfaccia astratta ben definita. In effetti, a noi è sempre riuscito, in tutti i progetti a cui abbiamo lavorato. Supponiamo però che non possiate isolarlo bene. E se aveste disperso certi enunciati in tutto il codice in piena libertà? Mettete quei requisiti nei metadati e usate qualche meccanismo automatico, come Aspects (Capitolo 1) o Perl, per inserire gli enunciati necessari nel codice stesso. Quale che sia il meccanismo che usate, *rendetelo reversibile*. Se una certa cosa viene aggiunta automaticamente, può anche essere tolta automaticamente.

Vedi anche

- *Disaccoppiamento e legge di Demetra*, Capitolo 5
- *Metaprogrammazione*, Capitolo 5
- *È solo una vista*, Capitolo 5

Sfide

- È tempo di un po' di meccanica quantistica con il gatto di Schrödinger. Supponiamo che abbiate messo un gatto in una scatola sigillata, in compagnia di una particella radioattiva. La particella ha esattamente il 50% di probabilità di dividersi in due; se accade, il gatto morirà, altrimenti se la caverà. Allora, il gatto è morto o vivo? Secondo Schrödinger, la risposta corretta è *entrambe le cose*. Ogni volta che avviene una reazione subnucleare che ha due esiti possibili, l'universo viene clonato. In uno l'evento si è verificato, nell'altro no. Il gatto è vivo in un universo, morto in un altro. Solo quando aprite la scatola scoprirete in quale universo siete.
  Non meraviglia che codificare per il futuro sia difficile.
  Pensate però l'evoluzione del codice come una scatola piena di gatti di Schrödinger: ogni decisione produce una diversa versione del futuro. Quanti futuri possibili supporta il vostro codice? Quali sono più probabili? Quanto sarà difficile adeguarvisi, quando verrà il momento?
  Avete il coraggio di aprire la scatola?

Proiettili traccianti

Pronti, fuoco, puntate…

Ci sono due modi per sparare con una mitragliatrice al buio. (Ce ne sarebbero molti, in realtà, compreso chiudere gli occhi e sparare a caso, ma questa è un'analogia e ci concediamo qualche libertà.) Potete stabilire esattamente dove sia il bersaglio (distanza, elevazione e azimuth). Potete stabilire le condizioni ambientali (temperatura, umidità, pressione atmosferica, vento ecc.). Potete determinare le specifiche esatte delle cartucce e dei proiettili che usate, e le loro interazioni con l'arma specifica che avete a disposizione. Poi potete usare delle tavole o un computer d'arma per calcolare direzione ed elevazione della canna. Se tutto funziona esattamente come specificato, se le vostre tavole sono corrette e se l'ambiente non cambia, i proiettili dovrebbero finire vicino al bersaglio.

Altrimenti potete usare proiettili traccianti. Questi proiettili vengono caricati a intervalli nella cartuccera fra le munizioni normali: quando vengono sparati, il fosforo che contengono si incendia e lasciano una traccia pirotecnica dall'arma a ciò che colpiscono. Se i traccianti colpiscono il bersaglio, lo faranno anche i proiettili normali.

Non meraviglia che si preferiscano i traccianti alla fatica dei calcoli. Il loro feedback è immediato e, poiché operano nello stesso ambiente delle

munizioni reali, gli effetti esterni sono ridotti al minimo.

L'analogia è un po' violenta, ma si applica ai nuovi progetti, in particolare quando dovete costruire qualcosa che non è mai stato costruito in precedenza. Come i tiratori, dovete cercare di colpire un bersaglio al buio. Poiché i vostri utenti non hanno mai visto un sistema simile in precedenza, i loro requisiti possono essere vaghi. Per parte vostra, magari usate algoritmi, tecniche, linguaggi o librerie che non vi sono molto familiari, e dovrete affrontare un bel numero di incognite. Poiché il completamento di un progetto richiede tempo, potete stare sicuri che l'ambiente in cui lavorate sarà cambiato, prima che finiate.

La risposta classica è quella di specificare il sistema fino alla morte: produrre pacchi di carta che riportano ogni requisito, cercano di risolvere ogni incognita e di vincolare l'ambiente. Un grande calcolo all'inizio, poi si spara e si spera.

I programmatori pragmatici, però, tendenzialmente preferiscono usare proiettili traccianti.

Codice che brilla nel buio

I proiettili traccianti funzionano perché operano nello stesso ambiente e sotto gli stessi vincoli dei proiettili reali. Arrivano rapidamente al bersaglio, perciò il tiratore ha un feedback immediato. Da un punto di vista pratico, sono una soluzione relativamente poco costosa.

Per ottenere lo stesso effetto con il codice, bisogna cercare qualcosa che ci porti da un requisito a qualche aspetto del sistema finale rapidamente, in modo visibile e ripetibile.

SUGGERIMENTO 15
Usate proiettili traccianti per trovare il bersaglio.

Una volta abbiamo intrapreso un progetto complesso di marketing su database client-server. Parte dei requisiti era la possibilità di specificare ed eseguire interrogazioni temporali. I server erano vari database relazionali e specializzati. La GUI del client, scritta in Object Pascal, usava un insieme di librerie C per fornire un'interfaccia ai server. Le interrogazioni degli utenti erano conservate nel server in una notazione simile al Lisp, per poi essere convertite in SQL ottimizzato subito prima

dell'esecuzione. Le incognite erano molte e molti gli ambienti diversi, e nessuno aveva le idee molto chiare su come dovesse comportarsi la GUI.

È stata un'ottima occasione per usare codice tracciante. Abbiamo sviluppato il framework per il front-end, librerie per rappresentare le interrogazioni e una struttura per convertire una interrogazione memorizzata in una interrogazione specifica per i database. Poi abbiamo assemblato il tutto e controllato che funzionasse. Per quella build iniziale, tutto quello che potevamo fare era inserire un'interrogazione che elencasse tutte le righe di una tabella, ma dimostrava che l'interfaccia utente poteva parlare con le librerie, le librerie potevano serializzare e de-serializzare un'interrogazione e che il server poteva generare SQL dal risultato. Nei mesi successivi abbiamo gradualmente arricchito questa struttura di base, aggiungendo funzionalità e ampliando in parallelo ogni componente del codice tracciante. Quando l'interfaccia utente aggiungeva un nuovo tipo di interrogazione, la libreria cresceva e la generazione di SQL veniva resa più sofisticata.

Il codice tracciante non è "a perdere": lo si scrive per conservarlo. Contiene tutti gli elementi di controllo degli errori, strutturazione, documentazione e autocontrollo che possiede qualsiasi pezzo di codice di produzione: semplicemente non è del tutto funzionale. Tuttavia, una volta ottenuta una connessione end-to-end fra i componenti del sistema, si può verificare quanto si è vicini al bersaglio, per poi apportare degli aggiustamenti se è il caso. Una volta che si è trovato il bersaglio, l'aggiunta di funzionalità è facile.

Lo sviluppo con traccianti è coerente con l'idea che un progetto non è mai finito: ci saranno sempre cambiamenti da apportare e funzioni da aggiungere. È un approccio incrementale.

L'alternativa convenzionale è una sorta di metodo da ingegneria pesante: il codice viene diviso in moduli, che vengono codificati nel vuoto. I moduli sono combinati in sotto-assemblaggi, che poi vengono ulteriormente combinati, finché un giorno non si ha un'applicazione completa. Solo a quel punto l'applicazione come un tutto può essere presentata all'utente e collaudata.

Il metodo del codice tracciante ha molti vantaggi.

- *Gli utenti possono vedere presto qualcosa che funziona*. Se avete comunicato efficacemente quello che state facendo (vedi *Grandi*

speranze, nel Capitolo 8), i vostri utenti sapranno di guardare qualcosa di immaturo. Non saranno delusi dall'assenza di funzionalità; saranno entusiasti d vedere qualche progresso concreto verso il loro sistema. Inoltre contribuiranno al progredire del progetto, accettandolo sempre di più. Quegli stessi utenti saranno probabilmente le persone che vi diranno, a ogni iterazione, quanto siete vicini al bersaglio.

- *Gli sviluppatori costruiscono una struttura in cui lavorare*. Il pezzo di carta che spaventa di più è quello su cui non è scritto nulla. Se avete elaborato tutte le interazioni della vostra applicazione e le avete trasformate in codice, il vostro team non dovrà immaginarsi tutto dal nulla. Tutti saranno più produttivi e ne guadagnerà la coerenza.
- *Avete una piattaforma di integrazione*. Quando il sistema viene connesso completamente, avete un ambiente a cui potete aggiungere nuovi pezzi di codice non appena hanno superato lo unit test. Anziché tentare una integrazione in stile "big bang", integrerete ogni giorno (e spesso molte volte al giorno). L'impatto di ogni nuovo cambiamento è più evidente e le interazioni sono più limitate, perciò debug e test sono più veloci e più accurati.
- *Avete qualcosa da dimostrare*. Gli sponsor del progetto e l'alta dirigenza hanno la tendenza a voler vedere una demo nei momenti meno opportuni. Con il codice di traccia, avrete sempre qualcosa da mostrare.
- *Avrete una percezione migliore dell'andamento del lavoro*. In uno sviluppo con codice tracciante, gli sviluppatori affrontano i casi d'uso uno alla volta. Fatto uno, passano al successivo. È molto più facile misurare le prestazioni e dimostrare l'avanzamento del lavoro all'utente. Poiché ogni singolo sviluppo è di minore entità, si evitano quei blocchi monolitici di codice che si danno completi al 95 per cento una settimana dopo l'altra.

I proiettili traccianti non sempre colpiscono il bersaglio

I proiettili traccianti vi fanno vedere che cosa state colpendo: non sempre sarà il bersaglio, ma potrete aggiustare la mira finché non lo centrate. Questo è il senso.

Succede lo stesso con il codice tracciante. Usate questa tecnica in situazioni in cui non siete sicuri al cento per cento di dove state andando. Non dovete meravigliarvi se i primi tentativi sono fuori bersaglio: l'utente dice "non è quel che avevo in mente", i dati che vi servono non sono disponibili quando ne avreste bisogno, oppure sembrano probabili problemi di prestazioni. Pensate come modificare quel che avete, per portarlo più vicino al bersaglio e ringraziate di aver usato una metodologia di sviluppo *lean*. Un corpus di codice di piccole dimensioni ha poca inerzia: si modifica facilmente e in fretta. Sarete in grado di raccogliere feedback sull'applicazione e di generare una nuova versione più accurata più rapidamente e a costi minori che con qualsiasi altro metodo. Poiché poi ogni componente importante dell'applicazione è rappresentato nel codice tracciante, i vostri utenti avranno fiducia che quel che vedono sia basato sulla realtà e non solo una specifica su carta.

Codice tracciante contro prototipazione

Potreste pensare che questa idea del codice tracciante non sia altro che prototipazione con un nome più aggressivo, ma c'è una differenza. Con un prototipo si mira a esplorare aspetti specifici del sistema finale. Con un vero prototipo, si butta via tutto quello che si è messo insieme per mettere alla prova l'idea e si ricodifica opportunamente sfruttando quel che si è imparato.

Per esempio, supponiamo che dobbiate produrre un'applicazione che aiuti gli spedizionieri a determinare come disporre in container scatole di dimensioni strane. Fra gli altri problemi, l'interfaccia utente deve essere intuitiva e gli algoritmi che usate per determinare la sistemazione ottimale sono molto complessi.

Potreste preparare il prototipo di un'interfaccia utente per i vostri utenti finali con uno strumento GUI. Scrivete solo il codice necessario perché l'interfaccia risponda alle azioni degli utenti. Una volta che siete d'accordo sull'impianto, potete buttar via tutto e ricodificare, questa volta con la logica di business alla base, utilizzando il linguaggio prescelto. Analogamente, potete prototipare un certo numero di algoritmi che diano

effettivamente una disposizione. Potete codificare dei test funzionali in un linguaggio di alto livello permissivo come il Perl, e test di prestazioni a basso livello in qualcosa di più vicino alla macchina. In ogni caso, una volta presa la decisione, dovete ricominciare da capo e codificare gli algoritmi nel loro ambiente finale, interfacciandoli al mondo reale. Questa è la *prototipazione*, ed è molto utile.

Il codice tracciante affronta un problema diverso. Avete bisogno di sapere come l'applicazione nel complesso stia insieme. Volete mostrare ai vostri utenti come funzioneranno in pratica le interazioni, e volete dare ai vostri sviluppatori uno scheletro architettonico su cui andare a inserire il codice. In questo caso potete costruire un tracciante costituito da un'implementazione banale dell'algoritmo di sistemazione dei pacchi (magari una cosa del tipo "primo arrivato, primo servito") e un'interfaccia utente semplice ma funzionante. Una volta che avete saldato insieme tutti i componenti dell'applicazione, avete un quadro generale da mostrare ai vostri utenti e ai vostri sviluppatori. Con il tempo, aggiungete a questo quadro nuove funzionalità e completate le routine solo abbozzate. Ma il quadro rimane intatto e sapete che il sistema continuerà a comportarsi come quando è stato completato il codice tracciante.

La distinzione è sufficientemente importante da meritare di essere ribadita. La prototipazione genera codice "a perdere". Il codice tracciante è *lean*, magro ma completo, e costituisce parte dello scheletro del sistema finale. Potete pensare la prototipazione come il lavoro di *intelligence* che viene condotto prima che venga sparato un solo proiettile tracciante.

Vedi anche

- *Software abbastanza buono*, Capitolo 1
- *Prototipi e Post-it*, Capitolo 2
- *La trappola delle specifiche*, Capitolo 7
- *Grandi speranze*, Capitolo 8

Prototipi e Post-it

Molti settori usano i prototipi per mettere alla prova idee specifiche: un prototipo costa molto meno di una produzione a scala reale. Le case

automobilistiche, per esempio, possono costruire molti prototipi diversi di una vettura di nuova concezione. Ognuno è fatto per sottoporre a test un aspetto specifico dell'auto, l'aerodinamica, lo stile, le caratteristiche strutturali e così via. Magari si costruisce un modello in argilla per la galleria del vento, magari uno in compensato e nastro isolante per l'estetica, e così via. Qualche azienda fa anche un passo in più, e ora gran parte del lavoro di modellazione si svolge al computer, riducendo ulteriormente i costi. In questo modo si possono mettere alla prova elementi rischiosi o incerti, senza doversi impegnare nella costruzione di un oggetto concreto.

Costruiamo prototipi del software in modo analogo e per gli stessi motivi - per analizzare ed esporre i rischi e per avere la possibilità di correggere il tiro a costi molto ridotti. Come le case automobilistiche, prepariamo un prototipo per sondare uno o più aspetti specifici di un progetto.

In genere pensiamo i prototipi come basati sul codice, ma non devono esserlo sempre per forza. Come nel mondo dell'auto, possiamo costruire prototipi con materiali diversi. I Post-it sono ottimi per prototipare cose dinamiche come il flusso di lavoro e la logica dell'applicazione. Un'interfaccia utente si può prototipare disegnando su una lavagna, con un mock-up non funzionale disegnato con un programma di disegno a punti, o con un costruttore di interfacce.

I prototipi devono rispondere solo a poche domande, perciò sono molto più economici e rapidi da sviluppare rispetto alle applicazioni che vanno in produzione. Il codice può ignorare dettagli non importanti - non importanti per voi in quel momento, ma probabilmente molto importanti per l'utente in seguito. Se prototipate una GUI, per esempio, potete passarla liscia anche con risultati o dati sbagliati. Se invece studiate proprio gli aspetti computazionali o di prestazione potete cavarvela con una GUI appena abbozzata o magari anche senza GUI.

Se però vi ritrovate in un ambiente in cui *non potete* rinunciare ai dettagli, dovete chiedervi se sia il caso di costruire proprio un prototipo. Magari uno sviluppo in stile proiettili traccianti in quel caso sarebbe più appropriato (vedi *Proiettili traccianti*, in questo stesso capitolo).

Cose da prototipare

Che genere di cose potete scegliere di studiare con un prototipo? Tutto ciò che comporta un rischio. Tutto ciò che non è stato mai provato prima, o che è assolutamente vitale per il sistema finale. Tutto ciò che non è dimostrato, che è sperimentale o dubbio. Tutto ciò che vi mette a disagio. Potete prototipare:

- architettura
- nuove funzionalità in un sistema esistente
- struttura o contenuti di dati esterni
- strumenti o componenti di terze parti
- problemi di prestazioni
- il design dell'interfaccia utente.

La prototipazione significa imparare dall'esperienza: il suo valore non sta nel codice prodotto, ma negli insegnamenti che se ne traggono. Questo è il senso della prototipazione.

SUGGERIMENTO 16
Prototipate per imparare.

Come usare i prototipi

Quando costruite un prototipo, quali dettagli potete ignorare?

- *Correttezza*. Potete usare dati fittizi, quando non crea problemi.
- *Completezza*. Il prototipo può funzionare solo in un senso molto limitato; magari con un solo gruppo di dati di input e una sola voce di menu.
- *Robustezza*. È probabile che la verifica degli errori sia incompleta o del tutto assente. Se uscite dal sentiero predefinito, il prototipo può andare in pezzi e bruciare con una favolosa esibizione pirotecnica. Va bene.
- *Stile*. È doloroso ammetterlo per iscritto, ma il codice dei prototipi probabilmente non ha neanche bisogno di molti commenti o di molta documentazione. Potete produrre tonnellate di documentazione come risultato dell'esperienza fatta con il prototipo, ma molto poca al confronto nel sistema prototipo stesso.

Poiché un prototipo deve sorvolare sui dettagli e concentrarsi su aspetti specifici del sistema considerato, potreste volerlo implementare in un linguaggio di altissimo livello, più alto del resto del progetto (magari un linguaggio come Perl, Python o Tcl). Un linguaggio di scripting di alto livello consente di rimandare molti dettagli (compresa la specifica dei tipi di dati), pur producendo codice funzionale (per quanto incompleto o lento). Se avete bisogno di prototipare interfacce utente, prendete in considerazione strumenti come Tcl/Tk, Visual Basic, Powerbuilder o Delphi. (Se però dovete studiare prestazioni assolute, e non relative, dovrete restringere la vostra scelta a un linguaggio che sia vicino per prestazioni al linguaggio di destinazione.)

Linguaggi di scripting funzionano bene come "colla" per unire pezzi di basso livello in nuove combinazioni. Sotto Windows, Visual Basic può fare da collante fra controlli COM. Più in generale, potete usare linguaggi come Perl e Python per legare insieme librerie C di basso livello, o manualmente, o automaticamente con strumenti come SWIG [URL 28], che è libero. Seguendo questa impostazione, potete assemblare rapidamente componenti preesistenti in nuove configurazioni, per vedere come funzionano le cose.

Prototipi di architettura

Spesso si costruiscono modelli dell'intero sistema considerato. A differenza di quanto accade con i traccianti, nessuno dei singoli moduli del sistema prototipo deve essere particolarmente funzionale. In effetti, si può anche non dover scrivere codice per prototipare l'architettura: lo si può fare su una lavagna, con Post-it o schedine per indici. Ciò che si cerca di capire è come il sistema stia insieme nel complesso, rimandando a più tardi i dettagli. Ecco alcune aree specifiche che si possono indagare nel prototipo di architettura.

- Le responsabilità dei componenti principali sono ben definite e appropriate?
- Le collaborazioni fra i componenti principali sono ben definite?
- L'accoppiamento è ridotto al minimo?

- Si possono identificare potenziali fonti di duplicazione?
- Le definizioni e i vincoli dell'interfaccia sono accettabili?
- Ogni modulo ha un percorso di accesso ai dati di cui ha bisogno in esecuzione? Vi ha accesso *quando* ne ha bisogno?

Quest'ultimo aspetto è quello che di solito genera più sorprese e per il quale l'esperienza di prototipazione fornisce i risultati più preziosi.

Come non usare i prototipi

Prima di imbarcarvi in una prototipazione basata sul codice, assicuratevi che sia chiaro a tutti che il codice che scriverà sarà "a perdere". I prototipi possono essere tanto attraenti da trare in inganno chi non sa che sono solo prototipi. Dovete mettere bene in chiaro che quel codice andrà buttato, sarà incompleto e non potrà mai essere completato.

È facile farsi fuorviare dall'apparente completezza di un prototipo dimostrativo, e gli sponsor del progetto o il management possono pretendere di mettere in campo il prototipo (o la sua progenie) se non create le aspettative giuste. Ricordate loro che potete costruire un bellissimo prototipo di una nuova auto in compensato e nastro isolante, ma che non vi azzardereste mai a guidarla nel traffico dell'ora di punta!

Se sospettate che nel vostro ambiente o nella vostra cultura ci siano buone possibilità che lo scopo del codice prototipo venga male interpretato, meglio passare ai proiettili traccianti. Vi ritroverete con un quadro generale solido su cui basare lo sviluppo futuro.

Se usato opportunamente, un prototipo può farvi risparmiare una grande quantità di tempo, denaro, fatica e sofferenza, identificando e correggendo i potenziali problemi in una fase iniziale del ciclo di sviluppo, in un momento in cui sistemare gli sbagli costa poco ed è facile.

Vedi anche

- *Il gatto mi ha mangiato il codice sorgente*, Capitolo 1
- *Comunicate!*, Capitolo 1
- *Proiettili traccianti*, Capitolo 2
- *Grandi speranze*, Capitolo 8

Esercizi

4. Il marketing vuole un incontro e un po' di brainstorming su alcuni progetti di pagine web. Stanno pensando a mappe immagine cliccabili che portino ad altre pagine e cose simili, ma non riescono a decidere un modello per l'immagine, magari un'auto, o un telefono o una casa. Avete un elenco di pagine e di contenuti target; vorrebbero vedere qualche prototipo. Ah, per inciso: avete un quarto d'ora. Che strumenti potreste usare?

Linguaggi di settore

> I limiti del linguaggio sono i limiti del mondo.
> - Ludwig Wittgenstein

I linguaggi informatici influenzano il modo in cui si pensa a un problema, e il modo in cui si pensa a comunicare. Ogni linguaggio ha un suo elenco di caratteristiche - termini altisonanti come tipizzazione statica o dinamica, early o late binding, modelli di eredità (singola, multipla, o nessuna), e tutte quelle caratteristiche possono suggerire o mettere in ombra certe soluzioni. Progettare una soluzione avendo in mente il Lisp produrrà risultati diversi da una soluzione basata sul pensiero in stile C, e viceversa. D'altra parte, e pensiamo sia molto più importante, il linguaggio di settore del problema può anche suggerire una soluzione di programmazione.

Noi cerchiamo sempre di scrivere codice utilizzando il vocabolario del campo applicativo (vedi *La fossa dei requisiti*, nel Capitolo 7, dove consigliamo l'uso di un glossario di progetto). In qualche caso, possiamo passare al livello successivo e programmare effettivamente con il vocabolario, la sintassi e la semantica (il linguaggio) di quel campo.

Se ascoltate gli utenti di un sistema proposto, possono dirvi esattamente come deve funzionare il sistema:

> Ascoltare le transazioni definite dalla Norma 12,3 dell'ABC su un insieme di linee X.25, tradurle nel formato 43B dell'azienda XUZ, ritrasmetterle sul canale satellitare e memorizzare per analisi future.

Se i vostri utenti hanno molte enunciazioni ben definite, potete inventare un mini-linguaggio su misura per quel campo applicativo, che esprima esattamente ciò che vogliono.

```
From X25LINE1 (Format=ABC123) {
  Put TELSTAR1 (Format=XYZ43B);
  Store DB;
}
```

Questo linguaggio non deve essere eseguibile. Inizialmente, potrebbe essere semplicemente un modo per catturare le esigenze dell'utente: una specifica. Però potete pensare di fare un passo ulteriore e implementare effettivamente il linguaggio. La specifica diventa codice eseguibile.

Dopo che avete scritto l'applicazione, gli utenti vi danno un nuovo requisito: le transazioni con saldo negativo non devono essere memorizzate ma vanno rispedite sulle linee X.25 nel formato originale:

```
From X25LINE1 (Format=ABC123) {
  if (ABC123.balance < 0) {
    Put X25LINE1 (Format=ABC123);
  }
  else {
    Put TELSTAR1 (Format=XYZ43B);
    Store DB;
  }
}
```

Facile, no? Con il supporto opportuno, potete programmare stando molto più vicini al campo applicativo. Non vogliamo dire che gli utenti finali programmino effettivamente in questi linguaggi, ma che vi potete dare uno strumento che vi consentirà di lavorare più vicino al loro settore.

SUGGERIMENTO 17
Programmate stando vicini al settore del problema.

Che si tratti di un linguaggio semplice per configurare e controllare un programma applicativo, o di un linguaggio più complesso per specificare regole o procedure, pensiamo sia bene cercare dei modi per avvicinare il più possibile il progetto al campo del problema. Codificando a un livello di astrazione più alto, siete liberi di concentrarvi sul risolvere i problemi di quel settore e potete ignorare i banali dettagli di implementazione.

Ricordate che un'applicazione ha molti utenti. Ci sarà l'utente finale, che conosce le regole di business e gli output richiesti, ma ci sono anche utenti secondari: lo staff operativo, i responsabili di configurazione e collaudo, i programmatori dell'assistenza e della manutenzione e le future generazioni di sviluppatori. Ciascuno di questi utenti ha il suo proprio campo di problemi e potete generare mini-ambienti e mini-linguaggi per tutti loro.

Errori specifici del settore
Se scrivete per uno specifico settore, potete eseguire anche una validazione specifica per quel settore, comunicando i problemi in termini comprensibili ai vostri utenti. Prendete l'applicazione di commutazione appena vista. Supponiamo che l'utente scriva male il nome del formato:

```
From X25LINE1 (Format=AB123)
```

Se succedesse in un linguaggio di programmazione standard, di uso generale, potreste ricevere un messaggio di errore standard, di uso generale:

```
Syntax error: undeclared identifier
```

Con un mini-linguaggio, invece, sareste in grado di emettere un messaggio di errore che usi il vocabolario del settore:

```
"AB123" is not a format. Known formats are ABC123,
         XYZ43B, PDQB, and 42.
```

Ricordate che un'applicazione ha molti utenti. Ci sarà l'utente finale, che conosce le regole di business e gli output richiesti, ma ci sono anche utenti secondari: lo staff operativo, i responsabili di configurazione e collaudo, i programmatori dell'assistenza e della manutenzione e le future generazioni di sviluppatori. Ciascuno ha il suo proprio campo di problemi e potete generare mini-ambienti e mini-linguaggi per tutti loro.

Implementare un mini-linguaggio

Nella forma più semplice, un mini-linguaggio può avere un formato orientato alla riga e facilmente analizzabile sintatticamente. Nella pratica, probabilmente usiamo questa forma più spesso di ogni altra. Può essere analizzata semplicemente utilizzando enunciati `switch`, o con le espressioni regolari in linguaggi di scripting come Perl. La risposta all'Esercizio 5, nell'Appendice B, presenta una semplice implementazione in C.

Potete anche implementare un linguaggio più complesso, con una sintassi più formale. Il trucco è definire prima la sintassi, usare una notazione come le forme normali di Backus-Naur (BNF), che permettono di specificare ricorsivamente grammatiche non contestuali. (Qualsiasi buon libro sulla costruzione di compilatori o il *parsing* tratta esaustivamente le forme BNF.) Una volta specificata la grammatica, normalmente è banale convertirla nella sintassi di input per un generatore di parser. Chi programma in C e C++ usa da anni `yacc` (o la sua implementazione liberamente disponibile, `bison` [URL27]). Questi programmi sono documentati a fondo nel libro *Lex and Yacc* [LMB92]. Chi programma in Java può provare javaCC, che si può trovare all'indirizzo [URL26]. La risposta all'Esercizio 7 (nell'Appendice B) presenta un parser scritto in `bison`. Come si vedrà, una volta che si

conosce la sintassi, non ci vuole molto lavoro per scrivere semplici mini-linguaggi.

Esiste un altro modo per implementare un mini-linguaggio: estenderne uno già esistente. Per esempio, potreste integrare funzionalità a livello di applicazione con, poniamo, Python [URL9] e scrivere qualcosa di questo genere (grazie a Eric Vought per l'esempio):

```
record = X25LINE1.get(format=ABC123)
if (record.balance < 0):
        X25LINE1.put(record, format=ABC123)
else:
        TELSTAR1.put(record, format=XYZ43B)
        DB.store(record)
```

Linguaggi di dati e linguaggi imperativi

I linguaggi che implementate possono essere usati in due modi diversi.

I *linguaggi di dati* producono qualche forma di struttura di dati, utilizzata da un'applicazione. Questi linguaggi vengono usati spesso per rappresentare informazioni di configurazione. Per esempio, in tutto il mondo si usa il programma `sendmail` per instradare la posta elettronica su Internet. Ha molte caratteristiche eccellenti e molti vantaggi, controllati da un file di configurazione di un migliaio di righe, scritto nel linguaggio di configurazione dello stesso `sendmail`.

```
Mlocal, P=/usr/bin/procmail,
        F=lsDFMAw5:/|@qSPfhn9,
        S=10/30, R=20/40,
        T=DNS/RFC822/X-Unix,
        A=procmail -Y -a $h -d $u
```

Chiaramente, la leggibilità non è uno dei punti di forza di `sendmail`.

Per anni, la Microsoft ha usato un linguaggio di dati che può descrivere menu, widget, finestre di dialogo e altre risorse Windows. La Figura 2.2 mostra un dettaglio di un tipico file di risorsa, che è molto più facile da leggere dell'esempio di `sendmail`, ma si usa esattamente nello stesso modo: viene compilato per generare una struttura di dati.

```
MAIN_MENU MENU
{
  POPUP "&File"
  {
    MENUITEM "&New". CM_FILENEW
    MENUITEM "&Open". CM_FILEOPEN
    MENUITEM "&Save". CM_FILESAVE
  }
```

```
}
MY_DIALOG_BOX DIALOG 6, 15, 292, 287

STYLE DS_MODALFRAME | WS_POPUP | WS_VISIBLE
                      WS_CAPTION | WS_SYSMENU

CAPTION "My Dialog Box"
FONT 8, "Ms Sans Serif"
{
  DEFPUSHBUTTON "OK" , ID_OK, 232, 16, 50, 14
  PUSHBUTTON "Help", ID_HELP, 232, 52, 50, 14
  CONTROL "Edit Text Control", ID_EDIT1,
         "EDIT", WS_BORDER | WS_TABSTOP, 16, 16, 80, 56
  CHECKBOX "Checkbox", ID_CHECKBOX1, 153, 65, 42, 38,
         BS_AUTOCHECKBOX | WS_TABSTOP
}
```

Figura 2.2 Un file .rc di Windows.

I *linguaggi imperativi* vanno un passo oltre. Qui il linguaggio viene effettivamente eseguito, perciò può contenere enunciati, costrutti di controllo e altre cose simili (come lo script visto poco fa nel paragrafo *Linguaggi di settore*).

Potete usare anche vostri linguaggi imperativi per facilitare la manutenzione dei programmi. Per esempio, vi può essere richiesto di integrare informazioni provenienti da un'applicazione legacy nel nuovo sviluppo di una GUI. Un modo comune sarebbe mediante lo *screen scraping*: la vostra applicazione si connette all'applicazione sul mainframe come se fosse un normale utente umano, invia i codici di tasti premuti e "legge" le risposte che ottiene. Potreste scrivere l'interazione con un mini-linguaggio.

```
locate prompt "SSN:"
type "%s" social_security_number
type enter
waitfor keyboardunlock
if text_at(10,14) is "INVALID SSN" return bad_ssn
if text_at(10,14) is "DUPLICATE SSN" return dup_ssn
# ecc...
```

> **NOTA**
> In effetti, è possibile acquistare tool che consentono proprio questo tipo di script. Potete anche considerare pacchetti open source come Expect, che mettono a disposizione possibilità analoghe [URL24].

Quando l'applicazione stabilisce che è il momento di inserire un numero di Sicurezza sociale, chiama l'interprete su questo script, che poi controlla la transazione. Se l'interprete è immerso nell'applicazione, i due

possono addirittura condividere dati direttamente (per esempio, con un meccanismo di callback).

Qui state programmando nel dominio dei programmatori della manutenzione. Se l'applicazione sul mainframe cambia e i campi si spostano, il programmatore può semplicemente aggiornare la vostra descrizione di alto livello, anziché dover andare a frugare nei particolari del codice C.

Linguaggi standalone e incorporati

Non c'è bisogno che un mini-linguaggio venga usato direttamente dall'applicazione perché sia utile. Molte volte usiamo un linguaggio di specifica per creare cose (fra cui metadati) che vengono compilate, lette o comunque usate dal programma stesso (vedi *Metaprogrammazione*, nel Capitolo 5).

Per esempio, nel Capitolo 3 descriviamo un sistema in cui abbiamo usato Perl per generare un gran numero di varianti dalle specifiche iniziali di uno schema. Abbiamo inventato un linguaggio comune per esprimere lo schema del database, poi abbiamo generato tutte le forme che ci servivano - SQL, C, pagine web, XML e altre ancora. L'applicazione non usava direttamente la specifica, ma si basava sull'output prodotto.

È comune incorporare linguaggi imperativi di alto livello direttamente nell'applicazione, in modo che vadano in esecuzione quando viene eseguito il codice. Chiaramente è una possibilità potente: si può cambiare il comportamento dell'applicazione cambiando gli script che legge, tutto senza compilare. Questo può semplificare notevolmente la manutenzione in un campo applicativo dinamico.

Sviluppo facile o manutenzione facile?

Abbiamo esaminato varie grammatiche, dai semplici formati orientati alla riga a quelli più complessi che si avvicinano molto ai linguaggi reali. Dato che per implementarla bisogna fare fatica in più, perché scegliere una grammatica più complessa?

Quel che si ottiene in cambio sono estendibilità e manutenzione. Il codice per l'analisi sintattica di un linguaggio "vero" può essere più difficile da scrivere, ma sarà molto più facile da capire per le persone e da estendere in futuro con nuove caratteristiche e funzionalità. L'analisi sintattica di linguaggi molto semplici può essere molto facile, ma il linguaggio può essere criptico, come l'esempio di `sendmail` appena visto.

Molte applicazioni superano le loro aspettative di vita, probabilmente ve la caverete meglio se stringete i denti e adottate subito il linguaggio più complesso ma più leggibile. La fatica iniziale si ripagherà molte volte con la riduzione dei costi di assistenza e manutenzione.

Vedi anche

- *Metaprogrammazione*, Capitolo 5

Sfide

- Qualcuno dei requisiti del vostro progetto corrente si potrebbe esprimere in un linguaggio specifico per quel settore? Sarebbe possibile scrivere un compilatore o un traduttore in grado di generare la maggior parte del codice necessario?
- Se decidete di adottare mini-linguaggi come modo di programmazione più vicino al settore del problema, siete disposti ad accollarvi la fatica in più necessaria per implementarli. Potete immaginare qualche modo in cui il framework che sviluppate per un progetto possa essere riusato in altri?

Esercizi

5. Vogliamo implementare un mini-linguaggio per controllare un semplice pacchetto di disegno (magari un sistema di grafica della tartaruga). Il linguaggio è costituito da comandi di una sola lettera. Alcuni comandi sono seguiti da un singolo numero. Per esempio, questo input disegnerebbe un rettangolo:

```
P 2 # seleziona penna 2
D   # penna giù
W 2 # disegna ovest 2cm
N 1 # poi nord 1
E 2 # poi est 2
S 1 # poi di nuovo a sud
U   # penna su
```

Implementate il codice che analizza sintatticamente il linguaggio. Deve essere progettato in modo che sia semplice aggiungere nuovi comandi.

6. Progettate una grammatica BNF per analizzare una specifica temporale. Devono essere accettati tutti gli esempi seguenti:

```
4pm, 7:38pm, 23:42, 3:16, 3:16am
```

7. Implementate un parser (analizzatore sintattico) per la grammatica BNF dell'Esercizio 6 utilizzando yacc, bison o qualche altro generatore di parser.
8. Implementate con Perl il parser delle indicazioni temporali. (Suggerimento: le espressioni regolari creano buoni parser.)

Stime

Quanto tempo ci vorrà per spedire *Guerra e pace* su una linea con un modem a 56k? Di quanto spazio su disco ci sarà bisogno per un milione di nomi e indirizzi? Quanto tempo impiega un blocco di 1000 byte a passare attraverso un router? Quanti mesi ci vorranno per consegnare il progetto finito?

A un certo livello, sono tutte domande senza senso: in tutti i casi mancano delle informazioni. A tutte però si può dare una risposta, se non vi mette in crisi fare delle stime. Mentre calcolate una stima, poi, finirete per capire meglio il mondo in cui abitano i vostri programmi.

Se imparate a stimare, e sviluppate questa capacità fino al punto in cui avrete una percezione intuitiva dell'ordine di grandezza delle cose, darete l'impressione di possedere una capacità magica di determinarne la fattibilità. Quando qualcuno dice "manderemo il backup su una linea ISDN al sito centrale" sarete in grado di capire intuitivamente se si tratti di un'idea praticabile oppure no. Quando codificate, sarete in grado di sapere quali sistemi hanno bisogno di un'ottimizzazione e quali invece possono essere lasciati così come sono.

SUGGERIMENTO 18
Stimate per evitare sorprese.

Come omaggio, alla fine di questa sezione vi riveleremo l'unica risposta corretta da dare quando qualcuno vi chiede una stima.

Quanto accurato è abbastanza accurato?

In una certa misura, tutte le risposte sono stime: qualcuna semplicemente è più precisa di altre. Perciò la prima domanda che dovete porvi quando qualcuno vi chiede una stima è quale sia il contesto in cui

verrà ricevuta la vostra risposta. Ha bisogno di una stima molto precisa, oppure sta cercando solo un'indicazione di massima?

- Se la nonna vi chiede quando arriverete, probabilmente si sta chiedendo se prepararvi il pranzo o la cena. Un sommozzatore bloccato sott'acqua che sta per finire la scorta d'aria probabilmente sarà interessato a una risposta precisa al secondo.
- Qual è il valore di pi greco? Se vi state chiedendo quanta siepe acquistare da mettere attorno a un'aiuola circolare, "3" probabilmente è una risposta sufficientemente precisa. Se siete a scuola, magari "21/7" è una buona approssimazione. Se invece lavorate alla NASA, può darsi che un valore approssimato alla dodicesima cifra decimale vada bene.

NOTA

"3" a quanto pare è una stima abbastanza precisa anche se siete legislatori. Nel 1897, una proposta di legge dello Stato dell'Indiana, la n. 246, voleva stabilire che da quel momento in poi pi greco avesse il valore "3". La proposta è stata accantonata indefinitamente, al secondo passaggio, quando un professore di matematica ha fatto notare che i poteri degli onorevoli non si estendevano fino alla promulgazione di leggi di natura.

Una delle cose interessanti sulle stime è che le unità che usate fanno una differenza nell'interpretazione del risultato. Se dite che una certa cosa richiederà circa 130 giornate lavorative, l'interlocutore si aspetterà che ci siate andati molto vicini. Se invece dite "Oh, circa sei mesi", sapranno di doverselo aspettare fra i cinque e i sette mesi da ora. Entrambi i numeri rappresentano lo stesso intervallo di tempo, ma "130 giorni" probabilmente suggerisce un grado di precisione molto più elevato di quello che pensate. Vi consigliamo di formulare le vostre previsioni di tempi in questo modo:

Durata	Indicate la stima in
1-1,5 giorni	giorni
3-8 settimane	settimane
8-30 settimane	mesi
Oltre 30 settimane	pensateci bene prima di fare una stima

Se, dopo aver fatto tutti i conti del caso, decidete che un progetto richiederà 125 giornate lavorative (25 settimane), potreste dare comunque una stima di “circa sei mesi”.

La stessa idea vale per le stime di qualsiasi grandezza: scegliete le unità in cui formulate la risposta in modo che rispecchino la precisione di cui volete dare l’impressione.

Da dove arrivano le stime?

Tutte le stime si basano su modelli del problema. Prima di entrare troppo a fondo nelle tecniche di costruzione di modelli, dobbiamo citare un trucco elementare che dà sempre buone risposte: chiedete a qualcuno che l’ha già fatto. Prima di impegnarvi troppo nella costruzione di un modello, guardatevi in giro e vedete se c’è qualcuno che si è trovato in passato in una situazione simile. Vedete come ha risolto il suo problema. È improbabile che troviate una corrispondenza esatta, ma rimarrete sorpresi da quante volte è possibile basarsi sulle esperienze degli altri.

Capire la domanda

La prima parte di un esercizio di stima è capire bene quello che ci viene chiesto. Oltre al problema del livello di precisione, bisogna afferrare bene l’ambito in cui ci si muove. Spesso è implicito nella domanda, ma è bene prendere l’abitudine di riflettere quale sia l’ambito d’azione prima di fare congetture. Spesso l’ambito che si sceglie entra a far parte della risposta: “Ipotizzando che non ci siano incidenti e che l’auto non sia a corto di carburante, dovrei arrivare in venti minuti”.

Costruire un modello del sistema

Questa è la parte divertente di una stima. In base alla comprensione della domanda, si costruisce un modello mentale approssimativo e ridotto all’osso. Se si devono stimare tempi di risposta, il modello può comprendere un server e qualche tipo di traffico in arrivo; per un progetto, il modello può essere costituito dai passi che la vostra

organizzazione usa durante lo sviluppo, insieme con un'immagine molto grezza di come possa essere implementato il sistema.

La costruzione di modelli può essere creativa, e anche utile sul lungo termine. Spesso porta a scoprire schemi e processi sottostanti che superficialmente non erano evidenti. Potreste essere spinti addirittura a riesaminare la domanda di partenza: "Hai chiesto una stima per fare X. Se però consideri Y, una variante di X, si potrebbe fare in circa metà tempo, e perderesti solo una funzione".

La costruzione del modello introduce imprecisioni nel processo di stima. È una cosa inevitabile, ma anche vantaggiosa. Si compensa la minore precisione con la maggiore semplicità. Raddoppiare lo sforzo sul modello può dare un aumento solo piccolo della precisione: l'esperienza vi dirà quando è il momento di smettere di perfezionare ulteriormente.

Suddividere il modello in componenti

Una volta che avete un modello, potete suddividerlo in componenti. Dovrete scoprire le regole matematiche che descrivono come quei componenti interagiscono fra loro. A volte il contributo di un componente è un singolo numero, che viene sommato al risultato. Alcuni componenti possono fornire come risultato valori che diventano poi fattori di una moltiplicazione, altri possono essere più complicati (per esempio quelli che simulano l'arrivo di traffico a un nodo).

Troverete che per ciascun componente in genere esistono parametri che influenzano il suo contributo al modello complessivo. In questa fase, limitatevi a identificare i singoli parametri.

Dare a ciascun parametro un valore

Una volta individuati i parametri, potete riesaminarli e assegnare un valore a ciascuno. Potete immaginare che in questo passaggio introdurrete qualche errore. Il trucco è stabilire quali parametri influiscono maggiormente sul risultato, e concentrarsi sul dare a questi un valore quasi giusto. In genere i parametri i cui valori vengono sommati a un risultato sono meno significativi di quelli che moltiplicano o dividono. Il raddoppio della velocità di una linea può raddoppiare la quantità di dati

ricevuti in un'ora, ma l'aggiunta di un ritardo di transito di 5 millisecondi non avrà effetti apprezzabili.

Dovete avere un metodo giustificabile per calcolare questi parametri critici. Per l'esempio delle code, potreste misurare il tasso di arrivo reale delle transazioni nel sistema esistente, o trovare un sistema simile da misurare. Analogamente, potreste misurare il tempo richiesto in quel momento per servire una richiesta, o formulare una stima utilizzando le tecniche descritte in questa sezione. In effetti, vi troverete spesso a basare una stima su altre stime parziali: è lì che si insinuano gli errori più gravi.

Calcolare le risposte

Solo nei casi più semplici una stima darà una singola risposta. Potete dire tranquillamente "Posso percorrere questi cinque isolati in 15 minuti", ma, quando i sistemi sono più complessi, vorrete mettere qualche paletto. Fate più calcoli, variando i valori dei parametri critici, finché non stabilite quali abbiano davvero un peso importante nel modello. Un foglio di calcolo può essere di grande aiuto. Poi formulate la risposta in funzione di quei parametri. "Il tempo di risposta è di circa tre quarti di secondo se il sistema ha un bus SCSI e 64 MB di memoria, e di un secondo con 48 MB di memoria." (Notate come "tre quarti di secondo" dia un senso di precisione ben diverso da 750 millisecondi.)

Nella fase di calcolo, potreste ottenere risposte che sembrano strane. Non abbiate troppa fretta di escluderle: se i calcoli sono giusti, può darsi che sia sbagliata la vostra comprensione del problema, o che non sia corretto il modello. Si tratta di informazioni preziose.

Tener traccia della propria abilità nelle stime

Ci sembra una buona idea tener traccia delle stime fatte, in modo da vedere quanto erano azzeccate. Se una stima complessiva comportava il calcolo di sottostime, tenete traccia anche di queste. Spesso scoprirete che le vostre stime sono molto buone: in effetti, dopo un po', ve lo aspetterete.

Quando una stima si rivela sbagliata, non limitatevi a scrollare le spalle e andare per la vostra strada: scoprite perché la realtà è stata diversa dalle vostre congetture. Magari avete scelto dei parametri che non erano adeguati alla realtà del problema, o magari il vostro modello era sbagliato. Quale che sia il motivo, prendete il tempo per scoprire che cosa è successo. Se lo fate, la vostra prossima stima sarà migliore.

Stimare i tempi di un progetto

Le regole normali per le stime possono mostrare la corda di fronte alla complessità e ai capricci dello sviluppo di un'applicazione di una certa mole. Spesso l'unico modo per stabilire la tabella di marcia di un progetto è fare dell'esperienza sullo stesso progetto. Non è un paradosso, se praticate lo sviluppo incrementale, ripetendo i passi seguenti.

- Verificare i requisiti.
- Analizzare i rischi.
- Progettare, implementare, integrare.
- Validare con gli utenti.

Inizialmente, magari avrete solo un'idea vaga di quante iterazioni saranno necessarie, o di quanto ciascuna durerà. Qualche metodo richiede che lo si stabilisca nella stesura del piano iniziale, ma è un errore, tranne forse che per i progetti più elementari. Se non dovete creare un'applicazione simile a una che avete già realizzato in precedenza, con lo stesso team e la stessa tecnologia, finireste per tirare a indovinare.

Dunque, completate la codifica e i test della funzionalità iniziale e lo segnate come fine del primo incremento. Sulla base di quell'esperienza, potete perfezionare la vostra congettura iniziale sul numero delle iterazioni e su quello che può essere incluso in ciascuna. Ogni volta la stima viene perfezionata sempre più, e con essa cresce la fiducia nel piano dei tempi.

SUGGERIMENTO 19
Iterate la pianificazione dei tempi con il codice.

Non farà magari la felicità del management, che in genere vuole un numero solo, chiaro e preciso, ancora prima di dare il via al progetto, ma

dovrete far capire che il team, la sua produttività e l'ambiente determineranno i tempi. Formalizzandolo, e perfezionando il piano dei tempi nell'ambito di ciascuna iterazione, darete le stime più precise che potete.

Che cosa dire quando viene chiesta una stima

Rispondete "Mi faccio sentire io".

Quasi sempre otterrete risultati migliori se rallentate il processo e perdete del tempo a esaminare i passi che descriviamo in questa sezione. Le stime fatte davanti alla macchinetta del caffè (come il caffè) vi si ritorceranno contro.

Vedi anche

- *Velocità degli algoritmi,* Capitolo 6

Sfide

- Tenete un registro delle vostre stime. Per ciascuna, verificate quanto si sia dimostrata precisa. Se l'errore era superiore al 50 per cento, cercate di scoprire dove siete finiti fuori strada.

Esercizi

9. Vi viene chiesto: "Che cosa ha una larghezza di banda maggiore, una linea di comunicazione a 1 Mbps o una persona che va da un computer all'altro con un nastro da 4 GB in tasca?". Quali vincoli indicherete, per essere sicuri che l'ambito della vostra risposta sia corretto? (Per esempio, potreste dire che il tempo di accesso al nastro viene ignorato.)
10. Allora, che cosa ha la larghezza di banda maggiore?

Gli strumenti di base

Ogni artigiano inizia il proprio percorso con un insieme di base di strumenti di buona qualità. Chi lavora il legno avrà bisogno di righelli, calibri, un paio di seghe, qualche buona pialla, buoni ceselli, trapani e punte, mazzuoli e morsetti. Questi strumenti saranno scelti con amore, saranno fatti per durare, serviranno a scopi specifici con poca sovrapposizione con altri strumenti e, forse la cosa più importante, il nostro ebanista in erba se li sentirà bene in mano.

Poi inizia un processo di apprendimento e di adattamento. Ogni strumento avrà la sua personalità e le sue peculiarità e dovrà essere trattato in un suo modo speciale. Ciascuno dovrà essere appuntito o affilato in un modo specifico, o tenuto in un certo modo. Con il tempo, ciascuno si logorerà a seconda dell'uso, fino a che l'impugnatura non sembrerà un calco delle mani dell'artigiano e la superficie di taglio non si allineerà perfettamente con l'angolo a cui lo strumento viene tenuto. A quel punto, gli strumenti porteranno direttamente dal cervello dell'artigiano al prodotto finito, saranno diventati estensioni delle sue mani. Con il tempo, aggiungerà nuovi strumenti, forme, seghe circolari a guida laser, maschere a coda di rondine - tutti ottimi prodotti tecnologici, ma potete scommetterci che sarà davvero felice solo quando avrà in mano quegli strumenti originali, quando sentirà la pialla cantare mentre scorre sul legno.

Gli strumenti amplificano il talento: quanto migliori sono i vostri strumenti, quanto meglio sapete usarli, tanto più produttivi potete essere. Iniziate con un insieme di base di strumenti di applicazione generale. Con l'esperienza, e quando incontrerete richieste speciali, amplierete con regolarità il vostro parco attrezzi. State sempre all'erta per individuare modi migliori di fare le cose. Se vi imbattete in una situazione in cui vi

sembra che gli strumenti che avete a disposizione non sarebbero stati adeguati, prendete nota di cercare qualcosa di diverso o di più potente, che avrebbe potuto essere d'aiuto. Lasciate che sia il bisogno a guidare i vostri acquisti.

Molti programmatori alle prime armi commettono l'errore di adottare un unico strumento di grande potenza, per esempio un particolare ambiente di sviluppo integrato (IDE, Integrated Development Environment) e non abbandonano mai la sua comoda interfaccia. È un errore: dobbiamo sentirci a nostro agio oltre i limiti imposti da un IDE, e l'unico modo è avere a portata di mano lo strumentario di base, affilato e pronto per l'uso.

In questo capitolo parleremo di investire nello strumentario di base. Come in ogni buona discussione sugli strumenti, inizieremo (in *Il potere del puro testo*) esaminando le materie prime, quelle a cui dovrete dare forma. Da lì passeremo al banco da lavoro, nel nostro caso il computer. Come potete usare il computer per trarre il massimo dagli strumenti che usate? Ne parleremo in *Giochi di shell*. Ora che abbiamo la materia prima e un banco su cui lavorare, passiamo allo strumento che userete più di ogni altro. In *Power Editing*, vi daremo qualche suggerimento su come diventare più efficienti.

Per essere sicuri di non perdere nulla del nostro prezioso lavoro, è bene usare sempre un sistema di *Controllo del codice sorgente*, anche per cose come la rubrica personale degli indirizzi! Dato poi che il signor Murphy in fondo era un ottimista, non diventerete mai grandi programmatori se non diventerete abilissimi nel *Debug*.

Avrete bisogno di un po' di colla per tenere insieme tutte le magie. Vedremo qualche possibilità, come awk, Perl e Python, in *Manipolazione del testo*.

Come chi lavora il legno spesso costruisce delle maschere per avere una guida nella realizzazione di pezzi complessi, i programmatori possono scrivere codice che scrive a sua volta codice. Ne parliamo in *Generatori di codice*.

Dedicate del tempo ad apprendere come usare questi strumenti, e a un certo punto scoprirete con un po' di meraviglia che le vostre dita si muovono sulla tastiera e manipolano il testo senza che dobbiate ragionarci coscientemente. Gli strumenti saranno diventati estensioni delle vostre mani.

Il potere del puro testo

Da programmatori pragmatici, il nostro materiale di base non è il legno o il ferro, ma la conoscenza. Raccogliamo requisiti come conoscenza, poi esprimiamo quella conoscenza nei nostri progetti, nelle implementazioni, nei test e nei documenti. Siamo convinti che il formato migliore per conservare a lungo la conoscenza sia il *puro testo*. Con il puro testo, abbiamo la possibilità di manipolare la conoscenza, manualmente o da programma, utilizzando praticamente tutti gli strumenti a nostra disposizione.

Che cos'è il puro testo?

Il *puro testo* (*plain text*) è costituito da caratteri stampabili, in forma direttamente leggibile e comprensibile da esseri umani. Per esempio, il frammento seguente è costituito da caratteri stampabili, ma è senza senso.

```
Field19=467abe
```

Il lettore non ha idea di quale possa essere il significato di `467abe`. Una scelta migliore sarebbe renderlo *comprensibile* agli esseri umani.

```
DrawingType=UMLActivityDrawing
```

Testo puro non significa testo non strutturato: XML, SGML e HTML sono ottimi esempi di puro testo con una struttura ben definita. Con il puro testo si può fare qualsiasi cosa si possa fare con un formato binario, anche il *versioning*.

Il testo puro tendenzialmente è a un livello più alto di una codifica binaria, che di solito è derivata direttamente dall'implementazione. Supponiamo che vogliate memorizzare una proprietà `uses_menus` che può essere `TRUE` o `FALSE`. Utilizzando il testo, potreste scriverla come

```
myprop.uses_menus=FALSE
```

Confrontatela con `0010010101110101`.

Con la maggior parte dei formati binari, il problema è che il contesto necessario per comprendere i dati è separato dai dati stessi. Si separano artificialmente i dati dal loro significato. I dati potrebbero anche essere cifrati; sono assolutamente senza senso, in mancanza della logica applicativa che li analizzi. Con il puro testo, invece, si può avere un

flusso di dati autodescrittivi, indipendente dall'applicazione che l'ha creato.

SUGGERIMENTO 20
Conservate la conoscenza in forma di puro testo.

Svantaggi

L'uso del puro testo presenta due svantaggi importanti: (1) può richiedere più spazio di un formato binario compresso e (2) può essere più costoso, dal punto di vista computazionale, interpretare ed elaborare un file di puro testo.

A seconda dell'applicazione, uno o entrambi questi svantaggi possono essere inaccettabili: per esempio, quando si conservano dati di telemetria da satellite o come formato interno di un database relazionale. Anche in questi casi, però, può essere accettabile conservare in forma di puro testo *metadati* sui dati grezzi (vedi *Metaprogrammazione*, nel Capitolo 5).

Qualche programmatore avrà il timore che l'espressione dei metadati in puro testo significhi esporli agli utenti del sistema, ma è un timore malriposto. I dati binari saranno più oscuri del puro testo, ma non sono più sicuri. Se avete paura che gli utenti vedano le password, cifratele. Se non volete che modifichino i parametri di configurazione, includete un *hash sicuro* di tutti i valori dei parametri nel file, sotto forma di somma di controllo (*checksum*).

NOTA
Per l'inclusione di hash sicuri si usa spesso MD5. Per un'eccellente introduzione al meraviglioso mondo della crittografia, vedi [Sch95].

Il potere del testo

Poiché *più ingombrante* e *più lento* non sono fra le caratteristiche desiderate più spesso dagli utenti, perché impegolarsi con il puro testo? Quali sono i benefici?

- Garanzia contro l'obsolescenza.
- Potenza.
- Maggiore facilità di test.

Garanzia contro l'obsolescenza

Dati in forma leggibile dagli esseri umani e dati che si autodescrivono sopravvivranno a ogni altra forma di dati e alle applicazioni che li hanno creati. Punto. Se i dati sopravvivono, avrete la possibilità di usarli, potenzialmente, anche molto dopo la definitiva scomparsa dell'applicazione originale che li ha scritti.

Potete analizzare sintatticamente un file simile con una conoscenza solo parziale del suo formato; nel caso dei file binari, dovete conoscere tutti i particolari dell'intero formato per poterne effettuare correttamente l'analisi sintattica.

Prendete un file di dati di qualche sistema legacy (e tutto il software diventa "legacy" non appena è stato scritto). Non sapete un granché dell'applicazione originale; tutto quello che vi importa è che manteneva una lista dei numeri di Sicurezza sociale dei clienti, che dovete trovare ed estrarre. Fra i dati, vedete

```
<FIELD10>123-45-6789</FIELD10>
...
<FIELD10>567-89-0123</FIELD10>
...
<FIELD10>901-23-4567</FIELD10>
```

Riconoscendo il formato di un numero di Sicurezza sociale, potete rapidamente scrivere un programmino per estrarre quei dati, anche se non avete alcuna informazione su tutto il resto che si trova nel file.

Immaginatevi invece se il file fosse stato formattato in questo modo:

```
AC27123456789B11P
...
XY43567890123QTYL
...
6T2190123456788AM
```

Non avreste riconosciuto con altrettanta facilità il significato dei numeri. Questa è la differenza fra *leggibile da un essere umano* e *comprensibile a un essere umano*.

Intanto che ci siamo, anche `FIELD10` non è di grande aiuto. Qualcosa come

```
<SSNO>123-45-6789</SSNO>
```

renderebbe l'esercizio banale, e garantirebbe che i dati sopravvivano a qualsiasi progetto li abbia creati.

Potenza

Praticamente ogni strumento nell'universo informatico, dai sistemi di gestione del codice sorgente agli ambienti di compilazione, dagli editor ai filtri standalone, può operare con il puro testo.

> **La filosofia Unix**
> Unix è famoso per essere stato progettato secondo la filosofia degli strumenti piccoli e affilati, ciascuno mirato a fare una cosa sola molto bene. Questa filosofia è resa possibile dall'uso di un formato di fondo comune, il file di puro testo orientato alla riga. I database utilizzati per l'amministrazione di sistema (utenti e password, configurazione di rete e così via) sono tutti conservati come file di puro testo. (Qualche sistema, per esempio Solaris, mantiene anche una forma binaria di certi database per l'ottimizzazione delle prestazioni: la versione in puro testo è conservata come interfaccia alla versione binaria.) Quando un sistema va in tilt, potreste avere accesso solo a un ambiente minimo per ripristinarlo (potreste non avere accesso ai driver grafici, per esempio). Situazioni come questa possono davvero farvi apprezzare la semplicità del puro testo.

Per esempio, supponiamo dobbiate mettere in esercizio una applicazione di grandi dimensioni con un file di configurazione complesso, specifico per il sito (viene in mente `sendmail`). Se il file è in puro testo, potete sottoporlo a un sistema di controllo del codice sorgente (vedi *Controllo del codice sorgente*, nel seguito di questo capitolo), in modo che venga generata automaticamente una cronologia di tutti i cambiamenti. Strumenti di confronto fra file come `diff` e `fc` consentono di vedere a colpo d'occhio quali cambiamenti siano stati apportati, mentre `sum` consente di generare una somma di controllo per verificare che il file non abbia subito modifiche accidentali (o malintenzionate).

Maggiore facilità di test

Se usate il puro testo per creare dati sintetici che guidino i test di sistema, è semplice aggiungere, aggiornare o modificare i dati di test *senza dover creare strumenti speciali a questo scopo*. Analogamente, output in puro testo da test di regressione può essere analizzato banalmente (con `diff`, per esempio) oppure essere sottoposto a un esame più completo con Perl, Python o qualche altro strumento di scripting.

Il minimo comun denominatore

Anche nel futuro di agenti intelligenti basati su XML che navigano autonomamente nell'Internet selvaggia e pericolosa, negoziando fra loro scambi di dati, l'onnipresente file di testo sarà ancora lì. In effetti, in ambienti eterogenei i vantaggi del puro testo compensano abbondantemente tutti i suoi svantaggi. Dovete essere sicuri che tutte le parti possano comunicare utilizzando uno standard comune, e il puro testo è quello standard.

Vedi anche

- *Controllo del codice sorgente*, in questo Capitolo
- *Generatori di codice*, in questo Capitolo
- *Metaprogrammazione*, Capitolo 5
- *Lavagne*, Capitolo 5
- *Automazione onnipresente*, Capitolo 8
- *È tutta scrittura*, Capitolo 8

Sfide

Progettate un piccolo database di rubrica personale (nome, numero di telefono ecc.) utilizzando una rappresentazione binaria immediata nel linguaggio che preferite. Fatelo prima di leggere il resto di questa sfida.

1. Traducete quel formato in un formato di puro testo mediante XML.
2. Per ciascuna versione, aggiungete un nuovo campo a lunghezza variabile, chiamato *directions*, in cui potete inserire le indicazioni per raggiungere l'abitazione di ciascuna persona.

Quali problemi sorgono, relativamente a gestione delle versioni ed estendibilità? Quale forma è stata più facile da modificare? E per quel che riguarda la conversione dei dati preesistenti?

Giochi di shell

Chiunque lavori il legno ha bisogno di un bel banco da lavoro, robusto e affidabile, un posto in cui tenere i pezzi a un'altezza comoda mentre li lavora. Il banco è il centro del laboratorio, e l'artigiano continua a ritornarvi mentre il pezzo prende forma.

Per un programmatore che manipola file di testo, il banco da lavoro è la shell di comando. Dal prompt della shell, potete invocare tutto il repertorio di strumenti, usando i *pipe* per combinarli in modi che gli sviluppatori originali mai si sarebbero sognati. Dalla shell potete lanciare applicazioni, debugger, browser, editor e utility. Potete cercare file, interrogare lo stato del sistema e filtrare l'output. Programmando la shell, poi, potete costruire macro-comandi complessi per attività che dovete svolgere spesso.

Ai programmatori cresciuti fra interfacce GUI e ambienti di sviluppo integrati (IDE) potrebbe sembrare una posizione estrema: in fin dei conti, non si può fare tutto altrettanto bene puntando e facendo clic con il mouse?

La semplice risposta è "no". Le interfacce GUI sono meravigliose e per certe operazioni semplici possono essere più veloci e più comode. Spostare file, leggere email codificate MIME e scrivere lettere sono tutte cose che si fanno benissimo in un ambiente grafico, ma se svolgete tutto il vostro lavoro con le GUI, non potete attingere a tutte le potenzialità dell'ambiente. Non potrete automatizzare attività, né usare tutta la potenza degli strumenti disponibili. E non sarete in grado di combinare quegli strumenti per creare *macrostrumenti* personalizzati. Un vantaggio delle GUI è il WYSIWYG, *what you see is what you get*, quello che vedi è quello che ottieni. Lo svantaggio è WYSIAYG, *what you see is all you get*, quello che vedi è tutto quello che ottieni.

Gli ambienti GUI normalmente sono limitati alle capacità previste dai loro progettisti. Se dovete andare al di là del modello fornito dal progettista, di solito siete sfortunati - e vi succederà spesso di dover andare al di là del modello. I programmatori pragmatici non si limitano a scrivere codice, o a sviluppare modelli a oggetti, o a scrivere documentazione, o ad automatizzare il processo di build: fanno *tutte* queste cose. La portata di qualsiasi strumento di solito è limitata alle attività che si prevede quello strumento debba svolgere. Per esempio, supponiamo che dobbiate integrare nel vostro IDE un preprocessore di codice (per implementare la progettazione per contratto, o pragma di multielaborazione, o qualcosa di simile). A meno che chi ha progettato l'IDE non abbia fornito esplicitamente gli "agganci" giusti, l'integrazione non sarà possibile.

Magari trovate già comodo lavorare con il prompt dei comandi, e in quel caso potete tranquillamente saltare questa sezione; in caso contrario, dovete convincervi che la shell è vostra amica. Da programmatori pragmatici, vorrete continuamente svolgere operazioni ad hoc, fare cose che la GUI probabilmente non consente. La riga di comando è più adatta, se volete combinare rapidamente un paio di comandi ed eseguire una interrogazione o qualche altra attività. Ecco qualche esempio.

Trovare tutti i file `.c` *modificati successivamente al Makefile.*

Shell: `find . -name '*.c' -newer Makefile -print`

GUI: Aprire Esplora file, navigare fino alla directory giusta, fare clic sul Makefile e annotare giorno e ora in cui è stato modificato. Poi selezionare la voce *Trova* nel menu *Strumenti* e inserire `*.c` come specificazione di file. Selezionare la scheda della data e inserire la data di cui si è preso nota dal Makefile nel primo campo data. Poi fare clic su OK.

Costruire un archivio zip/tar del codice sorgente

Shell: `zip archive.zip *.h *.c` *oppure* `tar cvf archive.tar *.h *.c`

GUI: Avviare una utility di compressione (per esempio la versione shareware di WinZip [URL41], selezionare *Crea nuovo archivio*, inserire il nome, selezionare la directory sorgente nella finestra di dialogo per l'aggiunta di file, impostare il filtro a `*.c`, fare clic su *Aggiungi*, impostare il filtro a `*.h`, fare clic su *Aggiungi*, poi chiudere l'archivio.

Quali file Java non sono stati modificati nell'ultima settimana?

Shell: `find . -name '*.java' -mtime +7 -print`

GUI: Fare clic e navigare fino a *Trova file*, fare clic sul campo *Nome* e scrivere *.java, selezionare la scheda *Data di modifica*. Poi selezionare *Fra*. Fare clic sulla data iniziale e inserire la data iniziale del progetto. Fare clic sulla data finale e inserire la data di una settimana fa (tenetevi sempre a portata di mano un calendario). Fare clic su *Trova*.

Di questi file, quali usano le librerie awt?

Shell: `find . -name '*.java' -mtime +7 -print | xargs grep 'java.awt'`

GUI: Caricare in un editor ogni file presente nella lista dell'esempio precedente e cercare la stringa `jawa.awt`. Scrivere il nome di ciascun file contenente una corrispondenza.

Chiaramente, l'elenco potrebbe continuare. I comandi della shell potranno essere oscuri o secchi, ma sono potenti e concisi. Dato poi che i

comandi di shell possono essere combinati in file di script (o file di comandi nei sistemi Windows), potete costruire sequenze di comandi per automatizzare cose che fate spesso.

SUGGERIMENTO 21
Usate la potenza delle shell di comando.

Acquisite familiarità con la shell, e la vostra produttività avrà un'impennata. Avete bisogno di un elenco di tutti i nomi di package unici importati esplicitamente dal vostro codice Java? Quanto segue lo memorizza in un file chiamato `list`.

```
grep '^import ' *.java |
  sed -e's/.*import *//' -e's/;.*$//' |
  sort -u >list
```

Se non avete dedicato molto tempo a esplorare le possibilità della shell di comando dei sistemi che usate, potrebbe sembrarvi inarrivabile, ma investite un po' di energia nell'acquisire familiarità con la vostra shell e le cose cominceranno subito ad andare a posto. Provate un po' la shell di comando e rimarrete sorpresi da quanto può rendervi più produttivi.

Utility di shell e sistemi Windows

Le shell di comando fornite con i sistemi Windows stanno gradualmente migliorando, ma le utility da riga di comando di Windows sono ancora inferiori alle loro corrispondenti Unix. Non tutto, però, è perduto.

Esiste un pacchetto chiamato Cygwin [URL31], sviluppato in origine dalla Cygnus Solutions, poi acquisita da Red Hat. Oltre a fornire uno strato di compatibilità Unix per Windows, Cygwin mette a disposizione una raccolta di oltre un centinaio di utility Unix, fra cui utility molto apprezzate come `ls`, `grep` e `find`. Utility e librerie si possono scaricare e usare liberamente, ma non dimenticate di leggere la licenza. La distribuzione Cygwin è fornita con la shell Bash.

NOTA
La GNU General Public License [URL57] è una sorta di "virus" legale che gli sviluppatori open source utilizzano per proteggere i loro (e vostri) diritti. Dovete dedicare un po' di tempo a leggerla. In sostanza, dice che potete usare e modificare software con licenza GPL, ma se distribuite qualche modifica deve essere accompagnata a sua volta da una licenza GPL (e indicata come tale) e dovete rendere disponibile il relativo codice sorgente. Questa è la parte "virale" - ogni volta che derivate un'opera da un'opera con GPL, anche la derivata deve

essere sotto GPL. Questo però non vi pone alcun limite se usate semplicemente gli strumenti - definire proprietà e tipo di licenza del software sviluppato utilizzando quegli strumenti è a vostra scelta.

Usare strumenti Unix sotto Windows

La disponibilità di strumenti Unix di alta qualità sotto Windows ci fa molto piacere e li usiamo quotidianamente. Dovete sapere però che ci sono problemi di integrazione. A differenza delle loro corrispondenti MS-DOS, queste utility sono sensibili alla differenza fra maiuscole e minuscole nei nomi di file, perciò `ls a*.bat` non troverà AUTOEXEC.BAT. Potete incontrare problemi anche con i nomi di file che contengono spazi e con differenze nei separatori dei percorsi. Infine, si trovano problemi interessanti quando si eseguono programmi MS-DOS che si aspettano argomenti in stile MS-DOS sotto le shell Unix. Per esempio, le utility Java di JavaSoft usano un segno di due punti come separatore CLASSPATH sotto Unix, ma usano un punto e virgola sotto MS-DOS. Di conseguenza, uno script Bash o ksh che gira in un sistema Unix girerà nello stesso identico modo sotto Windows, ma la riga di comando che passa a Java verrà interpretata in modo errato.

In alternativa, David Korn (quello della shell Korn) ha messo insieme un package con il nome UWIN, che ha le stesse finalità della distribuzione Cygwin - è un ambiente di sviluppo Unix sotto Windows. UWIN contiene una versione della shell Korn. Versioni commerciali sono disponibili dalla Global Technologies, Ltd [URL30]. Inoltre, la AT&T consente lo scaricamento libero del pacchetto a fini di valutazione e accademici. Anche in questo caso, leggete attentamente la licenza prima dell'uso.

Infine, Tom Christiansen sta assemblando (nel momento in cui scriviamo) Perl Power Tools, un tentativo di implementare tutte le familiari utility Unix in modo portabile, in Perl [URL32].

Vedi anche

- *Automazione onnipresente*, Capitolo 8

Sfide

- Ci sono cose che attualmente fate manualmente in una GUI? Non vi capita mai di passare a colleghi istruzioni che comportano un gran numero di "fai clic su questo pulsante", "seleziona questo elemento" e così via? Potreste automatizzarle?
- Ogni volta che passate a un nuovo ambiente, imponetevi di scoprire quali shell sono disponibili. Vedete se potete portare con voi la shell che usate normalmente.

- Studiate le alternative alla shell che usate normalmente. Se vi capita un problema che la vostra shell non può affrontare, magari una shell alternativa potrebbe esservi di aiuto.

Power editing

Abbiamo parlato di strumenti come estensione della mano. Beh, questo vale per gli editor più di qualsiasi altro strumento software. Dovete poter manipolare testo con il minimo sforzo possibile, perché il testo è la materia prima fondamentale della programmazione. Vediamo caratteristiche e funzioni comuni che possono aiutarvi a ottenere il massimo dal vostro ambiente di editing.

Un editor

Pensiamo che sia meglio conoscere molto bene un editor, e usarlo per tutte le attività di editing: codice, documentazione, memo, amministrazione di sistema e così via. Se non avete un unico editor, rischiate una Babele moderna. Potreste dover usare l'editor incorporato nell'IDE di ciascun linguaggio per la codifica, e un prodotto da ufficio per la documentazione, e magari un diverso editor ancora per spedire la posta elettronica. Anche le combinazioni di tasti da usare per modificar le righe di comando nella shell possono essere diverse. È difficile essere bravi in uno qualsiasi di questi ambienti se ciascuno ha un diverso insieme di convenzioni e di comandi.

NOTA
Idealmente, la shell che usate dovrebbe avere combinazioni di tasti corrispondenti a quelle utilizzate dal vostro editor. Bash, per esempio, supporta sia le combinazioni di vi, sia quelle di emacs.

Dovete essere competenti. Saper semplicemente scrivere alla tastiera e usare un mouse per il taglia e incolla non basta. Non potete essere efficaci quanto potete esserlo con un editor potente ben conosciuto. Battere ← o Backspace dieci volte per spostare il cursore a sinistra all'inizio di una riga non è efficiente quanto battere un solo tasto come ^A, Home o 0.

SUGGERIMENTO 22
Usate un unico editor, usatelo bene.

Scegliete un editor, imparatelo bene e usatelo per tutte le attività di editing. Se usate un unico editor (o un solo insieme di combinazioni da tastiera) per tutte le attività di redazione di testi, non dovrete fermarvi a pensare per effettuare qualche manipolazione del testo: i tasti che servono diventeranno istintivi. L'editor diventerà un'estensione della vostra mano, i tasti canteranno mentre procedete senza intoppi. Questo è l'obiettivo.

Assicuratevi che l'editor che scegliete sia disponibili su tutte le piattaforme che utilizzate. Emacs, vi, CRISP, Brief e altri sono disponibili per più piattaforme, spesso in versione sia GUI sia a caratteri.

Caratteristiche degli editor

Al di là di eventuali caratteristiche che trovate particolarmente utili e comode, ecco alcune capacità di base che pensiamo qualsiasi editor decente dovrebbe avere. Se il vostro editor è carente in uno qualsiasi di questi ambiti, forse è venuto il momento di pensare di passare a uno più avanzato.

- *Configurabile*. Tutti gli aspetti dell'editor devono essere configurabili secondo le vostre preferenze: tipi di caratteri, colori, dimensioni delle finestre, attribuzioni di tasti (quali comandi sono controllati dalle possibili combinazioni di tasti). L'uso della sola tastiera per operazioni di editing comuni è più efficace che impartire comandi con il mouse o da menu, perché le mani non lasciano mai la tastiera.
- *Estendibile*. Un editor non deve diventare obsoleto solo perché arriva un nuovo linguaggio di programmazione. Deve potersi integrare con qualsiasi ambiente di compilazione usiate. Deve essere possibile "insegnargli" le sfumature di qualsiasi nuovo linguaggio o di qualsiasi nuovo formato testuale (XML, HTML versione 9 e così via).
- *Programmabile*. Deve essere possibile programmare l'editor per fargli svolgere attività complesse, costituite da più passaggi. La programmazione può avvenire mediante macro o con un linguaggio di scripting incorporato (Emacs usa una variante del Lisp, per esempio).

Molti editor inoltre hanno caratteristiche specifiche per particolari linguaggi, per esempio:

- evidenziazione della sintassi;
- autocompletamento;
- rientri automatici;
- codice iniziale o modelli di documenti;
- collegamenti a sistemi di help;
- funzioni simili a quelle di un IDE (compilazione, debug e simili).

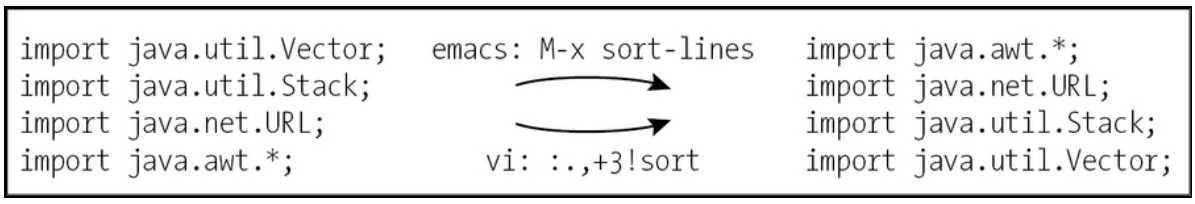


Figura 3.1 Ordinamento di righe in un editor.

L'evidenziazione della sintassi può sembrare un di più frivolo, ma in realtà può essere molto utile e migliorare la produttività. Quando ci si abitua a vedere le parole chiave in un colore o in un tipo di carattere diversi, una parola che non si presenta in quel modo perché non è stata scritta correttamente balza all'occhio molto prima che si faccia partire il compilatore.

Poter compilare e navigare direttamente agli errori nell'ambiente dell'editor è molto comodo nei grandi progetti. Emacs in particolare è adattissimo a questo stile di interazione.

Produttività

Abbiamo incontrato un numero sorprendente di persone che usano il Blocco note di Windows come utility per scrivere e modificare il codice sorgente. È un po' come usare un cucchiaino da caffè come se fosse un badile: battere alla tastiera e usare semplici operazioni di taglia e incolla con il mouse non è sufficiente.

Che tipo di cose dovrete fare che *non possono* essere fatte in questo modo?

Beh, tanto per cominciare, i movimenti del cursore. Singole combinazioni di tasti che vi fanno spostare per parole, per righe, per blocchi o per funzioni sono molto più efficienti che non premere tante volte un tasto che vi fa spostare di un carattere o di una riga alla volta.

Oppure, supponiamo che scriviate codice Java. Vi piace tenere gli enunciati `import` in ordine alfabetico, e qualcuno ha inserito un po' di file che non rispettano questo standard (può sembrare estremo, ma nel caso di un progetto di grandi dimensioni può far risparmiare molto tempo, quando si deve scorrere una lunga lista di enunciati `import`). Vorreste esaminare rapidamente una serie di file e ordinare alfabeticamente una piccola parte di essi. In editor come vi ed Emacs lo si può fare facilmente (Figura 3.1). Provateci in Blocco note.

Qualche editor può aiutare a rendere più fluide operazioni comuni. Per esempio, quando create un nuovo file in un particolare linguaggio, l'editor può fornirvi un modello, in cui possono essere inclusi:

- nome della classe o del modulo già compilati (ricavati dal nome del file);
- il vostro nome e/o le indicazioni di copyright;
- scheletri di costrutti in quel linguaggio (dichiarazioni di costruttori e distruttori, per esempio).

Un'altra caratteristica utile è il rientro automatico. Anziché dover rientrare manualmente (con la barra spaziatrice o con la tabulazione), l'editor rientra il testo automaticamente al momento opportuno (dopo l'apertura di una parentesi graffa, per esempio). L'aspetto piacevole di questa caratteristica è che potete usare l'editor per avere uno stile di rientri coerente per tutto il vostro progetto. Il kernel di Linux, per esempio, è sviluppato in questo modo. Gli sviluppatori sono dispersi in tutto il mondo, e molti lavorano sugli stessi pezzi di codice. Esiste un elenco pubblico di impostazioni (in questo caso per Emacs) che descrive lo stile di rientri richiesto.

E adesso?

Questo tipo di consigli è particolarmente difficile da dare, perché praticamente ciascun lettore avrà un diverso livello di agio e di esperienza con lo/gli editor che sta usando. Perciò, per riassumere e per dare qualche indicazione su dove andare adesso, se vi riconoscete in qualche affermazione della colonna di sinistra, vedete nella corrispondente posizione della colonna di destra quello che pensiamo dovreste fare.

Se vi riconoscete in questa affermazione...	Allora pensate a questo...
Uso solo le caratteristiche di base di molti editor diversi.	Scegliete un editor potente e imparate a usarlo bene.
Ho un editor preferito, ma non uso tutte le sue caratteristiche.	Imparatele. Riducete il numero dei tasti che dovete battere.
Ho un editor preferito e lo uso ogni volta che è possibile.	Provate a espanderlo e a usarlo per più attività di quelle per cui lo usate adesso.
Penso che siate matti. Blocco note è il miglior editor che sia mai stato fatto.	Se siete felici e produttivi, va bene così! Ma se vi capita di provare un po' di "invidia dell'editor" forse dovete rivalutare la vostra posizione.

Quali editor sono disponibili?

Dopo avervi raccomandato di imparare a usare bene un editor decente, quale consiglieremmo? Beh, questa volta ci tiriamo indietro; la scelta dell'editor è una scelta personale (qualcuno direbbe addirittura religiosa!). Però nell'Appendice A elenchiamo una serie di editor molto diffusi, con l'indicazione di dove trovarli.

Sfide

- Qualche editor usa un linguaggio vero e proprio per la personalizzazione e lo scripting. Emacs, per esempio, usa il Lisp. Fra i nuovi linguaggi che imparerete quest'anno, inserite il linguaggio usato dal vostro editor. Per tutte le cose che vi ritrovate a fare spesso, sviluppate una serie di macro (o qualcosa di equivalente) per gestirle.
- Sapete tutto quello che il vostro editor è in grado di fare? Provate a chiederlo a i vostri colleghi che usano lo stesso editor. Provate a svolgere qualsiasi attività di

editing utilizzando il minor numero possibile di tasti.

Controllo del codice sorgente

Il progresso, lungi dal consistere nel cambiamento, dipende dalla memoria. Chi non ricorda il passato è condannato a ripeterlo.
- George Santayana, *Life of Reason*

Una delle cose importanti che andiamo subito a cercare in un'interfaccia utente è il tasto *Undo* o *Annulla*, quel singolo pulsante che ci condona tutti gli errori. Meglio ancora se l'ambiente consente più livelli di annullamento e di ripetizione, in modo da poter tornare indietro e mettere rimedio a qualcosa che è successo qualche minuto fa. E se lo sbaglio fosse capitato la settimana scorsa, e nel frattempo avete spento e riacceso il computer decine di volte? Beh, questo è uno dei molti vantaggi di usare un sistema di controllo del codice: è un grande tasto Undo, una macchina del tempo a livello di progetto che può riportarvi ai bei tempi della settimana scorsa in cui il codice veniva compilato ed eseguito correttamente.

I sistemi di controllo del codice sorgente, o i sistemi di *gestione di configurazione*, di portata più generale, tengono traccia di ogni cambiamento nel codice sorgente e nella documentazione. I migliori possono tenere traccia anche delle versioni dei compilatori e dei sistemi operativi. Con un sistema di controllo del codice sorgente configurato opportunamente, *potete sempre tornare indietro a una versione precedente del vostro software*.

Ma un sistema di controllo del codice sorgente (SCCS, *source code control system*: usiamo la maiuscola per indicare un sistema generico, esiste un "sccs" in tutte minuscole, un sistema specifico rilasciato in origine con lo Unix System V di AT&T) fa molto più che rimediare agli errori. Un buon SCCS permette di tener traccia dei cambiamenti e di rispondere a domande del tipo: chi ha apportato cambiamenti a questa riga di codice? Qual è la differenza fra la versione corrente e quella della settimana scorsa? Quante righe di codice abbiamo cambiato in questa release? Quali file vengono modificati più spesso? Informazioni di questo

genere sono preziose per individuare i bug, per l'auditing, il controllo delle prestazioni e della qualità.

Un SCCS vi permetterà anche di identificare le release del vostro software. Una volta identificata, sarete sempre in grado di tornare indietro e rigenerare una release, indipendente dai cambiamenti che possono essere stati apportati in seguito.

Usiamo spesso un SCCS per gestire le diramazioni nell'albero dello sviluppo. Per esempio, una volta rilasciato del software, normalmente vorrete continuare a svilupparlo per la release successiva. Al contempo, dovrete risolvere i bachi della release corrente e inviare le versioni corrette ai clienti. Vorrete inserire tutte queste correzioni di bachi nella prossima release (se è il caso), ma non volete mandare ai clienti il codice in corso di sviluppo. Con un SCCS potete generare diramazioni dell'albero di sviluppo ogni volta che generate una release. Applicate le soluzioni per gli errori al codice in quel ramo, e continuate a sviluppare nel tronco principale. Poiché i rimedi degli errori possono essere rilevanti anche per il tronco principale, qualche sistema consente di fondere selettivamente i cambiamenti apportati nel ramo, in modo automatico, anche nel tronco principale.

I sistemi di controllo del codice sorgente possono conservare in un repository centrale i file che mantengono: un ottimo candidato per l'archiviazione.

Infine, qualche prodotto può consentire a due o più utenti di lavorare contemporaneamente sullo stesso gruppo di file, addirittura di apportare cambiamenti concorrenti allo stesso file. Il sistema poi gestisce la fusione di cambiamenti quando i file vengono rinviati al repository. Anche se apparentemente rischiosi, questi sistemi funzionano bene, nella pratica, per progetti di tutte le dimensioni.

SUGGERIMENTO 23
Usate sempre il controllo del codice sorgente.

Sempre. Anche se siete una squadra di una sola persona, impegnata in un progetto di una sola settimana. Anche se si tratta di un prototipo destinato a essere buttato. Anche se ciò a cui state lavorando non è codice sorgente. Assicuratevi che *tutto* sia sotto il controllo del codice sorgente: documentazione, elenchi di numeri telefonici, comunicazioni ai produttori, makefile, procedure di build e di rilascio, quel piccolo script per la shell che brucia il master del CD, proprio tutto. Usiamo

normalmente il controllo del codice sorgente su tutto quello che scriviamo (anche sul testo di questo libro). Anche se non lavoriamo a un progetto, il nostro lavoro quotidiano è al sicuro in un repository.

Controllo del codice sorgente e build

Vi è un grande beneficio nel mettere un intero progetto sotto l'ombrello di un sistema di controllo del codice sorgente: potete avere build del prodotto *automatiche* e *ripetibili*. Il meccanismo di build del progetto può estrarre l'ultimo sorgente dal repository in modo automatico. Può essere eseguito nel cuore della notte dopo che tutti (si spera) sono andati a casa. Potete eseguire test automatici di regressione per essere sicuri che ciò che è stato codificato quel giorno non abbia guastato alcunché.
L'automazione delle build garantisce la coerenza: non ci sono procedure manuali e non c'è bisogno che gli sviluppatori ricordino di copiare il codice in qualche speciale area per le build. La build è ripetibile perché potete sempre ricostruire il sorgente nello stato in cui era in una data ben precisa.

Ma il mio team non usa il controllo del codice sorgente

Male! Ci sembra una buona occasione per fare un po' di evangelizzazione! Però, mentre aspettate che gli altri vedano la luce, forse dovreste implementare il vostro controllo privato del sorgente. Usate uno qualsiasi degli strumenti liberamente disponibili elencati nell'Appendice A, e imponetevi di mantenere il vostro lavoro personale al sicuro in un repository (oltre a fare tutto quello che il progetto richiede). Può sembrare una duplicazione dello sforzo, ma possiamo garantirvi che vi risparmierà un sacco di fastidi (e farà risparmiare denaro al progetto), la prima volta che dovrete rispondere a domande come "Che cosa hai fatto al modulo xyz?" e "Che cosa ha fatto inceppare la build?". Questo metodo può anche contribuire a convincere i vostri capi che il controllo del codice sorgente funziona davvero. Non dimenticate che un SCCS si applica altrettanto bene a tutte le cose che fate al di fuori del lavoro.

Prodotti per il controllo del codice sorgente

Nell'Appendice A trovate gli indirizzi per recuperare sistemi di controllo del codice sorgente rappresentativi, alcuni commerciali e altri liberi. I prodotti disponibili sono molti di più - cercate indicazioni per le FAQ della gestione di configurazione. Per una introduzione al sistema di controllo delle versioni CVS, liberamente disponibile, potete vedere il nostro libro *Pragmatic Version Control* [TH03].

Vedi anche

- *Ortogonalità*, Capitolo 2
- *Il potere del puro testo*, in questo capitolo
- *È tutta scrittura*, Capitolo 8

Sfide

- Anche se non potete usare un SCCS al lavoro, installate RCS o CVS su un sistema personale. Usatelo per gestire i vostri progetti personali, i documenti che scrivete e (se possibile) le variazioni di configurazione applicate al sistema stesso.
- Date un'occhiata a qualcuno dei progetti open source per cui sono disponibili archivi accessibili pubblicamente sul Web (come Mozilla [URL51], KDE [URL54] e Gimp [URL55], Come ottenete aggiornamenti del sorgente? Come effettuate cambiamenti: il progetto regola l'accesso o arbitra l'inclusione dei cambiamenti?

Debug

È doloroso
Guardare ai propri guai e sapere
Che tu stesso e nessun altro li ha causati
- Sofocle, *Aiace*

La parola *bug* è stata usata per descrivere un "oggetto terrorizzante" sin dal XIV secolo. Si dice che sia stata il viceammiraglio Grace Hopper, inventrice del COBOL, la prima a osservare un *computer bug* - letteralmente, una farfalla che si era impigliata in un relè in uno dei primi sistemi di calcolo. Quando gli era stato chiesto di spiegare perché la

macchina non funzionava come doveva, un tecnico aveva riferito che c'era "un bug nel sistema" e scrupolosamente l'ha inserito, ben fissato con il nastro adesivo, ali e tutto, nel registro.

Purtroppo, abbiamo ancora "bug" nel sistema, anche se non del tipo con le ali. I difetti del software si manifestano in molti modi, da requisiti non compresi a errori di codifica. Purtroppo, i sistemi informatici moderni sono ancora limitati: fanno quello che *diciamo* loro di fare, non necessariamente quello che *vorremmo* facessero.

Nessuno scrive software perfetto, perciò è un dato che il debug richieda una parte cospicua della giornata. Vediamo alcuni dei problemi coinvolti nel debug e qualche strategia generale per trovare errori sfuggenti.

Psicologia del debug

Per molti sviluppatori il debug è in sé un tema delicato, carico di emotività. Anziché affrontarlo come un rompicapo da risolvere, troverete negazione, scaricabarile, scuse patetiche o pura e semplice apatia. Rendetevi conto che il debug è solo *risoluzione di problemi*, *problem solving*, e affrontatelo come tale.

Trovato un errore di qualcun altro, potete dedicare tempo ed energie a inveire e svergognare il colpevole. In qualche ambiente di lavoro fa parte della cultura e può essere anche catartico, ma nel campo tecnico dovete concentrarvi sul risolvere il *problema*, non sul trovare il colpevole.

SUGGERIMENTO 24
Risolvete il problema, non affibbiate la colpa.

Non importa realmente se l'errore sia colpa vostra o di qualcun altro: rimane comunque un problema vostro.

Una mentalità da debug

> Nessuno è più facile da ingannare di noi stessi.
> - Edward Bulwer-Lytton, *The Disowned*

Prima di iniziare il debug, è importante adottare la mentalità giusta. Dovete abbassare molte delle difese che usate ogni giorno per proteggere il vostro ego, escludere ogni pressione a cui potete essere sottoposti e mettervi comodi. Soprattutto, ricordate la prima regola del debug:

SUGGERIMENTO 25
Niente panico.

È facile farsi prendere dal panico, in particolare se si è a poca distanza da una scadenza, o se si ha un capo nervoso o un cliente che vi alita sul collo mentre cercate di trovare la causa dell'errore. Ma è molto importante fare un passo indietro e *pensare* veramente a che cosa può causare i sintomi che siete convinti indichino un "baco". Se la prima reazione quando vedete qualcosa che non va o vedete un *bug report* è "questo è impossibile", avete palesemente torto. Non sprecate un solo neurone a seguire il filo di pensieri che parte da "ma questo non può succedere", perché chiaramente *può* succedere, ed è successo.

Attenti alla miopia nel debug. Resistete all'impulso di mettere una pezza semplicemente ai sintomi che vedete: è più probabile che il guasto effettivo sia a distanza di parecchi passi da quello che state osservando, e può chiamare in causa un buon numero di altre cose con cui è in rapporto. Cercate sempre di scoprire la radice di un problema, non limitatevi a quella sua specifica occorrenza.

Da dove iniziare

Prima di *iniziare* a cercare l'errore, assicuratevi di lavorare su codice che sia stato compilato correttamente, senza generare warning. Di routine, noi fissiamo i livelli di avvertimento del compilatore il più alti possibile. Non ha senso perdere tempo a cercare un problema che il compilatore potrebbe trovare per voi. Dobbiamo concentrarci sui problemi più difficili.

Quando cercate di risolvere un problema, dovete raccogliere tutti i dati pertinenti. Purtroppo, il rilevamento degli errori non è una scienza esatta. È facile farsi fuorviare da coincidenze, e non potete permettervi di sprecare tempo nel debug di coincidenze. La prima cosa indispensabile è essere precisi nelle osservazioni.

La precisione nelle segnalazioni di bug è ulteriormente ridotta quando arrivano attraverso terzi, potreste davvero aver bisogno di *osservare* in azione l'utente che ha comunicato il bug, per avere un livello di dettaglio sufficiente.

Andy una volta ha lavorato a una grande applicazione di grafica. Vicino al rilascio, i tester hanno segnalato che l'applicazione andava in tilt ogni volta che dipingevano un tratto con un particolare pennello. Il programmatore responsabile ha risposto che non c'era nulla che non andasse: aveva provato a dipingere con quel pennello e funzionava perfettamente. Il dialogo è andato avanti per vari giorni, con il malumore in rapida crescita.

Alla fine, abbiamo messo entrambi nella stessa stanza. Il tester ha selezionato il pennello e dipinto un tratto dall'angolo superiore destro verso l'inferiore sinistro e l'applicazione è esplosa. "Oh", ha detto il programmatore sottovoce, che poi ha ammesso mestamente di aver provato solo disegnando tratti dalla sinistra in basso verso la destra in alto, il che non faceva emergere l'errore.

Ci sono due aspetti importanti in questa vicenda.

- Può essere necessario intervistare l'utente che ha segnalato l'errore per poter raccogliere più dati di quelli forniti inizialmente.
- Test artificiali (come il singolo tratto disegnato dal programmatore dal basso verso l'alto) non mettono abbastanza alla prova un'applicazione. Dovete sottoporre a test brutalmente sia condizioni limite, sia comportamenti realistici degli utenti finali. Dovete farlo sistematicamente (vedi *Test senza pietà* nel Capitolo 8).

Strategie di debug

Una volta che pensate di sapere che cosa succede, è il momento di scoprire che cosa *il programma* pensa stia succedendo.

Riproduzione degli errori

No, i nostri errori non si moltiplicano realmente (anche se qualcuno probabilmente è abbastanza vecchio da poterlo fare legalmente). Parliamo di un altro genere di riproduzione.

Il modo migliore per cominciare a risolvere un errore è renderlo riproducibile. In fin dei conti, se non riuscite a riprodurlo, come fare a sapere se l'avete risolto?

Vogliamo qualcosa di più della possibilità di riprodurre un errore seguendo una lunga serie di passi; vogliamo un errore che possa essere riprodotto con un *unico comando*. È molto più difficile sistemare un errore se dovete compiere sempre 15 passaggi per arrivare al punto in cui viene a galla. A volte, costringendovi a isolare le circostanze in cui l'errore si verifica, potrete addirittura avere un'idea di come sistemarlo.

Vedete *Automazione onnipresente*, nel Capitolo 8, per altre idee analoghe.

Visualizzate i vostri dati

Spesso, il modo più semplice per capire che cosa fa un programma (o che cosa farà) è dare una bella occhiata ai dati su cui agisce. L'esempio più semplice è un approccio diretto "nome di variabile = valore di dati", che può essere realizzato sotto forma di testo stampato o di campi in una finestra di dialogo GUI o in una lista.

Potete entrare molto più in profondità nei vostri dati utilizzando un debugger che vi consenta di *visualizzare* i dati e tutte le interrelazioni esistenti. Ci sono debugger che possono rappresentare i vostri dati come un paesaggio 3D in realtà virtuale visto a volo d'uccello, o come un grafico 3D a forma d'onda, o semplicemente come un diagramma strutturale, come si vede nella Figura 3.2. Avanzando passo per passo nel programma, immagini come questa possono valere molto più di mille parole, perché gli errori di cui stavate andando a caccia balzano di colpo all'occhio.

Anche se il debugger ha un supporto limitato per la visualizzazione di dati, potete comunque farlo voi - o a mano, con carta e matita, o con programmi esterni per i grafici.

Il debugger DDD ha alcune capacità di visualizzazione, ed è libero ([URL19]. È interessante notare che DDD funziona con molti linguaggi, fra cui C, C++, Fortran, Java, Modula, Pascal, Perl e Python (chiaramente ha un progetto ortogonale).

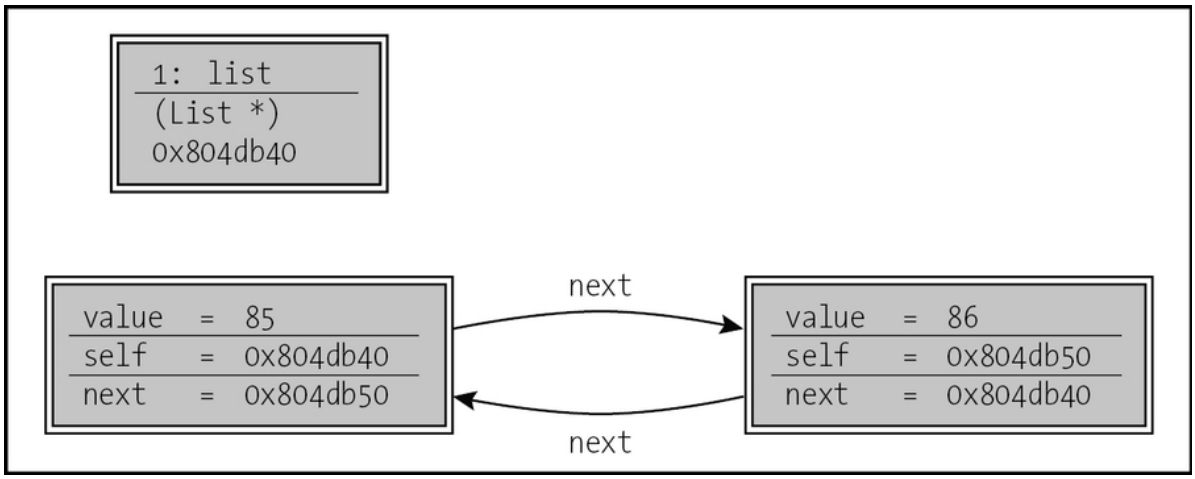


Figura 3.2 Esempio di diagramma da debugger di una lista concatenata circolare. Le frecce rappresentano puntatori a nodi.

Tracce

I debugger in genere si concentrano sullo stato del programma *al momento*. A volte avete bisogno di qualcosa di più, dovete osservare lo stato di un programma o di una struttura di dati nel tempo. Vedere una traccia di stack può dirvi solo come siete arrivati qui direttamente, non può dirvi che cosa stavate facendo prima di questa catena di chiamate, in particolare in sistemi basati su eventi.

Gli *enunciati di traccia* sono quei piccoli messaggi diagnostici che fate comparire sullo schermo o inviate a un file e che dicono cose come "arrivato qui" e "valore di x = 2". È una tecnica primitiva, rispetto ai debugger in stile IDE, ma è particolarmente efficace nella diagnosi di varie classi di errori che i debugger non possono diagnosticare. Il *tracing* è preziosissimo in qualsiasi sistema in cui il tempo sia a sua volta un fattore: processi concorrenti, sistemi in tempo reale, applicazioni basate su eventi.

Potete usare enunciati di traccia per "scavare a fondo" nel codice. Potete aggiungere enunciati di traccia, cioè, scendendo lungo l'albero delle chiamate.

I messaggi di traccia devono avere un formato regolare, coerente: potreste volerli analizzare automaticamente. Per esempio, se dovete rintracciare una perdita di risorse (per esempio il mancato bilanciamento di aperture e chiusure di file) potreste tracciare in un file di log ogni `open` e ogni `close`. Elaborando il file di log con Perl, potreste identificare chiaramente dove si trova l'`open` colpevole.

Variabili corrotte? Controllate nei dintorni

A volte esaminerete una variabile, aspettandovi di vedere un valore intero piccolo e invece vi ritrovate qualcosa come 0x6e69614d. Prima di tirarvi su le maniche per fare sul serio, date rapidamente un'occhiata alla memoria intorno alla variabile corrotta. Spesso ne ricaverete un indizio. Nel nostro caso, l'esame della memoria circostante sotto forma di caratteri ci dice

```
20333231 6e69614d 2c745320 746f4e0a
 1 2 3    M a i n    S t ,  \n N o t
2c6e776f 2058580a 31323433 00000a33
o w n ,   \n X X     3 4 2 1     3\n\0\0
```

Sembra che qualcuno abbia dipinto un indirizzo con lo spray sopra il nostro contatore. Ora sappiamo dove andare a guardare.

Fare la papera di gomma

Una tecnica molto semplice ma particolarmente utile per trovare la causa di un problema è semplicemente spiegare il problema a qualcun altro. L'altra persona deve mettersi alle vostre spalle e guardare lo schermo, fare costantemente cenno con la testa (come una paperella di gomma che va su e giù nella vasca da bagno). Non c'è bisogno che quella persona dica una parola: il semplice atto di spiegare, passo per passo, che cosa dovrebbe fare il codice spesso spinge il problema a saltar fuori da solo dallo schermo.

NOTA

Perché "fare la papera di gomma"? (In inglese, *rubber ducking* è ancora più divertente.) Quando era studente all'Imperial College di Londra, Dave ha lavorato molto con un assistente di ricerca che si chiamava Greg Pugh, uno dei migliori sviluppatori che Dave abbia mai conosciuto. Da mesi Greg portava con sé una piccola papera di gomma gialla, che collocava sul suo terminale mentre codificava. C'è voluto un bel po' prima che Dave trovasse il coraggio di chiedere…

Sembra semplice, ma nello spiegare il problema a un'altra persona dovete dire esplicitamente cose che potreste dare per scontate quando esaminate il codice da soli. Dovendo esprimere verbalmente qualcuna di quelle ipotesi, potreste improvvisamente vedere il problema in una nuova luce.

Processo di eliminazione

Nella maggior parte dei progetti, il codice che dovete controllare può essere una miscela di codice applicativo scritto da voi e da altri del vostro team, prodotti di terze parti (database, connettività, librerie grafiche, comunicazioni o algoritmi specializzati e così via) e piattaforma ambiente (sistema operativo, librerie di sistema e compilatori).

È possibile che ci sia un bug nel sistema operativo, nel compilatore o in un prodotto di terzi, ma non deve essere la prima cosa a cui pensare. È molto più probabile che l'errore stia nel codice applicativo in via di sviluppo. In generale è molto più remunerativo ipotizzare che il codice applicativo chiami in modo sbagliato una libreria che non assumere che sia la libreria stessa ad avere problemi. Anche se il problema *sta*

effettivamente in un prodotto di terzi, dovrete comunque eliminare la possibilità che la colpa sia del vostro codice, prima di inviare la segnalazione di errore.

Abbiamo lavorato a un progetto in cui un tecnico esperto era convinto che la chiamata di sistema `select` in Solaris fosse rovinata. Non c'era abilità persuasiva né logica in grado di fargli cambiare idea (il fatto che ogni altra applicazione di rete sulla macchina funzionasse perfettamente era irrilevante). Ha passato settimane a scrivere codice per aggirarla che, per qualche strana ragione, sembrava non risolvessero il problema. Quando finalmente l'abbiamo costretto a sedersi e a leggere la documentazione su `select`, ha scoperto il problema e l'ha corretto nel giro di pochi minuti. Adesso usiamo l'espressione "select si è rotto" come monito ironico, ogni volta che qualcuno di noi comincia a dare al sistema la colpa di un guasto che con tutta probabilità è frutto di un nostro errore.

SUGGERIMENTO 26
"select" non è rotto.

Ricordate: se vedete impronte di zoccoli, pensate a cavalli, non a zebre. Il sistema operativo probabilmente non ha problemi e anche il database probabilmente funziona benissimo.

Se avete "cambiato solo una cosa" e il sistema ha smesso di funzionare, quell'unica cosa è probabilmente la responsabile, direttamente o indirettamente, non importa quanto possa sembrare implausibile la spiegazione. A volte la cosa che è cambiata è al di fuori del vostro controllo: nuove versioni del sistema operativo, del compilatore, del database o di altro software di terze parti possono causare disastri in codice che in precedenza era corretto. Possono presentarsi nuovi bug; bug che avevate trovato il modo di aggirare sono stati sistemati e il vostro metodo per aggirarli è andato in tilt. Le API cambiano, cambiano le funzionalità; per farla breve, sono cambiate le regole del gioco e dovete sottoporre a test il sistema sotto le nuove condizioni. Perciò tenete d'occhio le vostre scadenze quando prendete in considerazione la possibilità di un aggiornamento: magari è meglio aspettare *dopo* il prossimo rilascio del vostro software.

Se però non c'è un posto ovvio in cui iniziare a cercare, potete sempre ricorrere a una buona vecchia ricerca dicotomica. Vedete se i sintomi sono presenti in due punti ben distanti del codice. Poi guardate nel mezzo. Se il problema è presente, allora l'errore sta fra il punto iniziale

quello mediano; altrimenti, sta fra il punto mediano e la fine. Potete continuare in questo modo finché non restringete il campo abbastanza per identificare il problema.

L'elemento sorpresa

Quando rimanete sorpresi da un errore (e magari mormorate "questo è impossibile", sottovoce, in modo da non farvi sentire da noi), dovete ripensare a verità che vi sono care. In quella routine a lista concatenata - quella che sapevate essere a prova di bomba e non potrebbe essere la causa di questo errore - avete verificato *tutte* le condizioni al contorno? Quell'altro pezzo di codice che usate da anni - non è possibile che abbia ancora dentro un baco. O no?

Ovviamente è possibile. L'entità della sorpresa che provate quando qualcosa va storto è direttamente proporzionale al grado di fiducia e di fede che avete riposto nel codice in esecuzione. Per questo, di fronte a un guasto "sorprendente", dovete rendervi conto che una o più delle vostre ipotesi di fondo potrebbero essere sbagliate. Non trascurate una routine o un pezzo di codice coinvolti nell'errore perché "sapete" che funzionano. Dimostratelo. Dimostratelo in *questo* contesto, con *questi* dati, con *queste* condizioni al contorno.

SUGGERIMENTO 27
Non datelo per scontato, dimostratelo.

Quando vi imbattete in un errore sorprendente, al di là di sistemarlo, dovete capire perché non è stato catturato prima. Riflettete se dovete modificare gli unit test o altri tipi di test, in modo che avrebbero identificato quell'errore.

Inoltre, se l'errore è il risultato di dati sbagliati che si sono propagati per un paio di livelli prima di mandare in tilt il tutto, vedete se una migliore verifica dei parametri in quelle routine l'avrebbe isolato prima. (Vedete le parti sui crash precoci e le asserzioni nel Capitolo 4.)

Intanto che ci siamo, esistono altri punti del codice che potrebbero essere suscettibili allo stesso errore? È il momento di scoprirlo e di sistemarli. Q*ualsiasi cosa* sia accaduta, dovete assicurarvi di venirlo a sapere, se succedesse di nuovo.

Se è stato necessario molto tempo per sistemare questo errore, chiedetevi perché. C'è qualcosa che potreste fare per rendere più facile

risolvere questo errore la prossima volta? Magari potreste inserire agganci migliori per i test, oppure scrivere un analizzatore di log file.

Infine, se l'errore è conseguenza di qualche ipotesi sbagliata di qualcun altro, discutete il problema con tutto il team: se una persona ha capito male, può darsi che non sia l'unica.

Fate tutte queste cose e, si spera, la prossima volta non resterete a bocca aperta.

Lista di controllo per il debug

- Il problema segnalato è un risultato diretto dell'errore sottostante o è solo un sintomo?
- Il baco è davvero nel compilatore? O è nel sistema operativo? O è nel vostro codice?
- Se doveste spiegare dettagliatamente il problema a un collega, che cosa gli direste?
- Se il codice sospetto passa i suoi unit test, i test sono sufficientemente completi? Che cosa succede se eseguite il test con questi dati?
- Le condizioni che hanno causato questo errore esistono anche da qualche altra parte nel sistema?

Vedi anche

- *Programmazione assertiva*, Capitolo 4
- *Programmazione per coincidenza*, Capitolo 6
- *Automazione onnipresente*, Capitolo 8
- *Test senza pietà*, Capitolo 8

Sfide

- Il debug è già una sfida sufficiente.

Manipolazione del testo

I programmatori pragmatici manipolano il testo come l'ebanista lavora il legno. Nelle pagine precedenti abbiamo analizzato alcuni strumenti specifici - shell, editor, debugger - che usiamo. Sono simili agli utensili di chi lavora il legno, utensili specializzati che fanno solo una o due cose, ma le fanno molto bene. Tuttavia, ogni tanto dobbiamo compiere qualche trasformazione che non è facilmente gestita dallo strumentario di base. Ci serve uno strumento di manipolazione del testo di tipo "generalista".

I linguaggi per la manipolazione del testo stanno alla programmazione come le fresatrici alla lavorazione del legno. Sono rumorose, fanno confusione e usano tanta forza bruta. Fate un errore con questi attrezzi, e pezzi interi saranno rovinati. Qualcuno giura che nel suo strumentario non entreranno mai. Nelle mani giuste, però, sia fresatrici che linguaggi di manipolazione del testo possono essere di potenza e versatilità incredibili. Potete rapidamente ritagliare qualcosa in una forma specifica, fare incastri e intagliare. Usati bene, questi strumenti hanno una finezza sorprendente, ma padroneggiarli richiede tempo.

Il numero dei buoni linguaggi di manipolazione del testo è in crescita. Gli sviluppatori Unix spesso amano usare la potenza delle loro shell di comando, affiancata da strumenti come `awk` e `sed`. Chi preferisce uno strumento più strutturato apprezza la natura orientata agli oggetti di Python [URL9]. Qualcuno usa Tcl [URL23] come attrezzo d'elezione. Noi preferiamo Ruby [TFH04] e Perl [URL8] per i nostri brevi script.

Questi linguaggi sono tecnologie abilitanti importanti. Usandoli, potete rapidamente mettere insieme utility e prototipare idee - attività che potrebbero richiede tempi cinque o dieci volte più lunghi con i linguaggi convenzionali. E quel fattore moltiplicativo è di importanza cruciale per il tipo di sperimentazione che facciamo. Dedicare 30 minuti a esplorare un'idea folle è molto meglio che perderci cinque ore. Passare una giornata ad automatizzare componenti importanti di un progetto è accettabile; perderci una settimana potrebbe non esserlo. Nel loro *The Practice of Programming* [KP99], Kernighan e Pike costruiscono lo stesso programma in cinque linguaggi diversi. La versione in Perl è la più breve (17 righe, contro le 150 del C). Con Perl, potete manipolare testo, interagire con i programmi, parlare sulle reti, pilotare pagine web, fare

calcoli aritmetici con precisione arbitraria e scrivere programmi che ricordano imprecazioni di Snoopy.

SUGGERIMENTO 28
Imparate un linguaggio di manipolazione del testo.

Per mostrarvi l'ampiezza di applicazione dei linguaggi di manipolazione del testo, ecco un campione di applicazioni che abbiamo sviluppato negli ultimi anni.

- *Manutenzione di schema di database*. Un insieme di script Perl prendeva un semplice file di puro testo contenente la definizione dello schema di un database e da questa generava:
- gli enunciati SQL per creare il database;
- file di dati piatti per popolare un dizionario di dati;
- librerie di codice C per accedere al database;
- script per verificare l'integrità del database;
- pagine web contenenti descrizioni e diagrammi dello schema;
- una versione XML dello schema.
- *Accesso a proprietà Java*. È buono stile di programmazione a oggetti limitare l'accesso alle proprietà di un oggetto, costringendo le classi esterne ad accedervi solo attraverso metodi `get` e `set`. Tuttavia, nel caso comune in cui una proprietà è rappresentata in una classe mediante una semplice variabile membro, creare un metodo `get` e un metodo `set` per ciascuna variabile è noioso e meccanico. Abbiamo uno script Perl che modifica i file sorgente e inserisce le definizioni corrette dei metodi per tutte le variabili opportunamente contrassegnate.
- *Generazione di dati di test*. Avevamo decine di migliaia di record di test, in vari file e formati, che dovevano essere riuniti e convertiti in una forma adatta al caricamento in un database relazionale. Perl l'ha fatto in un paio d'ore (e nel farlo ha trovato un paio di errori di coerenza nei dati originali).
- *Scrittura di libri*. Pensiamo sia importante che il codice presentato in un libro debba essere prima collaudato. La maggior parte del codice in questo libro lo è stato. Tuttavia, in base al principio DRY (vedi *I mali della duplicazione*, nel Capitolo 2) non volevamo copiare e incollare le righe di codice dai programmi collaudi al libro. Avrebbe voluto dire duplicare il codice, praticamente con la sicurezza che ci

saremmo scordati di aggiornare un esempio quando fosse cambiato il programma corrispondente. Per qualche esempio, poi, non volevamo annoiarvi con tutto il codice di contorno necessario per compilare ed eseguire l'esempio. Abbiamo fatto ricorso a Perl. Quando formattiamo il libro, chiamiamo uno script relativamente semplice, che estrae da un file sorgente un segmento dotato di nome, evidenzia la sintassi e converte il risultato nel linguaggio tipografico che usiamo.

- *Interfaccia da C a Object Pascal.* Un cliente aveva un team di sviluppatori che scriveva Object Pascal su PC. Il codice doveva interfacciarsi con un altro corpus di codice scritto in C. Abbiamo sviluppato un breve script Perl che analizzava i file di intestazione C, e ne estraeva le definizioni di tutte le funzioni esportate e le strutture di dati utilizzate. Poi generavamo unità Object Pascal con record Pascal per tutte le strutture C e importavamo le definizioni di procedura per tutte le funzioni C. Questo processo di generazione è entrato a far parte della build, così che, ogni volta che l'intestazione C cambiava, venisse costruita automaticamente una nuova unità Object Pascal.
- *Generazione di documentazione per il Web.* Moti team di progetto pubblicano la loro documentazione in siti web interni. Abbiamo scritto molti programmi Perl che analizzano schemi di database, file sorgente C o C++, makefile e altri sorgenti di progetto per produrre la necessaria documentazione HTML. Usiamo Perl per confezionare i documenti con intestazioni e piè di pagina standard e per trasferirli sul sito web.

Usiamo linguaggi di manipolazione del testo quasi ogni giorno. Molte delle idee di questo libro si possono implementare con maggiore semplicità in questi che in qualsiasi altro linguaggio che ci sia noto. Questi linguaggi rendono facile anche scrivere generatori di codice, il nostro prossimo tema.

Vedi anche

- *I mali della duplicazione,* Capitolo 2

Esercizi

11. Il vostro programma in C usa un tipo enumerato per rappresentare uno stato fra 100 possibili. Vorreste poter stampare lo stato come stringa (anziché come numero) a fini di debug. Scrivete uno script che legga dall'input standard un file contenente

```
name
state_a
state_b
 :    :
```

Fornite il file `name.h`, che contiene

```
extern const char* NAME_names[];
typedef enum {
   state_a,
   state_b,
    : :
 } NAME;
```

e il file `name.c`, che contiene

```
const char* NAME_names[] = {
   "state_a",
   "state_b",
    :    :
 };
```

12. A metà della scrittura di questo libro, ci siamo resi conto che non avevamo inserito la direttiva `use strict` in molti dei nostri esempi in Perl. Preparate uno script che esamini tutti i file `.pl` in una directory e aggiunga una `use strict` alla fine del blocco iniziale di commenti a tutti i file che già non la contengono. Ricordare di tenere una copia di riserva di tutti i file che modificate.

Generatori di codice

Quando chi lavora il legno ha il compito di produrre la stessa cosa molte volte, bara: si costruisce una forma o un modello. Se riesce a preparare la forma nel modo giusto una volta, poi può riprodurre l'oggetto molte volte. La forma elimina complessità e riduce la probabilità di errori, lasciando libero l'artigiano di concentrarsi sulla qualità.

Da programmatori, ci troviamo spesso in una posizione simile. Dobbiamo ottenere la stessa funzionalità, ma in contesti diversi. Dobbiamo ripetere informazioni in posti diversi. A volte dobbiamo semplicemente proteggerci dalla sindrome del tunnel carpale riducendo il numero dei movimenti delle dita sulla tastiera.

Come l'artigiano investe tempo nel creare una forma, un programmatore può costruire un generatore di codice. Una volta che l'ha

realizzato, può usarlo per tutta la durata del progetto praticamente senza costi.

SUGGERIMENTO 29
Scrivete codice che scrive codice.

Esistono due tipi di generatori di codice.

1. I *generatori di codice passivi* vengono eseguiti una volta per produrre un risultato. Da quel momento in poi, il risultato è autonomo - divorzia dal codice generatore. I maghi di cui parliamo in *Maghi cattivi*, nel Capitolo 8, e alcuni strumenti CASE, sono esempi di generatori di codice passivi.
2. I *generatori di codice attivi* vengono usati ogni volta che sono richiesti i loro risultati. Il risultato è a perdere - può sempre essere riprodotto dal generatore. Spesso i generatori attivi leggono una forma di script o di file di controllo per produrre i loro risultati.

Generatori di codice passivi

I generatori passivi fanno risparmiare scrittura alla tastiera. Fondamentalmente sono modelli parametrizzati, che generano un output dato a partire da un insieme di input. Una volta prodotto, il risultato è un file sorgente a pieno diritto nel progetto: verrà modificato, compilato e affidato al controllo del sorgente come qualsiasi altro file. Le sue origini saranno dimenticate. I generatori di codice passivi hanno molti usi:

- *Creare nuovi file sorgente*. Un generatore passivo può produrre modelli, direttive di controllo del codice sorgente, dichiarazioni di copyright e blocchi di commento standard per ogni nuovo file di un progetto. I nostri editor sono impostati in modo da farlo ogni volta che creiamo un nuovo file: creiamo un nuovo programma Java e il nuovo buffer dell'editor conterrà automaticamente un blocco di commento, una direttiva di pacchetto e lo scheletro di una dichiarazione di classe, tutto già compilato.
- *Eseguire conversioni una tantum fra linguaggi di programmazione*. Abbiamo iniziato a scrivere questo libro usando il sistema troff, ma

poi siamo passati a LATEX dopo che erano state completate 15 sezioni. Abbiamo scritto un generatore di codice che leggeva il sorgente troff e lo convertiva in LATEX. Era preciso al 90 per cento; il resto l'abbiamo fatto a mano. È una caratteristica interessante dei generatori di codice passivi: non devono essere per forza totalmente accurati. Bisogna scegliere quanto impegno mettere nel generatore, in rapporto all'energia che si deve spendere per sistemarne l'output.

- *Produrre tavole di consultazione e altre risorse* che sono costose da calcolare in fase di esecuzione. Invece di calcolare le funzioni trigonometriche, molti dei primi sistemi di grafica usavano tavole precalcolate dei valori di seno e coseno. Normalmente quelle tavole erano prodotte da un generatore passivo e poi copiate nel sorgente.

Generatori di codice attivi

Mentre quelli passivi sono semplicemente una comodità, i generatori di codice attivi sono una necessità, se si vuole seguire il principio DRY. Con un generatore attivo, si può prendere una singola rappresentazione di qualche pezzo di conoscenza e convertirla in tutte le forme di cui l'applicazione ha bisogno. Questa *non* è duplicazione perché le forme derivate sono a perdere, e vengono generate secondo necessità dal generatore (per questo sono generatori "attivi").

Ogni volta che vi trovate a cercare di far lavorare insieme due ambienti diversi, prendete in considerazione la possibilità di usare un generatore di codice attivo.

Magari dovete sviluppare un'applicazione di database. Qui avete due ambienti: il database e il linguaggio di programmazione che usate per accedervi. Avete uno schema, e dovete definire strutture di basso livello che rispecchiano la struttura di certe tabelle del database. Potreste semplicemente codificarle direttamente, ma questo viola il principio DRY: la conoscenza dello schema sarebbe espressa in due luoghi. Quando lo schema cambia, dovete ricordare di cambiare il codice corrispondente. Se si elimina una colonna da una tabella, ma la base di codice non cambia, magari non avrete nemmeno un errore di compilazione. Lo scoprirete solo quando i test cominciano a fallire (o quando gli utenti cominciano a telefonare).

Un'alternativa è usare un generatore di codice attivo: prende lo schema e lo usa per generare il codice sorgente delle strutture, come nella Figura 3.3. Ora, ogni volta che lo schema cambia, cambia automaticamente anche il codice per accedervi. Se una colonna viene eliminata, il campo corrispondente nella struttura scomparirà e qualsiasi codice di livello più alto che usa quella colonna non verrà compilato. Avete catturato l'errore al momento della compilazione, non in produzione. Ovviamente, questo procedimento funziona solo se create la parte di generazione del codice del processo stesso di build.

NOTA

Come procedete a costruire il codice da uno schema di database? Esistono vari modi. Se lo schema è conservato in un file piatto (per esempio, sotto forma di enunciati `create table`), uno script relativamente semplice può analizzarlo e generare il sorgente. In alternativa, se usate uno strumento per creare lo schema direttamente nel database, dovreste poter estrarre le informazioni di cui avete bisogno direttamente da dizionario dei dati del database. Perl mette a disposizione librerie che danno accesso alla maggior parte dei database principali.

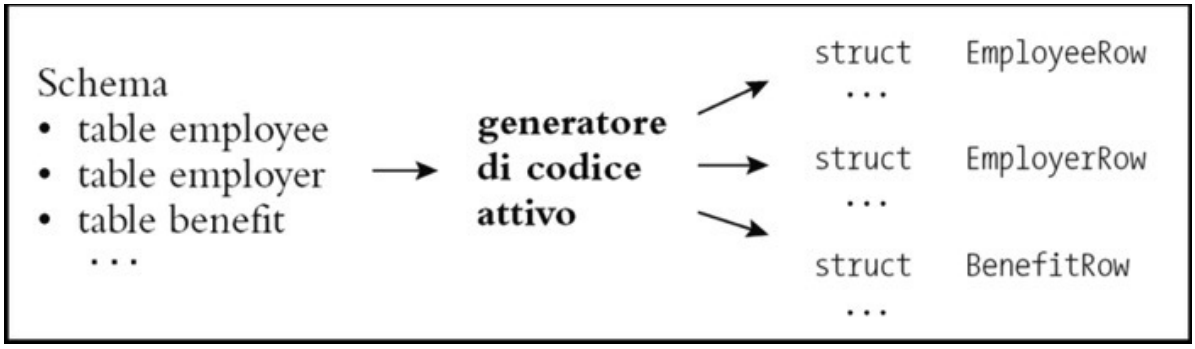


Figura 3.3 Un generatore di codice attivo genera codice da uno schema di database.

Un altro esempio di ambienti in cui l'uso di generatori di codice è particolarmente indicato sono quelli in cui si usano, nella stessa applicazione, linguaggi di programmazione diversi. Per comunicare, tutte le basi di codice devono avere alcune informazioni in comune: strutture di dati, formati di messaggio e nomi dei campi, per esempio. Anziché duplicare queste informazioni, usate un generatore di codice. A volte potete analizzare sintatticamente le informazioni nei file sorgente di un linguaggio e usarle per generare codice nell'altro. Spesso, però, è più agevole esprimerle in una rappresentazione più semplice, indipendente dal linguaggio, e generare il codice per entrambi i linguaggi, come nella Figura 3.4. Vedete anche la risposta all'Esercizio 13 (nell'Appendice B)

per un esempio di come separare l'analisi sintattica della rappresentazione in file piatto dalla generazione del codice.

I generatori di codice non devono essere per forza complessi

Tutto questo parlare di *attivo* e *passivo* vi avrà forse lasciato l'impressione che i generatori di codice siano bestie complicate, ma non devono esserlo per forza. Normalmente la parte più complessa è il parser, cioè l'analizzatore sintattico che analizza il file di input. Mantenete semplice il formato dell'input, e il generatore di codice diventa semplice. Guardate la risposta all'Esercizio 13 (Appendice B): il generatore effettivo è fatto fondamentalmente di enunciati `print`.

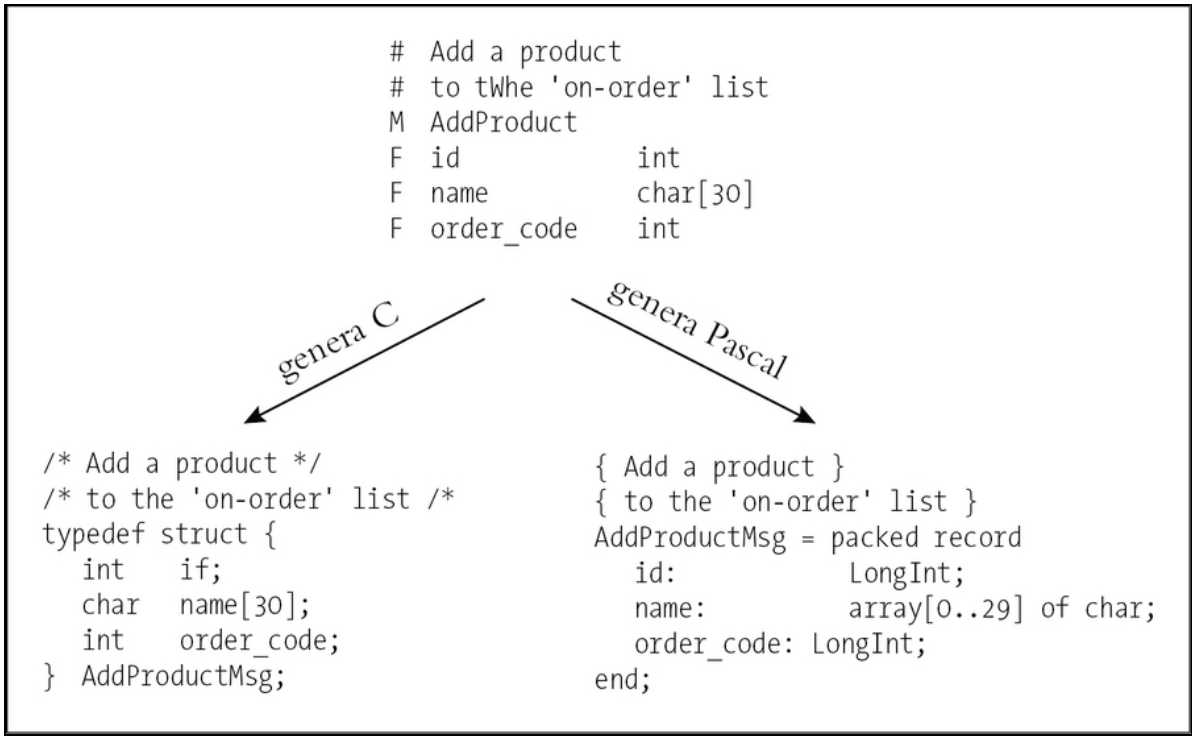


Figura 3.4 Generazione di codice a partire da una rappresentazione indipendente dal linguaggio. Nel file di input, le righe che iniziano con una M indicano l'inizio di una definizione di messaggio, le righe che iniziano con F definiscono campi (field), la E indica la fine del messaggio.

I generatori di codice non devono per forza generare codice

Molti esempi di questa sezione mostrano generatori di codice che producono sorgente di programma, ma non è sempre così. Potete usare i generatori di codice per scrivere quasi qualsiasi tipo di output: HTML, XML, puro testo, qualsiasi testo che possa diventare un input da qualche altra parte nel vostro progetto.

Vedi anche

- *I mali della duplicazione*, Capitolo 2
- *Il potere del puro testo*, in questo Capitolo
- *Maghi cattivi*, Capitolo 6
- *Automazione onnipresente*, Capitolo 8

Esercizi

13. Scrivete un generatore di codice che prenda il file di input della Figura 3.4 e generi output in due linguaggi di vostra scelta. Cercate di fare in modo che sia facile aggiungere altri linguaggi.

Paranoia pragmatica

SUGGERIMENTO 30
Non potete scrivere software perfetto.

Vi siete offesi? Non dovreste. Prendetelo come un assioma della vita. Adottatelo, festeggiatelo. Perché il software perfetto non esiste. Nessuno nella breve storia dell'informatica ha mai scritto un pezzo di software perfetto ed è improbabile che i primi a riuscirci siate proprio voi. Se non lo accettate come un fatto, finirete per sprecare tempo ed energie nell'inseguire un sogno impossibile.

Data quindi questa realtà deprimente, in che modo un programmatore pragmatico la trasforma in un vantaggio? È il tema di questo capitolo.

Tutti sappiamo di essere gli automobilisti migliori sulla Terra. Tutti gli altri al mondo sono lì solo per farci infuriare: non si fermano agli stop, fanno la gimcana fra una corsia e l'altra, non mettono le frecce quando svoltano, parlano al telefono o leggono il giornale mentre guidano e in generale non soddisfano i nostri standard. Perciò al volante stiamo sulla difensiva: stiamo attenti ai guai prima che si verifichino, prevediamo l'imprevedibile e non ci mettiamo mai in una condizione da cui non possiamo cavarcela.

L'analogia con la codifica è ovvia. Ci interfacciamo costantemente con il codice di altri, codice che potrebbe non essere all'altezza dei nostri elevatissimi standard, e abbiamo a che fare con input che magari sono validi e magari no. Perciò ci viene insegnato a codificare sulla difensiva. Se solo c'è un dubbio, verifichiamo tutte le informazioni che ci vengono passate. Usiamo asserzioni per identificare dati sbagliati. Facciamo controlli di coerenza, poniamo vincoli alle colonne dei database e in generale siamo molto soddisfatti di noi stessi.

I programmatori pragmatici vanno anche oltre: *non si fidano nemmeno di loro stessi*. Sapendo che nessuno, nemmeno loro, scrive codice perfetto, i programmatori pragmatici inseriscono nel codice difese contro i loro stessi errori. Descriviamo la prima misura difensiva in *Progettare per contratto*: clienti e fornitori devono essere d'accordo su diritti e responsabilità.

In *I progammi morti non mentono*, vogliamo assicurarci di non fare danni mentre cerchiamo di sradicare gli errori. Perciò cerchiamo di controllare le cose spesso e di terminare il programma se qualcosa va storto.

Programmazione assertiva descrive un metodo facile per controllare strada facendo: scrivere codice che verifica attivamente le nostre assunzioni.

Le eccezioni, come ogni altra tecnica, possono fare più male che bene, se non sono usate nel modo giusto. Ne parleremo in *Quando usare le eccezioni*.

Se i vostri programmi diventano più dinamici, vi troverete a fare i giocolieri con le risorse di sistema - memoria, file, dispositivi e simili. In *Come equilibrare le risorse*, suggeriamo qualche metodo per essere sicuri che il giocoliere non faccia mai cadere qualche pallina.

In un mondo di sistemi imperfetti, tempi ridicoli, strumenti che fanno ridere e pretese impossibili, vediamo di andare sul sicuro.

> Quando tutti sono realmente in giro a cercarti, la paranoia è solo pensar bene.
> - Woody Allen

Progettare per contratto

> Nulla mi stupisce quanto il buon senso e gli affari semplici.
> - Ralph Waldo Emerson, *Essays*

Avere a che fare con i sistemi informatici è dura, ma avere a che fare con le persone è ancora più dura. Come specie, però, abbiamo avuto più tempo per dipanare i problemi delle interazioni umane. Qualcuna delle soluzioni che abbiamo escogitato negli ultimi millenni si può applicare anche alla scrittura di software, e una delle migliori per garantire la semplicità degli affari è il *contratto*.

Un contratto definisce i vostri diritti e le vostre responsabilità, e anche quelli dell'altra parte. Inoltre, comprende anche un accordo sulle conseguenze, nel caso in cui una delle parti non rispetti gli impegni contrattuali.

Forse avete un contratto d'impiego che specifica le ore di lavoro e le regole di condotta da seguire; in cambio l'azienda vi paga uno stipendio e altri benefici. Ogni parte adempie ai suoi obblighi e tutti ne traggono beneficio.

È un'idea applicata in tutto il mondo, formalmente e informalmente, per aiutare gli esseri umani a interagire. Possiamo usare lo stesso concetto per favorire l'interazione dei moduli software? La risposta è "sì".

DBC

Bertrand Meyer [Mey97b] ha sviluppato il concetto di progettazione per contratto, *Design by Contract* (DBC, in breve) per il linguaggio Eiffel, basato in parte su lavori precedenti di Dijkstra, Floyd, Hoare, Wirth e altri. (Per maggiori informazioni su Eiffel stesso, vedete [URL10] e [URL11].) Si tratta di una tecnica semplice ma potente che si concentra sulla documentazione (e l'accordo) su diritti e responsabilità dei moduli software per garantire la correttezza dei programmi. Che cos'è un programma corretto? Un programma che fa quello che dice di fare, niente di più ma anche niente di meno. Documentare e verificare questa affermazione è il nocciolo della DBC.

Ogni funzione, ogni metodo in un sistema software *fa qualcosa*. Prima di iniziare a fare quel *qualcosa*, la routine può avere qualche aspettativa sullo stato del mondo, e può essere in grado di fare qualche affermazione sullo stato del mondo quando si conclude. Meyer descrive queste aspettative e queste affermazioni nel modo seguente.

- *Precondizioni*. Quel che deve essere vero perché la routine venga chiamata: i requisiti della routine. Una routine non deve mai venir chiamata quando le sue precondizioni sono violate. È responsabilità del chiamante passare buoni dati (vedi il riquadro "DBC e i parametri costanti").

- *Postcondizioni*. Quel che è garantito che la routine faccia; lo stato del mondo dopo che la routine ha completato il suo lavoro. Il fatto che la routine abbia una postcondizione comporta che *si concluda*: i cicli infiniti non sono ammessi.
- *Invarianti di classe*. Una classe garantisce che questa condizione sia sempre vera nella prospettiva di un chiamante. Durante l'elaborazione interna di una routine, l'invarianza può non valere, ma quando la routine esce e il controllo torna al chiamante, l'invariante deve essere vero. (Notate che una classe non può dare un accesso senza vincoli in scrittura a qualsiasi elemento di dati che faccia parte dell'invariante.)

Vediamo il contratto per una routine che inserisce un valore di dati in una lista ordinata unica. In iContract, un preprocessore per Java disponibile a [URL17], lo si specificherebbe in questo modo.

```
/**
  * @invariant forall Node n in elements() |
  *    n.prev() != null
  *      implies
  *         n.value().compareTo(n.prev().value()) > 0
  */

public class dbc_list {
  /**
    * @pre contains(aNode) == false
    * @post contains(aNode) == true
    */

  public void insertNode(final Node aNode) {
    // …
```

Qui diciamo che i nodi in questa lista devono essere sempre in ordine crescente. Quando si inserisce un nuovo nodo, non può essere già presente, e garantiamo che dopo l'inserimento lo si troverà nella lista.

Si scrivono queste precondizioni, postcondizioni e invarianti nel linguaggio di programmazione target, magari con qualche estensione. Per esempio, iContract mette a disposizione gli operatori della logica predicativa (`forall`, `exists`, `implies`) oltre ai normali costrutti Java. Le vostre asserzioni possono interrogare lo stato di qualsiasi oggetto a cui il metodo possa accedere, ma fate attenzione che l'interrogazione non abbia alcun effetto collaterale.

DBC e i parametri costanti

Spesso, una postcondizione userà parametri passati a un metodo per verificare la correttezza del comportamento. Se però alla routine è consentito modificare il parametro che le viene passato, si può aggirare il contratto. Eiffel non lo permette, ma Java sì. Qui usiamo la parola chiave Java `final` per indicare la nostra volontà che il parametro non sia modificato nel metodo. Non è un procedimento a prova di bomba: le sottoclassi possono ridichiarare il parametro come non finale. In alternativa, si può usare la sintassi `variabile@pre` di iContract per recuperare il valore che la variabile aveva in origine, prima di entrare nel metodo.

Il contratto fra una routine e ogni eventuale chiamante si può quindi leggere in questo modo.

> Se tutte le precondizioni della routine sono soddisfatte dal chiamante, la routine garantirà che tutte le postcondizioni e le invarianti saranno vere quando completerà il proprio lavoro.

Se una delle parti non rispetta i termini del contratto, viene chiamato in causa un rimedio (su cui si è convenuto in precedenza): viene sollevata un'eccezione, oppure il programma termina, per esempio. Qualsiasi cosa succeda, non fraintendete: il mancato rispetto del contratto non è un "baco". È qualcosa che non deve mai succedere, ragion per cui le precondizioni non devono essere usate per svolgere compiti come la validazione dell'input dell'utente.

SUGGERIMENTO 31
Progettate con i contratti.

In *Ortogonalità*, nel Capitolo 1, abbiamo consigliato di scrivere codice "timido"; qui, invece, l'accento è sul codice "pigro": siate molto rigidi per quel che accettate prima di iniziare, e promettete il minimo possibile in cambio. Ricordate: se il contratto dice che accetterete qualsiasi cosa e in cambio promettete il mondo, avrete un sacco di codice da scrivere!

Eredità e polimorfismo sono le pietre angolari dei linguaggi a oggetti e sono un campo in cui i contratti possono davvero brillare. Supponiamo che usiate l'eredità per creare una relazione "è un tipo di", dove una classe "è un tipo di" un'altra classe. Probabilmente vorrete aderire al *principio di sostituzione di Liskov* (Lis88):

> Le sottoclassi devono poter essere usate attraverso l'interfaccia della classe base senza bisogno che l'utente sappia la differenza.

In altre parole, volete essere sicuri che il nuovo sottotipo che avrete creato sia davvero "un tipo del" tipo base - cioè che ammetta gi stessi metodi, e che quei metodi abbiano lo stesso significato. Lo si può fare con i contratti. Dobbiamo specificare un contratto una sola volta, nella

classe base, in modo che venga applicato automaticamente a ogni futura sottoclasse. Una sottoclasse può, facoltativamente, accettare una gamma di input più ampia o fornire garanzie più ampie, ma deve accettare almeno quanto accetta la genitrice e garantire tutto quello che garantisce la genitrice.

Per esempio, prendiamo la classe base Java `java.awt.Component`. Potete trattare qualsiasi componente visuale in AWT o Swing come un `Component`, senza sapere se la sottoclasse effettiva è un pulsante, un canvas, un menu o qualche altra cosa. Ogni singolo componente può fornire ulteriori funzionalità specifiche, ma deve fornire almeno le capacità di base definite da `Component`. Non c'è nulla però che vi impedisca di creare un sottotipo di `Component` che metta a disposizione metodi, con nomi corretti, che fanno la cosa sbagliata. Potete creare facilmente un metodo `paint` che non dipinge un bel niente, o un metodo `setFont` che non imposta il tipo di carattere. AWT non ha contratti per far fronte al fatto che non rispettiate l'accordo.

Senza un contratto, tutto quello che può fare il compilatore è garantire che una sottoclasse sia conforme a una particolare "firma" di metodo. Se però introduciamo un contratto per la classe base, possiamo garantire che qualsiasi sottoclasse futura non possa alterare i *significati* dei nostri metodi. Per esempio, potreste definire un contratto per `setFont` simile al seguente, che garantisce che il tipo di carattere che impostate sia quello che otterrete:

```
/**
  *@pre f != null
  *@post getFont() == f
  */
  public void setFont(final Font f) {
    // ...
```

Implementare DBC

Il vantaggio principale di usare DBC può essere che mette in primo piano il problema dei requisiti e delle garanzie. Semplicemente enumerare al momento della progettazione qual è il dominio degli input, quali sono le condizioni al contorno e che cosa la routine promette di fornire - o, più importante, che cosa *non* promette di fornire - è un grande salto in avanti verso la scrittura di software migliore. Se non precisate

queste cose, tornate alla *programmazione per coincidenza* (Capitolo 6), che è dove molti progetti iniziano, finiscono e falliscono.

Nei linguaggi che non consentono DBC nel codice, può darsi che non possiate andare oltre - e non è troppo male. DBC, in fin dei conti, è una tecnica di *progettazione*. Anche senza controllo automatico, potete inserire il contratto nel codice sotto forma di commenti e averne comunque un vantaggio molto concreto. Se non altro, i contratti in forma di commento vi danno un luogo in cui cominciare a guardare quando arrivano i guai.

Asserzioni

Documentare questi assunti è un ottimo punto di partenza, ma potete ottenere un beneficio molto maggiore facendo controllare al compilatore il contratto al posto vostro. Potete emulare questa cosa in parte in alcuni linguaggi utilizzando le *asserzioni* (vedi *Programmazione assertiva*, più avanti in questo capitolo). Perché solo in parte? Non si possono usare le asserzioni per fare tutto quello che può fare DBC?

Purtroppo la risposta è no. Tanto per cominciare, non c'è supporto per la propagazione delle asserzioni lungo una gerarchia di eredità. Questo significa che, se si aggira un metodo della classe base che ha un contratto, le asserzioni che implementano quel contratto non verranno chiamate correttamente (a meno che non le duplichiate manualmente nel nuovo codice). Dovete ricordare di chiamare l'invariante di classe (e tutte le invarianti della classe base) manualmente prima di uscire da ogni metodo. Il problema di fondo è che il contratto non viene fatto rispettare automaticamente.

Inoltre non esiste un concetto incorporato di "vecchi" valori, cioè dei valori come erano al momento dell'ingresso in un metodo. Se usate le asserzioni per far valere i contratti, dovete aggiungere alla precondizione il codice per salvare tutte le informazioni che volete usare nella postcondizione. Fate il confronto con iContract, dove la postcondizione può semplicemente fare riferimento a `variable@pre`, o con Eiffel, che supporta `old expression`.

Infine, il sistema e le librerie della fase di esecuzione non sono progettate in modo da supportare i contratti, perciò queste chiamate non vengono controllate. È una perdita grave, perché spesso la maggior parte

dei problemi si scopre proprio al confine fra il nostro codice e le librerie che usa (vedi *I programmi morti non mentono* più avanti in questo capitolo per un'analisi più particolareggiata).

Supporto per i linguaggi

I linguaggi che incorporano un supporto per DBC (come Eiffel e Sather [URL12]) controllano automaticamente le precondizioni e le postcondizioni nel compilatore e nel sistema di esecuzione. In questo caso si ha il massimo beneficio, perché *tutta* la base di codice (anche le librerie) deve onorare i contratti.

Che cosa si può dire invece per linguaggi più diffusi, come C, C++ e Java? Per questi linguaggi, esistono preprocessori che elaborano i contratti incorporati nel codice sorgente originale sotto forma di commenti speciali. Il preprocessore espande questi commenti in codice che verifica le asserzioni.

Per C e C++, potete investigare Nana [URL18]. Nana non gestisce l'eredità, ma usa il debugger in fase di esecuzione per controllare le asserzioni in modo nuovo.

Per Java, esiste iContract [URL17], che prende i commenti (in formato JavaDoc) e genera un nuovo file sorgente con inclusa la logica dell'asserzione.

I preprocessori non sono efficaci quanto gli strumenti incorporati. Può essere complicato integrarli nei propri progetti, e altre librerie che si usano non avranno contratti. Possono comunque essere molto utili: quando si scopre un problema in questo modo (in particolare un problema che non avreste *mai* trovato) è quasi una magia.

DBC e crash precoci

DBC si sposa bene al nostro concetto di "crash precoce" (vedi *I programmi morti non mentono*, più avanti in questo capitolo). Supponiamo di avere un metodo che calcola le radici quadrate (come nella classe `DOUBLE` di Eiffel). Richiede una precondizione, che il dominio sia ristretto ai numeri positivi. Una precondizione in Eiffel si dichiara con la parola chiave `require` e una postcondizione si dichiara con `ensure`, perciò potremmo scrivere

```
sqrt: DOUBLE is
      -- Routine per la radice quadrata
   require
      sqrt_arg_must_be_positive: Current >= 0;
   --- ...
   --- qui calcola la radice quadrata
   --- ...
   ensure
       ((Result*Result) - Current).abs <= epsilon*Current.abs;
       -- Il risultato deve essere entro il limite di tolleranza degli errori
   end;
```

Chi è responsabile?
Chi ha la responsabilità di controllare la precondizione, il chiamante o la routine chiamata? Se l'implementazione fa parte del linguaggio, la risposta è "nessuno dei due": la precodizione viene verificata dietro le quinte, dopo che il chiamante ha invocato la routine, ma prima che si entri nella routine stessa. Quindi, se si deve effettuare un controllo esplicito dei parametri, deve essere compiuto dal *chiamante*, perché la routine stessa non vedrà mai parametri che violino la sua precondizione. (Per i linguaggi senza supporto interno, bisognerà racchiudere la routine *chiamata* con un preambolo o un "postambolo" che controlli quelle asserzioni.)
Consideriamo un programma che legga un numero dalla console, ne calcoli la radice quadrata (chiamando `sqrt`) e ne stampi il risultato. La funzione `sqrt` ha una precondizione: il suo argomento non deve essere negativo. Se l'utente inserisce alla console un numero negativo, sta al codice chiamante fare in modo che non venga mai passato a `sqrt`. Questo codice chiamante ha molte opzioni: potrebbe terminare, potrebbe emettere un avvertimento e leggere un altro numero, oppure potrebbe trasformare il numero in positivo e aggiungere una "*i*" al risultato restituito da `sqrt`. Qualsiasi cosa scelga, decisamente non è un problema di `sqrt`.
Esprimendo il dominio della funzione radice quadrata nella precondizione della routine `sqrt`, si trasferisce l'onere della correttezza al chiamante - dove è giusto che stia. Si può poi progettare la routine `sqrt` sapendo con certezza che il suo input *sarà* all'interno del suo dominio.

Se il vostro algoritmo per il calcolo della radice quadrata fallisce (o non rispetta i limiti di tolleranza per gli errori), ricevete un messaggio di errore e una traccia dello stack che vi presenta la catena di chiamate.

Se passate a `sqrt` un parametro negativo, il runtime di Eiffel stampa l'errore `sqrt_arg_must_be_positive`, insieme con una traccia dello stack. È meglio dell'alternativa in linguaggi come Java, C e C++, dove il passaggio di un numero negativo a `sqrt` restituisce il valore speciale `NaN` (*Not a Number*). Magari solo più tardi nel programma tenterete di fare qualche calcolo con `NaN`, e otterrete risultati del tutto imprevisti.

È molto più facile trovare e diagnosticare il problema con un crash precoce, nel punto dove il problema sorge.

Altri usi delle invarianti

Fin qui abbiamo discusso di precondizioni e postcondizioni che si applicano a singoli metodi e di invarianti che valgono per tutti i metodi di una classe, ma le invarianti si possono sfruttare anche in altri modi utili.

Invarianti di ciclo

Fissare correttamente le condizioni di limite di un ciclo non banale può essere problematico. I cicli sono soggetti al problema della banana (so come fare lo spelling di "banana", ma non so quando fermarmi), errori di paletti da recinzione (non so se contare i paletti o gli spazi fra l'uno e l'altro) e all'onnipresente errore "fuori di uno" [URL52].

In queste situazioni le invarianti possono essere d'aiuto: una *invariante di ciclo* è un enunciato dello scopo ultimo di un ciclo, ma generalizzato in modo che sia valido anche prima che il ciclo venga eseguito e a ogni iterazione del ciclo. Potete pensarla come un contratto in miniatura. L'esempio classico è una routine che trova il valore più grande in un array.

```
int m = arr[0]; // si assume arr.length > 0
int i = 1;
// Invariante di ciclo: m = max(arr[0:i-1])
while (i < arr.length) {
  m = Math.max(m, arr[i]);
  i = i + 1;
}
```

(`arr[m:n]` è una notazione di comodo che indica una sezione (*slice*) dell'array dall'indice *m* all'indice *n*.) L'invariante deve essere vera prima che il ciclo inizi, e il corpo del ciclo deve garantire che resti vera mentre il ciclo viene eseguito. In questo modo sappiamo che l'invariante è vera anche quando il ciclo termina, perciò il nostro risultato è valido. Le invarianti di ciclo possono essere codificate esplicitamente come asserzioni, ma sono utili anche come strumenti di progettazione e di documentazione.

Invarianti semantiche

Potete usare *invarianti semantiche* per esprimere requisiti inviolabili, una sorta di "contratto filosofico".

Una volta abbiamo scritto uno switch per transazioni con carta di debito. Un requisito importante era che una stessa transazione non venisse mai applicata due volte al conto di un utente di carta di debito. In altre parole, indipendentemente da quale fosse la modalità in cui si verifica, si doveva eventualmente sbagliare non elaborando una transazione anziché elaborando il duplicato di una transazione.

Questa semplice legge, derivata direttamente dai requisiti, si è dimostrata molto utile per risolvere situazioni di recupero da errori complessi, e ha guidato i dettagli della progettazione e dell'implementazione in molte aree.

Fate attenzione a non confondere requisiti che costituiscono leggi immutabili e inviolabili con quelli che sono semplicemente direttive che potrebbero cambiare con il management. Per questo parliamo di invarianti *semantiche*: devono essere fondamentali per il *significato* di una cosa, e non soggette ai capricci della politica aziendale (per questo ci sono le regole di business dinamiche).

Se trovate un requisito che soddisfa questa definizione, assicuratevi che diventi una parte ben conosciuta di tutta la documentazione che producete - che sia un elenco puntato nel documento dei requisiti che deve essere firmato in triplice copia o solo una grande annotazione sulla lavagna comune, visibile a tutti. Cercate di formularlo in modo chiaro e senza ambiguità. Nel caso delle carte di debito, per esempio, potremmo scrivere

SBAGLIA A FAVORE DEL COMSUMATORE.

Questa è un'indicazione chiara, concisa e priva di ambiguità, applicabile in molte aree diverse del sistema. È il nostro contratto con tutti gli utenti del sistema, la nostra garanzia di comportamento.

Contratti dinamici e agenti

Finora abbiamo parlato di contratti come specifiche fisse, immutabili, ma, in un panorama di agenti autonomi, non deve per forza essere così. In base alla definizione di "autonomi", gli agenti sono liberi di *rifiutare* richieste che non vogliono onorare. Sono liberi di rinegoziare il contratto: "Questo non posso farlo, ma me se mi dai questo, allora posso darti qualche altra cosa".

Certo qualsiasi sistema che si basi sulla tecnologia degli agenti dipende in modo *determinante* da accordi contrattuali, anche se sono generati dinamicamente.

Immaginate: avendo abbastanza componenti e agenti in grado di negoziare i propri contratti fra di loro per raggiungere un obiettivo, potremmo risolvere la crisi di produttività del software semplicemente lasciando che sia il software a risolverla per noi.

Se però non sappiamo usare i contratti manualmente, non riusciremo a usarli automaticamente. Perciò, la prossima volta che progettate un pezzo di software, progettate anche il suo contratto.

Vedi anche

- *Ortogonalità*, Capitolo 2
- *I programmi morti non mentono*, in questo Capitolo
- *Programmazione assertiva*, in questo Capitolo
- *Come equilibrare le risorse*, in questo Capitolo
- *Disaccoppiamento e legge di Demetra*, Capitolo 5
- *Accoppiamento temporale*, Capitolo 5
- *Programmazione per coincidenza*, Capitolo 6
- *Codice facile da sottoporre a test*, Capitolo 6
- *Team pragmatici*, Capitolo 8

Sfide

- Aspetti su cui riflettere: se DBC è così potente, perché non viene usata più ampiamente? È difficile formulare il contratto? Vi fa pensare a questioni che per il momento preferireste ignorare? Vi costringe a PENSARE!? Chiaramente, è uno strumento pericoloso!

Esercizi

14. Che cosa fa di un contratto un buon contratto? Chiunque può aggiungere precondizioni e postcondizioni, ma serviranno a qualcosa? Peggio ancora, faranno più male che bene? Per l'esempio sotto e per quelli degli Esercizi 15 e 16, stabilite se il contratto specificato è buono, brutto o cattivo e spiegate perché.

Come prima cosa, vediamo un esempio in Eiffel. Qui abbiamo una routine per aggiungere una `STRING` a una lista circolare doppiamente concatenata (ricordate che le precondizioni sono indicate con require, le postcondizioni con `ensure`).

```
-- Aggiunge un element unico a una lista doppiamente concatenata
-- e restituisce il NODE appena creato.
add_item (item : STRING) : NODE is
   require
      item /= Void              -- '/=' significa 'diverso da'.
```

```
      find_item(item) = Void     -- Deve essere unico
   deferred                      -- Classe base astratta.
   ensure
      result.next.previous = result  -- Controlla i link del nodo
      result.previous.next = result  -- appena aggiunto.
      find_item(item) = result       -- Deve trovarlo.
   end
```

15. Proviamo un esempio in Java, qualcosa di simile all'esempio dell'Esercizio 14. `insertNumber` inserisce un intero in una lista ordinata. Le precondizioni e postcondizioni sono etichettate come in `iContract` (vedi [URL17]).

```
private int data[];
/**
  *@post data[index-1] < data[index] &&
  *data[index] == aValue
  */
public Node insertNumber (final int aValue)
{
  int index = findPlaceToInsert(aValue);
. . .
```

16. Ecco un frammento di una classe stack in Java. Questo è un buon contratto?

```
/**
  *@pre anItem != null     // Richiede dati reali
  *@post pop() == anItem   // Verifica che siano
  *                        // sullo stack
  */
public void push(final String anItem)
```

17. Gli esempi classici di DBC (come negli Esercizi 14-16) mostrano una implementazione di un tipo di dati astratto, normalmente uno stack o una coda. Non molti però scrivono realmente classi di basso livello di questo tipo.
Perciò, per questo esercizio, progettiamo un'interfaccia per un miscelatore da cucina. Alla fine sarà un miscelatore basato su Web, abilitato a Internet e basato su CORBA, ma per il momento ci serve solo l'interfaccia per controllarlo. Ha dieci velocità (0 significa spento). Non si può usarlo vuoto e si può cambiare la velocità solo di una unità alla volta (si può passare cioè da 0 a 1 e da 1 a 2, ma non da 0 a 2).
Ecco i metodi. Aggiungete le opportune precondizioni e postcondizioni e una invariante.

```
int getSpeed()
void setSpeed(int x)
boolean isFull()
void fill()
void empty()
```

18. Quanti numeri ci sono nella successione 0, 5, 10, 15, …, 100?

I programmi morti non mentono

Avete mai notato che talvolta gli altri possono capire che qualcosa non va in voi prima ancora che siate consapevoli voi stessi del problema? Succede lo stesso con il codice degli altri. Se qualcosa comincia ad

andare storto in uno dei nostri programmi, talvolta se ne accorge prima una routine di libreria. Magari è un puntatore randagio che ci ha fatto sovrascrivere a un handle di file qualcosa che non ha senso. La prossima chiamata a `read` lo catturerà. Forse qualcosa è tracimato da un buffer e ha rovinato un contatore che stavamo per usare per determinare quanta memoria allocare. Magari avremo una segnalazione di errore da `malloc`. Un errore logico un paio di milioni di istruzioni fa significa che il selettore di un enunciato `case` non è più 1, 2 o 3 come previsto. Finiremo nel caso `default` (è questo un motivo per cui ogni enunciato `case/switch` deve avere una clausola di default: vogliamo sapere quando è successo l'"impossibile").

È facile cadere nella mentalità "non può succedere". Quasi tutti abbiamo scritto codice che non controllava se un file era stato chiuso correttamente, o se un enunciato di traccia era stato scritto come previsto. A parità di ogni altra cosa, è probabile che non ce ne fosse bisogno - il codice in questione non avrebbe dato errori in condizioni normali. Ma noi codifichiamo sulla difensiva. Cerchiamo puntatori sciolti in altre parti del programma che mandano all'aria lo stack. Verifichiamo che siano state effettivamente caricate le versioni corrette delle librerie condivise.

Tutti gli errori vi danno informazioni. Potreste convincervi che l'errore non può succedere, e scegliere di ignorarlo. I programmatori pragmatici invece si dicono che, se c'è un errore, deve essere successo qualcosa di molto, molto brutto.

SUGGERIMENTO 32
Andate in crash il più presto possibile.

Uno dei vantaggi del rilevare gli errori il più presto possibile è il crash precoce e, in molti casi, mandare in crash il programma è la cosa migliore che si possa fare. L'alternativa sarebbe continuare, magari scrivendo dati corrotti in qualche database fondamentale oppure comandando alla macchina di fare il ventesimo ciclo di risciacquo consecutivo.

Il linguaggio e le librerie Java hanno adottato questa filosofia. Quando nel sistema di runtime succede qualcosa di imprevisto, lancia una `RuntimeException`. Se non viene catturata, questa sale progressivamente fino al livello più alto del programma e lo fa fermare, visualizzando una traccia dello stack.

Potete fare la stessa cosa in altri linguaggi. Se non avete un meccanismo di eccezione, o se le vostre librerie non lanciano eccezioni,

assicuratevi di gestire in prima persona gli errori. In C, possono essere molto utili a questo scopo le macro:

```
#define CHECK(LINE, EXPECTED)                \
  { int rc = LINE;                           \
    if (rc != EXPECTED)                      \
        ut_abort(__FILE__, __LINE__, #LINE, rc, EXPECTED); }
void ut_abort(char *file, int ln, char *line, int rc, int exp) {
  fprintf(stderr, "%s line %dnn'%s': expected %d, got %dnn",
                  file, ln, line, exp, rc);
  exit(1);
}
```

Poi si possono racchiudere le chiamate che non devono mai fallire, usando

```
CHECK(stat("/tmp", &stat_buff), 0);
```

Se una chiamata dovesse fallire, si otterrebbe un messaggio scritto su `stderr`:

```
source.c line 19
'stat("/tmp", &stat_buff) ': expected 0, got -1
```

Chiaramente, a volte non è appropriato limitarsi a uscire da un programma in esecuzione. Ci sono magari risorse impegnate che così non vengono liberate, oppure è necessario scrivere messaggi di log, ripulire le transazioni aperte o interagire con altri processi. Le tecniche di cui parliamo in *Quando usare le eccezioni*, più avanti in questo capitolo, saranno utili. Il principio di base però resta lo stesso: quando il vostro codice scopre che qualcosa che si supponeva impossibile è appena successo, il vostro programma non è più praticabile. Qualsiasi cosa faccia da quel momento in poi è sospetto, perciò terminatelo il più presto possibile. Un programma morto normalmente fa meno danni di un programma azzoppato.

Vedi anche

- *Progettare per contratto*, in questo Capitolo
- *Quando usare le eccezioni*, in questo Capitolo

Programmazione assertiva

> L'autorimprovero è un lusso. Se ci diamo la colpa da soli abbiano la sensazione che nessun altro abbia diritto di rimproverarci.
> - Oscar Wilde, *Il ritratto di Dorian Gray*

Sembra ci sia un mantra che il programmatore deve memorizzare agli inizi della carriera. È un principio fondamentale dell'informatica, una convinzione cruciale che impariamo ad applicare a requisiti, progetti, codice, commenti, quasi tutto quello che facciamo. Dice

QUESTO NON POTRÀ MAI SUCCEDERE…

"Questo codice fra trent'anni non si userà più, perciò le date scritte con due cifre vanno benissimo." "Questa applicazione non verrà mai usata all'estero, perciò perché internazionalizzarla?" (`count` non può essere negativo." "Questa `printf` non può fallire."

Non lasciamoci cullare da questi autoinganni, in particolare quando scriviamo codice.

SUGGERIMENTO 33

Se non può succedere, usate le asserzioni per essere sicuri che non succeda.

Ogni volta che vi ritrovate a pensare "ma è ovvio che non può succedere", aggiungete del codice per verificarlo. Il modo più semplice è usare le asserzioni. Nella maggior parte delle implementazioni di C e C++, troverete qualche forma di macro `assert` o `_assert` che verifica una condizione booleana. Queste macro possono essere preziose. Se un puntatore passato a una procedura non deve mai essere `NULL`, verificate che sia così:

```
void writeString(char *string) {
    assert(string != NULL);
    ...
```

Le asserzioni costituiscono anche controlli utili sul funzionamento degli algoritmi. Magari avete scritto un brillante algoritmo di ordinamento: controllate anche che funzioni bene.

```
for (int i = 0; i < num_entries-1; i++) {
    assert(sorted[i] <= sorted[i+1]);
}
```

Ovviamente, la condizione passata a un'asserzione non deve avere un effetto collaterale. Ricordate anche che le asserzioni possono essere disattivate alla compilazione, perciò non mettete mai in una `assert` del codice che *debba* essere eseguito.

Non usate asserzioni al posto di una vera gestione degli errori. Le asserzioni controllano cose che non devono mai avvenire: non bisogna scrivere codice come

```
printf("Enter 'Y' or 'N': ");
ch = getchar();
```

```
assert((ch == 'Y') || (ch == 'N'));  /*pessima idea! */
```

E solo perché le macro `assert` fornite chiamano `exit` quando un'asserzione fallisce, non c'è ragione per cui debbano farlo le versioni che scrivete voi. Se avete bisogno di liberare risorse, fate in modo che il fallimento di un'asserzione generi un'eccezione, faccia un `longjmp` a un punto di uscita o chiami un gestore di errori. Assicuratevi solo che il codice eseguito in quei millisecondi prima della terminazione non si basi proprio sulle informazioni che hanno provocato il fallimento dell'asserzione.

Lasciate attive le asserzioni

Circola un malinteso, a proposito delle asserzioni, diffuso da chi scrive compilatori e ambienti per linguaggi. Suona più o meno così:

> Le asserzioni aggiungono del carico inutile al codice. Poiché controllano se si verificano cose che non devono succedere mai, vengono innescate solo da un baco nel codice. Una volta che il codice è stato collaudato e consegnato, non servono più e vanno disattivate, così il codice gira più veloce. Le asserzioni sono uno strumento per il debug.

Qui ci sono due presupposti falsi. In primo luogo, si presuppone che i test trovino tutti gli errori. In realtà, per qualsiasi programma complesso è improbabile che si sottoponga a prova nemmeno una minuscola percentuale delle permutazioni per cui dovrà passare il codice (vedi *Test senza pietà* nel Capitolo 8). In secondo luogo, gli ottimisti dimenticano che i programmi girano in un mondo pericoloso. Durante i test probabilmente un topo non addenterà un cavo di comunicazione, nessuno si metterà a giocare esaurendo la memoria e i file di log non riempiranno il disco. Queste cose possono succedere quando il programma gira in un ambiente di produzione. La vostra prima linea di difesa sta nel controllare qualsiasi possibile errore, la seconda nell'usare le asserzioni per cercare di identificare quelli che vi sono sfuggiti.

Disattivare le asserzioni quando si consegna un programma alla produzione è come camminare su un filo a grande altezza senza rete di protezione perché una volta si è riusciti a farcela. Il successo sarà grande, ma sarà difficile trovare un'assicurazione sulla vita.

Anche se *avete* problemi di prestazioni, disattivate solo le asserzioni che danno davvero fastidio. L'esempio dell'ordinamento visto prima può essere una parte critica dell'applicazione, e probabilmente deve essere

molto veloce. L'aggiunta di un controllo può voler dire far passare un'altra volta i dati, il che può essere inaccettabile. Rendete opzionale quel particolare controllo, ma lasciate gli altri.

NOTA

Nei linguaggi basati sul C, potete usare il preprocessore o enunciati `if` per rendere facoltative le asserzioni. Molte implementazioni disattivano la generazione del codice per la macro `assert` se è impostato (o non impostato) un flag per la fase di compilazione. Altrimenti, potete collocare il codice entro un enunciato `if` con una condizione costante, che molti compilatori (e anche i sistemi Java più comuni) elimineranno a fini di ottimizzazione.

Asserzioni ed effetti collaterali

È imbarazzante quando il codice che aggiungiamo per identificare errori in realtà finisce per crearne di nuovi. Questo può succedere se la valutazione della condizione ha effetti collaterali. In Java, per esempio, sarebbe una cattiva idea scrivere codice come questo:

```
while (iter.hasMoreElements()) {
  Test.ASSERT(iter.nextElement() != null);
  Object obj = iter.nextElement();
  // ....
}
```

La chiamata `.nextElement()` nell'`ASSERT` ha l'effetto collaterale di spostare l'iteratore oltre l'elemento che viene recuperato, perciò il ciclo elaborerà solo metà degli elementi della collezione. Sarebbe meglio scrivere

```
while (iter.hasMoreElements()) {
  Object obj = iter.nextElement();
  Test.ASSERT(obj != null);
  // ....
}
```

Il problema è una forma di "Heisenbug", debug che modifica il comportamento del sistema di cui si effettua il debug (vedi [URL52]).

Vedi anche

- *Debug*, Capitolo 3
- *Progettare per contratto*, in questo Capitolo
- *Come equilibrare le risorse*, in questo Capitolo
- *Programmazione per coincidenza*, Capitolo 6

Esercizi

19. Una veloce "verifica di realtà". Quali di queste cose "impossibili" possono succedere?

1. Un mese con meno di 28 giorni.
2. `stat(".",&sb) == -1` (cioè, non è possibile accedere alla directory corrente).

3. In C++: `a = 2; b = 3; if (a + b != 5) exit(1);`
4. Un triangolo la somma dei cui angoli interni è diversa da 180°.
5. Un minuto che non ha 60 secondi.
6. In Java: `(a + 1) <= a`.

20. Sviluppate una semplice classe di controllo delle asserzioni per Java.

Quando usare le eccezioni

In *I programmi morti non mentono* abbiamo detto che è buona pratica effettuare controlli per ogni possibile errore - in particolare quelli imprevisti. In pratica però questo può portare a codice molto brutto; la logica normale del programma può finire per essere totalmente oscurata dalla gestione degli errori, in particolare se siete della scuola "una routine deve avere un unico enunciato di ritorno" (noi non siamo di questa scuola). Abbiamo visto codice simile a questo:

```
retcode = OK;
if (socket.read(name) != OK) {
  retcode = BAD_READ;
}
else {
  processName(name);
  if (socket.read(address) != OK) {
    retcode = BAD_READ;
  }
  else {
    processAddress(address);
    if (socket.read(telNo) != OK) {
      retcode = BAD_READ;
    }
    else {
      // ecc, ecc...
    }
  }
}
return retcode;
```

Per fortuna, se il linguaggio di programmazione ammette le eccezioni, potete riscrivere questo codice in un modo molto più elegante:

```
retcode = OK;
try {
  socket.read(name);
  process(name);
  socket.read(address);
  processAddress(address);
  socket.read(telNo);
  // ecc, ecc...
}
catch (IOException e) {
```

```
  retcode = BAD_READ;
  Logger.log("Error reading individual: " + e.getMessage());
}
return retcode;
```

Il normale flusso di controllo ora è chiaro, e tutta la gestione degli errori è stata sistemata in un unico luogo.

Che cos'è eccezionale?

Uno dei problemi delle eccezioni è sapere quando usarle. Siamo convinti che debbano essere usate raramente come parte del flusso normale di un programma; le eccezioni devono essere riservate per eventi imprevisti. Ipotizziamo che un'eccezione non catturata termini il programma; ci chiediamo, "Questo codice girerà ancora se elimino tutti i gestori di eccezioni?". Se la risposta è "no", allora forse le eccezioni vengono utilizzate in circostanze non eccezionali.

Per esempio, se il codice cerca di aprire un file in lettura ma quel file non esiste, deve essere sollevata un'eccezione?

La nostra risposta è "dipende". Se il file *dovrebbe* essere lì, è giusta un'eccezione. È successo qualcosa di imprevisto - un file di cui si prevedeva l'esistenza sembra scomparso. Invece, se non avete idea se il file dovesse esistere o meno, non sembra eccezionale che non riusciate a trovarlo, e la cosa giusta è la restituzione di un errore.

Vediamo un esempio del primo caso. Il codice seguente apre il file `/etc/passwd`, che deve esistere in tutti i sistemi Unix. Se fallisce, passa al chiamante `FileNotFoundException`.

```
public void open_passwd() throws FileNotFoundException {
  // Questo può lanciare FileNotFoundException...
  ipstream = new FileInputStream("/etc/passwd");
  // ...
}
```

Il secondo caso invece può comportare l'apertura di un file specificato dall'utente sulla riga di comando. Qui non è giustificata un'eccezione, e il codice risulta diverso:

```
public boolean open_user_file(String name)
  throws FileNotFoundException {
  File f = new File(name);
  if (!f.exists()) {
    return false;
  }
  ipstream = new FileInputStream(f);
```

```
  return true;
}
```

Notate che la chiamata di `FileInutStream` può ancora generare un'eccezione, che viene trasmessa dalla routine. L'eccezione però verrà generata solo in circostanze davvero eccezionali; se si cerca semplicemente di aprire un file che non esiste si otterrà il ritorno di un errore convenzionale.

SUGGERIMENTO 34
Usate le eccezioni per problemi eccezionali.

Perché consigliamo questo approccio alle eccezioni? Beh, un'eccezione rappresenta un trasferimento di controllo immediato e non locale - è una sorta di `goto` a cascata. I programmi che usano le eccezioni come parte della loro elaborazione normale soffrono di tutti i problemi di leggibilità e manutenibilità del classico "codice a spaghetti". Questi programmi violano l'incapsulamento: le routine e i chiamanti sono accoppiati più strettamente mediante la gestione delle eccezioni.

I gestori degli errori sono un'alternativa

Un gestore di errori è una routine che viene chiamata quando si scopre un errore. Potete registrare una routine perché gestisca una categoria specifica di errori. Quando si verifica uno di questi errori, viene chiamato quel gestore.

Vi sono occasioni in cui è preferibile usare i gestori di errore, o al posto delle eccezioni o insieme a queste ultime. Chiaramente, se usate un linguaggio come il C, che non supporta le eccezioni, questa è una delle poche opzioni che avete a disposizione. Tuttavia, a volte si possono usare i gestori di errori anche in linguaggi (come Java) che incorporano un valido schema di gestione delle eccezioni.

Pensate all'implementazione di un'applicazione client-server che usi la tecnica RMI (*Remote Method Invocation*) di Java. Per il modo in cui è implementata RMI, ogni chiamata a una routine remota deve essere preparata a gestire una `RemoteException`. Aggiungere codice per gestire queste eccezioni può diventare noioso, e significa che è difficile scrivere codice che funzioni sia con routine locali che con routine remote. Un possibile modo di aggirare il problema è avvolgere gli oggetti remoti in una classe che non sia remota. Questa classe poi implementa

un'interfaccia di gestione degli errori, consentendo al codice client di registrare una routine da chiamare quando viene identificata una eccezione remota.

Vedi anche

- *I programmi morti non mentono*, in questo Capitolo

Sfide

- Linguaggi che non ammettono le eccezioni spesso hanno qualche altro meccanismo non locale di trasferimento del controllo (C ha `longjpm/setjmp`, per esempio). Pensate a come potreste implementare qualche meccanismo sostitutivo delle eccezioni utilizzando questi strumento. Quali sono i vantaggi e quali i pericoli? Quali passi speciali dovete intraprendere per essere sicuri che le risorse non restino orfane? Ha senso usare questo tipo di soluzione ogni volta che si codifica in C?

Esercizi

21. Nel progettare una nuova classe contenitore, identificate queste possibili condizioni di errore.

1. Mancata disponibilità di memoria per un nuovo elemento nella routine `add`.
2. Mancato reperimento dell'elemento richiesto nella routine `fetch`.
3. Passaggio di un puntatore `null` alla routine `add`. Come gestirle? Generare un errore, sollevare un'eccezione o ignorare la condizione?

Come equilibrare le risorse

> "Ti ho messo al mondo", diceva mio padre "e posso farti sparire. Per me non fa differenza. Semplicemente ne faccio un altro come te".
> - Bill Cosby, *Fatherhood*

Tutti gestiamo risorse quando codifichiamo: memoria, transazioni, thread, file, timer - ogni genere di cose con limiti di disponibilità. Quasi sempre, però, l'uso delle risorse segue un andamento prevedibile: si assegna la risorsa, la si usa, poi la si libera.

Molti sviluppatori però non hanno un piano coerente per trattare con l'allocazione e la liberazione di risorse. Abbiamo un suggerimento molto semplice.

SUGGERIMENTO 35
Finite quello che cominciate.

È un consiglio facile da applicare nella maggior parte delle circostanze. Vuol dire semplicemente che la routine o l'oggetto che assegna una risorsa deve avere la responsabilità di liberarla. Vediamo come si applica esaminando un esempio di codice cattivo - un'applicazione che apre un file, legge le informazioni di un cliente, aggiorna un campo e scrive il risultato nel file. Abbiamo eliminato la gestione degli errori perché l'esempio fosse più chiaro.

```
void readCustomer(const char *fName, Customer *cRec) {
  cFile = fopen(fName, "r+");
  fread(cRec, sizeof(*cRec), 1, cFile);
}
void writeCustomer(Customer *cRec) {
  rewind(cFile);
  fwrite(cRec, sizeof(*cRec), 1, cFile);
  fclose(cFile);
}
void updateCustomer(const char *fName, double newBalance) {
  Customer cRec;
  readCustomer(fName, &cRec);
  cRec.balance = newBalance;
  writeCustomer(&cRec);
}
```

A prima vista, la routine `updateCustomer` sembra buona. Sembra implementare la logica richiesta - leggere un record, aggiornare il saldo e riscrivere il record. Questa pulizia però nasconde un problema grave. Le routine `readCustomer` e `writeCustomer` sono strettamente accoppiate: condividono la variabile globale `cFile`. `readCustomer` apre il file e memorizza il puntatore di file in `cFile`, e `writeCustomer` usa quel puntatore salvato per chiudere il file quando ha finito. Questa variabile globale non compare nemmeno nella routine `updateCustomer`.

Perché non va bene? Prendiamo lo sfortunato manutentore a cui viene detto che le specifiche sono cambiate - il saldo va aggiornato solo se il nuovo valore non è negativo. Va al codice sorgente e modifica `updateCustomer`:

```
void updateCustomer(const char *fName, double newBalance) {
  Customer cRec;
  readCustomer(fName, &cRec);
  if (newBalance >= 0.0) {
    cRec.balance = newBalance;
```

```
    writeCustomer(&cRec);
}
}
```

Durante i test sembra che vada tutto bene, però quando il codice va in produzione, dopo un po' di ore collassa, lamentando un numero eccessivo di file aperti. Poiché `writeCustomer` in alcuni casi non viene chiamata, il file non viene chiuso.

Una pessima soluzione sarebbe trattare il caso speciale in `updateCustomer`:

```
void updateCustomer(const char *fName, double newBalance) {
  Customer cRec;
  readCustomer(fName, &cRec);
  if (newBalance >= 0.0) {
    cRec.balance = newBalance;
    writeCustomer(&cRec);
  }
  else
    fclose(cFile);
}
```

Questo risolverà il problema (il file ora verrà chiuso indipendentemente dal nuovo saldo), ma la soluzione significa che ora *tre* routine sono accoppiate attraverso la variabile globale `cFile`. Stiamo cadendo in una trappola e le cose andranno sempre peggio, se continuiamo su questa strada.

Il consiglio *finite quello che cominciate* ci dice che, idealmente, la routine che assegna una risorsa dovrebbe anche liberarla, Possiamo applicarlo qui ricostruendo un po' il codice:

```
void readCustomer(FILE *cFile, Customer *cRec) {
  fread(cRec, sizeof(*cRec), 1, cFile);
}
void writeCustomer(FILE *cFile, Customer *cRec) {
  rewind(cFile);
  fwrite(cRec, sizeof(*cRec), 1, cFile);
}
void updateCustomer(const char *fName, double newBalance) {
  FILE *cFile;
  Customer cRec;
  cFile = fopen(fName, "r+");          // >---
  readCustomer(cFile, &cRec);          //     |
  if (newBalance >= 0.0) {             //     |
    cRec.balance = newBalance;         //     |
    writeCustomer(cFile, &cRec);       //     |
  }                                    //     |
  fclose(cFile);                       // <---
}
```

Ora tutta la responsabilità per il file sta nella routine `updateCustomer`: apre il file e (concludendo quello che ha iniziato) lo chiude prima di uscire. La routine equilibra l'uso del file: apertura e chiusura stanno nello stesso luogo, ed è evidente che per ogni apertura ci sarà una chiusura

corrispondente. La ricostruzione inoltre ha permesso di eliminare una brutta variabile globale.

Annidare le allocazioni

Lo schema base per l'allocazione delle risorse si può estendere alle risorse che hanno bisogno di più di una risorsa alla volta. Ci sono solo due consigli in più.

1. Liberate le risorse in ordine contrario a quello in cui le avete allocate. In questo modo non ci saranno risorse orfane, se una risorsa conteneva riferimenti a un'altra.
2. Quando allocate lo stesso insieme di risorse in punti diversi del codice, assegnatele sempre nello stesso ordine, così ridurrete le possibilità di impasse. (Se il processo A ottiene `risorsa1` e chiede la `risorsa2`, mentre il processo B ha ottenuto la `risorsa2` e sta cercando di ottenere la `risorsa1`, i due processi rimarranno in attesa per sempre.)

Non importa che tipo di risorse stiate usando (transazioni, memoria, file, thread, finestre), vale sempre lo stesso schema di base: chi alloca una risorsa deve avere la responsabilità di liberarla. In qualche linguaggio, però, possiamo sviluppare l'idea ulteriormente.

Oggetti ed eccezioni

L'equilibrio fra allocazioni e de-allocazioni ricorda il rapporto fra costruttore e distruttore di classe. La classe rappresenta una risorsa, il costruttore vi dà un particolare oggetto di quel tipo di risorsa, il distruttore lo elimina dal vostro raggio d'azione.

Se programmate in un linguaggio a oggetti, troverete forse utile incapsulare le risorse in classi. Ogni volta che avete bisogno di un particolare tipo di risorsa, istanziate un oggetto di quella classe. Quando l'oggetto esce dal raggio d'azione, o è reclamato dal raccoglitore di spazzatura, il distruttore dell'oggetto libera la risorsa.

Questo approccio ha vantaggi in particolare quando si lavora con linguaggi come C++, dove le eccezioni possono interferire con la liberazione delle risorse.

Equilibri ed eccezioni

I linguaggi che ammettono le eccezioni possono complicare la liberazione delle risorse. Se viene lanciata un'eccezione, come si può essere sicuri che tutto ciò che era stato allocato prima dell'eccezione venga rimesso a posto? La risposta dipende, in parte, dal linguaggio.

Equilibrare risorse con le eccezioni di C++

C++ supporta un meccanismo di eccezione `try…catch`. Purtroppo, questo significa che esistono sempre almeno due percorsi possibili, quando si esce da una routine che cattura e poi rilancia un'eccezione:

```
void doSomething(void) {
  Node *n = new Node;
  try {
    // fa qualcosa
  }
  catch (...) {
    delete n;
    throw;
  }
  delete n;
}
```

Notate che il nodo che creiamo viene liberato in due punti, una volta nel normale percorso di uscita dalla routine e una volta nel gestore di eccezioni. Questa è un'ovvia violazione del principio DRY ed è un problema di manutenzione che prima o poi scoppierà.

Possiamo però usare la semantica di C++ a nostro vantaggio. Gli oggetti locali vengono distrutti automaticamente quando si esce dal blocco che li racchiude. Questo ci dà un paio di opzioni. Se le circostanze lo permettono, possiamo trasformare il puntatore `n` in un effettivo oggetto `Node` sullo stack:

```
void doSomething1(void) {
  Node n;
  try {
    // fa qualcosa
  }
  catch (...) {
    throw;
```

```
  }
}
```

Qui ci affidiamo a C++ per gestire automaticamente la distruzione dell'oggetto `Node`, che sia stata lanciata un'eccezione o meno.

Se il passaggio da puntatore a oggetto non è possibile, si può ottenere lo stesso effetto avvolgendo la risorsa (in questo caso, un puntatore `Node`) in un'altra classe.

```
// Classe wrapper per risorse Node
class NodeResource {
  Node *n;
public:
  NodeResource() { n = new Node; }
  ~NodeResource() { delete n; }
  Node *operator->() { return n; }
};
void doSomething2(void) {
  NodeResource n;
  try {
    // fa qualcosa
  }
  catch (...) {
    throw;
  }
}
```

Ora la classe wrapper, `NodeResource`, garantisce che quando i suoi oggetti vengono distrutti siano distrutti anche i nodi corrispondenti. Per comodità mette a disposizione un operatore -> di dereferenziamento, in modo che i suoi utenti possano accedere direttamente ai campi nell'oggetto `Node` che contiene.

Questa tecnica è così utile che la libreria standard del C++ mette a disposizione la classe modello `auto_ptr`, che fornisce wrapper automatici per oggetti allocati dinamicamente.

```
void doSomething3(void) {
auto_ptr<Node> p (new Node);
// Accede al Node come p->...
// Node cancellato automaticamente alla fine
}
```

Equilibrare le risorse in Java

A differenza del C++, Java implementa una forma pigra di distruzione automatica degli oggetti. Gli oggetti non referenziati sono considerati candidati per la raccolta della spazzatura, e se il raccoglitore di spazzatura dovesse mai volerseli portar via, viene chiamato il loro metodo `finalize`. Questa è una comodità per gli sviluppatori, che non

vengono più chiamati in causa per la maggior parte delle perdite di memoria, ma rende difficile implementare una pulizia delle risorse con lo schema del C++. Per fortuna, chi ha progettato il linguaggio Java ha saggiamente inserito una caratteristica che compensa, la clausola `finally`. Quando un blocco `try` contiene una clausola `finally`, si ha la certezza che il codice in quella clausola venga eseguito se viene eseguito un qualsiasi enunciato nel blocco `try`. Non importa se viene lanciata un'eccezione (e neanche se il codice nel blocco `try` esegue un `return`), il codice nella clausola `finally` verrà sempre eseguito. Questo significa che possiamo equilibrare il nostro uso delle risorse con codice come questo:

```
public void doSomething() throws IOException {
  File tmpFile = new File(tmpFileName);
  FileWriter tmp = new FileWriter(tmpFile);
  try {
    // fa un po' di lavoro
  }
  finally {
    tmpFile.delete();
  }
}
```

La routine usa un file temporaneo, che vogliamo eliminare, indipendentemente da come esce la routine. Il blocco `finally` ci permette di esprimerlo in modo conciso.

Quando non si possono equilibrare le risorse

Ci sono occasioni in cui lo schema base di allocazione delle risorse semplicemente non è appropriato. Succede in programmi che usano strutture di dati dinamiche. Una routine alloca un'area di memoria e la collega a qualche struttura più grande, dove può rimanere per un po' di tempo.

Il trucco qui è stabilire una invariante semantica per l'allocazione di memoria. Dovete decidere chi è responsabile dei dati in una struttura di dati aggregata. Che cosa succede quando liberate la struttura di livello più alto? Avete tre possibilità.

1. La struttura di livello più alto ha la responsabilità di liberare anche tutte le sottostrutture che contiene. Queste strutture poi eliminano

ricorsivamente i dati che contengono, e così via.
2. La struttura di livello più alto viene semplicemente liberata. Tutte le strutture a cui puntava (e che non hanno riferimenti altrove) sono orfane.
3. La struttura di livello più alto si rifiuta di liberarsi se contiene qualche sottostruttura.

La scelta dipende dalle circostanze di ogni singola struttura di dati. Però dovete esplicitarla in ciascun caso e implementare coerentemente la vostra decisione. Implementare queste opzioni in un linguaggio procedurale come il C può essere un problema: le strutture di dati stesse non sono attive. In queste circostanze preferiamo scrivere per ciascuna struttura importante un modulo che fornisca gli strumenti standard di allocazione e liberazione per quella struttura. (Questo modulo può anche mettere a disposizione strumenti come la stampa del debug, la serializzazione, la deserializzazione gli hook trasversali.)

Infine, se tener traccia delle risorse diventa complicato, potete scrivere la vostra forma di raccolta automatica e limitata della spazzatura implementando uno schema di conteggio dei riferimenti sugli oggetti allocati dinamicamente. Il libro *More Effective C++* [Mey96] dedica una sezione a questo argomento.

Controllare l'equilibrio

Dato che i programmatori pragmatici non si fidano di nessuno, nemmeno di loro stessi, pensiamo sia sempre una buona idea costruire codice che controlla effettivamente che le risorse vengano liberate nel modo giusto. Per la maggior parte delle applicazioni, questo di norma significa produrre wrapper per ogni tipo di risorsa e usarli per tener traccia di tutto ciò che viene allocato e liberato. In certi punti del codice, la logica del programma imporrà che le risorse siano in un certo stato: usate i wrapper per verificarlo. Per esempio, un programma che gira a lungo e deve servire delle richieste, probabilmente avrà un punto particolare, in testa al suo ciclo di elaborazione principale, in cui aspetta l'arrivo della richiesta successiva. Quello è un buon posto per assicurarsi che l'uso delle risorse non sia aumentato dall'ultima esecuzione del ciclo.

A un livello più basso, ma non meno utile, potete investire in strumenti che (fra le altre cose) tengono sotto controllo i programmi in esecuzione per identificare perdite di memoria. PurifyPlus (http://teamblue.unicomsi.com/products/purifyplus/) e Insure++ (http://www.parasoft.com/) sono due scelte molto diffuse.

Vedi anche

- *Progettare per contratto*, in questo Capitolo
- *Programmazione assertiva*, in questo Capitolo
- *Disaccoppiamento e legge di Demetra*, Capitolo 5

Sfide

- Anche se non c'è modo di essere sicuri di liberare sempre tutte le risorse, certe tecniche di progetto, se applicate coerentemente, saranno d'aiuto. Nel testo abbiamo esaminato come definire un'invariante semantica per le principali strutture di dati possa orientare le decisioni di deallocazione della memoria. Provate a pensare come *Progettazione per contratto*, agli inizi di questo capitolo, possa aiutarvi a perfezionare questa idea.

Esercizi

22. Qualche sviluppatore in C e C++ si impone di fissare un puntatore a `NULL` dopo aver liberato la memoria a cui fa riferimento. Perché è una buona idea?
23. Qualche sviluppatore in Java si impone di fissare una variabile di oggetto a `NULL` dopo aver finito di usare l'oggetto. Perché è una buona idea?

Piegarsi o spezzarsi

La vita non sta mai ferma. E nemmeno il codice che scriviamo. Per stare al passo con il ritmo quasi frenetico del cambiamento, dobbiamo fare di tutto per scrivere codice che sia il più aperto e flessibile, altrimenti scopriremo che il nostro codice diventa rapidamente obsoleto, o troppo fragile da riparare e alla fine viene lasciato indietro nella folle corsa verso il futuro.

In *Reversibiità*, nel Capitolo 2, abbiamo parlato dei pericoli delle decisioni irreversibili. In questo capitolo, vedremo come prendere decisioni *reversibili*, così che il nostro codice possa rimanere flessibile e adattabile, in vista di un futuro incerto.

Prima dobbiamo esaminare l'*accoppiamento*, ovvero le dipendenze fra moduli di codice. In *Disaccoppiamento e legge di Demetra* vedremo come tener distinti i concetti distinti, e diminuire l'accoppiamento.

Un buon modo per rimanere flessibili è scrivere *meno* codice. Ogni cambiamento del codice apre la porta alla possibilità di introdurre nuovi errori. *Metaprogrammazione* spiega come estrarre completamente dal codice i dettagli, mettendoli dove possono essere modificati con maggiore sicurezza e facilità.

In *Accoppiamento temporale*, esamineremo due aspetti del tempo in rapporto con l'accoppiamento. Dipendete dal fatto che il "tic" arrivi prima del "tac"? No, se volete rimanere flessibili.

Un concetto fondamentale nella creazione di codice flessibile è quello della separazione di un *modello* dei dati da una *vista*, o presentazione, di quel modello. Disaccoppiamo i modelli dalle visgte in *È solo una vista*.

Infine, esiste una tecnica per disaccoppiare ancora di più i moduli, fornendo un luogo di incontro in cui i moduli possono scambiarsi dati in modo anonimo e asincrono. Questo è il tema di *Lavagne*.

Armati di queste tecniche, potete scrivere codice che davvero possa adeguarsi ai tempi.

Disaccoppiamento e legge di Demetra

> Buone recinzioni fanno buoni vicini.
> - Robert Frost, *Mending Wall*

In *Ortogonalità*, nel Capitolo 2, e *Progettare per contratto*, nel Capitolo 4, abbiamo detto che è una buona cosa scrivere codice "timido". Ma la timidezza va in due direzioni: non rivelate molto di voi stessi agli altri e non interagite con troppe persone.

Spie, dissidenti, rivoluzionari e simili sono spesso organizzati in piccoli gruppi di persone, chiamati *cellule*. Tutti gli individui di una cellula possono conoscersi fra loro, ma non sanno nulla dei membri di altre cellule. Se viene scoperta una cellula, non c'è siero della verità che permetta di ricavare dai suoi membri i nomi di altri al di fuori della cellula. L'eliminazione delle interazioni fra cellule protegge tutti.

Ci sembra un buon principio da applicare anche alla codifica. Organizzate il vostro codice in cellule (moduli) e limitate l'interazione fra di essi. Se un modulo viene compromesso e deve essere sostituito, gli altri moduli devono poter andare avanti da soli.

Ridurre al minimo l'accoppiamento

Che cosa c'è di male nell'avere moduli che sanno gli uni degli altri? Niente, in linea di principio - non dobbiamo essere paranoici come spie o dissidenti. Però dobbiamo stare attenti a *quanti* sono gli altri moduli con cui interagiamo e, cosa ancora più importante, a *come* siamo finiti a interagire con loro.

Supponiamo che stiate ristrutturando casa, o costruendo una casa da zero. Di solito in questi casi ci si accorda con un'impresa che faccia il lavoro, ma quell'impresa può svolgere i lavori (in tutto o in parte) direttamente oppure offrirli a vari sottofornitori. In quanto clienti, non

siete coinvolti direttamente nelle trattative con questi terzi - l'impresa principale si assume il compito in vece vostra.

Vogliamo seguire lo stesso modello nel software. Se chiediamo un servizio a un oggetto, vorremmo che quel servizio venisse svolto per nostro conto. Non vogliamo che l'oggetto ci rifili un oggetto terzo con cui trattare per ottenere il servizio che ci serve.

Per esempio, supponiamo che scriviate una classe che genera un grafico a partire dai dati di un sensore scientifico. Avete sensori dispersi in tutto il mondo che registrano i dati e ciascun oggetto *registratore* contiene un oggetto *location* che ne dà la posizione e il fuso orario. Volete permettere ai vostri utenti di selezionare un sensore e di ottenere un grafico dei suoi dati, etichettato con il fuso orario corretto. Potreste scrivere

```
public void plotDate(Date aDate, Selection aSelection) {
  TimeZone tz =
    aSelection.getRecorder().getLocation().getTimeZone();
  ...
}
```

Ora però la routine che genera il grafico è inutilmente accoppiata con *tre* classi, `Selection`, `Recorder` e `Location`. Questo stile di codifica aumenta drasticamente il numero delle classi da cui dipende la nostra classe. Perché non va bene? Aumenta il rischio che un cambiamento da qualche altra parte, che non ha nulla a che fare con quello che facciamo qui, influenzi il *nostro* codice. Per esempio, se Fred apporta un cambiamento a `Location`, in modo che non contenga più direttamente una `TimeZone`, dove cambiare anche il vostro codice.

Anziché andare a scavare direttamente in una gerarchia, chiedete solo quello di cui avete bisogno direttamente:

```
public void plotDate(Date aDate, TimeZone aTz) {
   ...
}
plotDate(someDate, someSelection.getTimeZone());
```

Abbiamo aggiunto a `Selection` un metodo perché recuperi il fuso orario per conto nostro: alla routine per disegnare il grafico non interessa se il fuso orario arriva direttamente da `Recorder`, da qualche oggetto contenuto in `Recorder`, o se `Selection` si è inventata completamente un fuso orario diverso. La routine di selezione, per parte sua, dovrà probabilmente chiedere solo al registratore quale sia il suo fuso orario, lasciandogli il compito di recuperarlo dall'oggetto `Location` che contiene.

Percorrere direttamente le relazioni fra oggetti può portare rapidamente a un'esplosione combinatoria di relazioni di dipendenza. (Se *n* oggetti sanno tutto gli uni degli altri, un cambiamento a un solo oggetto può far sì che tutti gli altri *n* - 1 oggetti richiedano cambiamenti.) Si possono vedere sintomi di questo fenomeno in vari modi.

1. Grandi progetti C o C++ in cui il comando per collegare uno unit test è più lungo del programma di test stesso.
2. Cambiamenti "semplici" a un modulo che si propagano attraverso moduli che non dovrebbero avere alcun rapporto nel sistema.
3. Sviluppatori che hanno paura a modificare il codice perché non sono sicuri di che cosa potrebbe esserne influenzato.

Sistemi con molte dipendenze non necessarie sono molto difficili (e costosi) da manutenere, e tendono a essere molto instabili. Per mantenere le dipendenze al minimo, useremo la *legge di Demetra* per progettare i nostri metodi e le nostre funzioni.

La legge di Demetra per le funzioni

La legge di Demetra per le funzioni [LH89] cerca di ridurre al minimo l'accoppiamento fra moduli in qualsiasi programma. Cerca di impedirvi di raggiungere un oggetto per avere accesso ai metodi di un terzo oggetto. La legge è riassunta nella Figura 5.1.

Scrivendo codice "timido" che rispetta il più possibile la legge di Demetra possiamo raggiungere il nostro obiettivo.

SUGGERIMENTO 36
Riducete al minimo l'accoppiamento fra moduli.

Cambia davvero qualcosa?

Anche se in teoria sembra tutto bello, seguire la legge di Demetra aiuta davvero a creare codice più manutenibile?

Alcuni studi hanno mostrato [BBM97] che le classi in C++ con *insiemi di risposta* più grandi sono più esposte agli errori delle classi con insiemi

di risposta più piccoli (un *insieme di risposta* o *response set* è definito dal numero delle funzioni invocate direttamente dai metodi della classe).

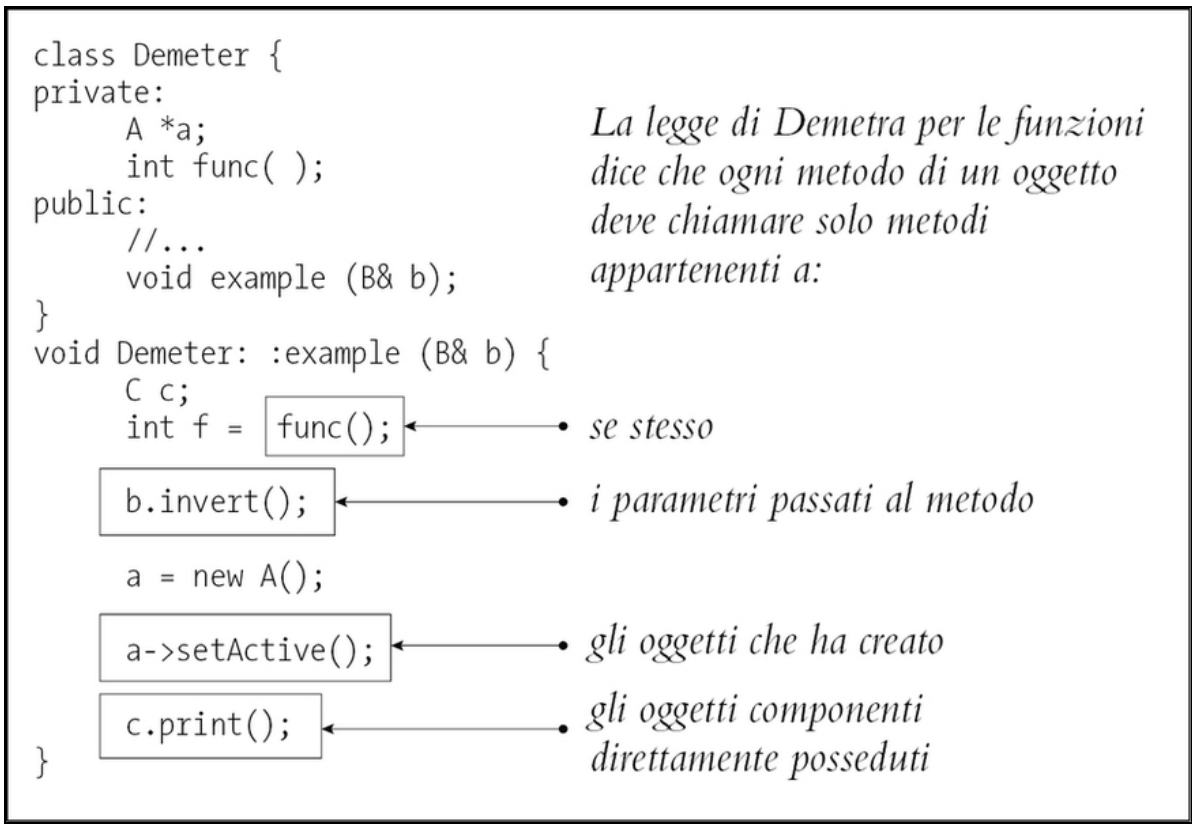


Figura 5.1 Legge di Demetra per le funzioni.

Seguire la legge di Demetra riduce la dimensione dell'insieme di risposta della classe chiamante, perciò ne segue che le classi progettate in questo tenderanno ad avere anche meno errori (vedi [URL56] per altri testi e informazioni sul progetto Demetra).

L'uso della legge di Demetra renderà il codice più adattabile e robusto, ma con un costo: in quanto "impresa principale", il vostro modulo deve delegare e gestire tutti i sottofornitori direttamente, senza chiamare in causa i clienti del vostro modulo. In pratica, questo significa che dovrete scrivere un gran numero di metodi wrapper, i quali semplicemente inoltreranno la richiesta a un delegato. Questi metodi imporranno al momento dell'esecuzione un costo e un sovraccarico per lo spazio, che in qualche applicazione possono essere significativi, o addirittura proibitivi.

Come per ogni tecnica, dovete bilanciare pro e contro per la *vostra* particolare applicazione. Nella progettazione di schemi di database, è

pratica comune “denormalizzare” lo schema per migliorare le prestazioni: violare le regole della normalizzazione per ottenere in cambio più velocità. Si può scendere a un compromesso simile anche qui. In effetti, ribaltando la legge di Demetra e accoppiando *strettamente* più moduli, si può ottenere un miglioramento di prestazioni significativo. Purché sia ben chiaro, e sia accettabile che quei moduli siano accoppiati, il progetto va bene. Altrimenti vi potreste essere messi sulla strada per un futuro fragile e non flessibile, o potreste non avere alcun futuro.

Disaccoppiamento fisico

In questa sezione ci siamo occupati in prevalenza di come progettare per mantenere le cose logicamente disaccoppiate entro i sistemi. Esiste però un altro tipo di interdipendenza che diventa molto significativo al crescere delle dimensioni dei sistemi. Nel suo libro *Large-Scale C++ Software Design* [Lak96], John Lakos affronta i problemi legati alle relazioni fra file, directory e librerie che costituiscono un sistema. I grandi progetti che ignorano questi problemi di *progettazione fisica* finiscono per arrivare a cicli di build che si misurano in giorni e unit test che possono trascinarsi nel sistema come codice di supporto, fra gli altri problemi. Lakos sostiene, ed è convincente, che la progettazione logica e fisica deve procedere in tandem: rimediare ai danni prodotti a un grande corpus di codice da dipendenze cicliche è estremamente difficile. Vi consigliamo questo libro se siete coinvolti in progetti su grande scala, anche se il C++ non è il linguaggio della vostra implementazione.

Vedi anche

- *Ortogonalità*, Capitolo 2
- *Reversibilità*, Capitolo 2
- *Progettazione per contratto*, Capitolo 4
- *Come equilibrare le risorse*, Capitolo 4
- *È solo una vista*, in questo Capitolo
- *Team pragmatici*, Capitolo 8
- *Test senza pietà*, Capitolo 8

Sfide

- Abbiamo analizzato come l’uso della delega faciliti il rispetto della legge di Demetra e quindi riduca l’accoppiamento. Scrivere tutti i metodi necessari per inoltrare le chiamate alle classi delegate però è noioso e sbagliare è facile. Quali sono i vantaggi e gli svantaggi della scrittura di un preprocessore che generi automaticamente queste chiamate? Il preprocessore dovrebbe essere eseguito solo una volta, o dovrebbe essere utilizzato come parte della build?

Esercizi

24. Abbiamo esaminato il concetto di disaccoppiamento fisico nella Figura 5.1. Quale dei seguenti file di intestazione C++ è accoppiato più strettamente al resto del sistema?

```
person1.h:   person2.h:
  #include "date.h"       class Date;
  class Person1 {     class  Person2 {
  private:       private:
    Date myBirthdate;       Date *myBirthdate;
  public:     public:
    Person1(Date &birthDate);            Person2(Date &birthDate
    // ...      // ...
```

25. Per l'esempio qui sotto e per quelli negli Esercizi 26 e 27, stabilite se le chiamate di metodi mostrate sono consentite, in base alla legge di Demetra. Questo primo esempio è in Java.

```
public void showBalance(BankAccount acct) {
  Money amt = acct.getBalance();
  printToScreen(amt.printFormat());
}
```

26. Anche questo esempio è in Java.

```
public class Colada {
  private Blender myBlender;
  private Vector myStuff;
  public Colada() {
    myBlender = new Blender();
    myStuff = new Vector();
  }
  private void doSomething() {
    myBlender.addIngredients(myStuff.elements());
  }
}
```

27. Questo esempio è in C++.

```
void processTransaction(BankAccount acct, int) {
  Person *who;
  Money amt;
  amt.setValue(123.45);
  acct.setBalance(amt);
  who = acct.getOwner();
  markWorkflow(who->name(), SET_BALANCE);
}
```

Metaprogrammazione

Non c'è grado di genialità che possa vincere la preoccupazione per i dettagli.
- *Ottava legge di Levy*

I dettagli fanno confusione nel nostro bel codice - in particolare se cambiano spesso. Ogni volta che dobbiamo entrare e cambiare il codice per rispondere a qualche cambiamento della logica di business, o della

normativa, o dei gusti personali del manager del giorno, corriamo il rischio di rompere il sistema - di introdurre un nuovo errore.

Perciò noi diciamo: “fuori i dettagli!”. Tirateli fuori dal codice. E intanto che ci siamo, possiamo rendere il codice altamente configurabile e “morbido”, cioè facilmente adattabile ai cambiamenti.

Configurazione dinamica

Innanzitutto, vogliamo rendere i nostri sistemi molto configurabili. Non solo per cose come i colori sullo schermo e il testo di invito, ma anche per aspetti molto interni come la scelta di algoritmi, prodotti di database, tecnologia middleware e stile di interfaccia utente. Questi elementi è bene che siano implementati come opzioni di configurazione, non per integrazione o ingegnerizzazione.

SUGGERIMENTO 37
Configurate, non integrate.

Usate *metadati* per descrivere opzioni di configurazione per un’applicazione, parametri di accordatura, preferenze degli utenti, directory di installazione e così via.

Che cosa sono esattamente i metadati? In senso stretto, i metadati sono dati su dati. L’esempio più comune probabilmente è uno schema di database o un dizionario di dati. Uno schema contiene dati che descrivono i campi (colonne) in termini di nomi, lunghezza e altri attributi. Dovete poter accedere a queste informazioni e manipolarle come fareste con qualsiasi altro dato nel database.

Qui usiamo il termine nel suo senso più generale: metadati sono tutti i dati che descrivono l’applicazione - come deve girare, quali risorse deve usare e così via. Normalmente ai metadati si accede e li si usa al momento dell’esecuzione, non in quello della compilazione. Usate i metadati continuamente - o almeno lo fanno i vostri programmi. Supponiamo che facciate clic su un’opzione per nascondere la barra degli strumenti del browser web. Il browser memorizzerà quella preferenza, sotto forma di metadato, in qualche tipo di database interno.

Il database può essere in un formato proprietario, oppure può usare un meccanismo standard. In Windows si usano normalmente o un file di inizializzazione (con il suffisso `.ini`) o voci nel Registro di sistema. Sotto

Unix, lo X Window System offre funzionalità simili con i file Application Default. Java usa file Property. In tutti questi ambienti, specificate una chiave per recuperare un valore. In alternativa, implementazioni dei metadati più potenti e flessibili usano un linguaggio di scripting incorporato (vedi *Linguaggi di settore* nel Capitolo 2).

Il browser Netscape ha implementato le preferenze con entrambe le tecniche. Nella versione 3, le preferenze veniva salvate come semplici coppie chiave/valore:

```
SHOW_TOOLBAR: False
```

Poi, nella versione 4 le preferenze erano più simili a JavaScript:

```
user_pref("custtolbar.Browser.Navigation_Toolbar.opne", false)
```

Applicazioni guidate dai metadati

Noi però vogliamo andare oltre l'uso dei metadati per le semplici preferenze. Vogliamo configurare e guidare il più possibile l'applicazione mediante metadati. Il nostro scopo è pensare dichiarativamente (specificare *che cosa* deve essere fatto, non *come*) e creare programmi molto dinamici e adattabili. Lo facciamo adottando una regola generale: programmare per il caso generale e mettere le specifiche da qualche altra parte - al di fuori della base di codice compilata.

SUGGERIMENTO 38
Mettete le astrazioni nel codice, i dettagli nei metadati.

Questa impostazione presenta numerosi vantaggi.

- Vi costringe a disaccoppiare la vostra progettazione, e il risultato sarà un programma più flessibile e adattabile.
- Vi costringe a creare progetti più robusti, più astratti, rimandando i dettagli - spedendoli fuori dal programma.
- Potete personalizzare l'applicazione senza ricompilarla. Potete anche usare questo livello di personalizzazione per fornire facili modi per aggirare errori critici in sistemi di produzione in opera.
- I metadati si possono esprimere in un modo più vicino al dominio del problema, rispetto a quello possibile con un linguaggio di programmazione "generalista" (vedi *Linguaggi di settore*, nel Capitolo 2).

- Potreste anche essere in grado di implementare più progetti diversi utilizzando lo stesso motore applicativo, ma con metadati diversi.

Vogliamo rimandare fino all'ultimo momento la definizione della maggior parte dei dettagli e lasciare i dettagli il più "morbidi" (cioè il più facili da modificare) possibile. Mettendo a punto una soluzione che ci consenta di apportare cambiamenti rapidamente, abbiamo migliori possibilità di affrontare bene il diluvio di cambiamenti di direzione che mette in crisi molti progetti (vedi *Reversibilità*, nel Capitolo 2).

Logica di business

Dunque, avete reso la scelta del motore di database un'opzione di configurazione e avete fornito metadati per determinare lo stile dell'interfaccia utente. Possiamo fare di più? Certo.

Poiché le politiche e le regole di business sono l'aspetto del progetto che è più facile cambi, ha senso mantenerle in un formato molto flessibile. Per esempio, la vostra applicazione per gli acquisti può tener conto di varie politiche aziendali. Magari pagate i piccoli fornitori a 45 giorni e i più grandi a 90 giorni. Fate in modo che le definizioni dei tipi di forniti, e anche i tempi di scadenza dei pagamenti, siano configurabili. Cogliete l'occasione per generalizzare.

Magari state scrivendo un sistema con orrendi requisiti per il flusso di lavoro. Le azioni partono e si fermano in base a complesse regole di business (che cambiano spesso). Pensate a codificarle in qualche tipo di sistema basato su regole (o esperto), incorporato nell'applicazione. In questo modo potrete configurarla scrivendo regole, non ritagliando codice.

Logica meno complessa può essere espressa con un mini-linguaggio, eliminando la necessità di ricompilare e rimettere in esercizio quando l'ambiente cambia. (Vedete un esempio sotto *Linguaggi di settore*, nel Capitolo 2.)

Un esempio: Enterprise Java Beans

Enterprise Java Beans (EJB) è un framework per semplificare la programmazione in un ambiente distribuito, basato sulle transazioni. Lo citiamo qui perché illustra come si possano usare metadati sia per configurare le applicazioni *sia* per ridurre la complessità di scrittura del codice.

Supponiamo che vogliate creare del software Java che partecipi a transazioni fra macchine diverse, fra database di produttori diversi e con modelli di thread e di bilanciamento del carico differenti.

Quando configurare

Come abbiamo detto in *Il potere del puro testo*, nel Capitolo 3, consigliamo di rappresentare i metadati di configurazione come puro testo - rende la vita molto più semplice.

Ma quando un programma deve leggere questa configurazione? Molti programmi esamineranno queste cose solo all'avvio, il che è una sfortuna. Se dovete modificare la configurazione, questo vi costringe a riavviare l'applicazione. Un metodo più flessibile è scrivere programmi che possano ricaricare la propria configurazione mentre sono in esecuzione. Questa flessibilità ha un costo: è più complessa da implementare.

Pensate allora come verrà usata la vostra applicazione: se è un lungo processo server, vorrete avere qualche modo per rileggere e applicare i metadati mentre il programma è in esecuzione. Per una piccola applicazione GUI client che si riavvia rapidamente potrebbe non essere necessario.

Questo fenomeno non è limitato al codice applicativo. Tutti diamo qualche segno di insofferenza quando qualche sistema operativo ci costringe al riavvio se installiamo qualche semplice applicazione o cambiamo un parametro innocuo.

La buona notizia è che non dovete preoccuparvi di nessuna di queste cose. Scrivete un *bean* - un oggetto autocontenuto che segue determinate convenzioni - e lo collocate in un *contenitore di bean* che gestisce gran parte dei dettagli di basso livello per vostro conto. Potete scrivere codice per un bean senza includere operazioni transazionali o gestione di thread; EJB usa i metadati per specificare come debbano essere gestite le transazioni.

L'allocazione dei thread e il bilanciamento del carico sono specificati come metadati al servizio di transazione sottostante, utilizzato dal contenitore. Questa separazione consente una maggiore flessibilità per configurare dinamicamente l'ambiente, al momento dell'esecuzione.

Il contenitore del bean può gestire transazioni per conto del bean in uno fra molti stili diversi (c'è anche la possibilità di controllare i propri commit e rollback). Tutti i parametri che influenzano il comportamento del bean sono specificati nel *descrittore di messa in esercizio*

(*deployment descriptor*) del bean, un oggetto serializzato che contiene i metadati che ci servono.

Sistemi distribuiti come EJB hanno aperto la strada a un nuovo mondo di sistemi dinamici configurabili.

Configurazione cooperativa

Abbiamo parlato di utenti e sviluppatori che configurano applicazioni dinamiche. Ma che cosa succede se fate in modo che le applicazioni si configurino a vicenda, ovvero software che adatta se stesso al proprio ambiente? Quella di configurazione non pianificata, sull'onda del momento, di software preesistente è un'idea potente.

Ci sono sistemi operativi che già si configurano in funzione dell'hardware al momento dell'avvio e browser web che si aggiornano automaticamente con nuovi componenti.

Le vostre applicazioni di maggiori dimensioni probabilmente hanno già problemi con la gestione di versioni diverse dei dati e release diverse di librerie e sistemi operativi. Forse un approccio più dinamico sarà utile.

Non scrivere codice dodo

Senza metadati, il vostro codice non sarà tanto adattabile o flessibile quanto potrebbe essere. È una brutta cosa? Beh, qui nel mondo reale, le specie che non si adattano scompaiono.

Il dodo non si è adattato alla presenza degli esseri umani e dei loro animali da allevamento sull'isola di Mauritius, e rapidamente si è estinto (certo, ha contribuito il fatto che i coloni pestassero a morte questi placidi uccelli con le mazze, tanto per sport). È stata la prima estinzione documentata di una specie a opera dell'uomo.

Non lasciate che il vostro progetto (o la vostra carriera) faccia la fine del dodo.

Vedi anche

- *Ortogonalità*, Capitolo 2
- *Reversibilità*, Capitolo 2
- *Linguaggi di settore*, Capitolo 2
- *Il potere del puro testo*, Capitolo 3

Sfide

- Per il progetto che state seguendo, provate a pensare quanta parte dell'applicazione potrebbe essere estratta dal programma stesso e inserita nei metadati. Come apparirebbe il "motore" risultante? Sareste in grado di riusarlo nel contesto di un'applicazione diversa?

Esercizi

28. Quali delle cose seguenti sarebbe meglio rappresentare come codice in un programma, e quali esternamente come metadati?

1. Assegnazioni delle porte di comunicazione.
2. Il supporto dell'editor per l'evidenziazione della sintassi di vari linguaggi.
3. Il supporto dell'editor per dispositivi grafici diversi.
4. Una macchina a stati per un analizzatore sintattico o uno scanner.
5. Valori e risultati campione da usare in unit test.

Accoppiamento temporale

Il tempo è un aspetto spesso ignorato delle architetture software. L'unico tempo di cui ci preoccupiamo è quello delle scadenze, il tempo che rimane prima della consegna - ma non è di questo che vogliamo parlare. Parliamo invece del ruolo del tempo come elemento progettuale del software stesso. Due aspetti del tempo sono importanti per noi: la concorrenza o concomitanza (cose che accadono allo stesso tempo) e l'ordinamento (le posizioni relative delle cose rispetto al tempo).

Di solito non affrontiamo la programmazione avendo in mente questi due aspetti. Quando qualcuno si mette lì a progettare un'architettura o a scrivere un programma, le cose sono tendenzialmente lineari. Questo è il modo in cui pensa la maggior parte delle persone: *fai questo* e poi sempre *fai quello*. Pensare in questo modo però porta all'*accoppiamento temporale*. Il metodo A va sempre chiamato prima del metodo B; si può eseguire un report solo alla volta; bisogna aspettare che lo schermo si ridisegni perché venga ricevuto il clic su un pulsante. Deve esserci tic prima di tac.

Questa impostazione non è molto flessibile, e non è molto realistica.

Dobbiamo consentire la concorrenza (non parleremo di programmazione parallela qui; un buon manuale di informatica vi fornirà le basi che vi servono) e dobbiamo pensare a disaccoppiare le dipendenze temporali o d'ordine. Così facendo, guadagniamo in flessibilità e riduciamo le dipendenze basate sul tempo in molte aree dello sviluppo: analisi del flusso di lavoro, architettura, progettazione e messa in esercizio.

Flusso di lavoro

In molti progetti, nell'ambito dell'analisi dei requisiti è necessario modellizzare e analizzare i flussi di lavoro degli utenti. Vogliamo scoprire che cosa *può* succedere contemporaneamente, e che cosa deve succedere in un ordine ben definito. Un modo per farlo è catturare la descrizione del flusso di lavoro con una notazione come i *diagrammi di attività UML* (per maggiori informazioni sui tipi di diagrammi UML, vedi [FS97]).

Un diagramma di attività è costituito da un insieme di azioni disegnate come rettangoli dagli angoli arrotondati. La freccia che esce da un'azione porta a un'altra azione (che può iniziare una volta completata la prima) o a una linea spessa chiamata *barra di sincronizzazione*. Quando *tutte* le azioni che portano a una barra di sincronizzazione sono complete, si può procedere seguendo una delle frecce che escono dalla barra. Un'azione verso cui non punta alcuna freccia può essere avviata in qualsiasi momento.

Potete usare i diagrammi di attività per massimizzare il parallelismo, identificando attività che si *potrebbero* eseguire in parallelo, ma non vengono eseguite così.

SUGGERIMENTO 39
Analizzate il flusso di lavoro per migliorare la concorrenza.

Per esempio, nel nostro progetto per il miscelatore (Esercizio 17), gli utenti potrebbero descrivere inizialmente il loro flusso di lavoro corrente come segue.

1. Apri il miscelatore.
2. Apri una confezione di piña colada.
3. Metti la piña colada nel miscelatore.

4. Misura ½ tazza di rhum bianco.
5. Versa il rhum.
6. Aggiungi 2 tazze di ghiaccio.
7. Chiudi il miscelatore.
8. Miscela per 2 minuti.
9. Apri il miscelatore.
10. Prendi i bicchieri.
11. Aggiungi degli ombrellini rosa.
12. Servi.

Anche se descrivono queste azioni in serie, e possono anche compierle effettivamente in sequenza, notiamo che molte potrebbero essere seguite in parallelo, come si vede nel diagramma di attività della Figura 5.2.

Può davvero essere illuminante capire dove vi sono realmente delle dipendenze. In questo caso, le attività del livello più alto (1, 2, 4, 10 e 11) possono tutte avvenire insieme, inizialmente. Le attività 3, 5 e 6 possono avvenire in parallelo più tardi. Se partecipaste a una gara di preparazione di piña colada, queste ottimizzazioni potrebbero cambiare tutto.

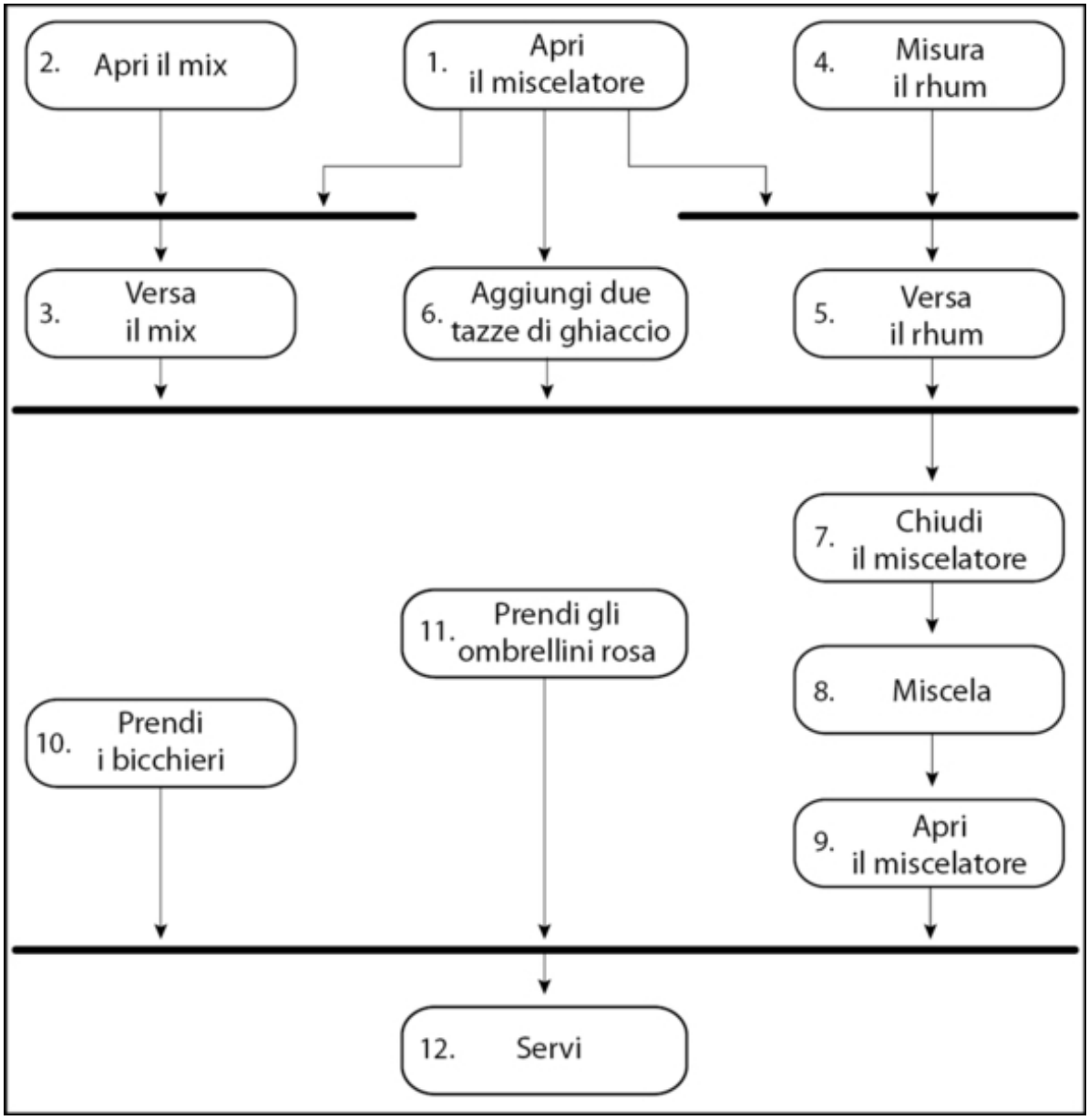


Figura 5.2 Diagramma di attività UML: preparazione di una piña colada.

Architettura

Qualche anno fa abbiamo scritto un sistema per l'elaborazione di transazioni online (OLTP, *On-Line Transaction Processing*). Nel caso più semplice, tutto quello che doveva fare il sistema era leggere una richiesta ed elaborare la transazione rispetto al database. Noi però abbiamo scritto

un'applicazione distribuita in multielaborazione e a tre livelli: ciascun componente era un'entità indipendente che girava in concorrenza con tutti gli altri componenti. Potrebbe sembrare più lavoro, ma non lo è stato: sfruttando il vantaggio del disaccoppiamento temporale, è stato *più facile* scriverlo. Diamo un'occhiata più da vicino a questo progetto.

Il sistema accetta in ingresso richieste da un gran numero di linee di comunicazione dati ed elabora le transazioni rispetto a un database di back-end.

Il progetto teneva conto dei vincoli seguenti.

- Il completamento delle operazioni su database richiede un tempo relativamente lungo.
- Per ogni transazione, non dobbiamo bloccare i servizi di comunicazione mentre viene elaborata una transazione sul database.
- Le prestazioni del database peggiorano se ci sono troppe sessioni concorrenti.
- Su ogni linea di dati sono in stato di avanzamento concorrente più transazioni.

La soluzione che ci ha dato le prestazioni migliori e l'architettura più pulita era simile a quella della Figura 5.3.

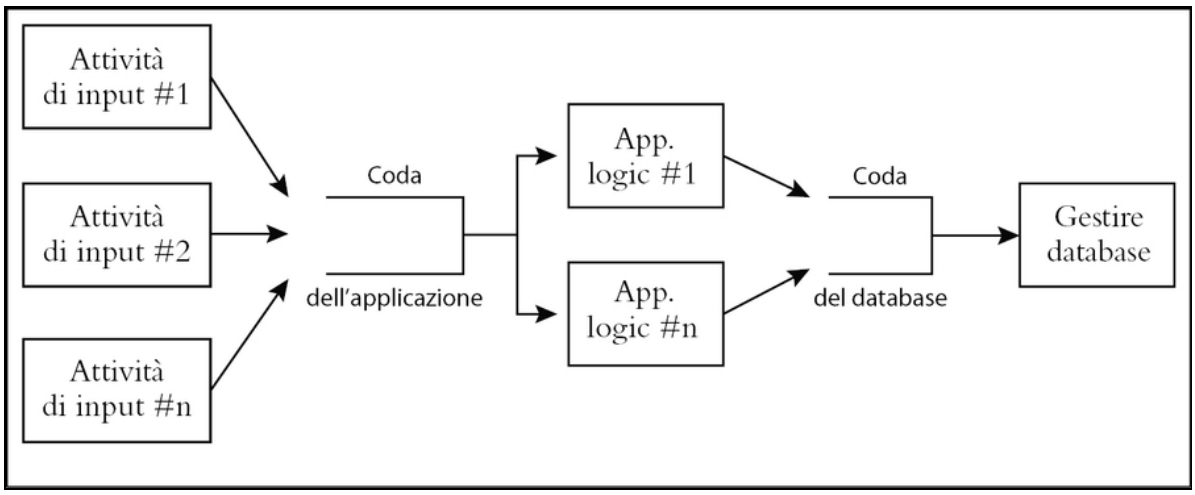


Figura 5.3 Panoramica dell'architettura OLTP.

Ogni riquadro rappresenta un processo distinto; i processi comunicano via code di lavoro. Ciascun processo di input tiene sotto controllo una

linea di comunicazione in ingresso e fa richieste al server applicativo. Tutte le richieste sono asincrone: non appena il processo di input fa la sua richiesta corrente, torna a tenere sotto controllo la linea, nel caso arrivi altro traffico. Analogamente, il server applicativo fa le richieste del processo di database e riceve una notifica quando la singola transazione è completa. (Anche se qui mostriamo il database come un'unica entità monolitica, non lo è. Il software di database è suddiviso in vari processi e thread client, ma questo è gestito internamente dal software di database e non rientra nel nostro esempio.)

Questo esempio inoltre mostra anche come ottenere un bilanciamento "brutale" del carico fra più processi consumatori: è il modello del *consumatore affamato*.

In questo modello, si sostituisce il pianificatore centrale con varie attività di consumo indipendenti e una coda di lavoro centralizzata. Ogni attività consumatrice prende un pezzo della coda di lavoro e si dedica al suo lavoro di elaborazione. Quando un'attività finisce il suo lavoro, torna alla coda per procurarsene altro. In questo modo, se una particolare attività procede lentamente, altre possono compensare e ciascun componente procede secondo il proprio ritmo: è disaccoppiato temporalmente dagli altri.

SUGGERIMENTO 40
Progettate usando i servizi.

Anziché componenti, abbiamo creato dei *servizi* ovvero oggetti concorrenti e indipendenti dietro interfacce coerenti e ben definite.

Progettazione per la concorrenza

L'uso crescente di Java come piattaforma ha esposto un numero maggiore di sviluppatori alla programmazione *multithread*. Programmare con i thread però impone qualche vincolo progettuale, ed è una buona cosa. Questi vincoli in effetti sono così utili che vogliamo rispettarli ogni volta che programmiamo. Ci aiuterà a disaccoppiare il nostro codice e a combattere la *programmazione per coincidenza* (vedi il Capitolo 6).

Con codice lineare, è facile fare ipotesi che portano a una programmazione trascurata, ma la concorrenza costringe a pensare le cose con un po' più di attenzione - non siete più soli alla festa. Dato che

ora le cose possono succedere “allo stesso tempo”, potete di colpo vedere certe dipendenze basate sul tempo.

Tanto per cominciare, ogni variabile globale o statica deve essere protetta da accessi concorrenti. È un buon momento per chiedervi *perché* mai avete bisogno di una variabile globale. Inoltre, dovete essere sicuri di presentare informazioni di stato coerenti, indipendentemente dall’ordine delle chiamate. Per esempio, quando è valido interrogare lo stato del vostro oggetto? Se l’oggetto è in uno stato non valido nell’intervallo fra certe chiamate, potreste confidare nella coincidenza che nulla chiami il vostro oggetto proprio in quel momento.

Supponiamo che abbiate un sottosistema a finestre in cui i widget vengono prima creati e poi visualizzati, in due passi distinti. Non è consentito impostare lo stato nel widget finché non viene visualizzato. A seconda di come è impostato il codice, forse vi fidate del fatto che nessun altro oggetto possa usare il widget creato fino a quando non l’avete visualizzato sullo schermo.

Questo però può non essere vero in un sistema concorrente. Gli oggetti devono essere sempre in uno stato valido quando vengono chiamati, e possono essere chiamati nei momenti più strani. Dovete essere sicuri che un oggetto sia in uno stato valido in *qualunque* momento possa essere chiamato. Spesso questo problema sorge con classi che definiscono routine separate di costruzione e di inizializzazione (dove il costruttore non lascia l’oggetto in uno stato inizializzato). Utilizzare le invarianti di classe, di cui abbiamo parlato in *Progettare per contratto* (Capitolo 4), vi aiuterà a non cadere in questa trappola.

Interfacce più pulite

Pensare a concorrenza e a dipendenze temporali può portare a progettare anche interfacce più pulite. Prendete la routine di libreria C `strtok`, che suddivide una stringa in token.

Il progetto di `strtok` non è *thread safe*. (Usa dati statici per mantenere la posizione corrente nel buffer; i dati statici non sono protetti dagli accessi concorrenti, perciò non è *thread safe*. Inoltre, manipola il primo argomento che le si passa, il che può portare a brutte sorprese.) E non è la parte peggiore: considerate la dipendenza dal tempo. Dovete fare la prima chiamata a `strtok` con la variabile che volete analizzare, e tutte le chiamate

successive, invece, con un `NULL`. Se passate un valore diverso da `NULL`, la routine riavvia invece l'analisi su quel buffer. Senza nemmeno considerare i thread, supponiamo che vogliate usare `strtok` per analizzare due stringhe diverse contemporaneamente:

```
char buf1[BUFSIZ];
char buf2[BUFSIZ];
char *p, *q;
strcpy(buf1, "this is a test");
strcpy(buf2, "this ain't gonna work");
p = strtok(buf1, " ");
q = strtok(buf2, " ");
while (p && q) {
  printf("%s %snn", p, q);
  p = strtok(NULL, " ");
  q = strtok(NULL, " ");
}
```

Così com'è, questo codice non funzionerà: in `strtok` vi è uno stato implicito conservato fra una chiamata e l'altra. Dovrete usare `strtok` su un buffer alla volta.

In Java il progetto di un analizzatore sintattico di stringhe deve essere diverso. Deve essere *thread safe* e presentare uno stato coerente.

```
StringTokenizer st1 = new StringTokenizer("this is a test");
StringTokenizer st2 = new StringTokenizer("this test will work");
while (st1.hasMoreTokens() && st2.hasMoreTokens()) {
  System.out.println(st1.nextToken());
  System.out.println(st2.nextToken());
}
```

`StringTokenizer` è un'interfaccia più pulita e più manutenibile. Non contiene sorprese e non provocherà errori misteriosi in futuro, come invece potrebbe fare `strtok`.

SUGGERIMENTO 41
Progettate sempre per la concorrenza.

Messa in esercizio

Una volta progettata un'architettura con un elemento di concorrenza, diventa più facile pensare a gestire *molti* servizi concorrenti: il modello diventa pervasivo.

Ora potete essere flessibili per la messa in esercizio dell'applicazione: stand-alone, client-server o *n*-tier. Con un'architettura a servizi indipendenti, potete rendere dinamica anche la configurazione del sistema. Pianificando per la concorrenza e disaccoppiando le operazioni

nel tempo, avete tutte queste opzioni, compresa quella stand-alone, che potete anche scegliere di non rendere concorrente.

Procedere in senso opposto (cercare di aggiungere la concorrenza a un'applicazione non concorrente) è *molto* più difficile. Se progettiamo per la concorrenza, possiamo soddisfare più facilmente requisiti di scalabilità o di prestazioni quando viene il momento, e se quel momento non dovesse venire mai, avremmo comunque il vantaggio di un progetto più elegante.

Vedi anche

- *Progettare per contratto*, Capitolo 4
- *Programmazione per coincidenza*, Capitolo 6

Sfide

- Quante attività eseguite in parallelo quando vi preparate per andare al lavoro il mattino? Potreste rappresentarle in un diagramma di attività UML? Potete trovare un modo per prepararvi più in fretta aumentando la concorrenza?

È solo una vista

> Eppure, un uomo ascolta
> quel che vuol sentire
> e trascura tutto il resto
> La la la…
> - Simon and Garfunkel, *The Boxer*

Ci è stato insegnato a non scrivere un programma come un unico grande blocco, e che dovevano invece applicare il principio del *divide et impera* e separare un programma in moduli. Ogni modulo ha le proprie responsabilità: in effetti, una buona definizione di modulo (o di classe) è che ha un'unica, ben definita responsabilità.

Una volta che si suddivide un programma in moduli diversi sulla base delle responsabilità, si ha un nuovo problema. Al momento dell'esecuzione, come si fa in modo che gli oggetti si parlino a vicenda? Come si gestiscono le dipendenze logiche? Cioè, come si sincronizzano i cambiamenti di stato (o gli aggiornamenti dei valori dei dati) in questi

oggetti diversi? Bisogna farlo in un modo pulito e flessibile - non vogliamo che gli oggetti sappiano troppe cose gli uni degli altri. Vogliamo che il modulo sia come l'uomo della canzone e senta solo quel che vuol sentire.

Partiamo con il concetto di *evento*. Un evento è semplicemente un messaggio speciale che dice "è appena successo qualcosa di interessante" (l'interesse, ovviamente, sta nell'occhio di chi guarda). Possiamo usare gli eventi per segnalare a un oggetto cambiamenti avvenuti in un altro oggetto a cui può essere interessato.

L'uso degli eventi in questo modo riduce al minimo l'accoppiamento fra quegli oggetti: il mittente dell'evento non deve avere alcuna conoscenza esplicita del ricevente. In effetti, i riceventi potrebbero essere molti, ciascuno concentrato su quel che deve fare (di cui il mittente è felicemente inconsapevole).

Dobbiamo stare un po' attenti a usare gli eventi, comunque. In una delle prime versioni di Java, per esempio, una routine riceveva *tutti* gli eventi destinati a una particolare applicazione: non proprio l'impostazione migliore per facilitare la manutenzione e l'evoluzione.

Pubblicare/abbonarsi

Perché non va bene far passare tutti gli eventi per una singola routine? Perché viola l'incapsulamento degli oggetti: quella routine ora ha una conoscenza profonda delle interazioni fra molti oggetti. Inoltre aumenta l'accoppiamento, mentre noi vogliamo *diminuirlo*. Dato che gli oggetti stessi debbono conoscere questi eventi, probabilmente violerete anche il principio DRY, l'ortogonalità e forse anche qualche clausola della Convenzione di Ginevra. Forse avrete visto questo tipo di codice: di solito è dominato da un enorme enunciato `case` o da una serie di `if-then`. Possiamo fare di meglio.

Gli oggetti devono essere in grado di registrarsi per ricevere solo gli eventi di cui hanno bisogno e non bisogna mai che vengano inviati loro eventi di cui non hanno bisogno. Non vogliamo inondare di spam i nostri oggetti! Useremo invece un protocollo *publish/subscribe* (pubblicazione/abbonamento), illustrato con il *diagramma di sequenza UML* della Figura 5.4. (Vedi anche lo schema Observer in [GHJV95] per ulteriori informazioni.)

Un diagramma di sequenza mostra il flusso dei messaggi fra vari oggetti disposti in colonne. Un messaggio è rappresentato come una freccia etichettata che va dalla colonna del mittente alla colonna del ricevente. Un asterisco nell'etichetta significa che è possibile l'invio di più di un messaggio di questo tipo.

Se siamo interessati a certi eventi generati da un `Publisher`, tutto quello che dobbiamo fare è registrarci. Il `Publisher` tiene traccia di tutti gli oggetti `Subscriber` interessati: quando genera un evento interessante, chiamerà a turno ciascun `Subscriber` e gli notificherà che si è verificato quell'evento.

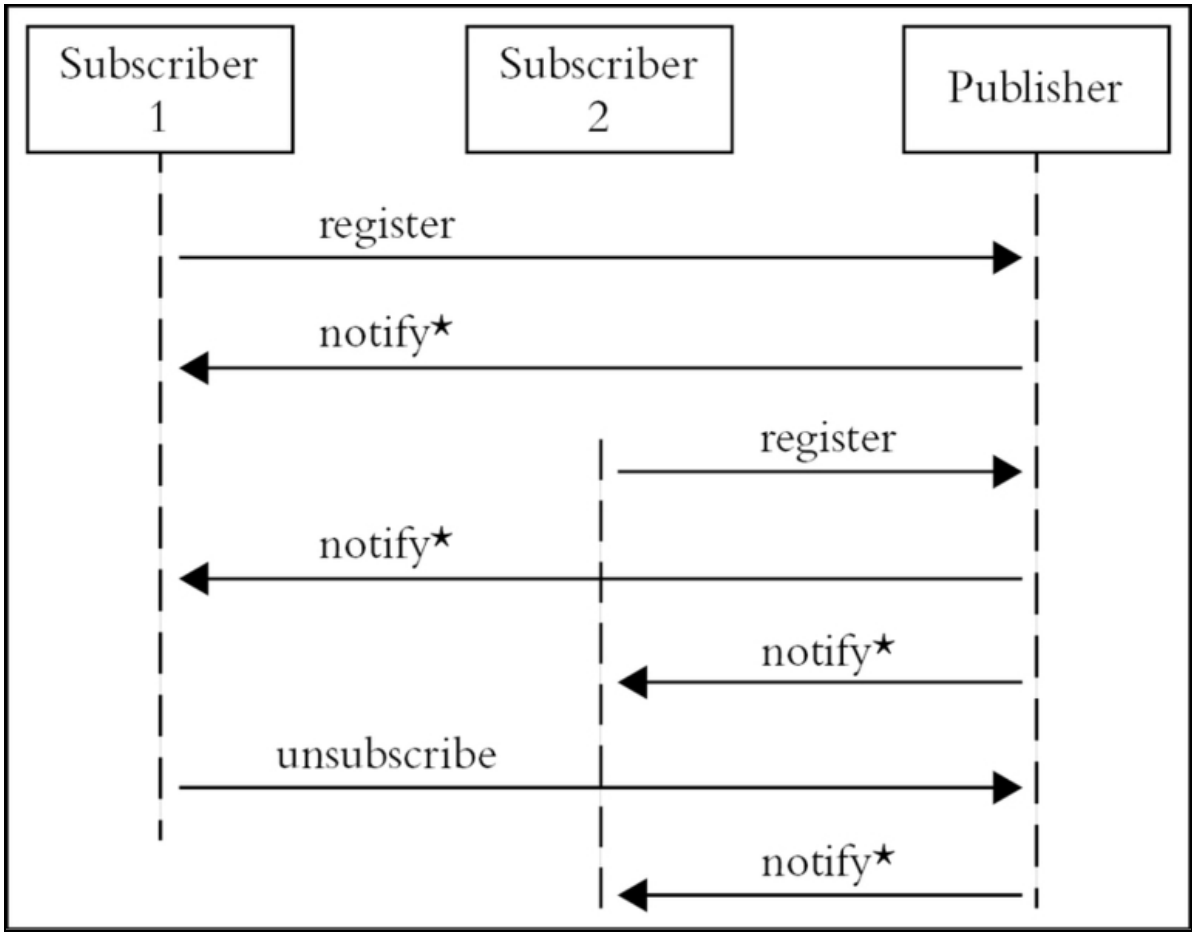


Figura 5.4 Protocollo publish/subscribe.

Esistono molte variazioni su questo tema, che rispecchiano altri stili di comunicazione. Gli oggetti possono usare lo schema pubblicazione/abbonamento su base paritetica (*peer-to-peer*, come abbiamo visto sopra); possono usare un *bus software*, dove un oggetto centralizzato mantiene il database degli ascoltatori e spedisce i messaggi

in modo opportuno. Si può avere addirittura uno schema in cui eventi critici vengono diffusi a tutti gli ascoltatori - abbonati o meno. Una possibile implementazione di eventi in un ambiente distribuito è illustrata dal CORBA Event Service, descritto nel riquadro omonimo.

Possiamo usare il meccanismo pubblicazione/abbonamento per implementare un concetto progettuale molto importante: la separazione tra un modello e le sue visualizzazioni. Cominciamo con un esempio basato su GUI, utilizzando il progetto di Smalltalk, in cui è nato questo concetto.

CORBA Event Service

Il CORBA Event Service consente agli oggetti partecipanti di inviare e ricevere notifiche di eventi tramite un bus comune, il *canale degli eventi.* Questo canale fa da arbitro nella gestione degli eventi e inoltre disaccoppia produttori e consumatori di eventi. Funziona in due modi fondamentali: *push* e *pull*.

In modalità push, i fornitori di eventi informano il canale che si è verificato un evento. Il canale poi distribuisce l'evento a tutti gli oggetti clienti che hanno registrato il loro interesse.

In modalità pull, i clienti vanno a vedere periodicamente il canale degli eventi, che a sua volta va a sondare il fornitore che offre dati di evento corrispondenti alla richiesta.

Anche se il CORBA Event Service può essere usato per implementare tutti i modelli degli eventi di cui abbiamo parlato, potete vederlo come un animale di un'altra specie. CORBA facilita le comunicazioni fra oggetti scritti in linguaggi di programmazione diversi, in esecuzione su macchine collocate in sedi geograficamente sparse e dotate di architetture differenti. Basandosi su CORBA, il servizio offre un modo disaccoppiato di interagire con le applicazioni di tutto il mondo, scritte da persone che non avete mai conosciuto, e che usano linguaggi di programmazione di cui non vi interessa nemmeno sapere il nome.

Modello-Vista-Controllo

Supponiamo di avere un'applicazione di foglio di calcolo. Oltre ai numeri nel foglio stesso, abbiamo anche un grafico che li rappresenta come istogramma e una finestra di dialogo con il totale corrente, che mostra la somma di una colonna nel foglio.

Ovviamente, non vogliamo avere tre copie distinte dei dati, perciò creiamo un *modello*: i dati stessi, con le operazioni comuni per manipolarli. Poi creiamo *viste* distinte che mostrano i dati in modi diversi: come foglio di calcolo, come grafico o in una casella dei totali. Ciascuna di queste viste può avere un suo *controllo*. La vista a grafico può avere un controllo che consente l'ingrandimento e la riduzione,

oppure la panoramica sui dati, per esempio. Nessuna di queste operazioni influenza i dati stessi, ma solamente quella vista.

Questo è il concetto fondamentale dello schema Modello-Vista-Controllo (*Model-View-Controller*, MVC): separare il modello sia dalla GUI che lo rappresenta, sia dai controlli che gestiscono la vista. (Vista e controllo sono strettamente accoppiati, e in qualche implementazione di MVC sono un componente unico.)

Facendo in questo modo, potete sfruttare alcune possibilità interessanti. Potete consentire più viste dello stesso modello di dati; potete usare visualizzatori comuni per molti modelli di dati diversi; potete addirittura prevedere più controlli, per fornire meccanismi di input non tradizionali.

SUGGERIMENTO 42
Separate le viste dai modelli.

Allentando l'accoppiamento fra modello e vista/controllo, guadagnate molta flessibilità a basso costo. In effetti, questa tecnica costituisce uno dei modi più importanti per mantenere la reversibilità (vedi il Capitolo 2).

La vista ad albero di Java

Un buon esempio di progetto MVC si può trovare nel widget a albero di Java, che visualizza un albero cliccabile e percorribile, che è in realtà un insieme di molte classi diverse organizzate in uno schema MVC.

Per produrre un widget a albero pienamente funzionale, tutto quello che si deve fare è predisporre una fonte di dati conforme all'interfaccia `TreeModel`. Il vostro codice ora diventa il modello per l'albero.

La visualizzazione è creata dalle classi `TreeCellRenderer` e `TreeCellEditor`, che possono essere ereditate e personalizzate per dare colori, tipi di caratteri e icone differenti nel widget. `JTree` funge da controller per il widget e mette a disposizione alcune funzionalità generali di visualizzazione.

Avendo disaccoppiato il modello dalla vista, semplifichiamo molto la programmazione. Non dovete più pensare alla programmazione di un widget a albero, dovete solo fornire una fonte di dati.

Supponiamo che arrivi la vicepresidente e voglia un'applicazione veloce che le permetta di navigare nel diagramma organizzativo dell'azienda, conservato in un database legacy sul mainframe. Scrivete semplicemente un wrapper che prende i dati dal mainframe, li presenta

come `TreeModel`, e *voilà*, avete un widget a albero completamente navigabile.

Ora potete sbizzarrirvi e cominciare a usare le classi di visualizzazione: potete modificare la presentazione dei nodi e usare icone, tipi di caratteri o colori speciali. Quando la vicepresidente torna da voi e dice che nuovi standard aziendali impongono l'uso di un'icona con il teschio e le ossa incrociate per certi dipendenti, potete apportare i cambiamenti in `TreeCellRenderer` senza toccare altro codice.

Oltre le GUI

MVC si insegna normalmente nel contesto dello sviluppo di GUI, ma è in realtà una tecnica di programmazione di uso generale. La vista è un'interpretazione del modello (magari un sottoinsieme) e non deve per forza essere grafica. Il controllo è un meccanismo di coordinamento e non deve per forza essere collegato a qualche tipo di dispositivo di input.

- *Modello*. Il modello astratto dei dati che rappresenta l'oggetto bersaglio. Il modello non sa direttamente nulla di viste o controlli.
- *Vista*. Un modo di interpretare il modello. Riceve le informazioni sui cambiamenti nel modello e sugli eventi logici dal controllo.
- *Controllo*. Un modo per controllare la vista e fornire al modello i nuovi dati. Pubblica gli eventi sia per il modello che per la vista.

Vediamo un esempio non grafico. Il baseball è un'istituzione unica. In quale altro campo si possono trovare gemme come "questa è la partita con il punteggio più alto mai giocata di giovedì, sotto la pioggia, con la luce artificiale, fra squadre il cui nome inizia con una vocale"? Supponiamo ci sia stato dato l'incarico di sviluppare un software di supporto per gli intrepidi cronisti che con grande senso del dovere comunicano punteggi, statistiche e informazioni curiose come quella.

Chiaramente, abbiamo bisogno di informazioni sulla partita in corso: le squadre in campo, le condizioni, il giocatore alla battuta, il punteggio e così via. Questi fatti costituiscono i nostri modelli: verranno aggiornate con l'arrivo di nuove informazioni (è cambiato il lanciatore, un giocatore fa uno strike, comincia a piovere…).

Avremo poi una serie di oggetti visualizzatori che usano questi modelli. Uno potrebbe tener conto dei *run* in modo da aggiornare il punteggio. Un altro potrebbe ricevere le notifiche sui nuovi battitori, e recuperare un breve riassunto delle loro statistiche nella stagione. Potremmo avere addirittura un visualizzatore di *trivia*, responsabile di scovare quei fatti strani e privi di ogni utilità che entusiasmano il pubblico degli spettatori.

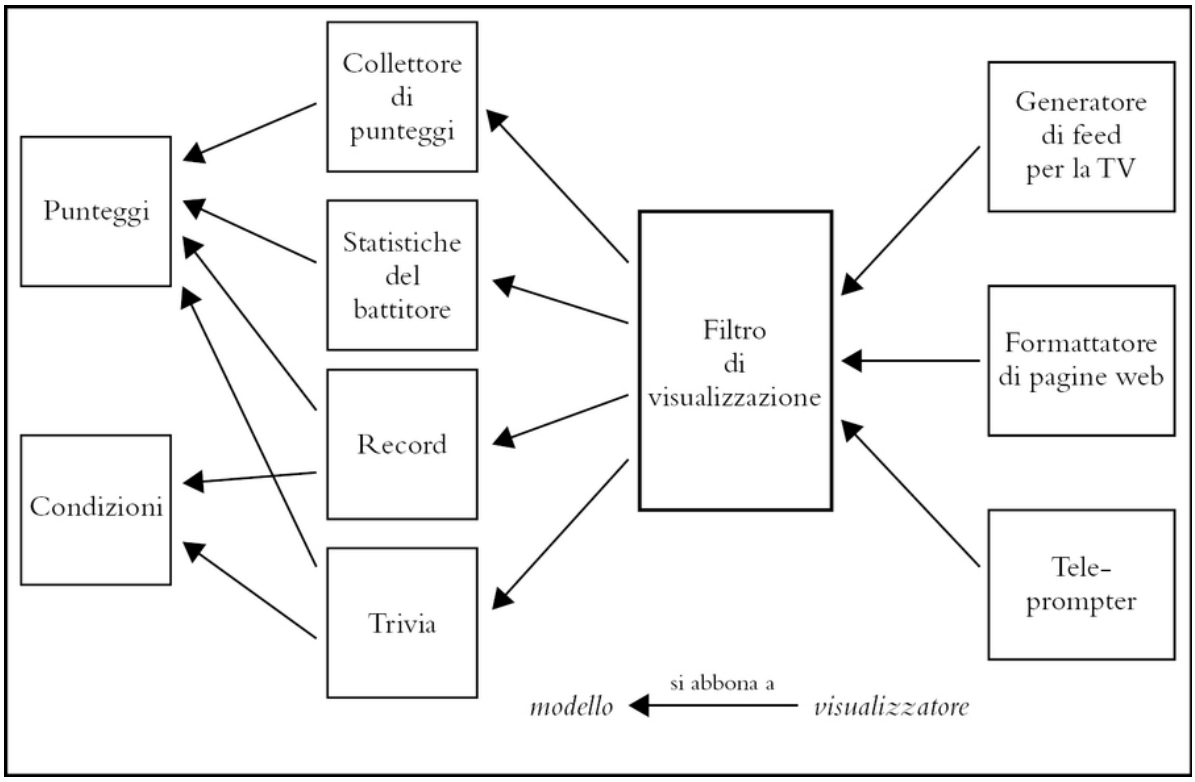


Figura 5.5 Notizie sul baseball: i visualizzatori si abbonano (sottoscrivono) ai modelli.

Noi però non vogliamo inondare il povero cronista con tutte queste viste direttamente. Faremo invece in modo che ogni vista generi notifiche di eventi "interessanti" e lasceremo a qualche oggetto di livello più alto il compito di pianificare che cosa venga mostrato. (Il fatto che sopra lo stadio voli un aereo probabilmente non è interessante, a meno che non sia il centesimo aereo a sorvolare lo stadio nel corso della serata.)

Questi oggetti visualizzatori sono diventati di colpo modelli per l'oggetto di livello superiore, che a sua volta potrebbe essere un modello per altri visualizzatori formattatori. Uno di questi potrebbe creare la scritta del teleprompter per il cronista, un altro potrebbe generare

didascalie video direttamente sul collegamento satellitare, un altro potrebbe aggiornare le pagine web della rete o della squadra (Figura 5.5).

Questo tipo di rete modello-visualizzatore è una tecnica di progettazione comune (e preziosa). Ogni collegamento disaccoppia i dati grezzi dagli eventi che li hanno creati - ogni nuovo visualizzatore è un'astrazione. Poiché poi le relazioni sono una rete (non una semplice catena lineare), abbiamo molta flessibilità. Ogni modello può avere *molti* visualizzatori, e un visualizzatore può lavorare con più modelli.

In sistemi avanzati come questo, può far comodo avere *viste di debug*, viste specializzate che mostrano in profondità i particolari del modello. Anche aggiungere uno strumento per tenere traccia di singoli eventi può far risparmiare molto tempo.

Ancora accoppiati (dopo tutti questi anni)

Nonostante la riduzione dell'accoppiamento che abbiamo ottenuto, ascoltatori e generatori di eventi (abbonati ed editori) hanno comunque *qualche* conoscenza gli uni degli altri. In Java, per esempio, devono essere d'accordo sulle definizioni comuni d'interfaccia e sulle convenzioni di chiamata.

Nella prossima sezione, vedremo modi per ridurre ulteriormente l'accoppiamento mediante una forma di pubblicazione/abbonamento in cui *nessuno* dei partecipanti deve sapere qualcosa degli altri, né deve chiamarli direttamente.

Vedi anche

- *Ortogonalità*, Capitolo 2
- *Reversibilità*, Capitolo 2
- *Disaccoppiamento e legge di Demetra*, in questo Capitolo
- *Lavagne*, in questo Capitolo
- *È tutta scrittura*, Capitolo 8

Esercizi

29. Poniamo di avere un sistema di prenotazioni aeree che include il concetto di "volo".

```
public interface Flight {
  // Restituisce false se il volo è tutto prenotato.
  public boolean addPassenger(Passenger p);
  public void addToWaitList(Passenger p);
  public int getFlightCapacity();
```

```
  public int getNumPassengers();
}
```

Se aggiungiamo un passeggero alla lista d'attesa, verrà automaticamente assegnato al volo se si rende disponibile un posto.
Un enorme sistema di reporting continua a tenere sotto controllo voli completi o in overbooking, per suggerire quando sarebbe opportuno prevedere voli ulteriori. Funziona bene, ma la sua esecuzione richiede ore.
Vorremmo avere un po' più di flessibilità nell'elaborare i passeggeri della lista d'attesa, e dobbiamo fare qualcosa a proposito del sistema di reporting che impiega troppo tempo. Usate le idee viste in questa sezione per riprogettare questa interfaccia.

Lavagne

Di solito non si associa la parola *eleganza* ai detective della polizia, che invece richiamano magari alla mente il luogo comune delle ciambelle con il caffè. Pensiamo però come i detective possano usare una *lavagna* per coordinare le indagini su un omicidio e risolverlo.

Supponiamo che l'ispettore capo cominci con il far installare una grande lavagna nella sala riunioni. Sulla lavagna, poi, scrive un'unica domanda:

H. DUMPTY (MASCHIO, UOVO): INCIDENTE O OMICIDIO?

Humpty è davvero caduto, o è stato spinto? Ogni detective può contribuire a questo potenziale caso di omicidio aggiungendo fatti, dichiarazioni dei testimoni, ogni prova forense che si scopra e così via. Mentre i dati si accumulano, un detective può notare un collegamento e scrivere sulla lavagna la sua osservazione o la sua ipotesi. Il processo continua, un turno dopo l'altro, con persone e agenti diversi, finché il caso non viene chiuso. (Un esempio di lavagna è nella Figura 5.6.)

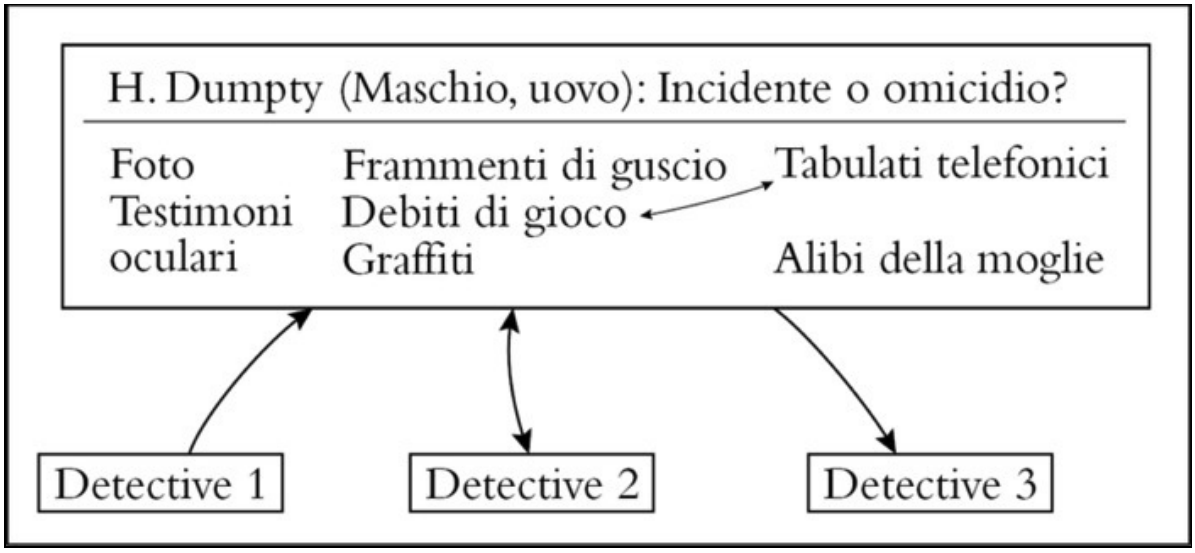


Figura 5.6 Qualcuno ha trovato un collegamento fra i debiti di gioco di Humpty e i tabulati telefonici. Forse riceveva telefonate di minaccia.

Alcune caratteristiche fondamentali del metodo della lavagna sono le seguenti.

- Nessuno dei detective deve sapere dell'esistenza di altri detective; ciascuno guarda la lavagna per scoprire le nuove informazioni e vi aggiunge quanto ha scoperto.
- I detective possono essere addestrati in campi diversi, possono avere livelli diversi di istruzione e di esperienza, possono non lavorare nello stesso distretto. Hanno in comune il desiderio di risolvere il caso, ma niente altro.
- Nel corso delle indagini possono andare e venire detective diversi, e possono lavorare in turni diversi.
- Non ci sono limitazioni su quello che può essere messo sulla lavagna. Possono essere foto, frasi, prove fisiche e così via.

Abbiamo lavorato a molti progetti che comportavano un flusso di lavoro o un processo di raccolta di dati distribuiti. In tutti i casi, progettare una soluzione sulla base di un semplice modello a lavagna ci ha fornito una metafora concreta su cui lavorare: tutte le caratteristiche elencate sopra parlando di detective sono applicabili a oggetti e moduli di codice.

Un sistema a lavagna permette di disaccoppiare completamente gli oggetti, gli uni dagli altri, mettendo a disposizione una "piazza" in cui consumatori e produttori di conoscenza possono scambiarsi dati in forma anonima e asincrona. Come potete immaginare, si riduce anche la quantità di codice da scrivere.

Implementazioni a lavagna

I sistemi a lavagna basati su computer sono stati inventati per l'uso in applicazioni di intelligenza artificiale dove i problemi da risolvere erano grandi e complessi: riconoscimento del parlato, sistemi di ragionamento basati sulla conoscenza e così via.

Sistemi distribuiti moderni simili a lavagne, come JavaSpaces e T Spaces [URL50, URL25] sono basati su un modello a coppie chiave/valore, presente per la prima volta in Linda [CG90], dove l'idea prendeva il nome di *tuple space* (spazio di ennuple).

Con questi sistemi, si possono memorizzare oggetti Java attivi (e non solo dati) sulla lavagna, e li si può poi recuperare per corrispondenza parziale di campi (attraverso schemi e jolly) o per sottotipi. Supponiamo per esempio di avere un tipo `Author`, con un sottotipo `Person`. Si potrebbe effettuare una ricerca in una lavagna contenente oggetti `Person` utilizzando uno schema `Author` con il valore "Shakespeare" per `lastName`. Si otterrà Bill Shakespeare l'autore ma non Fred Shakespeare il giardiniere.

Le operazioni principali in JavaSpaces sono:

Nome	Funzione
`read`	Cerca e recupera dati dallo spazio.
`write`	Inserisce un elemento nello spazio.
`take`	Simile a `read`, ma quando trova un elemento lo elimina anche dallo spazio.
`notify`	Imposta il verificarsi di una notifica ogni volta che viene scritto un oggetto corrispondente allo schema.

T Spaces ha un insieme simile di operazioni, ma con nomi diversi e una semantica leggermente diversa. Entrambi i sistemi sono costruiti come un prodotto database: mettono a disposizione operazioni atomiche e transazioni distribuite per garantire l'integrità dei dati.

Poiché possiamo memorizzare oggetti, possiamo usare una lavagna per progettare algoritmi sulla base di un *flusso di oggetti* e non solo di dati. È come se i nostri detective potessero appendere alla lavagna persone in carne e ossa, i testimoni e non solo le loro dichiarazioni. Chiunque può fare domande a un testimone nell'ambito del caso, appendere la trascrizione e spostare il testimone in un'altra area della lavagna, dove potrebbe rispondere in modo diverso (se si consente anche al testimone di leggere la lavagna).

Un grande vantaggio di sistemi come questi è che si ha un'unica interfaccia coerente verso la lavagna. Quando si costruisce un'applicazione distribuita convenzionale, si può dedicare molto tempo a mettere a punto specifiche chiamate API per ogni transazione e interazione distribuita nel sistema. Con l'esplosione combinatoria di interfacce e interazioni, il progetto può diventare rapidamente un incubo.

Lo stile di programmazione a lavagna elimina il bisogno di molte interfacce e permette di ottenere un sistema più elegante e coerente.

Organizzare la lavagna
Quando i detective lavorano su grandi casi, la lavagna può diventare molto ingombra, e può risultare difficile trovare i singoli dati. La soluzione è *suddividere in parti* la lavagna e iniziare in qualche modo a organizzare i dati.
Sistemi software diversi gestiscono questo partizionamento in modi diversi: qualcuno usa strutture piatte a *zone* o *gruppi di interesse*, altri adottano una struttura più gerarchica, ad albero.

Un esempio applicativo

Supponiamo di dover scrivere un programma che accetti ed elabori domande di mutuo o di prestito. Le norme che governano il settore negli Stati Uniti sono orrendamente complesse: le amministrazioni federali, statali e locali hanno tutte qualcosa da dire in merito. Il prestatore deve dimostrare di aver comunicato certe cose e deve chiedere certe informazioni, ma *non* deve fare certe altre domande e così via.

Al di là del marasma di leggi di cui tener conto, abbiamo da affrontare anche i problemi seguenti.

- Non c'è alcuna certezza sull'ordine in cui arriveranno i dati. Per esempio, le interrogazioni per una verifica del credito o per una

ricerca catastale possono richiedere quantità di tempo sostanziose, mentre informazioni come nome e indirizzo possono essere subito disponibili.

- La raccolta dei dati può essere fatta da persone diverse, distribuite in uffici diversi, in fusi orari diversi.
- In parte la raccolta dei dati può essere effettuata automaticamente da altri sistemi. Anche questi dati possono arrivare in modo asincrono.
- Ciononostante, certi dati possono essere comunque dipendenti da altri dati. Per esempio, magari non si può iniziare la ricerca di un'auto finché non si ha un documento di proprietà o di assicurazione.
- L'arrivo di nuovi dati può sollevare nuove domande e determinare nuove politiche. Supponiamo che la verifica del credito restituisca indicazioni tutt'altro che entusiasmanti: a questo punto serviranno altri cinque moduli e magari anche un campione di sangue.

Possiamo cercare di gestire ogni possibile combinazione e ogni possibile circostanza utilizzando un sistema a flusso di lavoro. Esistono molti sistemi del genere, ma possono essere complessi e richiedere molto lavoro di programmazione. Al variare dei regolamenti, il flusso di lavoro deve essere riorganizzato: le persone devono magari modificare le loro procedure e se il codice è troppo rigido deve essere riscritto.

Una lavagna, insieme con un motore di regole che incapsuli le norme legali, è una soluzione elegante a queste difficoltà. L'ordine di arrivo dei dati non è rilevante: quando viene comunicato un fatto, può attivare le regole appropriate. È facile anche gestire il feedback: l'output di qualsiasi insieme di regole può finire sulla lavagna e provocare l'attivazione altre regole pertinenti.

SUGGERIMENTO 43
Usate lavagne per coordinare il flusso di lavoro.

Possiamo usare la lavagna per coordinare fatti e agenti diversi, mantenendo però l'indipendenza e anche l'isolamento dei partecipanti, uno rispetto all'altro.

Si possono ottenere gli stessi risultati con metodi più basati sulla forza bruta, ovviamente, ma i sistemi che si otterrebbero sarebbero più fragili. Se si rompono, c'è il rischio che nessuno sia più in grado di far funzionare di nuovo il programma.

Vedi anche

- *Il potere del puro testo*, Capitolo 3
- *È solo una visya*, in questo Capitolo

Sfide

- Usate dei sistemi a lavagna nel mondo reale, per esempio la bacheca per i messaggi vicino al frigorifero o una grande lavagna bianca al lavoro? Che cosa li rende efficaci? I messaggi vengono inseriti con un formato coerente? È importante?

Esercizi

30. Per ciascuna delle applicazioni seguenti, un sistema a lavagna sarebbe appropriato oppure no? Perché?

1. *Elaborazione di immagini*. Vorreste avere una serie di processi in parallelo che prendono parti di un'immagine, le elaborano e poi rimettono a posto i pezzi completati.
2. *Calendari di gruppo*. Avete persone sparse per il pianeta, sotto fusi orari diversi e che parlano lingue diverse, e vogliono pianificare una riunione.
3. *Strumento di monitoraggio della rete*. Il sistema raccoglie statistiche sulle prestazioni e segnalazioni di gusti. Vorreste implementare qualche agente che usi queste informazioni per verificare i problemi nel sistema.

Mentre codificate

La saggezza convenzionale dice che, una volta che un progetto è nella fase di codifica, il lavoro è prevalentemente di tipo meccanico, si tratta solo di trascrivere il progetto in enunciati eseguibili. Pensiamo che questo atteggiamento sia il motivo principale per cui molti programmi sono brutti, inefficienti, male strutturati, non manutenibili e, in una parola, sbagliati.

La codifica non è meccanica. Se lo fosse tutti gli strumenti di CASE su cui si puntavano tante speranze nei primi anni Ottanta avrebbero sostituito i programmi già molto tempo fa. Ci sono decisioni da prendere ogni minuto, decisioni che richiedono riflessione attenta e capacità di giudizio, se si vuole che il programma risultante goda di una vita lunga, precisa e produttiva.

Gli sviluppatori che non pensano attivamente al loro codice programmano per coincidenza: il codice può funzionare, ma non esiste un motivo particolare per cui sia così. In *Programmazione per coincidenza*, ci facciamo sostenitori di un coinvolgimento più positivo nel processo di codifica.

La maggior parte del codice che scriviamo viene eseguita rapidamente, ma ogni tanto sviluppiamo algoritmi potenzialmente in grado di mettere in ginocchio anche i processori più veloci. In *Velocità degli algoritmi*, analizziamo modi per stimare la velocità del codice e diamo qualche suggerimento su come identificare i possibili problemi prima che si presentino davvero.

I programmatori pragmatici pensano criticamente a tutto il codice, incluso il loro. Cerchiamo continuamente spazi di miglioramento nei nostri programmi e nei nostri progetti. In *Refactoring*, esaminiamo

tecniche che ci aiutano a mettere a punto il codice esistente anche mentre siamo nel bel mezzo di un progetto.

Una cosa da avere sempre in mente quando si produce codice è che un giorno sarà necessario sottoporlo a test. Fate in modo che sia facile sottoporre a test il codice, e migliorerete le probabilità che venga effettivamente testato, un'idea che sviluppiamo in *Codice facile da sottoporre a test*.

Infine, in *Maghi cattivi*, consigliamo di fare attenzione a strumenti che scrivono per conto vostro grandi quantità di codice, a meno che non vi sia molto chiaro quel che state facendo.

La maggior parte di noi è in grado di guidare un'auto in gran parte in automatico; non comandiamo esplicitamente al nostro piede di schiacciare un pedale, o al nostro braccio di ruotare il volante, pensiamo semplicemente "rallenta e gira a destra". I buoni automobilisti attenti alla sicurezza però rivedono costantemente la situazione, stanno attenti ai problemi potenziali e cercano di essere sempre in una buona posizione, nell'eventualità che succeda l'imprevisto. Lo stesso vale per la codifica: può essere in gran parte un lavoro di routine, ma prestare sempre molta attenzione può evitare un disastro.

Programmazione per coincidenza

Avete mai visto i vecchi film di guerra in bianco e nero? Il soldato attento esce cautamente dalla boscaglia. Ha davanti una radura: il terreno sarà minato, o potrà attraversarla in sicurezza? Non vi sono indicazioni che si tratti di un campo minato, né segnali, né filo spinato, né crateri. Il soldato sonda con la baionetta il terreno davanti ai suoi piedi e chiude gli occhi, aspettandosi un'esplosione, ma non succede nulla. Allora va avanti, prudentemente, sondando continuamente il terreno. Alla fine, convinto ormai che la radura sia sicura, si raddrizza e marcia orgogliosamente avanti, e finisce in mille pezzi.

Il soldato all'inizio sonda l'eventuale presenza di mine e non ne trova, ma solo per fortuna; così arriva a una conclusione falsa, con risultati disastrosi.

Da sviluppatori, anche noi lavoriamo in campi minati. Centinaia di trappole non vedono l'ora di coglierci in fallo ogni giorno. Ricordando la

storia del soldato, dobbiamo fare molta attenzione a non trarre conclusioni sbagliate. Dobbiamo evitare di programmare per coincidenza, basandoci sulla fortuna e sui successi accidentali, e invece *programmare deliberatamente*.

Come programmare per coincidenza

Supponiamo che a Fred venga assegnato un compito di programmazione. Fred scrive un po' di codice, lo sottopone a test e sembra che funzioni. Scrive ancora un po' di codice, lo sottopone a test e sembra ancora che funzioni bene. Dopo varie settimane in cui procede così, il programma all'improvviso smette di funzionare; passa ore a cercare di sistemarlo, ma ancora non capisce il perché. Fred potrebbe anche perdere una gran quantità di tempo a caccia del pezzo di codice incriminato, senza riuscire mai a sistemarlo. Non importa quello che fa, sembra che non funzioni mai bene.

Fred non sa perché il codice vada male perché *non ha mai saputo perché funzionava prima*. Sembrava funzionare, con i "test" limitati che faceva, ma era solo per coincidenza. Sostenuto da quella falsa fiducia, Fred è andato avanti a testa bassa. Ora, la maggior parte delle persone intelligenti conoscerà qualcuno che assomiglia a Fred, ma *noi* siamo di tutt'altra pasta. Noi non ci basiamo sulle coincidenze, o no?

Qualche volta sì, ma qualche volta può essere molto facile confondere una coincidenza fortunata con un piano ben congegnato. Vediamo qualche esempio.

Accidenti di implementazione

Gli accidenti di implementazione sono cose che succedono solo perché quello è il modo in cui si scrive di solito il codice. Si finisce per basarsi su errori non documentati o condizioni limite.

Supponiamo che chiamiate una routine con dati errati. La routine risponde in un certo modo, e continuate a codificare in funzione di quella risposta. L'autore non pensava che la sua routine venisse usata in quel modo, non lo ha mai nemmeno preso in considerazione. Quando la routine viene "aggiustata", il vostro codice può non funzionare. Nel caso più estremo, la routine che avete chiamato magari non era stata neppure

progettata per fare quello che volevate, ma *sembrava* funzionare bene. Un problema simile è chiamare le cose in ordine sbagliato, o nel contesto sbagliato.

```
paint(g);
invalidate();
validate();
revalidate();
repaint();
paintImmediately(r);
```

Qui sembra che Fred stia disperatamente cercando di tirar fuori qualcosa dalla schermata, ma queste routine non sono mai state progettate per essere chiamate in questo modo: anche se sembra che funzionino, è proprio solo un caso, una coincidenza.

Per aggiungere al danno la beffa, quando il componente finalmente viene disegnato, Fred non tenta di tornare indietro e di eliminare le chiamate spurie. "Adesso funziona, meglio lasciarlo stare…"

È facile farsi ingannare da questi ragionamenti. Perché correre il rischio di andare a toccare qualcosa che funziona? Beh, ci vengono in mente vari motivi.

- Può essere che non funzioni davvero, forse sembra solo che funzioni.
- La condizione al contorno su cui fate affidamento può essere solo un caso. In circostanze diverse (con una risoluzione diversa dello schermo, magari) potrebbe comportarsi in altro modo.
- Un comportamento non documentato può cambiare con la successiva versione della libreria.
- Chiamate aggiuntive non necessarie rallentano il codice.
- Chiamate aggiuntive possono anche far aumentare il rischio di introdurre nuovi errori.

Per il codice che scrivete e che altri chiameranno, i principi fondamentali di una buona modularizzazione e dell'occultamento dell'implementazione dietro interfacce compatte e ben documentate possono essere d'aiuto. Un contratto ben specificato (vedi *Progettare per contratto*, nel Capitolo 4) può contribuire a eliminare i fraintendimenti.

Per le routine che chiamate, basatevi solo sui comportamenti documentati. Se non è possibile, per qualsiasi motivo, documentate molto

bene ciò che avete presupposto.

Accidenti del contesto

Potete avere anche “accidenti del contesto”. Supponiamo che stiate scrivendo un modulo di utility. Solo perché state codificando per un ambiente a GUI, il modulo deve fare affidamento sulla presenza di una GUI? Fate affidamento sul fatto che gli utenti parlino inglese? Su utenti che sanno leggere? Su che cos’altro fate affidamento, di cui non potete avere certezza?

Assunzioni implicite

Le coincidenze possono fuorviare a tutti i livelli - dalla generazione dei requisiti fino ai test. I test in particolare sono funestati da falsi rapporti di causalità ed esiti casuali. È facile dare per scontato che X causa Y ma, come abbiamo detto in *Debugging* (Capitolo 3), non datelo per scontato: dimostratelo.

A tutti i livelli, si procede avendo in testa molti presupposti - ma questi presupposti raramente sono documentati e spesso sono in conflitto con quelli di altri sviluppatori. I presupposti che non sono basati su fatti ben assodati sono la maledizione di tutti i progetti.

SUGGERIMENTO 44
Non programmate per coincidenza.

Come programmare consapevolmente

Vogliamo passare meno tempo a macinare codice, catturare e sistemare gli errori il più presto possibile nel ciclo di sviluppo, e comunque creare il minor numero possibile di errori. È utile programmare consapevolmente.

- Siate sempre consapevoli di quel che state facendo. Fred ha lasciato che le cose a poco a poco gli sfuggissero di mano, finché non è finito bollito come la rana in *Zuppa di pietre e rane bollite* (Capitolo 1).

- Non codificate alla cieca. Tentare di costruire un'applicazione che non si capisce fino in fondo, o usare una tecnologia con cui non si ha familiarità è un invito a farsi fuorviare dalle coincidenze.
- Procedete in base a un piano, che quel piano sia nella vostra testa, su un tovagliolo o in una stampata a misura di parete prodotta da uno strumento CASE.
- Fidatevi solo di cose affidabili. Non dovete dipendere dal caso o da ipotesi. Se non potete stabilire la differenza, in certe circostanze, assumete il peggio.
- Documentate i vostri presupposti. *Progettare per contratto* (Capitolo 4) può aiutarvi a chiarire i vostri presupposti nella vostra testa e aiutarvi a comunicarli agli altri.
- Non sottoponete a test solo il vostro codice, ma anche i vostri presupposti. Non tirate a indovinare: provateli. Scrivete un'asserzione per mettere alla prova i vostri presupposti (vedi *Programmazione assertiva*, nel Capitolo 4).
- Ordinate i vostri sforzi per priorità. Dedicate tempo agli aspetti importanti: più che probabilmente, sono le parti difficili. Se i fondamenti o l'infrastruttura non sono a posto, tutti gli orpelli, per quanto brillanti, saranno irrilevanti.
- Non siate schiavi della storia. Non lasciate che il codice esistente imponga il codice futuro. Tutto il codice può essere sostituito, se non è più adeguato. Anche all'interno di uno stesso programma, non lasciate che quel che avete già fatto vincoli quello che farete poi - siate pronti a ristrutturare tutto (vedi *Refactoring*, più avanti in questo capitolo). Questa decisione può avere conseguenze per la tempistica del progetto, ma l'ipotesi è che l'impatto sia minore del costo di *non* apportare le modifiche. (Ma non esagerate: abbiamo conosciuto uno sviluppatore che riscriveva tutto il sorgente che gli veniva passato per adeguarlo alle sue convenzioni sui nomi.)

Dunque: la prossima volta che qualcosa sembra funzionare, ma non sapete perché, assicuratevi che non sia solo una coincidenza fortuita.

Vedi anche

- *Zuppa di pietre e rane bollite*, Capitolo 1
- *Debug*, Capitolo 3

- *Progettare per contratto*, Capitolo 4
- *Programmazione assertiva*, Capitolo 4
- *Accoppiamento temporale*, Capitolo 5
- *Refactoring*, in questo Capitolo
- *È tutta scrittura*, Capitolo 8.

Esercizi

31. Potete identificare qualche coincidenza nel seguente frammento di codice C? Immaginate che questo codice sia sepolto in profondità in una routine di libreria.

```
fprintf(stderr,"Error, continue?");
gets(buf);
```

32. Questo pezzo di codice C può funzionare qualche volta su qualche macchina, ma anche no. Che cosa c'è di sbagliato?

```
/*Tronca una stringa ai suoi ultimi maxlen caratteri */
void string_tail(char *string, int maxlen) {
  int len = strlen(string);
  if (len > maxlen) {
    strcpy(string, string + (len - maxlen));
  }
}
```

33. Questo codice viene da una suite di traccia Java di uso generale. La funzione scrivere una stringa in un file di log. Supera il suo unit test, ma fallisce quando la usa uno degli sviluppatori web. Su quale coincidenza si basa?

```
public static void debug(String s) throws IOException {
  FileWriter fw = new FileWriter("debug.log", true);
  fw.write(s);
  fw.flush();
  fw.close();
}
```

Velocità degli algoritmi

In *Stime* (Capitolo 2), abbiamo parlato di stimare cose come il tempo necessario per attraversare una città o il tempo necessario per completare un progetto. Vi sono però anche altri tipi di stime che i programmatori pragmatici usano quasi quotidianamente: stime delle risorse usate dagli algoritmi - tempo, processore, memoria e così via.

Questo tipo di stime spesso è cruciale. Data la scelta fra due modi di fare qualcosa, quale scegliete? Sapete per quanto tempo girerà il vostro programma con 1000 record, ma come scalerà a un milione? Quali parti del codice richiedono un'ottimizzazione?

A queste domande spesso si può rispondere con il buon senso, un po' di analisi e un modo di scrivere le approssimazioni che prende il nome di notazione *big O* o *O-grande.*

Che cosa si intende per stima degli algoritmi?

La maggior parte degli algoritmi non banali gestisce qualche tipo di input variabile: ordinamento di n stringhe, inversione di una matrice $m \times n$, o decifrazione di un messaggio con una chiave a n bit. Normalmente, le dimensioni dell'input avranno un influsso sull'algoritmo: quanto più grande l'input, tanto maggiore il tempo di esecuzione o la quantità di memoria usata.

Se le relazioni fossero sempre lineari (per cui il tempo crescesse in proporzione diretta con il valore di n), questa sezione non sarebbe importante. La maggior parte degli algoritmi significativi però non è lineare. La buona notizia è che molti sono sublineari. Una ricerca binaria, per esempio, non deve per forza esaminare ogni possibilità alla ricerca di una corrispondenza. La cattiva notizia è che altri algoritmi sono molto peggio che lineari: i tempi di esecuzione o i requisiti di memoria crescono molto più rapidamente di n. Un algoritmo che impiega un minuto a elaborare dieci elementi può richiedere una vita per elaborarne 100.

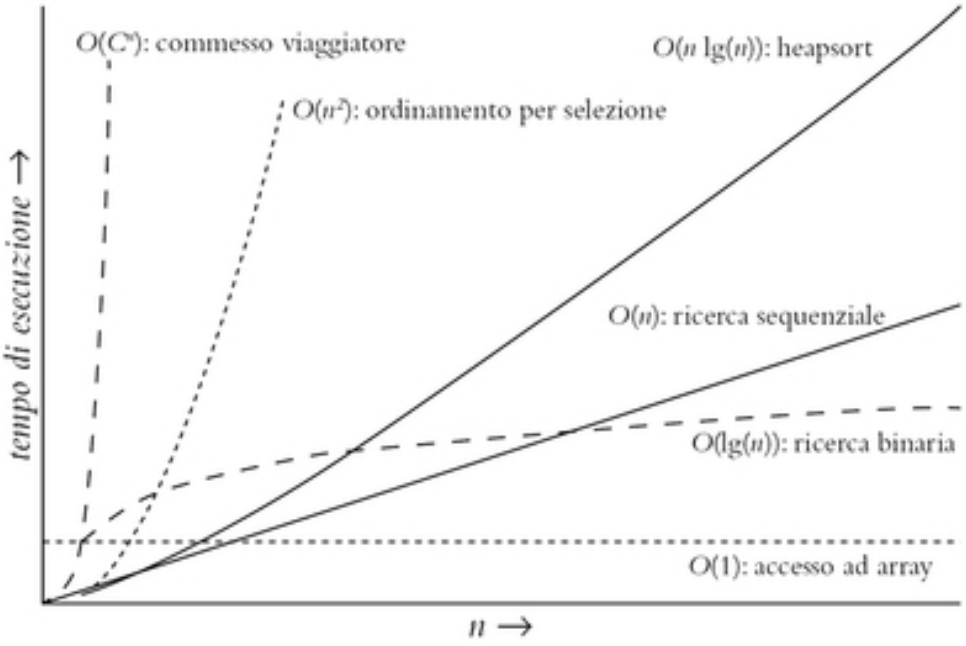
Ogni volta che scriviamo qualcosa che contiene cicli o chiamate ricorsive, nel subconscio controlliamo il tempo di esecuzione e i requisiti di memoria. Raramente è un processo formale, ma una conferma veloce che quel che stiamo facendo ha senso date le circostanze. Qualche volta, però, bisogna eseguire un'analisi più dettagliata, e in quel caso viene utile la notazione $O()$.

La notazione O()

La notazione $O()$ è un modo di esprimere le approssimazioni in forma matematica. Quando scriviamo che una particolare routine ordina n record in tempo $O(n2)$, vogliamo semplicemente dire che il tempo necessario nel caso peggiore varierà con il quadrato di n. Raddoppiate il

numero dei record e il tempo aumenterà di circa quattro volte. Pensate la *O* come una abbreviazione di *nell'ordine di*. La notazione *O*() indica un limite superiore ai valori della grandezza che stiamo misurando (tempo, memoria ecc.). Se diciamo che una funzione richiede un tempo *O*(*n*2), sappiamo che il tempo necessario per computarla non crescerà più rapidamente di *n*2. A volte troveremo funzioni *O*() molto complesse, ma poiché è il termine di ordine più alto che domina il valore al crescere di *n*, per convenzione si eliminano tutti i termini di ordine inferiore e non ci si dà la pena di mostrare gli eventuali fattori moltiplicativi costanti. *O*((*n*2/2) + 3*n*) è lo stesso che *O*(*n*2/2), che è equivalente a *O*(*n*2). Questo è in effetti un punto di debolezza della notazione: un algoritmo *O*(*n*2) può essere 1000 volte più veloce di un altro algoritmo *O*(*n*2), ma dalla notazione non lo si può capire.

La Figura 6.1 mostra varie notazioni *O*() comuni che vi capiterà di incontrare, con un grafico che mette a confronto i tempi di esecuzione di algoritmi di ciascuna categoria. Chiaramente, le cose cominciano rapidamente a sfuggire di mano non appena si supera *O*(*n*2).



Alcune notazioni O() di uso comune

$O(1)$	Constante (accesso a un elemento di un array, enunciati semplici)
$O(\lg(n))$	Logaritmico(ricerca binaria) [*La notazione lg(n) è una forma abbreviata per* $\log_2(n)$]
$O(n)$	Lineare (ricerca sequenziale)
$O(n \lg(n))$	Peggio che lineare, ma non tanto peggio (tempo di esecuzione medio di quicksort, heapsort)
$O(n^2)$	Quadratica (ordinamenti per selezione e inserzione)
$O(n^3)$	Cubica (prodotto di due matrici $n \times n$)
$O(C^n)$	Esponenziale (problema del commesso viaggiatore, partizionamento di un insieme)

Figura 6.1 Tempi di esecuzione di vari algoritmi.

Per esempio, supponiamo di avere una routine che impiega 1 secondo a elaborare 100 record. Quanto tempo ci vorrà perché ne elabori 1000? Se

il vostro codice è $O(1)$, ci vorrà sempre 1 secondo. Se è $O(\lg(n))$, probabilmente aspetterete circa 3 secondi. $O(n)$ vi darà un aumento lineare a 10 secondi, mentre $O(n \lg(n))$ vi porterà a 33 secondi circa. Se siete tanto sfortunati da avere una routine $O(n2)$, state tranquilli per 100 secondi intanto che fa il suo lavoro. E se usate un algoritmo esponenziale $O(2^n)$ potete andare tranquillamente a farvi una tazza di caffè: la vostra routine non finirà prima di 10263 anni. Fateci sapere come finisce l'universo.

La notazione $O()$ non vale solo per il tempo: potete usarla per rappresentare qualsiasi risorsa utilizzata da un algoritmo. Per esempio, spesso è utile poter modellizzare il consumo di memoria (vedi l'Esercizio 35 più avanti).

Stima a buon senso

Potete stimare l'ordine di grandezza di molti algoritmi di base con un po' di buon senso.

- *Cicli semplici*. Se un ciclo semplice va da 1 a n, è probabile che l'algoritmo sia $O(n)$ - il tempo cresce linearmente con n. Esempi sono le ricerche esaustive, l'identificazione del valore massimo in una matrice, la generazione di somme di controllo.
- *Cicli annidati*. Se si annida un ciclo all'interno di un altro ciclo, l'algoritmo diventa $O(m \times n)$, dove m e n sono i limiti dei due cicli distinti. Questo succede normalmente negli algoritmi di ordinamento semplici, come il *bubble sort*, dove il ciclo esterno esamina ciascun elemento della matrice alla volta, e il ciclo interno calcola dove collocare quell'elemento nel risultato ordinato. Questi algoritmi di ordinamento tendono a essere $O(n^2)$.
- *Processo dicotomico*. Se il vostro algoritmo dimezza l'insieme delle cose che prende in considerazione ogni volta che percorre il ciclo, è probabile che sia logaritmico, $O(\lg(n))$ (vedi l'Esercizio 37 più avanti). Una ricerca binaria in una lista ordinata, l'attraversamento di un albero binario e l'identificazione del primo bit 1 in una parola di macchina possono essere tutti $O(\lg(n))$.

- *Divide et impera*. Algoritmi che dividono il loro input, lavorano sulle due metà in modo indipendente e poi combinano i risultati possono essere $O(n \lg(n))$. L'esempio classico è il *quicksort*, che divide i dati in due metà ed effettua l'ordinamento ricorsivo di ciascuna metà, Anche se tecnicamente è $O(n^2)$ perché il suo comportamento degrada quando gli viene fornito un input ordinato, il tempo di esecuzione medio di quicksort è $O(n \lg(n))$.
- *Combinatoria*. Ogni volta che gli algoritmi comincia a considerare permutazioni di elementi, i tempi di esecuzione possono sfuggire di mano. Questo perché il numero delle permutazioni di n elementi è il fattoriale di n (ci sono $5! = 5\times4\times3\times2\times1=120$ permutazioni delle cifre da 1 a 5). Cronometrate un algoritmo combinatorio per cinque elementi; per sei elementi il tempo di esecuzione sarà sei volte maggiore, per sette elementi 42 volte (6×7) maggiore. Esempi sono gli algoritmi di molti dei problemi riconosciuti *difficili*, come il problema del commesso viaggiatore, la disposizione ottimale di oggetti in un contenitore, il partizionamento di un insieme di numeri in modo che ciascun insieme abbia lo stesso totale e così via. Spesso si usano euristiche per ridurre i tempi di esecuzione di questi tipi di algoritmi in particolari campi di problemi.

La velocità degli algoritmi in pratica

È improbabile che spendiate molto tempo, durante la vostra carriera, a scrivere routine di ordinamento. Quelle nelle librerie disponibili probabilmente saranno più veloci di tutto quello che potreste scrivere senza una fatica notevole. Tuttavia, i tipi di algoritmi descritti prima spuntano periodicamente. Ogni volta che vi ritrovate a scrivere un semplice ciclo, sapete di avere un algoritmo $O(n)$. Se quel ciclo contiene un ciclo interno, avete un algoritmo $O(m \times n)$. Dovete chiedervi quanto possono crescere quei valori. Se i numeri sono limitati, sapete quanto tempo impiegherà al massimo il vostro algoritmo. Se i numeri dipendono da fattori esterni (per esempio, il numero dei record in un lotto da elaborare durante la notte, o il numero dei nomi in un elenco di persone), potete magari fermarvi e riflettere sull'effetto che valori molto grandi possono avere sul tempo di esecuzione o sul consumo di memoria.

SUGGERIMENTO 45
Stimate l'ordine dei vostri algoritmi.

Ci sono cose che potete fare per affrontare i potenziali problemi. Se avete un algoritmo $O(n2)$, cercate un metodo *divide et impera* che vi faccia scendere a $O(n \lg(n))$.

Se non siete sicuri di quanto tempo richiederà il vostro codice, o di quanta memoria utilizzerà, provate a eseguirlo, variando il numero dei record in input o quello che è probabile influenzi di più il tempo di esecuzione. Poi riportate in un grafico i risultati. Dovreste avere presto una buona idea della forma della curva. Va verso l'alto, è una retta o va appiattendosi con il crescere della dimensione degli input? Tre o quattro punti dovrebbero già darvi un'idea.

Pensate anche solo a quello che fate nel codice stesso. Un semplice ciclo $O(n2)$ può dare prestazioni migliori di un complesso $O(n \lg(n))$ per valori piccoli di n, in particolare se l'algoritmo $O(n \lg(n))$ ha un ciclo interno costoso.

Dopo tutta questa teoria, non dimenticate che ci sono anche considerazioni pratiche. Il tempo di esecuzione può sembrare che cresca linearmente per piccoli insiemi di input. Ma date in pasto al codice milioni di record e di colpo il tempo peggiora e il sistema inizia a boccheggiare. Se testate una routine di ordinamento con chiavi di input casuali, potreste rimanere sorpresi la prima volta che incontra un input ordinato. I programmatori pragmatici cercano di coprire sia le basi teoriche sia quelle pratiche. Dopo tutte queste stime, l'unico tempo che conta è la velocità del vostro codice, nell'ambiente di produzione, con dati reali. Il che ci porta al nostro prossimo suggerimento.

SUGGERIMENTO 46
Sottoponete a test le vostre stime.

Se è difficile cronometrare tempi precisi, usate *profilatori di codice* per contare quante volte vengono eseguiti i diversi passi dell'algoritmo e riportate in grafico quei valori in funzione delle dimensioni dell'input.

Il meglio non è sempre il meglio

Dovete essere pragmatici anche nella scelta degli algoritmi appropriati: non sempre quello più veloce è il migliore per quel che si deve fare. Dato

un insieme di input piccolo, un ordinamento diretto per inserimento funzionerà bene quanto un quicksort, e ci vorrà meno tempo per scriverlo e sottoporlo a debug. Dovete anche stare attenti, se l'algoritmo che scegliete ha un costo elevato di impostazione. Per insiemi di input piccoli, l'impostazione può rendere trascurabile il tempo di esecuzione e rendere inappropriato l'algoritmo.

Fate attenzione anche all'*ottimizzazione prematura*. È sempre una buona idea essere sicuri che un algoritmo sia un collo di bottiglia, prima di investire il vostro tempo prezioso nel migliorarlo.

Vedi anche

- *Stime*, Capitolo 2

Sfide

- Ogni sviluppatore deve avere una percezione di come gli algoritmi sono progettati e analizzati. Robert Sedgewick ha scritto una serie di libri accessibili sul tema ([Sed83, SF96, Sed92] e altri). Vi consigliamo di aggiungere uno dei suoi libri alla vostra collezione, e di impegnarvi a leggerlo.
- Per quanti amano più dettagli di quelli forniti da Sedgewick: leggete i volumi della serie *Art of Computer Programming* di Dodald Knuth, che analizzano un'ampia gamma di algoritmi [Knu97a, Knu97b, Knu98].
- Nell'esercizio 34, consideriamo l'ordinamento di matrici di interi lunghi. Quali sono le conseguenze, se le chiavi sono più complesse e se il sovraccarico del confronto fra le chiavi è elevato? La struttura delle chiavi influenza l'efficienza degli algoritmi di ordinamento, o l'ordinamento più veloce è sempre il più veloce?

Esercizi

34. Abbiamo codificato un insieme di semplici routine di ordinamento, che possono essere scaricate dal nostro sito web (https://pragprog.com/the-pragmatic-programmer/source-code). Eseguitele sulle diverse macchine che avete a disposizione. I valori che ottenete seguono le curve previste? Che cosa potete dedurne circa la velocità relativa delle vostre macchine? Quali sono gli effetti di varie impostazioni di ottimizzazione del compilatore? Il radix sort è effettivamente lineare?

35. La routine qui sotto stampa i contenuti di un albero binario. Ipotizzando che l'albero sia bilanciato, all'incirca quanto spazio di stack userà la routine nello stampare un albero di 1.000.000 di elementi? (Ipotizzate che le chiamate di routine non impongano un overhead significativo allo stack.)

```
void printTree(const Node *node) {
  char buffer[1000];
  if (node) {
    printTree(node->left);
```

```
    getNodeAsString(node, buffer);
    puts(buffer);
    printTree(node->right);
  }
}
```

36. Vedete qualche modo per ridurre i requisiti di stack della routine nell'Esercizio 35 (a parte ridurre le dimensioni del buffer)?
37. Abbiamo sostenuto che un processo dicotomico è $O(\lg(n))$. Potete dimostrarlo?

Refactoring

Cambiamento e decadimento in tutto quel che vedo intorno…
- H. R. Lyte, *Abide With Me*

Con l'evoluzione di un programma, si renderà necessario ripensare decisioni prese in precedenza e rielaborare parti del codice. È un processo del tutto naturale: le esigenze evolvono, il codice non è una cosa statica.

Purtroppo, la metafora più comune per lo sviluppo di software è quella dell'edilizia: si costruisce un edificio (Bertrand Meyer [Mey97b parla di "Software Construction"). L'uso di questa metafora però implica alcuni passaggi.

1. Un architetto disegna le cianografie.
2. Le imprese appaltatrici scavano le fondamenta, costruiscono la sovrastruttura, installano i fili e le tubature, poi applicano le rifiniture.
3. I proprietari si trasferiscono e vivono per sempre felici e contenti, chiamando la manutenzione per sistemare gli eventuali problemi.

Il software non funziona proprio così. Più che all'edilizia, il software assomiglia al *giardinaggio*, è più materia organica che cemento armato. Piantate molte cose in un giardino in base a un piano iniziale e alle condizioni esistenti. Qualche specie si sviluppa bene, altre finiscono nel compost. Potete spostare un albero rispetto a un altro per sfruttare il gioco di luce e ombra, vento e pioggia. Piante troppo cresciute vengono potate, i colori che fanno a pugni possono essere spostati in posizioni esteticamente più piacevoli. Si strappano le erbacce, si dà fertilizzante a piante che hanno bisogno di un po' di aiuto. Si tiene costantemente

sott'occhio la salute del giardino e si fanno aggiustamenti (al suolo, alle piante, alla disposizione) secondo necessità.

Gli uomini d'azienda si trovano più a loro agio con la metafora della costruzione di un edificio: è più scientifica del giardinaggio, è ripetibile, esiste una gerarchia rigida per la gestione e così via. Ma noi non costruiamo grattacieli - non siamo altrettanto vincolati dai limiti della fisica e dal mondo reale.

La metafora del giardinaggio è molto più vicina alla realtà dello sviluppo del software. Forse una certa routine è diventata troppo grande, o tenta di fare troppo: bisogna dividerla in due. Le cose che non funzionano come previsto vanno strappate come se fossero erbacce o vanno potate.

Riscrivere, rielaborare e ristrutturare l'architettura del codice sono attività raggruppate collettivamente sotto il nome di *refactoring*.

Refactoring: quando?

Quando vi imbattete in un ostacolo perché il codice non quadra più, o notate due cose che in realtà andrebbero fuse insieme, o qualche altra cosa vi colpisce come "sbagliata", *non esitate a cambiare*. Non c'è come il presente. Sono molte le cose che possono spingere a un refactoring.

- *Duplicazione*. Avete scoperto una violazione del principio DRY (*I mali della duplicazione*, Capitolo 2).
- *Progettazione non ortogonale*. Avete scoperto codice o un progetto che potrebbero essere resi più ortogonali (*Ortogonalità*, Capitolo 2).
- *Conoscenza obsoleta*. Le cose cambiano, i requisiti si spostano e la vostra conoscenza del problema migliora. Il codice deve essere sempre all'altezza.
- *Prestazioni*. Dovete spostare funzionalità da un'area del sistema a un'altra, per migliorare le prestazioni.

Il refactoring del codice, ovvero lo spostamento di funzionalità e l'aggiornamento di decisioni precedenti, è in realtà un esercizio di *gestione del dolore*. Ammettiamolo, mettersi a cambiare codice sorgente può essere molto doloroso: quasi funzionava, e adesso è *proprio* a pezzi.

Molti sviluppatori sono riluttanti a fare a pezzi il loro codice solo perché non è del tutto giusto.

Complicazioni del mondo reale

Dunque, andate dal vostro capo o dal cliente e dite “Questo codice funziona, ma ho bisogno di un’altra settimana per ristrutturarlo”.

Per decenza, non possiamo riportare la risposta.

La pressione del tempo spesso viene usata come scusa per non operare il refactoring, ma la scusa non regge: se non ristrutturate ora, sarà necessario un investimento di tempo molto maggiore per sistemare il problema più avanti, quando ci saranno ancora più dipendenze con cui fare i conti. Ci sarà più tempo allora? In base alla nostra esperienza, non succede mai.

Potreste cercare di spiegare questo principio al vostro capo utilizzando un’analogia medica: pensate il codice che ha bisogno di refactoring come una “escrescenza”. Eliminarla richiede un intervento chirurgico invasivo. Potete procedere oggi ed eliminarla mentre è ancora piccola; oppure potete aspettare che cresca e si diffonda, ma a quel punto eliminarla sarà più costoso e anche più pericoloso. Aspettate ancora di più, e potreste perdere del tutto il paziente.

SUGGERIMENTO 47
Procedete al refactoring presto, procedete al refactoring spesso.

Tenete traccia delle cose che devono essere ristrutturate. Se non potete sottoporre qualcosa al refactoring subito, assicuratevi che venga inserita nel piano di lavoro. Assicuratevi che gli utenti del codice interessato *sappiano* che è pianificato un refactoring e quante ricadute potrà avere su di loro.

Come si procede al refactoring?

Di refactoring si è cominciato a parlare nella comunità di Smalltalk e, con altre tendenze (come gli schemi di progettazione), ha iniziato poi a diffondersi, ma rimane un argomento piuttosto nuovo: non c’è molto di pubblicato. Il primo libro importante sul refactoring [FBB+99] è stato pubblicato quasi contemporaneamente alla prima edizione di questo libro.

Fondamentalmente, il refactoring è riprogettazione, o *redesign*. Qualsiasi cosa abbiate progettato, voi o altri nel vostro team, può essere riprogettata alla luce di nuovi fatti, di una conoscenza più approfondita, di requisiti che sono cambiati e così via. Ma se procedete a fare a pezzi grandi quantità di codice con furia selvaggia potreste ritrovarvi in una posizione peggiore di quella da cui siete partiti.

Chiaramente, il refactoring è un'attività che deve essere intrapresa lentamente, consapevolmente con molta attenzione. Martin Fowler offre alcuni semplici suggerimenti su come effettuare il refactoring senza fare più danni che bene (vedete il riquadro a p. 30 di [FS97]).

1. Non tentate di effettuare un refactoring e di aggiungere funzionalità allo stesso tempo.
2. Assicuratevi di avere dei buoni test prima di iniziare il refactoring. Eseguite i test il più spesso possibile. Così saprete rapidamente se i cambiamenti hanno rovinato qualcosa.
3. Fate passi brevi e molto consapevoli: spostate un campo da una classe all'altra, fondete due metodi simili in una superclasse. Il refactoring spesso comporta molti cambiamenti puntuali che hanno come risultato un cambiamento su scala più grande. Se continuate a procedere per piccoli passi ed eseguite test dopo ogni passo, eviterete un lungo debug.

Refactoring automatico

Storicamente, gli utenti di Smalltalk hanno sempre avuto a disposizione un *browser di classe* nel loro IDE. Da non confondere con i browser web, i browser di classe consentono agli utenti di navigare ed esaminare gerarchie di classi e metodi.

Normalmente, i browser di classe consentono di modificare il codice, creare nuovi metodi e classi e così via. La variazione successiva sullo stesso tema è il *browser di refactoring*.

Un browser di refactoring può svolgere in modo semiautomatico operazioni comuni di refactoring: suddividere una lunga routine in routine più piccole, propagare automaticamente cambiamenti nei nomi di metodi e variabili, effettuare il drag and drop per facilitare lo spostamento del codice e così via.

Nel momento in cui è stata scritta la prima edizione di questo libro, questa tecnologia non era ancora uscita dal mondo Smalltalk, ma si è poi diffusa con una certa rapidità. Il browser di refactoring di Smalltalk si può trovare online a [URL20].

Parleremo ulteriormente di test a questo livello in *Codice facile da sottoporre a test* (più avanti in questo capitolo) e di test su grande scala in

Test senza pietà (Capitolo 8), ma il consiglio di Fowler di mantenere buoni test di regressione è la chiave per effettuare il refactoring con fiducia.

Può essere utile anche assicurarsi che cambiamenti drastici in un modulo (per esempio, una modifica all'interfaccia o alle sue funzionalità in modo incompatibile) facciano fallire la build. Vecchi clienti del codice, cioè, devono fallire nella compilazione. Potete quindi trovare facilmente i vecchi clienti e apportare le modifiche necessarie per aggiornarli.

Quindi, la prossima volta che vedete un pezzo di codice che non è proprio come dovrebbe essere, sistematelo e sistemate tutto quello che ne dipende. Gestite il dolore: se vi fa soffrire adesso, ma vi farà ancora più male in seguito, meglio darci subito un taglio. Ricordate gli insegnamenti di *Entropia del software* (Capitolo 1): non vivete con finestre rotte.

Vedi anche

- *Il gatto mi ha mangiato il codice sorgente*, Capitolo 1
- *Entropia del software*, Capitolo 1
- *Zuppa di pietre e rane bollite*, Capitolo 1
- *I mali della duplicazione*, Capitolo 2
- *Ortogonalità*, Capitolo 2
- *Programmazione per coincidenza*, in questo Capitolo
- *Codice facile da sottoporre a test*, in questo Capitolo
- *Test senza pietà*, Capitolo 8

Esercizi

38. Il codice seguente è stato ovviamente aggiornato varie volte nel corso degli anni, ma i cambiamenti non ne hanno certo migliorato la struttura. Sottoponetelo a refactoring.

```
if (state == TEXAS) {
  rate = TX_RATE;
  amt = base * TX_RATE;
  calc = 2*basis(amt) + extra(amt)*1.05;
}
else if ((state == OHIO) || (state == MAINE)) {
  rate = (state == OHIO) ? OH_RATE : ME_RATE;
  amt  = base * rate;
  calc = 2*basis(amt) + extra(amt)*1.05;
  if (state == OHIO)
    points = 2;
}
else {
  rate = 1;
  amt = base;
  calc = 2*basis(amt) + extra(amt)*1.05;
}
```

39. La classe Java che segue deve supportare qualche altra forma. Effettuate il refactoring della classe per prepararla all'aggiunta.

```
public class Shape {
  public static final int SQUARE = 1;
  public static final int CIRCLE = 2;
  public static final int RIGHT_TRIANGLE = 3;
  private int shapeType;
  private double size;
  public Shape(int shapeType, double size) {
    this.shapeType = shapeType;
    this.size = size;
}
// ... altri metodi ...

  public double area() {
    switch (shapeType) {
    case SQUARE:   return size*size;
    case CIRCLE:   return Math.PI*size*size/4.0;
    case RIGHT_TRIANGLE: return size*size/2.0;
    }
    return 0;
  }
}
```

40. Questo codice Java fa parte di un framework che verrà usato in tutto il vostro progetto. Sottoponetelo a refactoring in modo che sia più generale e che sia più facile estenderlo in futuro.

```
public class Window {
  public Window(int width, int height) { ... }
  public void setSize(int width, int height) { ... }
  public boolean overlaps(Window w) { ... }
  public int getArea() { ... }
}
```

Codice facile da sottoporre a test

La metafora del "software come circuito integrato" (*Software IC*) viene usata quando si parla di riusabilità e di sviluppo basato su componenti: l'espressione, a quanto pare, è stata usata per la prima volta nel 1986 da Cox e Novobilski nel loro *Object-Oriented Programming* [CN91]. L'idea è che i componenti software si dovrebbero combinare come i chip dei circuiti integrati; funziona però solo se è nota l'affidabilità dei componenti che si usano.

I chip sono progettati in modo da essere sottoposti a test, non solo in fabbrica, non solo quando vengono installati, ma anche sul campo quando vengono messi in esercizio. Chip e sistemi più complessi possono avere una funzione Built-In Self Test (BIST) che esegue internamente

qualche diagnosi di livello base, o un Test Access Mechanism (TAM) che mette a disposizione un sistema grazie al quale l'ambiente esterno può fornire stimoli e raccogliere risposte dal chip.

Possiamo fare la stessa cosa nel software. Come i nostri colleghi che lavorano sull'hardware dobbiamo incorporare sin dall'inizio nel software la possibilità di essere sottoposto a test, e collaudare ampiamente tutti i pezzi prima di cercare di collegarli fra loro.

Unit test

Il test a livello di chip per l'hardware è all'incirca equivalente allo *unit test* per il software, un test eseguito su ciascun modulo isolatamente, per verificare il suo comportamento. Possiamo avere un'idea migliore di come reagirà un modulo nel grande mondo se lo collaudiamo in condizioni controllate, anche artificiose.

Uno unit test è codice che mette alla prova un modulo. Normalmente, creerà qualche tipo di ambiente artificiale, poi invocherà le routine nel modulo e verificherà i risultati restituiti, confrontandoli con valori noti o con i risultati di precedenti esecuzioni dello stesso test (test di regressione).

In seguito, quando assembliamo i nostri "circuiti integrati software" in un sistema completo, avremo fiducia che le singole parti funzionino come previsto e poi potremo usare gli stessi strumenti di unit test per collaudare il sistema nel suo complesso. Parliamo di questo controllo su grande scala del sistema in *Test senza pietà* (Capitolo 8).

Prima di arrivare a quel punto, però, dobbiamo decidere che cosa sottoporre a test a livello di unità. Normalmente, i programmatori danno in pasto al codice un po' di bit casuali di dati e lo dichiarano collaudato, ma noi possiamo fare molto meglio, utilizzando le idee alla base della *progettazione per contratto*.

Test rispetto al contratto

Ci piace pensare lo unit test come *test rispetto al contratto* (vedi *Progettare per contratto*, nel Capitolo 4). Vogliamo scrivere casi di test che ci garantiscano che una data unità onora il suo contratto e questo ci

dirà due cose: se il codice rispetta il contratto e se il contratto significa quello che pensiamo significhi. Vogliamo verificare che il modulo fornisca proprio le funzionalità che promette, per un ampio spettro di casi di test e di condizioni limite.

Che cosa significa, in pratica? Prendiamo la routine della radice quadrata che abbiamo incontrato nel Capitolo 4. Il suo contratto è semplice:

```
require
  argument >= 0;
ensure
  ((Result * Result) - argument).abs <= epsilon*argument;
```

Questo ci dice che cosa sottoporre a test.

- Passiamo un argomento negativo e verifichiamo che venga rifiutato.
- Passiamo un argomento zero per essere sicuri che sia accettato (questo è il valore limite).
- Passiamo valori compresi fra zero e il massimo argomento esprimibile e verifichiamo che la differenza fra il quadrato del risultato e l'argomento originale sia minore di qualche frazione piccola dell'argomento.

Armati di questo contratto, e assumendo che la nostra routine faccia il suo controllo delle precondizioni e postcondizioni, possiamo scrivere uno script di test base per mettere alla prova la funzione radice quadrata.

```
public void testValue(double num, double expected) {
  double result = 0.0;
  try {                         // Possiamo lanciare una
    result = mySqrt(num);       // eccezione di precondizione
  }
  catch (Throwable e) {
    if (num < 0.0)              // Se l'input è < 0, allora
      return;                   // ci aspettiamo l'eccezione
    else                        // altrimenti
      assert(false);            // il test fallisce
  }
  assert(Math.abs(expected-result) < epsilon*expected);
}
```

Poi possiamo chiamare questa routine per collaudare la nostra funzione radice quadrata.

```
testValue(-4.0, 0.0);
testValue( 0.0, 0.0);
testValue( 2.0, 1.4142135624);
```

```
testValue(64.0, 8.0);
testValue(1.0e7, 3162.2776602);
```

Questo è un test molto semplice; nel mondo reale, ogni modulo non banale è probabile dipenda da vari altri moduli, perciò come procediamo a sottoporre a test la combinazione?

Supponiamo di avere un modulo `A` che usa una `LinkedList` e un `Sort`. Nell'ordine, sottoporremmo a test:

1. Il contratto completo di `LinkedList`;
2. Il contratto completo di `Sort`;
3. Il contratto di `A`, che si basa sugli altri contratti ma non li espone direttamente.

Questo stile di test richiede che si verifichino prima i sottocomponenti di un modulo. Una volta verificati quelli, si può sottoporre al test il modulo stesso.

Se `LinkedList` e `Sort` hanno passato il test, ma `A` no, possiamo essere abbastanza sicuri che il problema stia in `A`, o nel modo in cui `A` usa uno di quei sottocomponenti. Questa tecnica è un ottimo modo per ridurre la fatica del debug: ci possiamo concentrare rapidamente sulla probabile fonte del problema nel modulo `A` e non sprecare tempo a riesaminare i suoi sottocomponenti.

Perché ci diamo così tanto da fare? Soprattutto, vogliamo evitare di creare una "bomba a tempo", qualcosa che se ne sta lì senza farsi notare e poi scoppi in un momento strano più avanti nel progetto. Sottolineando il test rispetto al contratto, possiamo cercare di evitare il maggior numero possibile di quei disastri a valle.

SUGGERIMENTO 48
Progettate per sottoporre a test.

Quando progettate un modulo, o anche una singola routine, dovete progettare sia il suo contratto, sia il codice per sottoporre a test quel contratto. Progettando codice che superi un test e ottemperi al suo contratto, possiamo considerare condizioni limite e altri problemi che altrimenti magari non verrebbero in mente. Non esiste modo migliore di risolvere gli errori che cercare di evitarli in partenza. In effetti, costruendo i test *prima* di implementare il codice, mettete alla prova l'interfaccia prima di impegnarvi ad adottarla.

Scrivere unit test

Gli unit test per un modulo non devono essere nascosti in qualche angolo lontano dell'albero del sorgente, devono essere collocati in una posizione comoda. Per progetti di piccole dimensioni, potete incorporare lo unit test per un modulo nel modulo stesso; nel caso di progetti più grandi, consigliamo di spostare ciascun test in una sottodirectory. In un modo come nell'altro, ricordate che, se non è facile trovarlo, non verrà usato.

Rendendo facilmente accessibile il codice di test, fornite agli sviluppatori che possono usare il vostro codice due risorse preziosissime.

1. Esempi di come usare tutte le funzionalità del vostro modulo.
2. Un mezzo per costruire test di regressione e validare qualsiasi futuro cambiamento nel codice.

È comodo, ma non sempre fattibile, che ogni classe o modulo contenga il proprio unit test. In Java, per esempio, ogni classe può avere la propria `main`. In tutti i file, tranne in quello della classe principale dell'applicazione, si può usare la routine `main` per eseguire unit test; verrà ignorata quando viene eseguita l'applicazione stessa. Questo ha il vantaggio che il codice che consegnate conterrà ancora i test, che potranno essere usati per diagnosticare problemi sul campo.

In C++ potete ottenere lo stesso effetto (alla compilazione) utilizzando `#ifdef` per compilare selettivamente il codice di unit test. Per esempio, ecco un semplicissimo unit test in C++ incorporato nel nostro modulo, che verifica la nostra funzione radice quadrata utilizzando una routine `testValue` simile a quella Java vista in precedenza.

```
#ifdef __TEST__
int main(int argc, char **argv)
{
  argc--; argv++; // salta il nome del programma
  if (argc < 2) { // esegue i test standard se non ci sono argomenti
    testValue(-4.0, 0.0);
    testValue( 0.0, 0.0);
    testValue( 2.0, 1.4142135624);
    testValue(64.0, 8.0);
    testValue(1.0e7, 3162.2776602);
  }
  else { // altrimenti usa gli argomenti
    double num, expected;
```

```
    while (argc >= 2) {
      num = atof(argv[0]);
      expected = atof(argv[1]);
      testValue(num,expected);
      argc -= 2;
      argv += 2;
    }
  }
  return 0;
}
#endif
```

Questo unit test o eseguirà un insieme minimo di test o, se gli vengono dati gli argomenti, consentirà il passaggio di dati dal mondo esterno. Uno script di shell potrebbe usare questa possibilità per eseguire un insieme di test molto più ricco.

Che cosa fate se la risposta corretta per uno unit test è uscire, o abortire il programma? In quel caso dovete poter selezionare il test da eseguire, magari specificando un argomento dalla riga di comando. Dovrete anche passare dei parametri, se dovete specificare condizioni iniziali differenti per i vostri test.

Fornire unit test non è sufficiente. Dovete eseguirli, ed eseguirli spesso. È utile se la classe *supera* i suoi test ogni tanto.

Test harness

Di solito scriviamo *molto* codice di test e svolgiamo una gran quantità di test, perciò ci renderemo la vita più facile e svilupperemo un *test harness* (letteralmente una "imbragatura di test") per il progetto. La `main` vista nella sezione precedente è un caso semplice di test harness, ma di solito ci serviranno funzionalità più estese.

Un test harness può gestire operazioni comuni come registrare lo stato, analizzare l'output in cerca di risultati previsti e selezionare ed eseguire i test. Gli harness possono essere guidati da GUI, possono essere scritti nello stesso linguaggio del resto del progetto, oppure possono essere implementati come una combinazione di `makefile` e script Perl. Un semplice test harness è presentato nella risposta all'Esercizio 41 (Appendice B).

In linguaggi e ambienti a oggetti, si può creare una classe base che fornisca queste operazioni comuni. I singoli test possono essere sottoclassi, con l'aggiunta di codice specifico. Potete usare una

convenzione standard per i nomi e la riflessione in Java per costruire dinamicamente una lista di test. Questa tecnica è un bel modo di applicare il principio DRY: non dovete mantenere una lista di test disponibili. Prima di partire a scrivere il vostro harness, potete magari vedere lo xUnit di Kent Beck ed Erich Gamma [URL22] e magari anche il nostro libro *Pragmatic Unit Testing* [HT03] per una introduzione a JUnit.

Indipendentemente dalla tecnologia che decidete di usare, i test harness devono includere queste capacità:

- un modo standard per specificare impostazione e pulizia;
- un metodo per selezionare singoli test o tutti i test disponibili:
- uno strumento per analizzare l'output e trovare i risultati attesi (o quelli inattesi);
- una forma standardizzata di segnalazione dei fallimenti.

I test devono essere componibili: in altre parole, un test può essere composto da sottotest di sottocomponenti fino a qualsiasi profondità. Possiamo usare questa caratteristica per sottoporre a test con pari facilità parti specifiche del sistema o tutto il sistema, utilizzando gli stessi strumenti.

Test ad hoc
Nel corso del debug, possiamo finire per creare qualche test particolare al volo. Può trattarsi di qualcosa di molto semplice come un enunciato `print`, o di un pezzo di codice inserito interattivamente in un debugger o nell'ambiente IDE.
Alla fine della sessione di debug, dovete formalizzare il test ad hoc. Se il codice ha dato un errore una volta, è probabile che si blocchi ancora. Non buttate via il test che avete creato; aggiungetelo allo unit test esistente.

Per esempio, usando JUnit (il membro Java della famiglia xUnit), potremmo scrivere il nostro test per la radice quadrata in questo modo:

```
public class JUnitExample extends TestCase {
  public JUnitExample(final String name) {
    super(name);
  }
  protected void setUp() {
    // Carica dati di test...
    testData.addElement(new DblPair(-4.0,0.0));
    testData.addElement(new DblPair(0.0,0.0));
    testData.addElement(new DblPair(64.0,8.0));
    testData.addElement(new DblPair(Double.MAX_VALUE,
```

```
                                        1.3407807929942597E154));
  }
  public void testMySqrt() {
    double num, expected, result = 0.0;
    Enumeration enum = testData.elements();
    while (enum.hasMoreElements()) {
      DblPair p = (DblPair)enum.nextElement();
      num       = p.getNum();
      expected  = p.getExpected();
      testValue(num, expected);
    }
  }
  public static Test suite() {
      TestSuite suite= new TestSuite();
      suite.addTest(new JUnitExample("testMySqrt"));
      return suite;
  }
}
```

JUnit è progettato in modo da essere componibile: possiamo aggiungere quanti test vogliamo a questa suite, e ciascun test potrebbe essere a sua volta una suite. Inoltre, si ha la scelta fra un'interfaccia grafica e un'interfaccia batch per controllare i test.

Costruire una finestra di test

Anche i migliori gruppi di test è improbabile trovino tutti gli errori: c'è qualcosa nelle condizioni di un ambiente di produzione che sembra spingerli in modo particolare a venire allo scoperto.

Questo significa che spesso dovrete sottoporre a test un pezzo di software dopo che è stato messo in esercizio, con dati del mondo reale che fluiscono nelle sue vene. A differenza di una piastra circuitale o di un chip, non abbiamo piedini di test nel software, ma *possiamo* avere varie visualizzazioni dello stato interno di un modulo, senza usare il debugger (che può essere scomodo o impossibile in un'applicazione in produzione).

I file di log che contengono messaggi di traccia sono uno di questi meccanismi. I messaggi di log devono essere in un formato regolare e coerente: potreste volerli analizzare automaticamente per dedurre i tempi di elaborazione o i percorsi logici seguiti dal programma. Diagnostiche formattate male o in modo incoerente sono difficili da leggere e impraticabili per un analizzatore sintattico.

Un altro meccanismo per guardare nel codice in esecuzione è la sequenza dei "tasti caldi": quando si preme questa particolare

combinazione di tasti, compare una finestra di controllo diagnostico con messaggi di stato e così via. È una cosa che di solito non si rivela agli utenti finali, ma può essere molto comoda per l'assistenza.

Per codice server più grande e più complesso, una buona tecnica per vedere il suo funzionamento è includere un server web. Chiunque può mandare un browser web alla porta http dell'applicazione (che di solito è un numero non standard, per esempio 8080) e vedere lo stato interno, le voci di log e magari anche qualche forma di pannello di controllo per il debug. Può sembrare difficile da implementare, ma non lo è. Server web HTTP liberamente disponibili e incorporabili si possono trovare in vari linguaggi moderni. Un buon posto da cui iniziare l'esplorazione è [URL58].

Una cultura del test

Tutto il software che scriverete *verrà* sottoposto a test - se non da voi e dal vostro team, dagli eventuali utenti - perciò meglio pianificare di metterlo ampiamente alla prova. Un po' di considerazione in anticipo può fare molto per ridurre al minimo i costi di manutenzione e le chiamate all'help desk.

Nonostante la fama di hacker, la comunità di Perl è molto dedita ai test di unità e di regressione. La procedura standard di installazione dei moduli Perl supporta un test di regressione, invocando

```
% make test
```

Non c'è nulla di magico in Perl sotto questo aspetto. Perl rende più facile raccogliere e analizzare i risultati dei test per garantire la conformità, ma il grande vantaggio è semplicemente che si tratta di uno standard: i test vanno in un certo posto e hanno un certo output previsto. *Il test è più un fatto culturale che tecnico*: possiamo instillare questa cultura del test in un progetto, indipendentemente dal linguaggio utilizzato.

SUGGERIMENTO 49
Sottoponi a test il tuo software, o lo farà il tuo utente.

Vedi anche

- *Il gatto mi ha mangiato il codice sorgente*, Capitolo 1

- *Ortogonalità*, Capitolo 2
- *Progettare per contratto*, Capitolo 4
- *Refactoring*, in questo Capitolo
- *Test senza pietà*, Capitolo 8

Esercizi

41. Progettate una sessione di test per l'interfaccia del miscelatore descritta nella risposta all'Esercizio 17 (Appendice B). Scrivete uno script di shell che esegua un test di regressione per il miscelatore. Dovete sottoporre a test funzionalità base, condizioni di errore e al limite e tutti gli obblighi contrattuali. Quali restrizioni sono imposte ai cambiamenti di velocità? Sono rispettate?

Maghi cattivi

È innegabile: è sempre più difficile scrivere le applicazioni. Le interfacce utente in particolare sono sempre più raffinate. Una trentina di anni fa, una applicazione in media avrebbe avuto una semplice interfaccia da telescrivente (sempre ammesso che avesse un'interfaccia). I terminali asincroni di norma avevano una visualizzazione interattiva a caratteri, mentre i dispositivi interrogabili (come gli onnipresenti IBM 3270) permettevano di riempire una intera schermata prima di premere il pulsante di invio. Ora gli utenti si aspettano interfacce grafiche con aiuto sensibile al contesto, taglia e incolla, trascinamento, integrazione OLE e MDI o SDI. Gli utenti vogliono l'integrazione con il browser web e il supporto per i client "magri".

Le applicazioni stesse diventano sempre più complesse. Nella maggior parte dei casi gli sviluppi usano un modello *multitier*, magari con qualche livello di middleware o un monitor delle transazioni. Ci si aspetta che questi programmi siano dinamici e flessibili e che siano in grado di interoperare con applicazioni scritte da terze parti.

Oh, abbiamo dimenticato di dire che serve tutto per la settimana prossima?

Gli sviluppatori fanno fatica a stare al passo. Se usassimo gli stessi tipi di strumenti con cui si producevano le applicazioni di base per i terminali stupidi trent'anni fa, non riusciremmo a combinare mai niente.

I produttori di strumenti e i fornitori di infrastruttura hanno tirato fuori dal cappello la pallottola magica, il *wizard* (letteralmente: mago), la procedura di autocomposizione. I wizard sono grandi. Hai bisogno di una

applicazione MDI con supporto per contenitori OLE? Fai clic su un pulsante, rispondi a un paio di semplici domande e il maghetto genera automaticamente lo scheletro di codice per te. L'ambiente Microsoft Visual C++ crea automaticamente oltre 1200 righe di codice per questo scenario. I wizard sono attivissimi anche in altri contesti. Puoi usarli per creare componenti server, implementare bean Java e gestire interfacce di rete - tutti campi complessi, in cui è bello avere l'aiuto di un esperto.

Usare un wizard progettato da un guru non fa automaticamente del nostro amico Pietro uno sviluppatore altrettanto esperto. Pietro può sentirsi bravissimo - ha appena prodotto una gran quantità di codice e un programma che fa una gran figura. Aggiunge le funzionalità specifiche dell'applicazione ed è pronto per la consegna. Ma, a meno che non capisca davvero il codice che è stato prodotto per suo conto, si sta prendendo in giro. Programma per coincidenza. I wizard sono una via a senso unico: ritagliano il codice per voi, poi vanno per la loro strada. Se il codice che producono non è del tutto giusto, o se le circostanze cambiano e dovete adattare il codice, dovete cavarvela da soli.

Non siamo contrari ai wizard. Al contrario, abbiamo dedicato un'intera sezione (*Generatori di codice*, nel Capitolo 3) a come scriverne di propri. Ma se *usate* un wizard, e non capite tutto il codice che produce, non avrete il controllo della vostra applicazione. Non sarete in grado di manutenerla e sarete in difficoltà quando verrà il momento del debug.

SUGGERIMENTO 50
Non usate il codice dei wizard se non lo capite.

Qualcuno pensa che sia una posizione da estremisti. Dicono che lo sviluppatore si basa sempre su cose che non capisce del tutto - la meccanica quantistica dei circuiti integrati, la struttura degli interrupt del processore, gli algoritmi utilizzati per pianificare i processi, il codice nelle librerie fornite, e così via, Siamo d'accordo, e penseremmo la stessa cosa dei wizard se fossero semplicemente un insieme di chiamate a libreria o di servizi standard del sistema operativo, su cui chi sviluppa potrebbe fare affidamento - ma non lo sono. I wizard generano codice che diventa parte integrante dell'applicazione di Pietro. Il codice non è nascosto dietro una bella interfaccia, è intrecciato riga per riga con le funzionalità che Pietro scrive. Alla fine, smette di essere codice del wizard e comincia a essere codice di Pietro. E nessuno deve produrre codice che non capisce fino in fondo. (Esistono però anche altre tecniche

che aiutano a gestire la complessità: ne abbiamo viste due in *Ortogonalità*, nel Capitolo 2.)

Vedi anche

- *Ortogonalità*, Capitolo 2
- *Generatori di codice*, Capitolo 3

Sfide

- Avete a disposizione un wizard che costruisce GUI e lo usate per generare lo scheletro di un'applicazione. Esaminate ogni riga del codice prodotto. Vi è tutto chiaro? Avreste potuto produrlo anche da soli? L'avreste prodotto così anche voi, o fa cose di cui non avete bisogno?

Prima del progetto

Vi è mai capitato di avere la sensazione che il vostro progetto fosse condannato al fallimento, ancor prima di iniziare? E a volta può essere davvero così, se non si stabiliscono prima delle solide regole di base; altrimenti, meglio consigliare di chiudere subito tutto e far risparmiare un po' di soldi allo sponsor.

Proprio all'inizio di un progetto, dovete stabilire quali siano i requisiti: stare ad ascoltare gli utenti non basta, leggete in proposito *La fossa dei requisiti*.

La saggenzza convenzionale e la gestione dei vincoli sono i temi di *Risolvere rompicapo impossibili*. Problemi difficili spunteranno in tutte le fasi, da quella dei requisiti all'analisi, dalla codifica ai test. Nella maggior parte dei casi non saranno in realtà così difficili come appaiono a prima vista.

Quando pensate di aver risolto i problemi, potreste comunque sentirvi a disagio e non aver voglia di buttarvi a capofitto. È semplice procrastinazione, o è qualcosa di più? *Non finché non siete pronti* offre qualche consiglio su quando esercitare un po' di prudenza e dare ascolto alla vocina dentro la vostra testa.

Iniziare troppo presto è un problema, ma aspettare troppo può essere ancora più grave. Ne *La trappola delle specifiche*, analizzeremo i vantaggi della specifica mediante esempi.

Infine, in *Cerchi e frecce*, considereremo alcuni degli ostacoli dei processi e delle metodologie di sviluppo formale. Non importa quanto sia stato ben ponderato e non importa quali "buone pratiche" includa, non c'è metodo che possa sostituire la capacità di *pensare*.

Con questi aspetti fondamentali affrontati *prima* che il progetto sia avviato, potete essere in una posizione migliore per evitare la "paralisi

dell'analisi" e iniziare davvero un progetto di successo.

La fossa dei requisiti

> La perfezione si ottiene non quando non resta più nulla da aggiungere, ma quando non resta più niente da eliminare.
> - Antoine de St. Exupery, *Vento, sabbia e stelle*, 1949

Molti libri e molti tutorial indicano nella *raccolta dei requisiti* (*requirement gathering*) una delle prime fasi del progetto. Il termine "raccolta" (come l'inglese *gathering*) suscita l'immagine di una tribù di analisti felici, che bottinano frammenti di saggezza sparsi sul terreno tutto attorno, mentre in sottofondo l'orchestra esegue la *Sinfonia Pastorale*. "Raccolta" sembra implicare che i requisiti siano già lì e che si tratti soltanto di vederli, metterli in un cestino e andare avanti allegramente per la propria strada. Ma non funziona così. Raramente i requisiti sono lì in superficie: normalmente sono sepolti sotto strati di presupposti, idee sbagliate e preoccupazioni politiche.

SUGGERIMENTO 51
I requisiti non si "raccolgono", bisogna scavare per identificarli.

Scavare alla ricerca dei requisiti

Come fate a riconoscere un vero requisito mentre scavate in mezzo allo sporco circostante? La risposta è al tempo stesso semplice e complessa.

La risposta semplice è che un requisito è l'enunciazione di qualcosa che deve essere ottenuto. Buoni requisiti possono essere i seguenti.

- Il record di un dipendente può essere visto solo da un gruppo di persone indicate esplicitamente.
- La temperatura della testa del cilindro non deve superare un valore critico, che varia da motore a motore.
- L'editor evidenzierà le parole chiave, che verranno selezionate in funzione del tipo di file che si sta modificando.

Pochi requisiti però sono così netti, ed è questo che rende complessa l'analisi dei requisiti.

Il primo enunciato nell'elenco precedente potrebbe essere stato formulato dagli utenti come "Solo i supervisori di un dipendente e l'ufficio del personale possono vedere i record di quel dipendente". Questa affermazione è un vero requisito? Forse oggi, ma incorpora in un enunciato assoluto un elemento di norma aziendale: questa cambia regolarmente, perciò probabilmente non vogliamo che sia inserita in modo rigido nei nostri requisiti. Il nostro consiglio è di documentare queste norme in forma separata dal requisito, e creare un collegamento ipertestuale fra le due cose. Il requisito diventa un enunciato generale e agli sviluppatori viene fornita l'informazione sulle norme aziendali come esempio del tipo di cose che dovranno supportare nell'implementazione. Alla fine, nell'applicazione le norme possono essere rappresentate come metadati.

È una distinzione abbastanza sottile, ma per gli sviluppatori avrà implicazioni profonde. Se il requisito è formulato come "Solo l'ufficio del personale può vedere il record di un dipendente", lo sviluppatore può finire per codificare un test esplicito ogni volta che l'applicazione accede a questi file. Se l'enunciato invece è "Solo gli utenti autorizzati possono accedere al record di un dipendente", lo sviluppatore probabilmente progetterà e realizzerà qualche tipo di sistema di controllo degli accessi. Quando le norme aziendali cambieranno (e cambieranno sicuramente), dovranno essere aggiornati solo i metadati di quel sistema. In effetti, raccogliere i requisiti in questo modo porta con naturalezza a un sistema che è ben strutturato per supportare i metadati.

La distinzione fra requisiti, norme aziendali e implementazione può assumere contorni molto sfumati quando si parla di interfacce utente. "Il sistema deve permettere di scegliere la scadenza di un prestito" è l'enunciazione di un requisito. "Ci serve una casella di riepilogo per selezionare la scadenza del prestito" può esserlo e può non esserlo. Se gli utenti devono avere assolutamente una casella di riepilogo, allora è un requisito; se invece stanno semplicemente indicando la possibilità di effettuare una scelta, ma usano il termine *casella di riepilogo* come un esempio, può non esserlo. Il box che segue, *Qualche volta l'interfaccia è il sistema*, analizza un progetto che è andato terribilmente storto perché sono stati ignorati i bisogni dell'interfaccia utente.

È importante scoprire il motivo di fondo *per cui* gli utenti fanno una particolare cosa, non solo *il modo* in cui oggi lo fanno. Alla fine, il vostro sviluppo deve risolvere il loro *problema di business*, non semplicemente soddisfare i requisiti come li hanno formulati. Documentare le ragioni alla base dei requisiti darà al vostro team informazioni preziose, quando dovrete prendere quotidianamente decisioni di implementazione.

Esiste una tecnica semplice, per "entrare dentro" i requisiti degli utenti, tecnica che purtroppo non è usata abbastanza spesso: diventare un utente. Dovete scrivere un sistema per l'help desk? Passate un paio di giornate a osservare le telefonate con una persona esperta dell'assistenza. Dovete automatizzare un sistema di controllo manuale delle giacenze? Lavorate in magazzino per una settimana. (Una settimana vi sembra troppo? Non lo è, in particolare se considerate processi in cui il management e i lavoratori occupano mondi diversi. Il management vi darà un'idea di come funzionano le cose, ma quando andate dove il lavoro viene concretamente svolto, troverete una realtà ben diversa, e avrete bisogno di tempo per assimilarla.) Oltre a darvi un'idea di come il sistema viene *realmente* usato, resterete stupiti da come la richiesta "Posso sedermi qui vicino per una settimana mentre svolge il suo lavoro?" aiuti a creare fiducia e stabilisca una base per comunicare con gli utenti. State solo attenti a non essere d'intralcio!

SUGGERIMENTO 52
Lavorate con un utente per pensare come un utente.

Il processo di "scavo" dei requisiti rappresenta anche il momento per iniziare a costruire un rapporto con la base degli utenti, scoprire quali sono le loro aspettative e che cosa sperano di ottenere dal sistema che dovete costruire. Vedete *Grandi speranze*, nel Capitolo 8, per un approfondimento.

Documentare i requisiti

Dunque, siete seduti vicino agli utenti e cercate di estrarre da loro i requisiti reali. Vi ritroverete con alcuni scenari probabili che descrivono quello che l'applicazione deve fare. Sempre professionali, volete scrivere tutto e rendere pubblico un documento che tutti possano usare come base

di discussione - gli sviluppatori, gli utenti finali e chi ha voluto il progetto.

Un pubblico piuttosto ampio.

Ivar Jacobson [Jac94] ha proposto il concetto di *casi d'uso* per catturare i requisitivi. Permettono di descrivere un particolare *uso* del sistema - non in termini di interfaccia utente, ma in modo più astratto. Purtroppo il libro di Jacobson era un po' vago sui dettagli, perciò ora esistono molte opinioni diverse su che cosa debba essere un caso d'uso. È formale o informale, prosa semplice o un documento strutturato (come un modulo)? Qual è il livello di dettaglio appropriato (ricordate quanto ampio sia il pubblico dei destinatari)?

Qualche volta l'interfaccia è il sistema

In un articolo pubblicato su *Wired* (gennaio 1999, p. 176), Brian Eno, produttore e musicista, descriveva un incredibile oggetto tecnologico - il non plus ultra della console di missaggio. Fa ai suoni tutto quel che è possibile fare eppure, anziché consentire ai musicisti di fare musica meglio, o di produrre un disco più rapidamente o a costi più bassi, è d'intralcio: confonde il processo creativo.

Per capire perché, bisogna considerare come lavorano i tecnici nello studio di registrazione. Bilanciano i suoni in modo intuitivo. Con gli anni, sviluppano un circuito di retroazione fra le orecchie e la punta delle dita, che fanno scorrere cursori, ruotano manopole e così via. L'interfaccia del nuovo mixer però non sfruttava quelle loro capacità, ma li costringeva a scrivere su una tastiera o a fare clic con un mouse. Le funzioni che metteva a disposizione erano estesissime, ma erano confezionate in modi non familiari, esotici. Le funzioni di cui i tecnici avevano bisogno a volte erano nascoste dietro nomi oscuri, o erano ottenute con combinazioni non intuitive di strumenti di base.

Quell'ambiente ha un requisito: sfruttare le capacità esistenti. Duplicare pedissequamente quel che già esiste non consente di progredire, ma bisogna rendere possibile una *transizione* verso il futuro.

Per esempio, si sarebbe fatto un servizio migliore per i tecnici con qualche tipo di interfaccia basata su touchscreen: ancora tattile, montata ancora magari come una console di missaggio tradizionale, ma con un software che andava al di là di manopole e interruttori. Fornire una transizione comoda grazie a metafore familiari è un modo per contribuire all'adozione.

Questo esempio illustra anche la nostra convinzione che gli strumenti di successo si adattano alle mani che li usano. In questo caso, sono gli strumenti che costruite per altri che devono essere adattabili.

Un modo per considerare i casi d'uso è sottolinearne la natura orientata a uno scopo. Alistair Cockburn ha scritto un articolo che descrive questo approccio, e ha realizzato dei modelli che si possono usare (più o meno fedelmente) come punto di partenza ([Coc97a] e online a [URL46]). La

Figura 7.1 mostra un esempio abbreviato del suo modello, mentre la Figura 7.2 mostra un suo esempio di caso d'uso.

1. INFORMAZIONI CARATTERISTICHE
   - Obiettivo nel contesto
   - Raggio d'azione
   - Livello
   - Precondizioni
   - Condizione finale di successo
   - Condizione finale di fallimento
   - Attore primario
   - Innesco
2. SCENARIO PRINCIPALE DI SUCCESSO
3. ESTENSIONI
4. VARIAZIONI
5. INFORMAZIONI CORRELATE
   - Priorità
   - Obiettivo di prestazioni
   - Frequenza
   - Caso d'uso sovraordinato
   - Casi d'uso subordinati
   - Canale all'attore primario
   - Attori secondari
   - Canale agli attori secondari
6. PIANIFICAZIONE TEMPORALE
7. PROBLEMI APERTI

Figura 7.1 Modello di caso d'uso di Cockburn.

Utilizzando un modello formale come promemoria, potete essere sicuri di includere tutte le informazioni che vi servono in un caso d'uso: caratteristiche delle prestazioni, altre parti coinvolte, priorità, frequenza, i vari errori e le eccezioni che emergono ("requisiti non funzionali"). È anche un ottimo posto per registrare commenti degli utenti del tipo "oh, tranne quando ci troviamo nella condizione *xxx*, perché allora dobbiamo fare invece *yyy*". Il modello funge anche da ordine del giorno già confezionato per le riunioni con gli utenti.

Questo tipo di organizzazione consente la strutturazione gerarchica dei casi d'uso, con l'annidamento di casi d'uso più dettagliati all'interno di quelli di livello superiore. Per esempio, *invia debito* e *invia credito* sono elaborazioni di *invia transazione*.

Diagrammi di casi d'uso

Il flusso di lavoro può essere catturato con diagrammi di attività UML e i diagrammi di classe a livello concettuale a volte possono essere utili per modellizzare l'attività in questione. I veri casi d'uso però sono descrizioni testuali, con una gerarchia e rimandi incrociati: possono contenere collegamenti ipertestuali ad altri casi d'uso e possono essere annidati uno dentro l'altro.

CASO D'USO 5: ACQUISTO DI MERCI

A. INFORMAZIONI CARATTERISTICHE

- **Obiettivo nel contesto:** L'acquirente emette un ordine diretto alla nostra azienda, e si aspetta che le merci vengano spedite e fatturate.
- **Raggio d'azione:** Azienda
- **Livello:** Riepilogo
- **Precondizioni:** Conosciamo l'acquirente, il suo indirizzo ecc.
- **Condizione finale di successo:** L'acquirente ha le merci, noi abbiamo il denaro corrispondente.
- **Condizione finale di fallimento:** Noi non abbiamo spedito le merci, l'acquirente non ci ha spedito il denaro.
- **Attore primario:** L'acquirente, qualsiasi agente (o computer) che opera per conto del cliente.
- **Innesco:** Arriva un ordine d'acquisto.

B. SCENARIO PRINCIPALE DI SUCCESSO

1. L'acquirente chiama ed effettua un ordine d'acquisto.
2. L'azienda registra nome dell'acquirente, indirizzo, merci ordinate ecc.
3. L'azienda fornisce all'acquirente informazioni sulle merci, i prezzi, le date di consegna ecc.
4. L'acquirente firma l'ordine.
5. L'azienda prepara le merci per evadere l'ordine e le spedisce all'acquirente.
6. L'azienda invia una fattura all'acquirente.
7. L'acquirente paga quanto fatturato.

C. ESTENSIONI

3a. L'azienda ha esaurito uno dei prodotti ordinati: l'ordine viene rinegoziato.
4a. L'acquirente paga direttamente con carta di credito: si accetta il pagamento con carta di credito (caso d'uso 44).
7a. L'acquirente restituisce la merce: gestione delle merci rese (caso d'uso 105).

D. VARIAZIONI

1. L'acquirente può ordinare per telefono, fax, modulo d'ordine via web, scambio elettronico.
7. L'acquirente può pagare in contanti, con bonifico, assegno o carta di credito.

E. INFORMAZIONI CORRELATE

- **Priorità:** Massima.
- **Obiettivo di prestazioni:** 5 minuti per l'ordine, pagamento a 45 giorni.
- **Frequenza:** 200 al giorno
- **Caso d'uso sovraordinato:** Gestione della relazione con il cliente (caso d'uso 2).
- **Casi d'uso subordinati:** Creazione dell'ordine (15). Gestione del pagamento via carta di credito (44). Gestione delle merci rese (105).
- **Canale all'attore primario:** Può essere il telefono, un file, o interattivo.
- **Attori secondari:** Società delle carte di credito, banca, spedizioniere.

F. PIANIFICAZIONE TEMPORALE

- Data di consegna: Release 1.0

G. PROBLEMI APERTI

- Che cosa succede se possiamo evadere l'ordine solo parzialmente?
- Che cosa succede se la carta di credito risulta rubata?

Figura 7.2 Un esempio di caso d'uso.

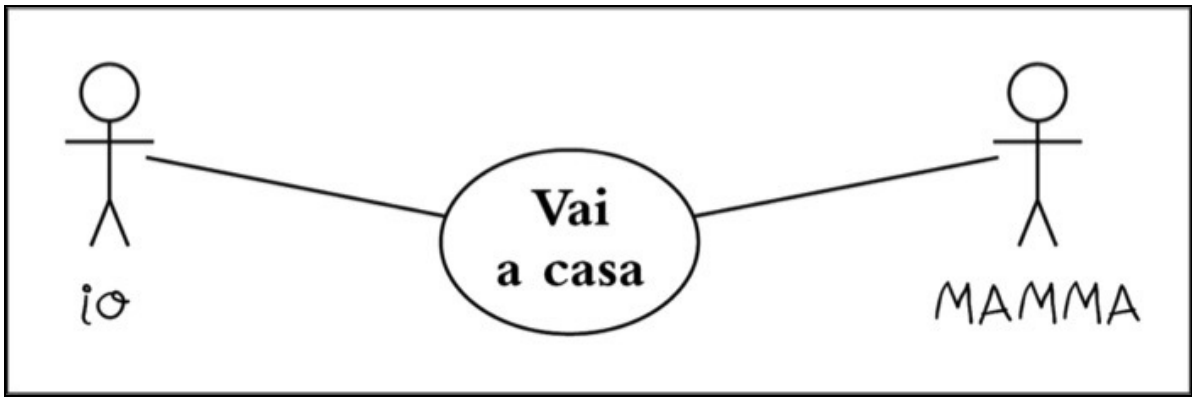


Figura 7.3 Casi d'uso UML: tanto semplici che potrebbe usarli un bambino.

A noi sembra incredibile che qualcuno prenda seriamente in considerazione la possibilità di documentare informazioni così dense utilizzando pupazzetti stilizzati come quelli della Figura 7.3. Non rendetevi schiavi di una notazione, quale che sia: usate il metodo che comunica nel modo migliore i requisiti al vostro pubblico.

Specificazioni eccessive

Un pericolo serio nella produzione di un documento di requisiti è scendere troppo nello specifico. I buoni documenti di requisiti restano astratti. L'enunciazione più semplice che rispecchia con precisione l'esigenza di business è la cosa migliore. Questo non significa che potete essere vaghi: dovete catturare le invarianti semantiche come requisiti e documentare le pratiche di lavoro specifiche o correnti come norme.

I requisiti non sono architettura; non sono progetto, e non sono interfaccia utente. I requisiti sono *esigenze*.

Guardare oltre

Spesso si è data la colpa del "baco del Duemila" alla scarsa preveggenza dei programmatori, disperati di risparmiare qualche byte nell'indicazione delle date, quando i mainframe avevano meno memoria di quella che ha oggi il telecomando di un televisore.

Ma non era colpa dei programmatori, e non era davvero un problema di uso della memoria: casomai, era colpa degli analisti di sistema e dei progettisti. Il problema aveva due cause principali: non aver saputo guardare al di là della pratica di business corrente, e una violazione del principio DRY.

Le aziende usavano la forma stenografica a due cifre molto prima che entrassero in scena i computer, era una pratica comune. Le prime applicazioni di elaborazione dei dati hanno semplicemente automatizzato processi di business preesistenti e non hanno fatto altro che ripetere l'errore. Anche se l'architettura richiedeva anni indicati con due cifre per l'input, il reporting e la memorizzazione, avrebbe dovuto essere un'astrazione della data che "sapeva" che due cifre erano una forma abbreviata della data reale.

SUGGERIMENTO 53
Le astrazioni vivono più a lungo dei dettagli.

"Guardare più in là" significa prevedere il futuro? No, significa generare enunciati come

> Il sistema fa uso attivo di un'astrazione DATA. Il sistema implementa i servizi di DATA, come formattazione, memorizzazione e operazioni matematiche, in modo coerente e universale.

I requisiti specificheranno solo che vengono usate le date. Può far intuire che sulle date si svolgono dei calcoli matematici. Può dire che le date verranno memorizzate in varie forme di dispositivi di memoria secondaria. Questi sono requisiti genuini di un modulo o di una classe per le date.

Solo una cosina ancora…

Molti fallimenti di progetti sono attribuiti a un allargamento del raggio d'azione, al progressivo aumento delle caratteristiche da inserire (in inglese si usano espressioni curiose come *feature bloat*, *creeping featurism*, *requirements creep*). È un aspetto della sindrome della rana bollita (vedi *Zuppa di pietre e rane bollite*, nel Capitolo 1).

Nella letteratura, troverete descrizioni di molte metriche, come numero degli errori segnalati e sistemati, densità dei difetti, coesione, accoppiamento, punti funzione, righe di codice e così via. Queste metriche possono essere calcolate manualmente o con software. Purtroppo invece non molti progetti sembrano tener traccia attivamente dei requisiti, e questo significa che non hanno modo di segnalare i cambiamenti del raggio d'azione: chi ha richiesto una caratteristica, chi l'ha approvata, il numero totale delle richieste approvate e così via.

La chiave per gestire la crescita dei requisiti è evidenziare agli sponsor del progetto l'impatto che ogni nuova caratteristica ha sull'andamento del progetto nel tempo. Quando il progetto è in ritardo di un anno sulla stima iniziale e cominciano a volare le accuse, può essere utile avere un quadro preciso e completo di come e quando ha avuto luogo la crescita dei requisiti.

È facile essere risucchiati nel vortice del "solo ancora una caratteristica", ma tenendo traccia dei requisiti si può avere un'idea più chiara che "solo ancora una caratteristica" è in realtà la quindicesima aggiunta questo mese.

Mantenere un glossario

Non appena si comincia a discutere di requisiti, utenti ed esperti del settore useranno certi termini che hanno per loro un significato specifico.

Possono magari distinguere fra un “client” e un “customer”, per esempio, e sarebbe inappropriato usare una parola o l’altra a caso nel sistema.

Create e tenete sempre aggiornato un *glossario di progetto*, un luogo unico in cui sono definiti tutti i termini specifici utilizzati nel progetto. Tutti i partecipanti al progetto, dagli utenti finali al personale dell’assistenza, devono usare il glossario per garantire la coerenza. Questo comporta che il glossario deve essere ampiamente accessibile, un buon argomento a sostegno della documentazione basata sul Web (approfondiremo tra poco).

SUGGERIMENTO 54
Usate un glossario di progetto.

È molto difficile avere successo in un progetto in cui utenti e sviluppatori si riferiscono alla stessa cosa con nomi diversi o, ancora peggio, usano lo stesso nome per parlare di cose diverse.

Fate girare parola

In *È tutta scrittura*, nel Capitolo 8, parliamo della pubblicazione dei documenti di progetto su siti web interni per garantire un facile accesso a tutti i partecipanti. Questo metodo di distribuzione è particolarmente utile per i documenti dei requisiti.

Presentando i requisiti come un documento ipertestuale, si possono meglio affrontare le esigenze di un pubblico differenziato: possiamo dare a ciascun lettore quello che vuole. Gli sponsor del progetto possono rimanere a un livello elevato di astrazione, per essere sicuri che gli obiettivi di business siano soddisfatti; i programmatori possono usare i collegamenti ipertestuali per “scendere in profondità” a livelli crescenti di dettaglio (anche facendo riferimento a definizioni appropriate o a specifiche tecniche).

La distribuzione via Web inoltre evita i tipici faldoni di dieci centimetri con l’etichetta *Analisi dei requisiti* che nessuno mai legge e che sono obsoleti nell’istante in cui l’inchiostro si deposita sulla carta. Se è sul Web, magari persino i programmatori potrebbero leggerlo.

Vedi anche

- *Zuppa di pietre e rane bollite*, Capitolo 1
- *Software abbastanza buono*, Capitolo 1

- *Cerchi e frecce*, in questo Capitolo
- *È tutta scrittura*, Capitolo 8
- *Grandi speranze*, Capitolo 9

Sfide

- Potete usare il software che scrivete? È possibile avere una buona percezione dei requisiti *senza* essere in grado di usare in prima persona il software?
- Scegliete un problema, non legato al computer, che dovete risolvere. Generate dei requisiti per una soluzione che non faccia uso del computer.

Esercizi

42. Quali fra i seguenti sono probabilmente requisiti genuini? Riformulate quelli che non lo sono in modo da renderli più utili (se possibile).

1. Il tempo di risposta deve essere inferiore a 500 millisecondi.
2. Le finestre di dialogo avranno uno sfondo grigio.
3. L'applicazione sarà organizzata sotto forma di una serie di processi di front-end e un server di back-end.
4. Se un utente inserisce caratteri non numerici in un campo numerico, il sistema emetterà un avviso acustico e non li accetterà.
5. Il codice dell'applicazione e i dati devono stare in 256 kB.

Risolvere rompicapo impossibili

Gordio, re di Frigia, una volta fece un nodo che nessuno riusciva a sciogliere. Si diceva che chi avesse risolto il rompicapo del nodo gordiano sarebbe stato il dominatore di tutta l'Asia. Arrivò Alessandro Magno, che fece a pezzi il nodo con la spada. Solo una piccola differenza di interpretazione dei requisiti, ecco tutto… e finì per dominare tutta l'Asia.

Ogni tanto, nel mezzo di un progetto, vi troverete nei pasticci quando si presenta un rompicapo davvero difficile: qualche aspetto tecnico che non riuscite a dominare, o magari qualche parte di codice che si rivela molto più difficile da scrivere di quel che pensavate. Magari sembra addirittura impossibile. Ma è davvero così difficile come sembra?

Pensate ai puzzle, quel tremendi tasselli di legno, di plastica o di cartoncino che sembrano fra i regali natalizi prediletti o che si trovano nei

mercatini dell'usato. Tutto quel che dovete fare è togliere l'anello, o trovare la collocazione dei pezzi a forma di T, o qualche altra cosa simile. Se lo fate, scoprite subito che le soluzioni ovvie non funzionano. Il puzzle non si può risolvere in quel modo. Anche se è ovvio, questo non impedisce di tentare la stessa cosa, più e più volte, pensando che ci debba essere un modo.

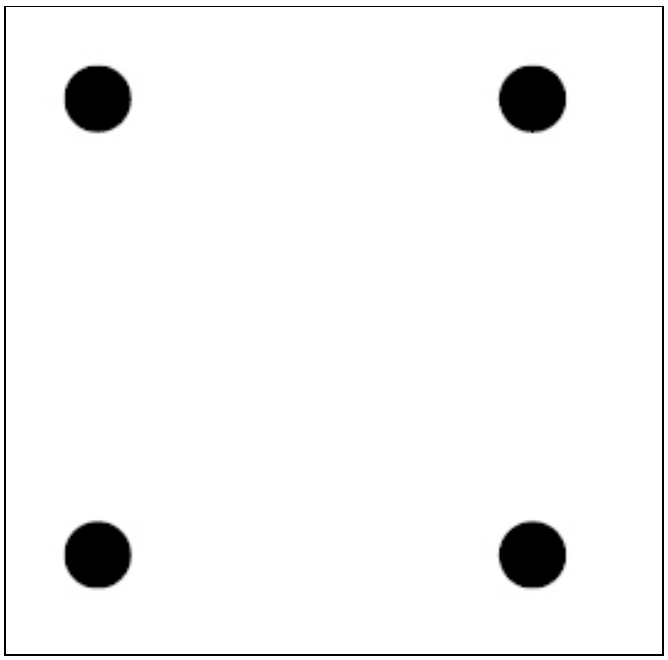
Ovviamente, il modo non c'è. La soluzione sta altrove. Il segreto per la risoluzione di un rompicapo è identificare i vincoli reali (non immaginati) e trovare una soluzione in base a quelli. Alcuni vincoli sono *assoluti*; altri sono solo *idee preconcette*. I vincoli assoluti *devono* essere rispettati, per quanto sgradevoli o stupidi possano apparire. Alcuni vincoli apparenti, invece, non sono affatto vincoli. C'è il vecchio trucco in cui si prende una bottiglia nuova, ancora sigillata, di champagne e si scommette che si riuscirà a berne della birra. Il trucco consiste nel capovolgere la bottiglia e mettere un po' di birra nell'incavo alla base della bottiglia stessa. Molti problemi di software possono essere altrettanto subdoli.

Gradi di libertà

L'idea di "pensare fuori dagli schemi" ci spinge a riconoscere i vincoli che possono non essere applicabili e a ignorarli. L'espressione però non è del tutto corretta: se lo "schema" è il confine segnato da vincoli e condizioni, allora il trucco consiste nel *trovare* lo schema, che può essere notevolmente più ampio di quel che si pensa.

La chiave per risolvere rompicapo consiste nel riconoscere i vincoli imposti e anche i gradi di libertà che si *hanno*, perché lì si trova la soluzione. Per questo i rompicapo sono così efficaci: si possono escludere troppo in fretta soluzioni potenziali.

Per esempio, potete collegare tutti i punti della figura seguente e tornare al punto di partenza con tre soli segmenti di retta, senza sollevare la penna dalla carta o ripercorrere uno stesso passo [Hol78]?



Dovete sfidare qualsiasi idea preconcetta e valutare se si tratti di vincoli reali, concreti o no. Non si tratta di pensare dentro o fuori gli schemi: il problema è *trovare* lo schema, identificare i vincoli reali.

SUGGERIMENTO 55
Non pensate fuori dagli schemi, *trovate* lo schema.

Di fronte a un problema intrattabile, enumerate *tutte* le possibili strade che avete davanti. Non escludete nulla, non importa quanto sembri inutilizzabile o stupido. Ora scorrete la lista e spiegate perché non si può prendere una certa strada. Ne siete sicuri? Potete *dimostrarlo*?

Pensate al cavallo di Troia - una soluzione nuova a un problema intrattabile. Come si fa a far entrare un esercito in una città fortificata da mura senza farsi scoprire? Potete scommettere che inizialmente la risposta “passando per la porta principale” era stata accantonata come un invito al suicidio.

Classificate i vostri vincoli e attribuite loro delle priorità. Quando un falegname inizia un progetto, taglia prima i pezzi più lunghi, poi ricava i più piccoli dal legno che resta. Analogamente, vogliamo identificare prima i vincoli più restrittivi e poi sistemare dentro quelli i vincoli restanti.

Per inciso, una soluzione al rompicapo dei quattro punti si trova nell’Appendice B.

Deve esserci un modo più facile!

A volte vi troverete a lavorare su un problema che sembra molto più difficile di quel che pensavate dovesse essere. Magari avete l'impressione di avere imboccato la strada sbagliata, che debba esistere un modo più facile. Magari state mancando la scadenza, o disperate di riuscire a far mai funzionare il sistema perché questo particolare problema è "impossibile".

È il momento di fare un passo indietro e porsi qualche domanda.

- Esiste un modo più facile?
- State tentando di risolvere il problema giusto, o siete stati distratti da un tecnicismo periferico?
- *Perché* questa cosa è un problema?
- Che cosa lo rende così difficile da risolvere?
- Deve essere fatto in questo modo?
- Deve proprio essere fatto?

Molte volte cercare di dare una risposta a queste domande produce una rivelazione sorprendente. Molte volte una reinterpretazione dei requisiti può far sparire un intero insieme di problemi, come nel caso del nodo gordiano.

Tutto quello che serve sono i vincoli veri, i vincoli fuorvianti e l'intelligenza di distinguerli.

Sfide

- Date uno sguardo attento a un problema difficile in cui siete incappati oggi. Potete tagliare il nodo gordiano? Ponetevi le domande che abbiamo elencato prima, in particolare "Deve essere fatto in questo modo?".
- Quando avete assunto il progetto corrente, vi è stato passato un insieme di vincoli? Sono tutti ancora applicabili e la loro interpretazione è ancora valida?

Non finché non siete pronti

Colui che esita a volte si salva.
- James Thurber, *The Glass in the Field*

I grandi hanno in comune una caratteristica: sanno quando iniziare e quando aspettare. Il tuffatore sta sul trampolino, e aspetta il momento perfetto per saltare. Il direttore sta davanti all'orchestra, con le braccia alzate, finché non sente che è venuto il momento giusto per dare l'attacco.

Anche voi siete grandi esecutori. Dovete ascoltare la vocina che vi sussurra: "aspetta": Se vi sedete per cominciare a scrivere e c'è qualche dubbio che vi tormenta, lasciate perdere.

SUGGERIMENTO 56
Ascoltate i dubbi che vi assillano. Iniziate quando siete pronti.

Esisteva uno stile di allenamento al tennis che si chiamava "tennis interiore". Si passavano ore a spedire palline oltre la rete, non preoccupandosi eccessivamente della precisione. Invece si doveva esprimere verbalmente dove finiva la pallina rispetto a un certo bersaglio (spesso una sedia). L'idea era che il feedback avrebbe addestrato il subconscio e i riflessi, in modo da migliorare senza sapere consapevolmente come e perché.

Da sviluppatori, avete fatto la stessa cosa per tutta la vostra carriera. Avete provato cose e visto quali funzionavano e quali noi. Avete accumulato esperienza e saggezza. Quando vi assilla un dubbio, o quando provate una certa riluttanza di fronte a un compito, fermatevi. Forse non sarete in grado di puntare il dito proprio su quello che non va, ma datevi tempo e i dubbi probabilmente si cristallizzeranno in qualcosa di più solido, qualcosa che potete affrontare. Lo sviluppo del software non è ancora una scienza. Lasciate che i vostri istinti contribuiscano alla vostra performance.

Sano giudizio o procrastinazione?

Tutti hanno paura del foglio bianco. Iniziare un nuovo progetto (o anche solo un nuovo modulo in un progetto già avviato) può essere un'esperienza snervante. Molti preferirebbero rinviare quel momento decisivo in cui si inizia a scrivere. E allora, come si fa a dire se si sta semplicemente procrastinando e non aspettando responsabilmente che tutti i pezzi vadano al loro posto?

Una tecnica che ha funzionato per noi in queste circostanze è iniziare a prototipare. Scegliete un settore che vi sembra si rivelerà difficile e cominciate a produrre qualche sorta di dimostrazione del concetto. Succederà, in genere, una di due cose. Subito dopo aver iniziato, avrete la sensazione di sprecare il vostro tempo. La noia probabilmente sarà un buon indizio che la riluttanza iniziale era solo desiderio di rimandare l'impegno di iniziare. Lasciate perdere il prototipo e buttatevi nello sviluppo reale.

Nell'altro caso, con il procedere del prototipo potreste avere uno di quei momenti di illuminazione, in cui ci si rende conto che qualche premessa di base era sbagliata. Non solo, ma si vede chiaramente come rimetterla in sesto. A quel punto vi sentire a vostro agio, potrete lasciare il prototipo e lanciarvi nel progetto. Il vostro istinto aveva ragione, e avete appena risparmiato a voi stessi e al vostro team una quantità notevole di lavoro sprecato.

Quando prendete la decisione di prototipare per capire che cosa vi mette a disagio, non dimenticate perché lo state facendo. L'ultima cosa che vorrete è trovarvi ad aver lavorato settimane con grande impegno prima di ricordarvi che dovevate solo scrivere un prototipo.

Un po' cinicamente, iniziare a lavorare su un prototipo può anche essere politicamente più accettabile che annunciare semplicemente "Non ho voglia di iniziare" e lanciare il solitario.

Sfide

- Discutete con i vostri colleghi la sindrome della paura-di-iniziare. Altri hanno la stessa sensazione? Le danno ascolto? Quali trucchi usano per superarla? Un gruppo può aiutare a superare la riluttanza individuale, o sarebbe solo pressione dei colleghi?

La trappola delle specifiche

Il Pilota d'atterraggio è il Pilota non ai comandi fino alla chiamate della "altitudine di decisione", quando il Pilota ai comandi non d'atterraggio passa i comandi al Pilota di atterraggio non ai comandi, a meno che quest'ultimo chiami "go-around", nel qual caso il Pilot non d'atterraggio ai comandi continua a stare ai comandi e il Pilota di atterraggio non ai comandi continua a non tenere i comandi fono alla successiva chiamata di "land" o

> “go-around”, come opportuno. Considerata la recente confusione su queste regole, si è ritenuto necessario ribadirle chiaramente.
>
> - Memorandum della British Airway, citato in *Pilot Magazine*, dicembre 1996

La specifica del programma è il processo per cui si prende un requisito e lo si riduce fino al punto in cui può subentrare la competenza di un programmatore. È un atto di comunicazione, che spiega e chiarisce il mondo in modo tale da eliminare le ambiguità importanti. Oltre a parlare allo sviluppatore che eseguirà l’implementazione iniziale, la specifica è un documento per le generazioni future di programmatori che dovranno manutenere e migliorare il codice. La specifica è anche un accordo con l’utente - un modo di codificare i suoi bisogni e un contratto implicito che il sistema sarà in linea con quel requisito.

Scrivere una specifica è una notevole responsabilità.

Il problema è che molti progettisti trovano difficile fermarsi. Hanno la sensazione che se non fissano con terribile precisione e grande dovizia di dettagli ogni singolo particolare non si saranno guadagnati la loro paga giornaliera. Ma è un errore, per vari motivi. In primo luogo, è ingenuo pensare che una specifica possa mai catturare ogni particolare e ogni sfumatura di un sistema o dei suoi requisiti. In campi di problemi molto limitati, esistono metodi formali per descrivere un sistema, ma richiedono comunque che il progettista spieghi il significato della notazione agli utenti finali: esiste ancora una interpretazione che può confondere le cose. Anche senza i problemi intrinseci a questa interpretazione, è molto improbabile che l’utente medio che entra in un progetto sappia esattamente di che cosa ha bisogno. Potrà dire di capire i requisiti, e potrà anche firmare il documento di 200 pagine che avete prodotto, ma potete stare sicuri che, non appena vedrà girare il sistema, vi inonderà di richieste di cambiamenti.

In secondo luogo, esiste un problema legato al potere espressivo del linguaggio stesso. Tutte le tecniche della diagrammazione e tutti i metodi formali si basano ancora su espressioni in linguaggio naturale delle operazioni da svolgere. (Esistono tecniche formali che cercano di esprimere le operazioni algebricamente, ma sono usate di rado nella pratica; richiedono comunque che gli analisti spieghino il significato agli utenti finali.) Il linguaggio naturale proprio non è all’altezza del compito. Guardate il testo di qualsiasi contratto: nel tentativo di essere precisi, gli avvocati devono piegare la lingua nei modi più innaturali.

Ecco una sfida per voi: scrivete una breve descrizione di come allacciarsi le scarpe. Provateci!

Se siete come noi, probabilmente avete lasciato perdere quando siete arrivati a “adesso piega pollice e indice in modo che il capo libero passi sotto e dentro il laccio sinistro…”. È una cosa incredibilmente difficile da esprimere, eppure la maggior parte di noi riesce ad allacciarsi le scarpe senza stare tanto a pensarci consapevolmente.

SUGGERIMENTO 57

Alcune cose si fa prima a farle che a descriverle.

Infine, c’è l’effetto “camicia di forza”. Un progetto che non lascia alcuno spazio di interpretazione al codificatore priva il lavoro di programmazione di ogni elemento di professionalità e di artisticità. Qualcuno direbbe che è per il meglio, ma si sbaglia. Spesso solo durante la codifica diventano evidenti certe opzioni. Mentre codificate, potete pensare, “Guarda un po’. Visto il modo particolare in cui ho codificato questa routine, potrei aggiungere questa ulteriore funzionalità quasi senza far fatica” oppure “Le specifiche dicevano di fare questo, ma potrei ottenere un risultato quasi identico facendo in un modo diverso, e ci potrei impiegare metà tempo”. Chiaramente, non dovete mettervi lì e fare questi cambiamenti, ma non avreste nemmeno notato questa opportunità se foste stati vincolati da una progettazione eccessivamente prescrittiva.

Da programmatori pragmatici, dovete tendere a vedere la raccolta dei requisiti, la progettazione e l’implementazione come facce diverse dello stesso processo - la realizzazione di un sistema di qualità. Diffidate degli ambienti in cui i requisiti vengono raccolti, le specifiche vengono scritte e poi inizia la codifica, tutto isolatamente. Cercate di adottare invece una impostazione senza soluzione di continuità: specifiche e implementazione sono semplicemente aspetti diversi dello stesso processo - un tentativo di catturare e codificare un requisito. Ciascuno deve fluire direttamente nel successivo, senza confini artificiali. Troverete che un processo di sviluppo sano favorisce il passaggio di feedback dall’implementazione e dai test nel processo di specifica.

Per essere chiari, non siamo contro la generazione di specifiche: sappiamo che ci sono occasioni in cui specifiche incredibilmente dettagliate sono obbligatorie - per ragioni contrattuali, per l’ambiente in cui si lavora o per la natura del prodotto che si sta sviluppando. (Specifiche dettagliate sono chiaramente appropriate per sistemi *life-*

critical. Devono essere prodotte per interfacce e librerie che saranno usate da altri. Quando tutto l'output viene visto come un insieme di chiamate di routine, sarà meglio essere sicuri che quelle chiamate siano ben specificate.) Ricordate semplicemente che si raggiunge un punto in cui i ritorni diminuiscono o diventano addirittura negativi, quando le specifiche diventano sempre più dettagliate. State attenti anche a costruire specifiche stratificate sopra specifiche senza implementazione o prototipazione di supporto: è facilissimo specificare qualcosa che non si può costruire.

Quanto più permettete che le specifiche siano coperte di sicurezza, che proteggono gli sviluppatori dal mondo terrificante della scrittura di codice, tanto più difficile sarà spostarsi sulla produzione del codice. Non cadete in questa spirale delle specifiche: a un certo punto dovete iniziare a codificare! Se trovate il vostro team tutto avvolto in calde e comode specifiche, liberatelo. Pensate alla prototipazione, oppure prendete in considerazione uno sviluppo a proiettili traccianti.

Vedi anche

- *Proiettili traccianti*, Capitolo 2

Sfide

- Quello dei lacci delle scarpe citato nel testo è un esempio interessante dei problemi che incontrano le descrizioni scritte. Avete pensato alla possibilità di descrivere il processo con diagrammi invece che con parole? Fotografie? Qualche notazione formale ricavata dalla topologia? Modelli con lacci di scarpe? Come insegnereste a un bambino piccolo?
- A volte un'immagine vale più di qualsiasi numero di parole. A volte invece non vale niente. Se vi trovate a specificare in maniera eccessiva, immagini o notazioni speciali sarebbero d'aiuto? Quanto dovrebbero essere dettagliate? Quando uno strumento di disegno è meglio di una lavagna?

Cerchi e frecce

> [fotografia] con cerchi e frecce e un paragrafo sul retro di ciascuna che spiega che cosa rappresenta, da usarsi come prova contro di noi…
> - Arlo Guthrie, *Alice's Restaurant*

Dalla programmazione strutturata, attraverso i team di programmatori principali, strumenti CASE, sviluppo a cascata, modello a spirale, Jackson, diagrammi ER, nuvole di Booch, OMT, Objectory e Coad/Yourdon, fino all'UML, l'informatica non è mai stata priva di metodi che volevano rendere la programmazione più simile all'ingegneria. Ogni metodo ha i suoi discepoli e gode di un periodo di popolarità, poi viene sostituito dal successivo. Fra tutti, forse solo il primo, la programmazione strutturata, ha avuto una vita lunga.

Qualche sviluppatore, però, alla deriva in un mare di progetti che affondano, continua ad afferrarsi all'ultima moda come le vittime di un naufragio si aggrappano a un pezzo di legno che passa loro accanto. Quando accanto a loro galleggia qualche altro pezzo, lo raggiungono faticosamente a nuoto, sperando che sia meglio del precedente. Alla fine della giornata, però, non importa quanto siano buoni i relitti galleggianti, gli sviluppatori sono ancora alla deriva senza spiagge in vista.

Non fraintendeteci. Ci piacciono (certe) tecniche formali e (certi) metodi. Ma siamo convinti che adottare ciecamente una tecnica senza collocarla nel contesto delle proprie pratiche di sviluppo e delle proprie capacità sia una ricetta per la delusione.

SUGGERIMENTO 58
Non siate schiavi di metodi formali.

I metodi formali hanno alcuni svantaggi gravi.

- La maggior parte dei metodi formali cattura i requisiti con una combinazione di diagrammi e un po' di parole di supporto. Queste immagini rappresentano la comprensione che hanno i progettisti dei requisiti. In molti casi però quei diagrammi non hanno alcun significato per gli utenti finali, perciò i progettisti devono interpretarli. Perciò non esiste un controllo formale reale dei requisiti da parte dell'effettivo utente del sistema - tutto si basa sulle spiegazioni dei progettisti, come nei tradizionali requisiti scritti. Vediamo qualche vantaggio nel catturare i requisiti in questo modo, ma, dove possibile, preferiamo mostrare all'utente un prototipo e lasciare che ci giochi.
- I metodi formali sembrano incoraggiare la specializzazione. Un gruppo di persone lavora a un modello dei dati, un altro esamina l'architettura, mentre i raccoglitori di requisiti collezionano casi

d’uso (o i loro equivalenti). Abbiamo visto tutto questo portare a cattive comunicazioni e a lavoro sprecato. Vi è anche una tendenza a ricadere in una mentalità del *noi verso loro*, progettisti contro codificatori. Preferiamo capire tutto il sistema a cui stiamo lavoriamo. Forse non sarà possibile avere un’idea in profondità di ogni aspetto di un sistema, ma dovete sapere come interagiscono i componenti, dove stanno i dati e quali sono i requisiti.

- Ci piace scrivere sistemi adattabili, dinamici, utilizzando i metadati per consentirci di modificare il carattere delle applicazioni in fase di esecuzione. La maggior parte degli attuali metodi formali combina un oggetto statico o un modello dei dati con qualche tipo di meccanismo per rilevare gli eventi o le attività. Non ne abbiamo mai incontrato uno che ci permettesse di illustrare il tipo di dinamismo che secondo noi i sistemi dovrebbero esibire. In effetti, la maggior parte dei metodi formali vi porterà fuori strada, incoraggiandovi a impostare relazioni statiche fra oggetti che in realtà dovrebbero essere intrecciati insieme dinamicamente.

I metodi pagano?

In un articolo del 1999 [Gla99b], Robert Glass passa in rassegna le ricerche sui miglioramenti di produttività e di qualità ottenuti utilizzando sette diverse tecnologie di sviluppo (linguaggi di quarta generazione, tecniche strutturate, strumenti CASE, metodi formali, metodologia della “clean room”, modelli di processo e orientamento agli oggetti): il gran rumore iniziale su ciascuno di questi metodi risulta essere stato sempre eccessivo. Vi è un’indicazione che alcuni metodi hanno dei vantaggi, ma questi vantaggi cominciano a manifestarsi solo dopo una significativa caduta di produttività e qualità mentre la tecnica viene adottata e chi la usa si forma. Mai sottostimare il costo dell’adozione di nuovi strumenti e metodi. Siate preparati a trattare i primi progetti con queste tecniche come un’esperienza di apprendimento.

Dobbiamo usare metodi formali?

Assolutamente sì. Ma ricordate sempre che i metodi formali di sviluppo sono solo un altro strumento nella vostra cassetta degli attrezzi. Se, dopo attenta analisi, avete la sensazione di dover usare un metodo formale, abbracciatelo - ma ricordate sempre chi comanda. Non dovete mai diventare schiavi di una metodologia: cerchi e frecce sono cattivi padroni. I programmatori pragmatici considerano le metodologie criticamente, poi estraggono il meglio da ciascuna e le fondono in un insieme di pratiche operative che migliorano ogni mese. È fondamentale. Dovete lavorare costantemente per perfezionare e migliorare i vostri processi. Non accettate mai i confini rigidi di una metodologia come se fossero i limiti del vostro mondo.

Non cedete alla falsa autorità di un metodo. Qualcuno può andare a una riunione con un chilometro quadrato di diagrammi di classe e 150 casi d'uso, ma tutta quella carta è ancora solo la sua fallibile interpretazione di requisiti e progetto. Cercate di non pensare a quanto costa uno strumento, quando esaminate il suo output.

SUGGERIMENTO 59

Strumenti costosi non producono progetti migliori.

I metodi formali sicuramente hanno il loro posto nello sviluppo. Tuttavia, se vi imbattete in un progetto la cui filosofia è "il diagramma di classe è l'applicazione, il resto è codifica meccanica", sapete che avete di fronte un team di progetto che fa acqua e che vi aspetta una lunga nuotata per tornare a casa.

Vedi anche

- *La fossa dei requisiti*, in questo Capitolo

Sfide

- I diagrammi dei casi d'uso fanno parte dei processi UML per la raccolta dei requisiti (vedi *La fossa dei requisiti*, in questo capitolo). Sono un modo efficace per comunicare con i vostri utenti? Se no, perché li usate?
- Come potete stabilire se un metodo formale porta dei benefici al vostro team? Che cosa potete misurare? Che cosa costituisce un miglioramento? Potete distinguere fra benefici dello strumento e maggiore esperienza da parte dei membri del team?
- Dove sta il punto di *break-even* per l'introduzione di nuovi metodi nel team? Come valutate il compromesso fra benefici futuri e perdita di produttività nell'immediato, quando viene introdotto lo strumento?

- Strumenti che funzionano per grandi progetti vanno bene per quelli piccoli? E il contrario?

Progetti pragmatici

Quando il progetto è avviato, bisogna passare dai problemi di filosofia individuale e di codifica a questioni più generali, a livello di intero progetto. Non entreremo nei particolari della gestione di progetto, ma parleremo di una serie di aree critiche che possono far arrivare a termine oppure far naufragare qualsiasi progetto.

Non appena a un progetto lavorano più di una persona, dovete stabilire delle regole di base e delegare parti del progetto di conseguenza. In *Team pragmatici*, vedremo come si possa farlo rispettando la filosofia pragmatica.

Il singolo fattore di maggiore importanza per far funzionare coerentemente e affidabilmente le attività a livello di progetto è l'automazione delle procedure. Spiegheremo perché, e vedremo qualche esempio reale, in *Automazione onnipresente*.

In precedenza, abbiamo parlato di eseguire test mentre si codifica. In *Test senza pietà*, facciamo il passo successivo, andando alla filosofia e agli strumenti dei test a livello di progetto, in particolare quando non si ha a disposizione un ampio personale.

L'unica cosa che gli sviluppatori odiano più dei test è la documentazione. Sia che abbiate un *technical writer* a disposizione, sia che il compito tocchi a voi, vedremo come rendere questo compito meno doloroso e più produttivo in *È tutta scrittura*.

Il successo sta nell'occhio dell'osservatore - lo sponsor del progetto. La percezione di successo è quella che conta, e in *Grandi speranze* mostreremo qualche trucco per deliziare qualsiasi sponsor di progetto.

L'ultimo suggerimento del libro è una conseguenza diretta di tutto il resto. In *Orgoglio e pregiudizio*, vi incoraggiamo a firmare il vostro lavoro, e ad essere orgogliosi di quel che fate.

Team pragmatici

> Al Group L, Stoffel controlla sei programmatori di altissimo livello, un'impresa di gestione all'incirca paragonabile a fare il guardiano di bestiame.
> - *The Washington Post Magazine*, 9 giugno 1985

Fin qui in questo libro abbiamo esaminato tecniche pragmatiche che possono aiutare un singolo a essere un programmatore migliore. Questi metodi possono funzionare anche per i team?

La risposta è un sonoro "sì!". Esistono vantaggi nell'essere un individuo pragmatico, ma questi vantaggi si moltiplicano molte volte se l'individuo lavora in un team pragmatico.

In questa sezione esamineremo brevemente come le tecniche pragmatiche si possano applicare ai team nel loro complesso. Queste note sono solo un punto d'inizio. Una volta che avete un gruppo di sviluppatori pragmatici che lavorano in un ambiente abilitante, svilupperanno e perfezioneranno rapidamente le dinamiche di gruppo che per loro funzionano meglio.

Riprendiamo osservazioni delle sezioni precedenti in termini di team.

Niente finestre rotte

La qualità è un problema di squadra. Lo sviluppatore più diligente in un team a cui non importa di nulla troverà difficile mantenere l'entusiasmo necessario per sistemare le difficoltà. Il problema è ulteriormente aggravato se il team scoraggia attivamente lo sviluppatore dal dedicare tempo a risolvere quegli elementi.

I team nel loro complesso non devono tollerare le finestre rotte - quelle piccole imperfezioni che nessuno sistema. Il team *deve* assumersi la responsabilità della qualità del prodotto, sostenendo gli sviluppatori che comprendono la filosofia del *niente finestre rotte* che abbiamo descritto in *Entropia del software*, nel Capitolo 1, e incoraggiando quelli che ancora non l'hanno scoperta.

Alcune metodologie di gruppo hanno un *rappresentante della qualità*, qualcuno a cui il gruppo delega la responsabilità per la qualità di ciò che deve essere prodotto. Questo è chiaramente ridicolo: la qualità può venire solo dai contribuiti individuali di *tutti* i membri del team.

Rane bollite

Ricordate la povera rana nella pentola piena d'acqua, in *Zuppa di pietre e rane bollite*, nel Capitolo 1? Non nota il cambiamento graduale nel suo ambiente, e finisce cotta. Lo stesso può succedere alle persone che non sono vigili. Può essere difficile tenere sott'occhio l'ambiente complessivo, nel calore dello sviluppo del progetto.

È ancora più facile per i gruppi nel loro insieme finire bolliti. Ciascuno dà per scontato che un problema sia gestito da qualcun altro, o che il leader del team abbia dato il benestare a un cambiamento che l'utente richiede. Anche le squadre meglio intenzionate possono non prestare attenzione a cambiamenti significativi nei loro progetti.

Combattete tutto questo. Assicuratevi che tutti controllino attivamente l'ambiente per identificare i cambiamenti. Magari nominate un *chief water tester*, un capo scandagliatore delle acque. Questa persona deve controllare costantemente se ci sono ampliamenti del raggio d'azione, riduzioni delle scale temporali, caratteristiche aggiuntive, nuovi ambienti, qualunque cosa non sia nell'accordo originale. Adottate delle metriche per i nuovi requisiti. Il team non deve rifiutare i cambiamenti, dovete semplicemente essere consapevoli del loro verificarsi. Altrimenti, sarete *voi* nell'acqua bollente.

Comunicate

È ovvio che gli sviluppatori in un gruppo debbano parlarsi. Abbiamo dato qualche suggerimento su come facilitarlo, in *Comunicate!*, nel Capitolo 1. È facile però dimenticare che il team stesso ha una presenza dentro l'organizzazione: come entità, deve comunicare chiaramente con il resto del mondo.

Per chi sta al di fuori, i peggiori team di progetto sono quelli che appaiono cupi e reticenti. Tengono riunioni senza struttura, in cui nessuno vuol parlare. I loro documenti sono un pasticcio: non ce ne sono due con lo stesso aspetto, e ciascuno usa la propria terminologia.

I grandi team di progetto hanno una personalità ben definita. La gente non vede l'ora di riunirsi con loro, perché sa che assisterà a una performance ben preparata che fa sentir bene tutti. La documentazione che producono è elegante, accurata e coerente. Il gruppo parla con una

sola voce, può addirittura avere un senso dello humor. (Il gruppo parla con una sola voce verso l'esterno: internamente, consigliamo fortemente un dibattito vivace e robusto. I buoni sviluppatori in genere sono appassionati per quanto riguarda il loro lavoro.)

Esiste un semplice trucco di marketing che aiuta i team a comunicare come una cosa sola: generare un brand. Quando iniziate un progetto, trovategli un nome, idealmente qualcosa di un po' matto. (In passato ai nostri progetti abbiamo dato nomi come quelli dei pappagalli killer che predano pecore, di illusioni ottiche e di città mitiche.) Perdete 30 minuti per escogitare un logo bizzarro e usatelo nei vostri memo e nelle vostre relazioni. Usate il nome del team ampiamente quando parlate con altri. Suona stupido, ma dà al team una identità da sfruttare, e al mondo qualcosa di memorabile da associare al vostro lavoro.

Non ripetetevi

Ne *I mali della duplicazione*, nel Capitolo 2, abbiamo parlato delle difficoltà di eliminare i doppioni di lavoro fra membri del gruppo. Questa duplicazione porta a fatica sprecata e può avere come risultato un incubo per la manutenzione. Chiaramente qui sono utili buone comunicazioni, ma a volte serve qualcosa di più.

Qualche team nomina bibliotecario di progetto uno dei membri, a cui viene data la responsabilità di coordinare la documentazione e i repository del codice. Altri membri del gruppo possono usare quella persona come primo interlocutore quando cercano qualcosa. Un buon bibliotecario sarà in grado anche di identificare rischi di duplicazione, leggendo il materiale che deve gestire.

Quando il progetto è troppo grande per un bibliotecario solo (o quando nessuno vuole avere questo ruolo), scegliete delle persone come punti focali di vari aspetti funzionali del lavoro. Se qualcuno vuol parlare di gestione delle date, deve sapere di parlare con Mary. Se c'è un problema con lo schema del database, si deve parlare con Fred.

E non dimenticate il valore dei sistemi di groupware e dei newsgroup locali di Usenet per comunicare e conservare domande e risposte.

Ortogonalità

L'organizzazione tradizionale dei team si basa sul vecchio metodo a cascata della costruzione del software. Ai singoli vengono assegnati ruoli in base alla loro funzione aziendale. Troverete analisti di business, architetti, progettisti, programmatori, tester, documentatori e cose simili. (In *The Rational Unified Process: An Introduction*, l'autore identifica 27 ruoli distinti all'interno di un team di progetto! [Kru98]) Qui è implicita una gerarchia: quanto più potete avvicinarvi all'utente, tanto più in alto siete.

Estremizzando, alcune culture dello sviluppo impongono divisioni rigide di responsabilità: i codificatori non possono parlare ai tester, che a loro volta non possono parlare all'architetto capo e così via. Alcune organizzazioni peggiorano ulteriormente il problema imponendo che sottogruppi diversi afferiscano a catene di gestione separate.

È un errore pensare che le attività di un progetto (analisi, progettazione, codifica, test) possano avvenire in isolamento. Non possono. Sono viste diverse sullo stesso problema e separarle artificialmente può causare una barca di problemi. I programmatori che sono a due o tre livelli di distanza dagli utenti effettivi del loro codice con tutta probabilità non saranno consapevoli del contesto in cui il loro lavoro deve essere usato. Non saranno in grado di prendere decisioni informate.

SUGGERIMENTO 60
Organizzate in base alle funzionalità, non alle funzioni aziendali.

Siamo favorevoli a una suddivisione dei team per funzionalità. Dividete le persone in piccoli team, ciascuno responsabile di un particolare aspetto funzionale del sistema finale. Lasciate che i team si organizzino da soli al loro interno, basandosi come possono sui punti di forza individuali. Ciascun team ha responsabilità verso gli altri nel progetto, come definito dagli impegni che hanno concordato. L'esatto insieme di impegni varia da progetto a progetto, e lo stesso vale per la suddivisione delle persone fra i team.

Funzionalità qui non significa necessariamente casi d'uso degli utenti finali. Conta lo strato di accesso al database, come conta il sottosistema della guida in linea. Puntiamo a team coesi, in gran parte auto-contenuti, esattamente gli stessi criteri che dobbiamo usare quando modularizziamo il codice. Esistono segni che avvertono quando l'organizzazione del team

è sbagliata: un esempio classico sono due sotto-team che lavorano allo stesso modulo o alla stessa classe nel programma.

In che modo è di aiuto questo stile funzionale di organizzazione? Organizzate le risorse con le stesse modalità utilizzate per organizzare il codice, applicando tecniche come i contratti (*Progettare per contratto*, Capitolo 4), il disaccoppiamento (*Disaccoppiamento e legge di Demetra*, Capitolo 5) e l'ortogonalità (*Ortogonalità*, Capitolo 2), e contribuite a isolare il team nel suo complesso dagli effetti del cambiamento. Se l'utente decide all'improvviso di cambiare il fornitore del database, solo il team del database deve esserne influenzato. Se il marketing all'improvviso decide di usare uno strumento già pronto per la funzione calendario, il gruppo del calendario soltanto assorbirà il colpo. Se ben seguita, questa forma di impostazione dei gruppi può ridurre drasticamente il numero delle interazioni fra le varie persone, riducendo i tempi, aumentando la qualità e abbassando il numero dei difetti. Questa impostazione può anche portare a un gruppo di sviluppatori più impegnato. Ogni team sa di essere l'unico responsabile di una particolare funzione, perciò sente anche maggiormente come proprio ciò che produce.

Questa impostazione però funziona solo con sviluppatori responsabili e un forte management di progetto. Creare una serie di team autonomi e lasciarli a briglia sciolta senza leadership è una ricetta per il disastro. Il progetto ha bisogno di almeno due "teste", una tecnica, l'altra amministrativa. La testa tecnica fissa filosofia e stile di sviluppo, assegna le responsabilità ai gruppi e fa da arbitro nelle inevitabili "discussioni" fra le persone. Tiene anche costantemente sott'occhio il quadro generale, cercando di identificare gli elementi comuni non necessari fra gruppi diversi, che potrebbero ridurre l'ortogonalità del lavoro complessivo. La testa amministrativa, il manager di progetto, pianifica le risorse di cui i team hanno bisogno, controlla e riferisce sull'andamento del lavoro e contribuisce a decidere le priorità in funzione delle esigenze di business. La testa amministrativa può fungere anche da ambasciatore del team quando si tratta di comunicare con il mondo esterno.

I team di progetti più ampi hanno bisogno di ulteriori risorse: un bibliotecario che indicizzi e archivi codice e documentazione, un costruttore di strumenti che fornisca strumenti e ambienti comuni, sostegno operativo e così via.

Questo tipo di organizzazione del team è simile, nel suo spirito, al vecchio concetto di *chief programmer team*, documentato per la prima volta nel 1972 [Bak72].

Automazione

Un ottimo modo per garantire coerenza e precisione è automatizzare tutto ciò che il team fa. Perché formattare manualmente il codice quando l'editor che usate può farlo per voi automaticamente mentre scrivete? Perché completare moduli di test quando la build notturna può eseguire i test in automatico?

L'automazione è una componente essenziale di ogni team di progetto - abbastanza importante per dedicarle un'intera sezione (la prossima). Per essere sicuri che le cose vengano automatizzate, nominate uno o più membri del team *costruttori di strumenti* (*tool builder*), perché costruiscano e mettano in esercizio gli strumenti che automatizzano la macchina del progetto. Fate produrre loro makefile, script di shell, modelli dell'editor, programmi di servizio e cose simili.

Sapere quando smettere di aggiungere vernice

Ricordate che i team sono fatti di individui. Date a ciascun membro la possibilità di mettersi in luce nel modo che gli è congeniale. Date loro quel tanto di struttura che basta a sostenerli e ad avere la certezza che il progetto vada in porto rispettando i requisiti. Poi, come il pittore in *Software abbastanza buono* (Capitolo 1), resistete alla tentazione di aggiungere altra vernice.

Vedi anche

- *Entropia del software*, Capitolo 1
- *Zuppa di pietre e rane bollite*, Capitolo 1
- *Software abbastanza buono*, Capitolo 1
- *Comunicate!*, Capitolo 1
- *I mali della duplicazione*, Capitolo 2
- *Ortogonalità*, Capitolo 2
- *Progettare per contratto*, Capitolo 4

- *Disaccoppiamento e legge di Demetra*, Capitolo 5
- *Automazione onnipresente*, in questo Capitolo

Sfide

- Guardatevi intorno: ci sono team di successo al di fuori del campo dello sviluppo di software? Che cosa li rende team di successo? Usano qualcuno dei processi di cui si è parlato in questa sezione?
- La prossima volta che avviate un progetto, provate a convincere le persone a dargli un brand. Date alla vostra organizzazione il tempo di abituarsi all'idea, poi fate una rapida indagine per vedere quale differenza ha prodotto, sia all'interno del team, sia esternamente.
- Algebra dei team: a scuola, ci venivano assegnati problemi del tipo "Se 4 operai impiegano 6 ore per scavare un fossato, quanto tempo impiegheranno 8 operai?". Nella vita reale, però, quali fattori influenzano la risposta alla domanda: "Se 4 programmatori impiegano 6 mesi a sviluppare un'applicazione, quanto tempo impiegheranno 8 programmatori?". In quanti scenari il tempo si riduce realmente?

Automazione onnipresente

> La civiltà avanza estendendo il numero delle operazioni importanti che possiamo svolgere senza pensare.
> - Alfred North Whitehead

All'alba dell'era dell'automobile, le istruzioni per far partire una Ford Model-T erano lunghe più di due pagine. Nelle auto moderne, basta girare una chiave: la procedura di accensione è automatica e a prova di stupido. Una persona che segue un elenco di istruzioni può ingolfare il motore, lo starter automatico no.

Anche se quello dell'informatica è un campo ancora allo stadio della Model-T, non possiamo permetterci di ripercorrere continuamente due pagine di istruzioni per svolgere qualche operazione comune. Che si tratti della procedura di build e rilascio, della documentazione della revisione del codice o di qualsiasi altra attività che si ripete nel corso del progetto, deve essere automatica. Dovremo magari costruire starter e iniettore del carburante da zero ma, una volta fatto, da quel momento basterà girare la chiave.

Inoltre, vogliamo garantire al progetto coerenza e ripetibilità. Le procedure manuali affidano la coerenza al caso; la ripetibilità non è garantita, in particolare se ci sono aspetti della procedura aperti all'interpretazione di persone diverse.

Tutto in automatico

Una volta siamo stati nella sede di un cliente, dove tutti gli sviluppatori usavano lo stesso IDE. L'amministratore di sistema aveva dato a ogni sviluppatore le istruzioni per installare pacchetti aggiuntivi all'IDE e le istruzioni riempivano varie pagine - piene di fai clic qui, scorri là, trascina questo, fai doppio clic su quello e poi fallo ancora.

Non sorprende che le macchine dei vari sviluppatori fossero tutte leggermente diverse e che quando sviluppatori diversi eseguivano lo stesso codice si verificassero piccole differenze nel comportamento dell'applicazione. Su una macchina comparivano errori che nelle altre non emergevano. Tracciare le differenze di versione di qualsiasi componente di solito era fonte di qualche sorpresa.

SUGGERIMENTO 61
Non usate procedure manuali.

Le persone non sono "ripetibili" come i computer. Né dovremmo aspettarci che lo siano. Uno script di shell o un file batch eseguirà le stesse istruzioni nello stesso ordine ogni volta. Lo si può sottoporre a controllo del sorgente, in modo da poter esaminare cambiamenti nella procedura nel corso del tempo ("ma ha sempre funzionato…").

Un altro strumento di automazione è `cron` (o `at` in Windows NT). Permette di pianificare attività periodiche senza intervento umano - di solito nel cuore della notte. Per esempio, il seguente file `crontab` specifica che un comando `nightly` di un progetto venga eseguito cinque minuti dopo mezzanotte ogni giorno, che il backup venga eseguito alle 3:15 del mattino nei giorni lavorativi e che `expense_reports` venga eseguito a mezzanotte il primo giorno di ogni mese.

```
# MIN HOUR DAY MONTH DAYOFWEEK    COMMAND
# ------------------------------------------------------------
    5    0   *      *    *        /projects/Manhattan/bin/nightly
   15    3   *      *  1-5        /usr/local/bin/backup
    0    0   1      *    *        /home/accounting/expense_reports
```

Con `cron` possiamo pianificare backup, la build notturna, la manutenzione del sito web e qualsiasi altra cosa si debba fare - senza intervento umano, automaticamente.

Compilazione del progetto

La compilazione del progetto è un'attività che deve essere affidabile e ripetibile. In genere compiliamo i progetti con makefile, anche quando usiamo un ambiente IDE. L'uso di makefile ha vari vantaggi. È una procedura automatica mediante script. Possiamo includere agganci per generare il codice per noi e per eseguire automaticamente test di regressione. Gli IDE hanno i loro vantaggi, ma con gli IDE da soli può essere difficile ottenere il livello di automazione che cerchiamo. Vogliamo controllare, effettuare la build, sottoporre a test e spedire con un unico comando.

Generare codice

Ne *I mali della duplicazione* (Capitolo 2), abbiamo sostenuto la generazione di codice per derivare conoscenza da fonti comuni. Possiamo sfruttare il meccanismo di analisi delle dipendenze di `make` per rendere facile questo processo. È molto semplice aggiungere regole a un makefile per generare automaticamente un file da qualche altro sorgente. Per esempio, supponiamo di voler prendere un file XML, generare da questo un file Java e compilare il risultato:

```
.SUFFIXES: .java .class .xml
.xml.java:
        perl convert.pl $< > $@
.java.class:
        $(JAVAC) $(JAVAC_FLAGS) $<
```

Scrivete `test.class` e `make` cercherà automaticamente un file con il nome `test.xml`, costruirà un file `.java` eseguendo uno script Perl, poi compilerà quel file per produrre `test.class`.

Possiamo usare lo stesso genere di regole per generare automaticamente anche codice sorgente, file di intestazione o documentazione da altre forme (vedi *Generatori di codice*, Capitolo 3).

Test di regressione

Potete usare il makefile anche per eseguire test di regressione, o per un singolo modulo o per un intero sottosistema. Potete facilmente sottoporre a test l'*intero* progetto con un solo comando in testa all'albero del sorgente, oppure potete sottoporre a test un singolo modulo utilizzando lo stesso comando in una sola directory. Vedete *Test senza pietà*, più avanti in questo capitolo, per ulteriori informazioni sui test di regressione.

Make ricorsivo
Molti progetti impostano makefile gerarchici ricorsivi per le build e il test dei progetti. Dovete fare attenzione però ad alcuni problemi potenziali.
`make` calcola le dipendenze fra i vari target che deve costruire, ma può analizzare solo le dipendenze esistenti nell'ambito di una singola invocazione di `make`. In particolare, un `make` ricorsivo non sa nulla delle dipendenze che possono avere altre invocazioni di `make`. Se siete attenti e precisi, potete ottenere i risultati giusti, ma è facile provocare lavoro ulteriore non necessario - oppure perdere una dipendenza e *non* ricompilare quando sarebbe necessario.
Inoltre, le dipendenze in fase di build possono non essere identiche alle dipendenze in fase di test, e potrebbero essere necessarie gerarchie separate.

Automazione delle build

Una *build* è una procedura che prende una directory vuota (e un ambiente di compilazione noto) e costruisce il progetto da zero, producendo quello che speriamo di produrre come oggetto finale consegnabile, l'immagine master di un CD-ROM o un archivio auto-estraibile, per esempio. Normalmente una build di progetto comprenderà i passi seguenti.

1. Estrae il codice sorgente dal repository.
2. Costruisce il progetto da zero, normalmente da un makefile di massimo livello. Ogni build è contrassegnata da qualche forma di numero di release o di versione, o magari da una indicazione di data.
3. Crea una immagine distribuibile. Questa procedura può comportare la definizione di proprietà e permessi del file e la produzione di tutti gli esempi, la documentazione, i file README e tutto ciò che verrà spedito con il prodotto, nell'esatto formato che sarà necessario per la pubblicazione. (Se producete un CD-ROM in formato ISO9660, per

esempio, eseguireste il programma che produce un'immagine bit per bit del file system 9660. Perché aspettare la notte prima della consegna per verificare che funzioni?)

4. Esegue test specifici (`make test`).

Per la maggior parte dei progetti, questo livello di build viene eseguito automaticamente ogni notte. Nella build notturna, normalmente si eseguono test più completi di quelli che il singolo può eseguire quando costruisce qualche parte specifica del progetto. L'aspetto importante è che la build completa esegua *tutti* i test disponibili. Volete sapere se un test di regressione è fallito a causa di uno dei cambiamenti apportati oggi al codice. Identificando il problema vicino alla fonte, avete migliori probabilità di identificarlo e di risolverlo.

Se non eseguite regolarmente i test, potreste scoprire che l'applicazione è andata in tilt per un cambiamento apportato al codice tre mesi fa. Vi auguriamo di riuscire a scoprirlo.

Build finali

Le *build finali*, quelle che si intende diffondere come prodotti, possono avere requisiti diversi dalle normali build notturne. Una build finale può richiedere che il repository sia bloccato, o etichettato con un numero di release, che i flag di ottimizzazione e di debug siano impostati in modo diverso, e via di questo passo. Noi di solito usiamo un target distinto per `make` (per esempio `make final`), che imposti tutti questi parametri in una volta sola.

Ricordate che, se il prodotto è compilato in modo diverso rispetto a versioni precedenti, bisogna sottoporre a test di nuovo *questa* versione.

Automazione delle attività amministrative

Non sarebbe bello se i programmatori potessero dedicare tutto i loro tempo alla programmazione? Purtroppo, raramente è così. Ci sono email a cui rispondere, scartoffie da compilare, documenti da pubblicare sul Web e così via. Potete decidere di creare uno script di shell che faccia un

po’ del lavoro sporco, ma dovrete comunque ricordare di mandare in esecuzione lo script quando serve.

Dato che la memoria è la seconda cosa che si perde con l’avanzare dell’età, non vogliamo farci troppo affidamento. Possiamo eseguire script che svolgano automaticamente delle procedure per noi, sulla base dei *contenuti* del codice sorgente e dei documenti. Il nostro obiettivo è mantenere un flusso di lavoro automatico, senza bisogno di intervento umano, guidato dai contenuti.

Generazione del sito web

Molti team di sviluppo usano un sito web interno per le comunicazioni del progetto, e pensiamo sia un’ottima idea. Non vogliamo però dedicare troppo tempo alla manutenzione del sito web e non vogliamo neanche che diventi obsoleto. Informazioni fuorvianti sono peggio dell’assenza di informazioni.

La documentazione estratta da codice, analisi dei requisiti, documenti di progettazione e tutti i disegni e i grafici devono essere pubblicati sul Web con regolarità. A noi piace pubblicare questi documenti in modo automatico nell’ambito della build notturna o come un “aggancio” nella procedura di controllo d’ingresso del codice sorgente.

Comunque si faccia, i contenuti web dovrebbero essere generati automaticamente dalle informazioni presenti nel repository e pubblicate *senza* intervento umano. Questa è in effetti un’altra applicazione del principio DRY: le informazioni esistono in una sola forma come codice e documenti accettati. La visualizzazione da un browser web è semplicemente questo, solo una visualizzazione. Non deve essere necessario manutenere la visualizzazione manualmente.

Qualsiasi informazione generata dalla build notturna deve essere accessibile sul sito web di sviluppo: i risultati della build stessa (presentati magari come riepilogo di una pagina con avvertimenti del compilatore, errori e stato corrente), test di regressione, statistiche di prestazioni, metriche della codifica e ogni altro tipo di analisi statica, e via elencando.

Procedure di approvazione

Alcuni progetti hanno vari flussi di lavoro amministrativo da seguire. Per esempio, si devono pianificare revisioni del codice o del progetto, che poi vanno seguite, bisogna avere una procedura di approvazione e così via. Si può usare l'automazione (e in particolare il sito web) per alleggerire il carico delle scartoffie.

Supponiamo che vogliate automatizzare la pianificazione e l'approvazione della revisione del codice. Potreste inserire un marcatore speciale in ogni file di codice sorgente:

```
/*Status: needs_review */
```

Un semplice script potrebbe esaminare tutto il codice sorgente e identificare tutti i file con uno status `needs_review`, che indica che sono pronti per la revisione. Poi potreste pubblicare un elenco di quei file come pagina web, inviare automaticamente un messaggio di posta elettronica alle persone giuste, o addirittura fissare automaticamente una riunione mediante un software di calendario.

Potete impostare sulla pagina web un modulo con cui i revisori possano registrare l'approvazione o meno. Dopo la revisione, lo stato può essere modificato automaticamente in `reviewed`. Sta a voi decidere se effettuare un esame del codice con tutti i partecipanti; potete comunque svolgere automaticamente il lavoro amministrativo. (In un articolo dell'aprile 1999, Robert Glass riassume ricerche da cui sembra emergere che, mentre l'ispezione del codice è efficace, condurre le revisioni nelle riunioni non lo sia [Gla99a].)

I figli del calzolaio

I figli del calzolaio hanno le scarpe rotte. Spesso, le persone che sviluppano software usano per il loro lavoro gli strumenti più scalcinati.

Noi però abbiamo tutte le materie prime necessarie a fabbricare strumenti migliori. Abbiamo `cron`. Abbiamo `make`, Ant e Cruisecontrol per l'automazione (vedi [Cla04]). E abbiamo Ruby, Perl e altri linguaggi di scripting di alto livello per sviluppare rapidamente strumenti personalizzati, generatori di pagine web, generatori di codice, test harness e così via.

Che sia il computer a fare tutte le cose ripetitive e di routine, farà un lavoro migliore di noi. Noi abbiamo cose più importanti e più difficili da

fare.

Vedi anche

- *Il gatto mi ha mangiato il codice sorgente*, Capitolo 1
- *I mali della duplicazione*, Capitolo 2
- *Il potere del puro testo*, Capitolo 3
- *Giochi di shell*, Capitolo 3
- *Debug*, Capitolo 3
- *Generatori di codice*, Capitolo 4
- *Team pragmatici*, in questo Capitolo
- *Test senza pietà*, in questo Capitolo
- *È tutta scrittura*, in questo Capitolo

Sfide

- Tenete nota delle vostre abitudini nel corso della giornata lavorativa. Vedete attività ripetitive? Scrivete continuamente la stessa sequenza di comandi? Provate a scrivere qualche script di shell per automatizzare il processo. Fate continuamente clic sulla stessa sequenza di icone? Potete creare una macro che lo faccia per voi?
- Quanta parte del lavoro amministrativo del vostro progetto può essere automatizzata? Dato il costo elevato dello staff di programmazione, determinate quanta parte del budget del progetto va sprecata in procedure amministrative. Potete giustificare il tempo necessario a predisporre una soluzione automatizzata, sulla base dei risparmi complessivi che potreste ottenere?

Test senza pietà

La maggior parte degli sviluppatori odia i test. Tende a fare test morbidi, sapendo nel subconscio dove il codice fallirà ed evitando i punti deboli. I programmatori pragmatici sono diversi. Noi siamo *motivati* a trovare i nostri errori *adesso*, in modo da non doverci vergognare quando altri troveranno i nostri errori più tardi.

Trovare errori è come andare a pesca senza una rete. Usiamo reti fini e piccole (unit test) per catturare i pesciolini e grandi reti più grezze (test di integrazione) per catturare gli squali killer. A volte i pesci riescono a sfuggire, così ripariamo i buchi che troviamo, con la speranza di catturare sempre più difetti scivolosi che nuotano nel mare del nostro progetto.

SUGGERIMENTO 62

Fate test presto. Fate test spesso. Fate test automaticamente.

Vogliamo cominciare a fare test non appena abbiamo del codice. Questi pesciolini hanno la pessima abitudine di diventare rapidamente giganteschi squali mangiauomini e catturare uno squalo è molto più difficile. Ma non vogliamo eseguire tutti i test manualmente.

Molti team sviluppano per i loro progetti complicati piani di test. A volte li usano effettivamente. Abbiamo scoperto però che i team che usano test automatizzati hanno molte più possibilità di successo. I test che vengono eseguiti a ogni build sono molto più efficaci dei piani di test che restano a prendere polvere sugli scaffali.

Quanto prima si identifica un errore, tanto meno costoso è porvi rimedio. "Codifica un po', testa un po'" è un detto diffuso nel mondo Smalltalk, e possiamo adottare quel mantra scrivendo codice di test contemporaneamente alla scrittura del codice di produzione (o addirittura prima). L'eXtreme Programming [URL45] definisce questo concetto "integrazione continua, test senza sosta".

In effetti, un buon progetto può avere *più* codice di test che codice di produzione. Il tempo necessario a produrre quel codice di test è ben impiegato. Alla lunga i costi saranno molto minori, e si ha effettivamente la possibilità di costruire un prodotto con un numero di difetti vicino a zero.

Inoltre, sapere di aver superato il test dà un forte senso di fiducia che un pezzo di codice sia "finito".

SUGGERIMENTO 63

La codifica non è finita finché non sono stati eseguiti con successo tutti i test.

Solo perché avete finito di mettere insieme un pezzo di codice non significa che possiate andare dal vostro capo o dal cliente a dire che è *completo*. Non lo è. Innanzitutto, il codice non è mai davvero finito. Cosa più importante, non potete sostenere che sia usabile da qualcuno finché non supera tutti i test disponibili.

Dobbiamo esaminare tre aspetti dei test a livello di progetto: che cosa sottoporre a test, come sottoporlo a test e quando sottoporlo a test.

Che cosa sottoporre a test

Esistono vari tipi importanti di test del software che dovete eseguire:

- unit test;
- test di integrazione;
- validazione e verifica;
- esaurimento delle risorse, errori e recupero;
- test di prestazioni;
- test di usabilità.

L'elenco non è affatto completo e progetti specializzati avranno bisogno anche di vari altri tipi di test, ma ci dà comunque un buon punto di partenza.

Unit test

Uno *unit test* è codice che mette alla prova un modulo. Ne abbiamo parlato in *Codice facile da sottoporre a test* (Capitolo 6). Questo tipo di test è il fondamento di tutte le altre forme di test di cui parleremo in questa sezione. Se le parti non funzionano isolatamente, probabilmente non funzioneranno bene neanche insieme. Tutti i moduli che usate devono superare i loro unit test prima di poter procedere.

Una volta che tutti i moduli pertinenti hanno superato i loro test individuali, siete pronti per la fase successiva. Dovete verificare come tutti i moduli interagiscano fra loro in tutto il sistema.

Test di integrazione

I *test di integrazione* servono a verificare che i principali sottosistemi che costituiscono il progetto lavorino bene insieme. Se sono stati definiti e ben verificati dei buoni contratti, gli eventuali problemi di integrazione si possono identificare facilmente; altrimenti, l'integrazione diventa un fertile terreno di coltura per gli errori. In effetti, spesso è la fonte di errori più grande nel sistema.

I test di integrazione sono in realtà solo un'estensione degli unit test che abbiamo descritto, solo che ora si sottopone a test come i sottosistemi nel loro complesso onorino i loro contratti.

Validazione e verifica

Non appena avete un'interfaccia utente eseguibile o un prototipo, dovete dare una risposta a una domanda importantissima: gli utenti vi hanno detto quello che volevano, ma è quello di cui hanno bisogno?

Soddisfa i requisiti funzionali del sistema? Anche questo deve essere sottoposto a test. Un sistema libero da bachi che risponde alla domanda sbagliata non è molto utile. Dovete essere consapevoli degli schemi di accesso degli utenti finali e di come si differenziano dai dati di test degli sviluppatori (per un esempio, vedete il caso delle pennellate in *Debug*, nel Capitolo 3).

Esaurimento delle risorse, errori e recupero

Ora che avete una buona idea che il sistema si comporterà correttamente in condizioni ideali, dovete scoprire come si comporterà nelle condizioni del *mondo reale*. Nel mondo reale, i vostri programmi non avranno risorse illimitate, e potrebbero restare a corto di qualcosa. Fra i limiti che il vostro codice potrebbe incontrare vi sono:

- memoria;
- spazio su disco;
- ampiezza di banda della CPU;
- tempo;
- ampiezza di banda del disco;
- ampiezza di banda della rete;
- tavolozza dei colori;
- risoluzione del video.

Forse fate già verifiche sui problemi di allocazione di spazio sul disco o di memoria, ma quanto spesso sottoponete a test gli altri aspetti? La vostra applicazione troverà posto su uno schermo 640 × 480 con 256 colori? Girerà su uno schermo da 1600 × 1280 con colori a 24 bit senza ridursi a un francobollo? Il lavoro in batch finirà prima che inizi l'archiviazione?

Potete identificare limitazioni ambientali, come le specifiche del video, e applicare gli adattamenti opportuni. Non tutti i guasti però sono recuperabili. Se il vostro codice scopre che la memoria è esaurita, le

vostre opzioni sono limitate: magari non avete più risorse per fare altro che fallire.

Quando il sistema fallisce, fallisce con grazia? Prova, nel modo migliore possibile, di salvare il proprio stato e di impedire la perdita del lavoro, oppure crollerà miseramente davanti agli occhi dell'utente?

Test di prestazioni

Anche i test di prestazioni, stress test o test sotto carico possono essere un aspetto importante del progetto.

Chiedetevi se il software soddisfa i requisiti di prestazione nelle condizioni del mondo reale, con il numero previsto di utenti, di connessioni o di transazioni al secondo. È scalabile?

Per alcune applicazioni, avrete bisogno di hardware o software specializzato per simulare in modo realistico i carichi.

Test di usabilità

I test di usabilità sono diversi dai tipi di test di cui abbiamo parlato fin qui. Vengono eseguiti con utenti reali, in condizioni ambientali reali.

Pensate l'usabilità in termini di fattori umani. Ci sono stati fraintendimenti nell'analisi dei requisiti che devono essere risolti? Il software si adatta all'utente come se fosse un'estensione della sua mano? (Non solo vogliamo che i nostri strumenti si adattino alle nostre mani, vogliamo che gli strumenti che creiamo per gli utenti si adattino alle loro mani.)

Come per la validazione e la verifica, dovete eseguire test di usabilità il più presto possibile, quando c'è ancora tempo per apportare correzioni. Per progetti più grandi, potete coinvolgere specialisti dei fattori umani. (Se non altro, è divertente giocare con gli specchi unidirezionali.)

Come sottoporre a test

Abbiamo visto *che cosa* sottoporre a test; ora dobbiamo rivolgere la nostra attenzione a *come* sottoporlo a test, prestando particolare attenzione a:

- test di regressione;
- dati di test;
- collaudo dei sistemi GUI;
- test dei test;
- test esaustivi.

Test di regressione

Un test di regressione confronta l'output del test corrente con valori precedenti (o valori noti). Possiamo così assicurarci che gli errori risolti oggi non hanno avuto conseguenze negative su cose che ieri funzionavano bene. È una rete di sicurezza importante, e riduce nettamente la possibilità di sorprese sgradevoli.

Tutti i test che abbiamo citato fin qui possono essere eseguiti come test di regressione, dandoci la sicurezza di non aver perso nulla nello sviluppare nuovo codice. Possiamo eseguire regressioni anche per verificare prestazioni, contratti, validità e così via.

Test di progettazione/metodologia
Potete sottoporre a test la progettazione del codice stesso e la metodologia utilizzata per costruire il software? In un certo senso, sì, è possibile. Lo potete fare analizzando le *metriche* - le misurazioni dei vari aspetti del codice. La metrica più semplice (e spesso anche la più interessante) sono le *righe di codice*: quanto è grande il codice stesso?
Esistono molte altre metriche diverse che si possono usare per esaminare il codice, fra cui:

- MacCabe Cyclomatic Complexity Metric (misura la complessità delle strutture di decisione);
- Inheritance fan-in (numero di classi base) e fan-out (numero di moduli derivati che usano quello in oggetto come genitore);
- insieme di risposta (vedi *Disaccoppiamento e legge di Demetra*, Capitolo 5);
- rapporti di accoppiamento fra classi (vedi [URL48]).

Alcune metriche sono pensate in modo da dare un "voto di idoneità", mentre altre sono utili solo per confronto: si calcolano, cioè, queste metriche per tutti i moduli del sistema e si vede come si comporta un certo modulo rispetto agli altri. Qui di solito si applicano tecniche statistiche standard, come la media e la deviazione standard.
Se si trova un modulo le cui metriche sono nettamente diverse da quelle di tutti gli altri, bisogna chiedersi se è giusto. Per qualche modulo sarà un comportamento giustificato, ma per quelli che *non* hanno una buona scusa può indicare la possibile esistenza di problemi.

Dati di test

Da dove prendiamo i dati per eseguire tutti questi test? Esistono solo due tipi di dati: dati del mondo reale e dati di sintesi. In effetti dobbiamo usare entrambi, perché la differente natura di questi tipi di dati esporrà errori diversi nel nostro software.

I dati reali vengono da qualche fonte reale, magari raccolti da un sistema preesistente, dal sistema di un concorrente o da un prototipo di qualche tipo. Rappresentano tipici dati d'utente. Le grandi sorprese vengono quando si scopre che cosa significa *tipici*. È il momento in cui è più probabile vengano alla luce difetti e incomprensioni nell'analisi dei requisiti.

I dati di sintesi sono generati artificialmente, magari nel rispetto di determinati vincoli statistici. I motivi per cui si possono usare dati di sintesi sono numerosi.

- Servono molti dati, più di quelli che potrebbe fornire qualsiasi campione estratto dal mondo reale. Potreste riuscire a usare i dati reali come seme per generare un campione più grande e aggiustare qualche campo che deve avere valori univoci.
- Servono dati per mettere alla prova condizioni limite. Questi dati possono essere completamente di sintesi: campi data che contengono il 29 febbraio 1999, record di dimensioni enormi o indirizzi con codici postali esteri.
- Servono dati con determinate proprietà statistiche. Volete vedere che cosa succede se una transazione ogni tre fallisce? Ricordate l'algoritmo di ordinamento che rallenta enormemente quando gli sono forniti dati già ordinati? Potete presentare dati in ordine casuale o già ordinati per mettere in luce questo tipo di debolezza.

Mettere alla prova sistemi GUI

Per poter mettere alla prova sistemi in cui hanno molto spazio interfacce grafiche spesso sono necessari strumenti di test specializzati. Questi strumenti possono essere basati su un semplice modello di cattura/riproduzione degli eventi, oppure possono richiedere script

costruiti appositamente per pilotare la GUI. Alcuni sistemi combinano elementi di entrambi i tipi.

Strumenti meno sofisticati creano un grado elevato di accoppiamento fra la versione del software sottoposta a test e lo script di test stesso: se spostate una finestra di dialogo o riducete le dimensioni di un pulsante, il test può non essere in grado di trovarli e può fallire. La maggior parte degli strumenti moderni di test per le GUI usa più tecniche diverse per aggirare questo problema e cercare di adattarsi a piccole differenze di layout.

Non si può automatizzare tutto. Andy ha lavorato a un sistema di grafica che consentiva all'utente di creare e visualizzare effetti visivi non deterministici che simulavano vari fenomeni naturali. Purtroppo, nei test non si poteva semplicemente prendere una bitmap e confrontare l'output con una esecuzione precedente, perché per la natura del programma ogni volta la forma era diversa. Per situazioni di questo genere, non si ha altra scelta che fare affidamento sull'interpretazione manuale dei risultati dei test.

Uno dei molti vantaggi della scrittura di codice disaccoppiato (vedi *Disaccoppiamento e legge di Demetra*, Capitolo 5) è la maggiore modularità dei test. Per esempio, per applicazioni di elaborazione dati con una GUI di front-end, il progetto deve essere sufficientemente disaccoppiato da poter sottoporre a test la logica dell'applicazione *senza* la presenza della GUI. Quest'idea è simile a quella di sottoporre a test per primi i sottocomponenti. Validata la logica dell'applicazione, diventa più facile identificare gli errori che si manifestano quando si introduce l'interfaccia utente (è probabile che gli errori siano stati generati dal codice dell'interfaccia utente).

Test dei test

Non è possibile scrivere software perfetto, perciò ne consegue che non è possibile neanche scrivere software di test perfetto. Dobbiamo sottoporre a test i test.

Possiamo pensare il nostro insieme di gruppi di test come un sistema di sicurezza complesso, progettato per far partire un allarme quando si presenta un errore. Quale modo migliore per collaudare un sistema di sicurezza che tentare di forzare la porta d'ingresso?

Scritto un test per identificare un particolare errore, *provocate* deliberatamente l'errore e assicuratevi che il test entri in azione. Questo garantisce che il test catturi l'errore se si verifica realmente.

SUGGERIMENTO 64
Usate dei sabotatori per mettere alla prova i vostri test.

Se volete essere *davvero* seri nei vostri test, vorrete nominare un *sabotatore di progetto*. Il ruolo del sabotatore è prendere una copia dell'albero del sorgente, introdurvi di proposito degli errori e verificare che i test li catturino.

Quando scrivete dei test, assicuratevi che gli allarmi suonino quando devono.

Test esaustivi

Quando siete fiduciosi che i vostri test siano corretti e che trovino gli errori che create, come fate a sapere se avete sottoposto a test sufficientemente completi la base di codice?

La risposta, in breve, è "non potete saperlo" e non ci riuscirete mai. Ma esistono sul mercato prodotti che possono essere d'aiuto. Questi strumenti di *analisi della copertura* osservano il vostro codice durante i test e tengono traccia di quali righe di codice siano state eseguite e quali no. Questi strumenti contribuiscono a dare un'idea generale di quanto siano esaustivi i test, ma non aspettatevi di vedere una copertura del 100 per cento.

Anche se per caso venisse chiamata in causa ogni riga di codice, questo non esaurisce il quadro. Quel che è importante è il numero degli stati che può avere il vostro programma, e gli stati non sono equivalenti a righe di codice. Per esempio, supponiamo che abbiate una funzione che prende due interi, ciascuno dei quali può essere un numero fra 0 e 999.

```
int test(int a, int b) {
  return a / (a + b);
}
```

In teoria, questa funzione di sole tre righe ha 1.000.000 di stati logici, 999.999 dei quali funzioneranno correttamente e uno no (quando `a + b` è uguale a zero). Sapere semplicemente che questa riga di codice è stata eseguita non dice molto - dovete identificare tutti i possibili stati del programma. Purtroppo, in generale questo è un problema *davvero difficile*.

SUGGERIMENTO 65
Sottoponete a test la copertura degli stati, non quella del codice.

Anche con una buona copertura del codice, i dati che si usano per i test hanno un forte impatto e, cosa ancora più importante, l'*ordine* in cui si percorre il codice può avere le conseguenze più estese.

Quando sottoporre a test

In molti progetti si tende a lasciare i test per l'ultimo minuto, proprio quando si è vicini a una scadenza. Bisogna cominciare molto prima: non appena esiste del codice di produzione, bisogna sottoporlo a test.

La maggior parte dei test deve essere eseguita in automatico. È importante notare che dicendo "in automatico" intendiamo dire che anche i *risultati* dei test vengono interpretati automaticamente. (Vedi *Automazione onnipresente* in questo stesso capitolo, per un approfondimento.)

A noi piace fare i test con la massima frequenza possibile, e sempre prima che il codice sia ammesso al repository del sorgente. Alcuni sistemi di controllo del codice sorgente possono farlo automaticamente; altrimenti, scriviamo semplicemente

```
% make test
```

Di solito non è un problema eseguire regressioni su tutti i singoli unit test e sui test di integrazione con la frequenza necessaria. Potrebbe però non essere facile eseguire alcuni test con tanta frequenza. Gli stress test, per esempio, possono richiedere una configurazione o apparecchiature speciali e un po' di controllo diretto. Questi test si possono eseguire meno spesso - una volta alla settimana o al mese, magari. È importante però che vengano eseguiti con regolarità, in modo pianificato. Se non possono essere eseguiti automaticamente, è importante che compaiano nei piani di lavoro, con tutte le risorse necessarie associate a questa attività.

Stringere la rete

Infine, vogliamo rivelarvi il concetto in assoluto più importante per i test. È un concetto ovvio e praticamente tutti i manuali dicono di fare in

questo modo, ma, per qualche motivo, la maggior parte dei progetti non lo fa.

Se un errore sfugge alla rete dei test esistenti, dovete aggiungere un nuovo test per catturarlo la prossima volta.

SUGGERIMENTO 66
Trovate gli errori una volta sola.

Vedi anche

- *Il gatto mi ha mangiato il codice sorgente*, Capitolo 1
- *Debug*, Capitolo 3
- *Disaccoppiamento e legge di Demetra*, Capitolo 5
- *Refactoring*, Capitolo 6
- *Codice facile da sottoporre a test*, Capitolo 6
- *Automazione onnipresente*, in questo Capitolo

Sfide

- Potete sottoporre a test in modo automatico il vostro progetto? Molti team sono costretti a rispondere “no”. Perché? Perché è troppo difficile definire i risultati accettabili? In questo modo non diventa difficile dimostrare agli sponsor che il progetto è “finito”?
- È difficile sottoporre a test la logica dell’applicazione indipendentemente dalla GUI? Che cosa dice questo a proposito della GUI? E a proposito dell’accoppiamento?

È tutta scrittura

L’inchiostro più sbiadito è meglio della migliore memoria.
- Proverbio cinese

Normalmente, gli sviluppatori non pensano molto alla documentazione. Nel migliore dei casi è una necessità spiacevole; nel peggiore dei casi è trattata come un compito a bassa priorità, nella speranza che il management se ne dimentichi alla fine del progetto.

I programmatori pragmatici considerano la documentazione una parte integrante del processo complessivo di sviluppo. Si può rendere più facile la scrittura della documentazione non duplicando gli sforzi e non

perdendo tempo, e tenendo sempre la documentazione a portata di mano, nel codice stesso, se possibile.

Non sono idee particolarmente originali o nuove; quella di sposare codice e documentazione compare già nel lavoro di Donald Knuth sulla programmazione competente e nell'utility JavaDoc della Sun, solo per fare un paio di esempi. Vogliamo far sparire la dicotomia fra codice e documentazione e trattarli invece come due viste sullo stesso modello (vedi *È solo una vista*, nel Capitolo 5). In realtà, vogliamo andare un po' oltre e applicare *tutti* i nostri principi pragmatici alla documentazione come al codice.

SUGGERIMENTO 67
Trattate la vostra lingua come se fosse proprio un altro linguaggio di programmazione.

Esistono fondamentalmente due tipi di documentazione prodotta per un progetto: interna ed esterna. La documentazione interna comprende commenti al codice sorgente, documenti di progettazione e test e così via. La documentazione esterna è tutto ciò che viene pubblicato nel mondo esterno, come i manuali per l'utente. Indipendentemente dal pubblico a cui è destinata, o dal ruolo di chi la scrive (sviluppatore o technical writer), tutta la documentazione è uno specchio del codice. Se esiste una discrepanza, quello che conta, nel bene o nel male, è il codice.

SUGGERIMENTO 68
Mettete la documentazione dentro, non applicatela poi all'esterno.

Cominciamo con la documentazione interna.

Commenti nel codice

Produrre documenti formattati a partire dai commenti e dalle dichiarazioni nel codice sorgente è abbastanza immediato, ma prima bisogna essere sicuri che ci *siano* davvero commenti nel codice. Il codice deve avere commenti, ma troppi possono essere un male come troppo pochi.

In generale, i commenti devono dire *perché* è stata fatta una certa cosa, il suo senso e il suo scopo. Il codice già mostra *come* è stata fatta, perciò ogni commento in proposito è ridondante, ed è una violazione del principio DRY.

Quando si commenta il codice sorgente si ha l'occasione perfetta per documentare quelle particolarità sfuggenti del progetto che non possono essere documentate altrove: compromessi tecnici, ragioni per cui sono state prese certe decisioni, quali alternative sono state scartate e così via.

A noi piace vedere un *semplice* livello di intestazione a livello di modulo, commenti significativi per i dati e le dichiarazioni di tipo e una breve intestazione per classe e metodo, che descriva come viene usata la funzione e tutto ciò che fa, che non sia ovvio.

I nomi delle variabili, ovviamente, devono essere scelti bene e devono essere significativi. `Foo`, per esempio, non ha senso, come `doit` o `manager` o `stuff`. La notazione ungherese (dove si codificano nel nome anche le informazioni sul tipo della variabile) è inappropriata per i sistemi a oggetti. Ricordate che voi (e altri dopo di voi) *leggerete* il codice centinaia di volte, ma lo *scriverete* solo poche volte. Datevi il tempo di scrivere `connectionPool` anziché `cp`.

Ancor peggio dei nomi senza senso sono i nomi *fuorvianti*. Avete mai dovuto chiedere a qualcuno di spiegare incoerenze in codice legacy come "la routine chiamata `getData` in realtà scrive dati su disco"? Il cervello umano fa continuamente di queste cose: si chiama *effetto Stroop* [Str35]. Potete fare voi stessi un esperimento per vedere gli effetti di questa interferenza: prendete dei pennarelli colorati e usateli per scrivere i nomi dei colori, ma non scrivete mai il nome di un colore con un pennarello di quel colore. Potete scrivere "blu" in verde, "marrone" in rosso e così via. (In alternativa, abbiamo un campione di colori già disegnati sul nostro sito web, https://pragprog.com/the-pragmatic-programmer/stroop-effect) Una volta scritti i nomi dei colori, cercate di dire ad alta voce il colore in cui è scritta ciascuna parola, il più rapidamente possibile. A un certo punto inciamperete e comincerete a dire i nomi dei colori, e non i colori. I nomi sono molto significativi per il nostro cervello, e nomi fuorvianti aggiungono confusione al codice.

Potete documentare i parametri, ma chiedetevi se sia davvero necessario in tutti i casi. Il livello di commento suggerito dallo strumento JavaDoc sembra appropriato:

```
/**
  *Find the peak (highest) value within a specified date
  *range of samples.
  *
  *@param   aRange Range of dates to search for data.
  *@param   aThreshold Minimum value to consider.
```

```
*@return  the value, or <code>null</code> if no value found
*         greater than or equal to the threshold.
*/
public Sample findPeak(DateRange aRange, double aThreshold);
```

Ecco un elenco di cose che *non* devono comparire nei commenti al codice sorgente.

- *Un elenco delle funzioni esportate dal codice nel file.* Ci sono programmi che analizzano il sorgente per voi: usateli e la lista sarà sicuramente aggiornata.
- *Cronologia delle revisioni.* A questo scopo ci sono i sistemi di controllo del codice sorgente (vedi *Controllo del codice sorgente* nel Capitolo 3). Può essere utile però includere l'informazione sulla data dell'ultimo cambiamento e su chi l'ha effettuato. (Questo tipo di informazione, così come il nome del file, è fornito dal tag `$Id$` di RCS.)
- *Un elenco degli altri file usati da questo file.* Può essere compilato con maggiore precisione da strumenti automatici.
- *Il nome del file.* Se deve comparire nel file, non gestitelo manualmente. RCS e altri sistemi simili possono mantenere aggiornata automaticamente questa informazione. Se spostate il file o ne cambiate il nome, non dovrete ricordare di modificare l'intestazione.

Una delle informazioni più importanti che *deve* comparire nel file sorgente è il nome dell'autore, non necessariamente chi ha modificato per ultimo il file, ma il proprietario. Legare al codice sorgente la responsabilità fa meraviglie per l'onestà delle persone (vedi *Orgoglio e pregiudizio*, più avanti in questo capitolo).

Il progetto può anche richiedere che in ogni file sorgente compaiano certe dichiarazioni di copyright e altre informazioni legali standard. Fate in modo che l'editor inserisca queste cose automaticamente al posto vostro.

Se esistono commenti significativi, strumenti come JavaDoc [URL7] e DOC++ [URL21] possono estrarli e formattarli in modo da produrre automaticamente documentazione a livello di API. Questo è un esempio specifico di una tecnica più generale che utilizziamo, i *documenti eseguibili*.

Documenti eseguibili

Supponiamo di avere una specifica che elenca le colonne in una tabella di database. Avremo poi un insieme separato di comandi SQL per creare la tabella effettiva nel database, e probabilmente qualche struttura di record in un linguaggio di programmazione per ospitare i contenuti di una riga nella tabella. Le stesse informazioni sono ripetute tre volte: cambiate una qualsiasi di queste tre fonti e le altre due sono subito obsolete. Questa è una chiara violazione del principio DRY.

Per eliminare il problema, dobbiamo scegliere la fonte autorevole dell'informazione: può essere la specifica, lo strumento per lo schema del database o può essere una terza fonte diversa. Scegliamo il documento di specifica come fonte: ora è il nostro *modello* per questo processo. Poi dobbiamo trovare un modo per esportare le informazioni che contiene come *viste* diverse - uno schema di database e un record in linguaggio di alto livello, per esempio. (Vedi *È solo una vista* nel Capitolo 5, per maggiori informazioni su modelli e viste.)

Se il documento è conservato come puro testo con comandi di markup (con HTML, LATEX o troff, per esempio), potete usare strumenti come Perl per estrarre lo schema e riformattarlo automaticamente. Se il documento è nel formato binario di un word processor, il riquadro seguente presenta qualche opzione.

Il vostro documento ora è parte integrante dello sviluppo del progetto. L'unico modo per cambiare lo schema è modificare il documento. Avete la garanzia che specifica, schema e codice sono fra loro allineati. Riducete al minimo la quantità di lavoro da fare per ogni cambiamento e potete aggiornare automaticamente le visualizzazioni del cambiamento.

E se il mio documento non è in puro testo?
Purtroppo, sempre più spesso i documenti di progetto sono scritti con word processor che salvano i file su disco in formato proprietario. Diciamo "purtroppo", perché questo riduce drasticamente le possibilità di elaborazione automatica del documento. Avete comunque un paio di opzioni.

- Scrivete macro. La maggior parte dei word processor più avanzati ora ha un linguaggio per le macro. Con un po' di pazienza potete programmarli in modo che esportino sezioni marcate del vostro documento nelle forme alternative che vi servono. Se programmare a questo livello è troppo noioso, potete sempre esportare la sezione opportuna in un formato di file standard di puro

testo, e poi usare uno strumento come Perl per convertire quest'ultimo nella forma finale.

- Rendete subordinato il documento. Anziché usare come fonte definitiva il documento, usate un'altra rappresentazione. (Nell'esempio del database, potreste usare lo schema come informazione autorevole.) Poi scrivete uno strumento che esporti quelle informazioni in una forma importabile nel documento. Dovete assicurarvi che le informazioni vengano importate ogni volta che il documento viene stampato, anziché una volta sola, al momento della creazione del documento.

Possiamo generare documentazione a livello di API da codice sorgente utilizzando in modo simile strumenti come JavaDoc e DOC++. Il modello è il codice sorgente: una visualizzazione del modello può essere compilata; altre dovranno essere stampate o visualizzate sul Web. Il nostro obiettivo è sempre lavorare sul modello, che il modello sia il codice stesso o qualche altro documento, e fare in modo che tutte le visualizzazioni siano aggiornate automaticamente (vedi *Automazione onnipresente*, in questo stesso capitolo, per maggiori informazioni sui processi automatici).

Di colpo, la documentazione non è poi così male.

Technical writer

Fin qui abbiamo parlato solo di documentazione interna, scritta dai programmatori stessi. Che cosa accade invece quando nel progetto sono coinvolti technical writer professionisti? Molto spesso i programmatori si limitano a lanciare "di là dal muro" dei materiali ai technical writer e lasciano che se la cavino loro a produrre manuali utente, materiali promozionali e così via.

Questo è un errore. Solo perché i programmatori non scrivono questi documenti non vuol dire che possiamo abbondonare i principi pragmatici. Vogliamo che i technical writer abbraccino gli stessi principi fondamentali a cui si attiene un programmatore pragmatico, in particolare che rispettino il principio DRY, l'ortogonalità, il concetto di modello e visualizzazione e l'uso dell'automazione e dello scripting.

O lo stampi o lo intrecci

Un problema intrinseco alla documentazione cartacea pubblicata è che è obsoleta non appena viene stampata. La documentazione in qualsiasi forma è solo un'istantanea.

Perciò cerchiamo di produrre tutta la documentazione in una forma che possa essere pubblicata online, sul Web, con tutti i suoi collegamenti ipertestuali. È più facile mantenere aggiornata questa visualizzazione della documentazione che rintracciare ogni copia cartacea esistente, bruciarla e ristampare e ridistribuire nuove copie. È anche un modo migliore per soddisfare i bisogni di un pubblico vasto. Ricordate, però, di mettere una data o un numero di versione su ogni pagina web: in questo modo il lettore può avere un'idea chiara di che cosa è aggiornato, di che cosa è stato cambiato di recente e di che cosa no.

Molte volte è necessario presentare la stessa documentazione in formati diversi: un documento a stampa, pagine web, guida online o magari uno *slide show*. La soluzione tipica fa pesantemente affidamento sul taglia-incolla, con la creazione di vari nuovi documenti dall'originale. È una cattiva idea: la presentazione di un documento deve essere indipendente dai suoi contenuti.

Se usate un sistema di marcatura, avete la flessibilità di realizzare tanti formati di output diversi quanti sono quelli che vi servono. Potete scegliere di far sì che

```
<H1>Titolo di capitolo</H1>
```

Generi un nuovo capitolo nella versione a stampa del documento e il titolo di una nuova slide in una presentazione. Tecnologie come XSL (*eXtensible Stylesheet Language*) e CSS (*Cascading Syle Sheets*), che servono proprio a separare presentazione e contenuto, si possono usare per generare più formati di output da un solo documento con marcatori.

Se usate un word processor, probabilmente non avrete a disposizione una possibilità simile. Se vi siete ricordati di usare gli stili per identificare i diversi elementi del documento, applicando fogli stile diversi potete drasticamente modificare l'aspetto del prodotto finale. La maggior parte dei word processor ora consente la conversione dei documenti in formati come HTML per la pubblicazione sul Web.

Linguaggi di marcatura

Infine, per la documentazione di progetti su grande scala, consigliamo di prendere in considerazione qualcuno degli schemi più moderni per la marcatura.

Molti tecnici ora usano DocBook per definire i loro documenti. DocBook è uno standard di marcatura basato su SGML che identifica con precisione ogni componente di un documento. Il documento può essere poi passato a un processore DSSSL per ottenerne un gran numero di formati diversi. Il progetto di documentazione di Linux usa DocBook per pubblicare informazioni in RTF, TEX, info, PostScript e HTML.

Purché la marcatura originale sia abbastanza ricca da esprimere tutti i concetti necessari (compresi i collegamenti ipertestuali), la traduzione in altre forme pubblicabili può essere facile e automatica. Potete produrre una guida online, manuali da pubblicare, schede prodotto per il sito web e anche un calendario "un suggerimento al giorno", tutto a partire dalla stessa fonte, che ovviamente è sotto controllo del sorgente ed è costruita insieme alla build notturna (vedi *Automazione onnipresente* in questo stesso capitolo).

Documentazione e codice sono visualizzazioni diverse dello stesso modello sottostante, ma la visualizzazione è *tutto* ciò che deve essere diverso. Non lasciate che la documentazione diventi un cittadino di serie B, bandito dal flusso di lavoro principale del progetto. Trattate la documentazione con la stessa cura che dedicate al codice, e gli utenti (e i manutentori che seguiranno) canteranno le vostre lodi.

Vedi anche

- *I mali della duplicazione*, Capitolo 2
- *Ortogonalità*, Capitolo 2
- *I poteri del puro testo*, Capitolo 3
- *Controllo del codice sorgente*, Capitolo 3
- *È solo una vista*, Capitolo 5
- *Programmazione per coincidenza*, Capitolo 6
- *La fossa dei requisiti*, Capitolo 7
- *Automazione onnipresente*, in questo Capitolo

Sfide

- Avete scritto un commento esplicativo per il codice sorgente che avete appena scritto? Perché no? Avete poco tempo? Non siete sicuri se il codice funzionerà davvero - stavate solo mettendo alla prova un'idea con un prototipo? Poi

butterete via il codice, giusto? Non entrerà nel progetto senza commenti e allo stadio sperimentale, vero?

- A volte mette a disagio documentare il progetto del codice sorgente perché il progetto non è ancora chiaro nella vostra mente: è ancora in evoluzione. Vi sembra che non sia giusto sprecare fatica descrivendo quel che fa qualcosa fino a che non lo fa veramente. Non vi fa venire in mente la programmazione per coincidenza (Capitolo 6)?

Grandi speranze

> Stupitene, o cieli; inorridite come non mai.
> - Geremia, 2:12

Un'azienda annuncia profitti record e le sue azioni scendono del 20 per cento. Il notiziario finanziario la sera spiega che l'azienda non ha soddisfatto le aspettative degli analisti. Una bambina apre un costoso regalo di Natale e scoppia in lacrime - non è la bambola economica che sperava di avere. Un team di progetto fa miracoli per realizzare un'applicazione di complessità fenomenale solo per vederla snobbata dai suoi utenti perché non ha un sistema di guida in linea.

In un senso astratto, un'applicazione ha successo se realizza correttamente le sue specifiche; purtroppo, però, questo paga solo le bollette astratte.

In realtà, il successo di un progetto si misura da quanto soddisfa le *aspettative* dei suoi utenti. Un progetto che rimane al di sotto delle loro aspettative è considerato un fallimento, non importa quanto sia buono in termini assoluti. Però, come il genitore della bambina che si aspettava una bambola da quattro soldi, andate troppo in là e anche voi sarete un fallimento.

SUGGERIMENTO 69
Superate di poco le aspettative dei vostri utenti.

L'esecuzione di questo suggerimento però richiede un po' di lavoro.

Comunicare aspettative

Gli utenti inizialmente arrivano da voi con una certa idea di ciò che vogliono. Magari sarà incompleta, incoerente o tecnicamente impossibile,

ma è la *loro* idea e, come la bambina a Natale, hanno investito parecchie emozioni che non potete semplicemente ignorare.

Quando la vostra comprensione dei loro bisogni si sviluppa, troverete aree in cui le loro aspettative non possono essere soddisfatte, altre in cui sono fin troppo prudenti. Fa parte del vostro ruolo comunicarlo. Lavorate con i vostri utenti in modo che la loro comprensione di quello che realizzerete sia precisa. E fatelo per tutto il processo di sviluppo. Non perdete mai di vista i problemi di business che la vostra applicazione deve risolvere.

Qualche consulente chiama questo processo "gestione delle aspettative": controllare attivamente quello che gli utenti devono sperare di ottenere dai loro sistemi. Pensiamo che sia una posizione un po' elitaria: il nostro ruolo non è controllare le speranze degli utenti; dobbiamo invece lavorare con loro per arrivare a una comprensione comune del processo di sviluppo e del prodotto finale, comprese quelle aspettative che non hanno ancora espresso verbalmente. Se il team comunica correttamente con il mondo esterno, il processo è quasi automatico: tutti dovrebbero capire che cosa ci si aspetta e come verrà realizzato.

Esistono alcune tecniche importanti che si possono usare per agevolare questo processo: i *Proiettili traccianti* (Capitolo 2) e *Prototipi e Post-it* (Capitolo 2) sono le più importanti. Entrambe consentono al team di costruire qualcosa che l'utente può vedere; sono modi ideali di comunicare quello che avete capito dei loro requisiti; entrambe infine consentono a voi e a vostri utenti di esercitarvi a comunicare gli uni con gli altri.

Il miglio extra

Se lavorate a stretto contatto con i vostri utenti, condividendo le loro aspettative e comunicando quello che fate, non ci saranno molte sorprese quando il progetto viene consegnato.

Ma questa è una CATTIVA COSA. Cercate di sorprendere i vostri utenti. Non spaventateli, badate bene, ma *deliziateli*.

Date loro qualcosa più di quello che si aspettavano. Quel minimo di fatica in più necessaria per aggiungere al sistema qualche caratteristica

orientata all'utente sarà abbondantemente ripagata molte volte in benevolenza.

Ascoltate i vostri utenti mentre il progetto procede, per carpire indizi di quali caratteristiche li delizierebbero davvero. Cose che potete aggiungere con relativa facilità e che mediamente sono apprezzate in media dagli utenti sono:

- aiuto a fumetti o con ToolTip;
- scorciatoie da tastiera;
- una guida di consultazione rapida a complemento del manuale utente;
- colori;
- analizzatori di file di log;
- installazione automatica;
- strumenti per controllare l'integrità del sistema;
- la possibilità di eseguire più versioni del sistema per l'addestramento;
- una schermata iniziale personalizzata per la loro organizzazione.

Sono tutte cose relativamente superficiali e non sovraccaricano realmente il sistema con un eccesso di caratteristiche. Tutte però dicono ai vostri utenti che il team di sviluppo aveva a cuore la produzione di un ottimo sistema, pensato per essere davvero usato. Ricordate solo di non introdurre errori nel sistema aggiungendo queste nuove caratteristiche.

Vedi anche

- *Software abbastanza buono*, Capitolo 1
- *Proiettili traccianti*, Capitolo 2
- *Prototipi e Post-it*, Capitolo 2
- *La fossa dei requisiti*, Capitolo 7

Sfide

- A volte i critici più severi di un progetto sono le persone che ci hanno lavorato. Vi è mai capitato di restare delusi perché qualcosa che avevate prodotto non era all'altezza delle vostre aspettative? Com'è possibile? Forse qui non si tratta solo di logica.

- Su che cosa fanno commenti i vostri utenti quando consegnate il vostro software? La loro attenzione è rivolta alle varie aree dell'applicazione in proporzione alla fatica che avete investito in ciascuna di esse? Che cosa li delizia?

Orgoglio e pregiudizio

Ci hai deliziato abbastanza a lungo.
- Jane Austen, *Orgoglio e pregiudizio*

I programmatori pragmatici non sfuggono alle responsabilità: anzi, sono felici di accettare le sfide e di rendere ben nota la loro competenza. Se siamo responsabili di un progetto, o di un pezzo di codice, facciamo un lavoro di cui possiamo essere orgogliosi.

SUGGERIMENTO 70
Firmate il vostro lavoro.

Gli artigiani di un tempo erano orgogliosi di firmare il loro lavoro, e anche voi dovreste esserlo.

I team di progetto sono ancora fatti di persone, però, e questa regola può essere fonte di guai. In qualche progetto, l'idea di *proprietà del codice* può causare problemi di cooperazione: le persone diventano territoriali o non disponibili a lavorare sugli elementi delle fondamenta comuni. Il progetto può finire come una schiera di piccoli regni isolati. Si genera il pregiudizio a favore del proprio codice e contro gli altri colleghi.

Non è quello che vogliamo. Non dovete difendere gelosamente il vostro codice contro gli intrusi; al tempo stesso, dovete trattare con rispetto il codice degli altri. La Regola aurea ("Fai agli altri quello che vorresti facessero a te") e una base di mutuo rispetto fra gli sviluppatori sono vitali per far funzionare questo suggerimento.

L'anonimato, in particolare nei grandi progetti, può favorire un terreno di coltura per i germi della trascuratezza, gli errori, la pigrizia e il codice scritto male. Diventa troppo facile vedersi come una semplice rotella in un grande meccanismo e avanzare scuse patetiche in infiniti report di stato, anziché contribuire con buon codice.

Il codice deve essere di proprietà di qualcuno, ma non necessariamente di un individuo. In effetti, un metodo di successo come l'eXtreme

Programming di Kent Beck [URL45] consiglia la proprietà condivisa del codice (ma questo richiede anche altre pratiche, come la programmazione in coppia, per proteggersi contro i pericoli dell'anonimato).

Vogliamo vedere l'orgoglio della proprietà: "L'ho scritto io e mi prendo la paternità del mio lavoro". La vostra firma deve essere riconosciuta come un indicatore di qualità. Le persone devono vedere il vostro nome su un pezzo di codice e immaginare subito che sia solido, ben scritto, collaudato e documentato. Un vero lavoro da professionisti, scritto da un vero professionista.

Un pragmatic programmer.

Risorse

Siamo riusciti a parlare di così tante cose, in questo libro, solo perché abbiamo esaminato i nostri temi da una grande altezza. Se avessimo dovuto trattarli nel modo che avrebbero meritato, il libro sarebbe stato lungo dieci volte di più.

Abbiamo iniziato il libro proponendo l'idea che i programmatori pragmatici non devono mai smettere di imparare. In questa appendice elenchiamo risorse che possono esservi di aiuto su questo fronte.

Nella sezione *Associazioni professionali*, diamo le indicazioni relative all'IEEE e all'ACM. Consigliamo ai programmatori pragmatici di associarsi a una (o a entrambe). Poi, in *Costruirsi una biblioteca*, presentiamo periodici, libri e siti web che contengano informazioni pertinenti e di alta qualità (o che sono semplicemente divertenti).

In tutto il libro abbiamo accennato a molte risorse accessibili via Internet. Nella sezione *Risorse Internet*, elenchiamo gli URL di quelle risorse, con una breve descrizione di ciascuna. La natura del Web però è tale che molti di questi potrebbero essere cambiati nel momento in cui leggerete il libro. Potete ricorrere a uno dei molti motori di ricerca per trovare link più aggiornati, oppure visitare il nostro sito web e consultare la nostra sezione di link.

Infine troverete la bibliografia del libro.

Associazioni professionali

Esistono due associazioni professionali di livello mondiale per i programmatori:.

- *Association for Computing Machinery* (ACM, www.acm.org, ACM Member Services, PO Box 11414, New York, NY 10286, USA);
- *IEEE Computer Society* (www.computer.org, 1730 Massachusetts Avenue NW, Washington, DC 20036-1992, USA.

Consigliamo a tutti i programmatori di iscriversi a una o a entrambe. Chi lavora al di fuori degli Stati Uniti può anche iscriversi alla propria associazione nazionale.

Far parte dei una associazione professionale dà molti vantaggi. I convegni e gli incontri locali danno ottime occasioni per incontrare persone con interessi simili, i gruppi di interesse speciali e i comitati tecnici danno la possibilità di partecipare alla definizione di standard e direttive. Otterrete anche molte delle loro pubblicazioni, da discussioni di alto livello sulle pratiche del settore alla teoria informatica di basso livello.

Costruirsi una biblioteca

Leggiamo molto. Come abbiamo notato in *Il portafoglio delle conoscenze* (Capitolo 1), un buon programmatore non smette mai di imparare. Restare aggiornati con libri e periodici può essere di aiuto. Qui ne indichiamo alcuni che abbiamo trovato particolarmente interessanti.

Periodici

Se siete come noi, accumulerete vecchie riviste e periodici finché le pile non sono tanto alte da far diventare i fascicoli che stanno in fondo sottili come carta velina. Questo significa che conviene essere selettivi. Ecco alcuni dei periodici che noi leggiamo.

- *IEEE Computer*. Inviato ai membri della IEEE Computer Society, *Computer* è orientato prevalentemente alla pratica, ma non disdegna la teoria. Alcuni fascicoli sono dedicati a un tema specifico, altri sono raccolte di articoli interessanti. [All'indirizzo

www.computer.org/web/publications/magazines si trova l'elenco di tutti i periodici pubblicati dalla IEEE.]

- *IEEE Software.* Un'altra ottima pubblicazione bimestrale della IEEE Computer Society per chi si occupa di software.
- *Communications of the ACM.* La rivista che ricevono tutti i membri dell'ACM. *CACM* è da decenni lo standard del settore e probabilmente ha pubblicato più articoli fondamentali di qualsiasi altra fonte.
- *SIGPLAN.* Prodotta dallo ACM Special Interest Group on Programming Languages, *SIGPLAN* è ottenibile opzionalmente con l'iscrizione alla ACM. Spesso è utilizzata per pubblicare le specifiche dei linguaggi, oltre ad articoli che possono interessare chiunque voglia approfondire la programmazione.
- *Dr. Dobbs Journal.* Rivista mensile, ha cessato le pubblicazioni alla fine del 2014. L'archivio completo è disponibile online, sul sito www.drdobbs.com. Proponeva "buoni materiali per sviluppatori seri", articoli di attualità ma soprattutto sulle tecniche di programmazione.
- *The Perl Journal.* Rivista dedicata al Perl, ha cessato le pubblicazioni (www.tpj.com).
- *Software Development Magazine.* Rivista mensile dedicata ai temi generali di gestione di progetti e sviluppo software. Si è fusa con il *Dr. Dobbs Journal* nel 2006 e ne ha seguito la sorte.

Settimanali

Esistono vari settimanali per gli sviluppatori e i loro manager. Sono in gran parte una raccolta di comunicati stampa, riconfezionati in forma di articoli. I contenuti sono comunque apprezzabili: permettono di sapere che cosa succede, di rimanere aggiornati sui nuovi prodotti e di seguire le vicende delle aziende del settore. Non aspettatevi grandi approfondimenti tecnici, però.

Libri

I libri di informatica sono spesso costosi, ma se li scegliete con cura sono un investimento prezioso. Potete trovare il catalogo dei nostri libri

sul nostro sito. Esistono però anche molti altri libri che abbiamo trovato utili e interessanti.

Analisi e progettazione

- *Object-Oriented Software Construction. Seconda edizione.* Il testo epico di Bertrand Meyer sui fondamenti dello sviluppo a oggetti: tutto in circa 1300 pagine [MEY97b].
- *Design Patterns.* Un *design pattern*, ovvero uno schema progettuale, descrive un modo per risolvere una particolare classe di problemi a un livello più alto rispetto al linguaggio di programmazione. Questo libro ormai classico [GHJV95] della *Gang of Four* descrive 23 schemi progettuali, fra cui Proxy, Visitor e Singleton.
- *Analysis Patterns.* Una ricca raccolta di schemi di architettura di alto livello, ricavati da molti progetti reali e distillati in forma di libro. Un modo relativamente rapido per apprendere le idee di molti anni di esperienza nella modellazione [Fow96].

Team e progetti

- *The Mythical Man Month.* Classico testo di Fred Brooks sui problem dell'organizzazione dei team di progetto [Bro95].
- *Dynamics of Software Development.* Una serie di brevi saggi sulla costruzione di software in grandi team. Si concentra sulle dinamiche fra i membri del team e fra il team e il resto del mondo [McC95].
- *Surviving Object-Oriented Projects: A Manager's Guide.* I "diari dal fronte" di Alistair Cockburn illustrano molti dei pericolic e delle trappole della gestione di un progetto OO, in modo particolare se è il primo del genere che affrontate. Cockburn preenta suggerimenti e tecniche per affrontare i problem più comuni [Coc97b].

Ambienti specifici

- Unix. W. Richard Stevens ha scritto vari libri eccellenti, fra cui *Advanced Programming in the Unix Environment* e la serie *Unix Network Programming* [Ste92, Ste98, Ste99].
- *Windows. Win32 System Services* di Marshall Brain [Bra95] è una guida di riferimento compatta alle API di basso livello. *Programming Windows* di Charles Petzold [Pet98] è la bibbia per lo sviluppo di GUI Windows.
- C++. Se vi ritrovate in un progetto in C++, andate di corsa in libreria e trovate una copia di *Effective C++*, e magari anche di *More Effective C++* di Scott Meyer [Mey97a, Mey96]. Per costruire sistemi di dimensioni notevoli, vi serve *Large-Scale C++ Software Design* di John Lakos [Lak96]. Per le tecniche avanzate, *Advanced C++ Programming Styles and Idioms* di Jim Caplien [Cop92].

Inoltre, la serie *Nutshell* di O'Reilly (www.ora.com) offre volumi brevi e compatti ma esaurienti su vari temi e vari linguaggi, come perl, yacc, sendmail, il funzionamento di Windows, le espressioni regolari.

Il Web

Trovare buoni contenuti sul Web è difficile. Ecco alcuni siti che noi visitiamo almeno una volta alla settimana.

- Slashdot (slashdot.org). Presentato come "Notizie per nerd. Cose importanti", Slashdot è uno dei punti di incontro della comunità Linux. Oltre a notizie e aggiornamenti regolari su Linux, il sito offre informazioni su tecnologie "calde" e su problemi che riguardano gli sviluppatori.
- WikiWikiWeb (www.c2.com/ppr/). Il Portland Pattern Repository, con discussioni sugli schemi. Non è solo una grande risorsa: è anche un esperimento interessante di redazione collettiva di idee.

Risorse Internet

Quelli che seguono sono indirizzi a risorse disponibili in Internet. Erano validi al momento in cui abbiamo scritto, ma, conoscendo come vanno le cose in Rete, potrebbero anche essere obsoleti nel momento in cui leggete queste pagine. [Abbiamo comunque cercato di segnalare criticità e alternative quando possibile, *NdR*.]

Editor

Emacs e vi non solo sono editor cross-platform, ma sono anche disponibili liberamente e ampiamente utilizzati. Una rapida scorsa a una rivista come *Dr. Dobbs* vi permetterà di scoprire numerose alternative commerciali.

Emacs

Sia Emacs che XEmacs sono disponibili per piattaforme Unix e Windows.

[URL1] L'editor Emacs (www.gnu.org). Il non plus ultra dei grandi editor, contiene tutte le caratteristiche che si possono desiderare in un editor. Ha una curva di apprendimento quasi verticale, ma l'impegno viene ripagato abbondantemente. Costituisce anche un ottimo lettore di mail e news, una rubrica, un calendario con agenda, un gioco d'avventura.

[URL2] L'editor XEmacs (www.xemacs.org). Nato dall'Emacs originale, ha fama di una costruzione interna più pulita e ha un'interfaccia più elegante.

Vi

Esistono almeno 15 diversi cloni di vi. Fra questi, vim è probabilmente quello presente sulla maggior parte delle piattaforme, perciò può essere una buona scelta se vi accade di lavorare in ambienti diversi.

[URL3] L'editor vim (www.vim.org). Dalla documentazione: "Esistono molti miglioramenti rispetto a vi: undo a molti livelli, finestre e buffer, evidenziazione della sintassi, editing da riga di comando, completamento dei nomi di file, guida in linea, selezione visiva ecc.".

[URL4] L'editor elvis (elvis.the-little-red-haired-girl.org). Un clone migliorato, con supporto per X.

[URL5] Emacs Viper Mode (www.cs.sunysb.edu/~kifer/emacs.html). Viper è un insieme di macro che rendono Emacs più simile a vi. Qualcuno dubita che sia sensato prendere l'editor più grande del mondo ed estenderlo per fargli emulare un editor il cui punto di forza è invece la compattezza. Altri dicono invece che così si ha il meglio di entrambi i mondi.

Compilatori, linguaggi e strumenti di sviluppo

[URL6] Il compilatore GNU C/C++ (gcc.gnu.org). Uno dei compilatori C e C++ più diffusi. Serve anche per Objective C.

[URL7] Il linguaggio Java (della Sun, oggi Oracle) (www.oracle.com/it/java/index.html). Il sito dedicato a Java: vi si trovano SDK, documentazione, tutorial, notizie e altro ancora.

[URL8] Perl Language Home Page (www.perl.com). Articoli e notizie sulla programmazione in Perl. Per il linguaggio, www.perl.org.

[URL9] Python (www.python.org). Python, linguaggio per la programmazione a oggetti e interpretato e interattivo, con una sintassi leggermente strana e un seguito ampio e fedele.

[URL10] Smart Eiffel (SmartEiffel.loria.fr). Il compilatore GNU Eiffel (inizialmente chiamato Small Eiffel, ora Smart Eiffel) gira su tutte le macchine che hanno un compilato C ANSI e un ambiente di runtime Posix.

[URL11] ISE Eiffel (www.eiffel.com). La Interactive Software Engineering è l'azienda da cui ha avuto origine la progettazione per contratto (Design by Contract), e vende un compilatore Eiffel commerciale e altri strumenti collegati.

[URL12] Sather (www.icsi.berkeley.edu/~sather). Sather è un linguaggio sperimentale derivato da Eiffel. Ha lo scopo di supportare funzioni di ordine superiore e l'astrazione dell'iterazione così come Common Lisp, CLU o Scheme, e di essere efficiente quanto C, C++ o Fortran. [Lo sviluppo + stato trasferito alla GNU, lo si può rovare ora a www.gnu.org/software/sahter/.]

[URL13] VisualWorks (www.cincomsmalltalk.com). Il sito dell'ambiente VisualWorks Smalltalk. Sono disponibili gratuitamente, per uso personale, versioni non commerciali per Windows e Linux.

[URL14] L'ambiente del linguaggio Squeak (squeak.org). Squeak è un'implementazione libera e portabile di Smalltalk-80 scritta in Squeak; può produrre codice C per prestazioni più elevate.

[URL15] Il linguaggio di programmazione TOM (www.gerbil.org/tom). Un linguaggio molto dinamico che ha le sue radici nell'Objective-C. [Lo sviluppo di TOM è cessato da tempo.]

[URL16] Il progetto Beowulf (www.beowulf.org). Un progetto per la costruzione di computer ad alte prestazioni da cluster di macchine Linux economiche in rete.

[URL17] iContract - strumento di progettazione per contratto per Java (www.reliable-systems.com). Il formalismo di precondizioni, postcondizioni e invianti della progettazione per contratto, implementato come preprocessore per Java. Rispetta l'eredità, implementa quantificatori esistenziali e altro ancora.

[URL18] Nana: log e asserzioni per C e C++ (savannah.gnu.org/nana/). Supporto migliorato per il controllo delle asserzioni e i log in C e C++. Offre anche un certo supporto per la progettazione per contratto.

[URL19] DDD: Data Display Debugger (www.gnu.org/software/ddd/). Un front- end grafico libero per debugger Unix.

[URL20] Il Refactoring Browser (www.refactory.com/tools/refactoring-browser) di John Brant. Un browser di refactoring molto diffuso per Smalltalk.

[URL21] DOC++, generatore di documentazione (docpp.sourceforge.net). DOC++ è un sistema di documentazione per C/C++ e Java che genera output LATEX e HTML per la consultazione online della documentazione direttamente dall'intestazione C++ o da file di classi Java. [Lo sviluppo di questo sistema risulta interrotto.]

[URL22] xUnit, framework per unit test (www.XProgramming.com). Idea semplice ma potente, il framework per unit test xUnit offre una piattaforma coerente per sottoporre a test software scritto in vari linguaggi. [Non più attivo, ma vedi la voce "XUnit" in Wikipedia per un elenco di framework.]

[URL23] Il linguaggio Tcl (www.tcl.tk). Tcl (Tool Command Language) è un linguaggio di scripting che si può incorporare facilmente in un'applicazione).

[URL24] Expect: automazione dell'interazione con programmi (expect.sourceforge.net). Estensione costruita su Tcl, expect permette lo

scripting di interazioni con programmi. Oltre ad aiutare a scrivere file di comandi che (per esempio) recuperano file da server remoti o estendono le possibilità della shell, expect può essere utile quando si eseguono test di regressione. Una versione grafica, expectk, permette di dotare di un front-end grafico applicazioni non GUI.

[URL25] T Spaces (www.almaden.ibm.com/cs/TSpaces). Dalla pagina web: "T Spaces è un buffer di comunicazione di rete con capacità di database. Consente le comunicazioni fra applicazioni e dispositivi in una rete di computer e sistemi operativi eterogenei. T Spaces offre servizi di comunicazione di gruppo, servizi di database, servizi di trasferimento file basati su URL e servizi di notifica di eventi.

[URL26] javaCC - *Java Compiler-Compiler* (javacc.org). Un generatore di parser strettamente legato al linguaggio Java.

URL 27] Bison, generatore di parser (www.gnu.org/software/bison/). Bison prende in input le specifiche di una grammatica e da queste genera il codice sorgente C di un analizzatore sintattico adeguato.

[URL28] SWIG: *Simplified Wrapper and Interface* Generator (www.swig.org). SWIG è uno strumento di sviluppo software che connette programmi scritti in C, C++ e Objective-C con tutta una serie di linguaggi di programmazione di alto livello come Perl, Python e Tcl/Tk, nonché con Java, Eiffel e Guile.

[URL29] The *Object Management Group*, Inc. (www.omg.org). L'OMG è alfiere di varie specifiche per la produzione di sistemi a oggetti distribuiti. Fra i loro lavori rientrano la *Common Object Request Broker Architecture* (CORBA) e lo *Internet Inter-ORB* Protocol (IIOP). Insieme, queste specifiche consentono agli oggetti di comunicare fra loro, anche se sono scritti in linguaggi differenti e se sono eseguiti su computer di tipo diverso.

Strumenti Unix sotto DOS

[URL30] The UWIN Development Tools, Global Technologies, Inc., Old Bridge, NJ (www.gtlinc.com/uwin.html). Il pacchetto UWIN fornisce librerie dinamiche (DLL, *Dynamic Link Libraries*) per Windows che emulano una buona parte dell'interfaccia di libreria a livello di Unix C. Con questa interfaccia, la GTL ha portato in Windows un gran numero di

strumenti Unix da riga di comando. Vedi anche [URL31]. [Lo sviluppo di questo pacchetto appare cessato.]

[URL31] Cygwin (www.cygwin.com). Il pacchetto (sviluppato in origine dalla Cygnus, poi acquisita da Red Hat) emula l'interfaccia di libreria Unix C e offre un gran numero di strumenti Unix da riga di comando sotto il sistema operativo Windows.

[URL32] Perl Power Tools (www.perlpowertools.com) Un progetto per reimplementare l'insieme dei comandi classici di Unix in Perl, rendendoli disponibili a tutte le piattaforme che supportano Perl (e sono davvero tante). [Il progetto originale di Tom Christiansen è stato interrotto, ripreso poi nel 2014 da Brian DFoy.]

Strumenti di controllo del codice sorgente

[URL33] RCS: *Revision Control System* (www.gnu.org/software/rcs/). Sistema di controllo del codice sorgente GNU per Unix e Windows NT.

[URL34] CVS: *Concurrent Version System* (www.nongnu.org/cvs/). Sistema di controllo del codice sorgente, liberamente disponibile, per Unix e Windows NT. Estende RCS supportando un modello client-server e l'accesso concorrente ai file.

[URL35] Aegis Transaction-Based Configuration Management (aegis.sourceforge.net/). Uno strumento di controllo delle revisioni, orientate al processo, che impone standard di progetto (come la verifica che il codice in ingresso superi i test).

[URL36] ClearCase (www-03.ibm.com/software/products/it/clearcase). Controllo delle versioni, gestione dello spazio di lavoro e delle build, controllo di processo. [Sviluppato dalla Rational Software, poi acquisita dalla IBM.]

[URL37] MKS Source Integrity. Controllo delle versioni e gestione di configurazione. Alcune versioni incorporano caratteristiche che consentono agli sviluppatori remoti di lavorare simultaneamente sullo stesso file (in modo molto simile a CVS). [Originariamente sviluppato dalla MKS, acquisita nel 2011 dalla PTC. Il pacchetto ora è disponibile commercialmente con il nome di PTC Product Lifecycle Management, www.ptc.com/en/products/plm.]

[URL38] PVCS Configuration Management (www.microfocus.com/products/pvcs/). Un sistema di controllo del codice

sorgente, molto diffuso per sistemi Windows. [Ha avuto una storia abbastanza complessa: passato attraverso varie aziende, è approdato alla Micro Focus nel 2016.]

[URL39] Visual SourceSafe (www.microsoft.com). Un sistema di controllo delle versioni che si integra con gli strumenti di sviluppo visuale della Microsoft.

[URL40] Perforce (www.perforce.com). Un sistema di gestione della configurazione di software client-server. [Il sistema ora ha preso il nome di Helix, sempre prodotto dalla Perforce.]

Altri strumenti

[URL41] WinZip: Archive Utility for Windows, Nico Mak Computing, Inc., Mansfield, CT (www.winzip.com). Una utility di compressione di file per Windows. Supporta entrambi i formati ZIP e TAR.

[URL42] The Z Shell (www.zsh.org/). Una shell per l'uso interattivo, anche se è anche un potente linguaggio di scripting. In zsh sono state incorporate molte caratteristiche utili di bash, ksh e tcsh, a cui sono state aggiunte molte funzioni originali.

[URL 43] SMB Client per sistemi Unix (www.samba.org/). Samba permette di condividere file e altre risorse fra sistemi Unix e Windows.

Articoli e pubblicazioni

[URL44] The comp.object FAQ (www.faqs.org/faqs/by-newsgroup/comp/comp.object.html). FAQ ampie e ben organizzate per il newsgroup comp.object. [Da tempo non aggiornato.]

[URL45] eXtreme Programming (www.extremeprogramming.org/). Citiamo dal sito: Molte delle pratiche di XP sono state create e collaudate nell'ambito del progetto Chrysler-C3, un sistema di paghe di grande successo, implementato in Smalltalk. [L'eXtreme Programming rientra fra le tecnologie di sviluppo agile: vedi in proposito it.wikipedia.org/wiki/Metodologia_agile.]

[URL46] Home page di Alistair Cockburn (alistair.cockburn.us/). Cercate "Structuring Use Cases with Goals" e usate i modelli per i casi d'uso.

[URL47] Home page di Martin Fowler (martinfowler.com/). Fowler è l'utore di *UML Distilled e Refactoring: Improving the Design of Existing Code.* Nel suo sito discute i temi dei suoi libri e il suo lavoro con UML.

[URL48] Home page di Robert C. Martin (sites.google.com/site/unclebobconsultingllc/ vedi anche blog.cleancoder.com/). Materiali vari sulla programmazione.

[URL49] *Aspect-Oriented Programming* (www.parc.xerox.com/csl/projects/aop/). Un metodo per aggiungere funzionalità al codice, ortogonale e dichiarativo. [La pagina non risulta più attiva. Consultate en.wikipedia.org/wiki/Aspect-oriented_programming per maggiori informazioni.]

[URL50] "Getting started with JavaSpaces technology" (www.oracle.com/technetwork/articles/javase/javaspaces-140665.html). Un articolo introduttivo a JavaSpaces, un sistema simile a Linda per Java, che supporta persistenza distribuita e algoritmi distribuiti.

[URL51] Netscape Source Code (www.mozilla.org). Il sorgente di sviluppo del browser Netscape (ora Firefox).

[URL52] The Jargon File (www.catb.org/~esr/jargon/). Di Eric. S. Raymond, contiene definizioni di molti termini comuni (e non tanto comuni) in campo informatico, nonché una buona dose di folklore. [Non risulta più aggiornato dal 2003.]

[URL53] Saggi di Eric S. Raymond (catb.org/~esr/). I saggi di Eric Raymond, *The Cathedral and the Bazaar* e *Homesteading the Noosphere*, che descrivono le basi psicosociali e le implicazioni del movimento Open Source.

[URL54] The K Desktop Environment (www.kde.org/). Citiamo dal sito: "KDE è un potente ambiente grafico desktop per stazioni di lavoro Unix. KDE è un progetto Internet ed è veramente aperto in ogni senso".

[URL55] *The GNU Image Manipulation Program* (www.gimp.org/). Gimp è un programma distribuito liberamente utilizzato per la creazione di immagini, la composizione e il ritocco.

[URL56] The Demeter Project (Progetto Demetra, www2.ccs.neu.edu/research/demeter/). Ricerche su come rendere il software più facile da manutenere e farlo evolvere mediante l'Adaptive Programming.

Vari

[URL57] The GNU Project (www.gnu.org/). La Free Software Foundation (Boston, MA) è una fondazione senza fini di lucro che raccoglie fondi per il progetto GNU. Obiettivo del progetto GNU è produrre un sistema di tipo Unix completo e libero. Molti degli strumenti che sono stati sviluppati lungo la strada sono diventati standard del settore.

[URL58] Web Server Survey (news.netcraft.com/archives/category/web-server-survey/). Indagini mensili condotte dalla Netcraft (una società inglese) sulla diffusione dei principali web server.

Bibliografia

[Bak72] F. T. Baker, "Chief programmer team management of production programming", in *IBM Systems Journal*, 11(1), pp. 56-73, 1972.

[BBM96] V. Basili, L. Briand, W. L. Melo, "A validation of object-oriented design metrics as quality indicators", in *IEEE Transactions on Software Engineering*, 22(10), pp. 751-761, ottobre 1996.

[Ber96] Albert J. Bernstein, *Dinosaur Brains: Dealing with All Those Impossible People at Work*, Ballantine Books, New York, NY, 1996.

[Bra95] Marshall Brain, *Win32 System Services, Prentice Hall*, Englewood Cliffs, NJ, 1995.

[Bro95] Frederick P. Brooks, Jr., *The Mythical Man Month: Essays on Software Engineering,* Addison-Wesley, Reading, MA, anniversary edition, 1995.

[CG90] N. Carriero, D. Gelenter. *How to Write Parallel Programs: A First Course,* MIT Press, Cambridge, MA, 1990.

[Cla04] Mike Clark, *Pragmatic Project Automation, The Pragmatic Programmers,* LLC, Raleigh, NC e Dallas, TX, 2004.

[CN91] Brad J. Cox, Andrex J. Novobilski, *Object-Oriented Programming, An Evolutionary Approach*, Addison-Wesley, Reading, MA, 1991.

[Coc97a] Alistair Cockburn, "Goals and use cases", in *Journal of Object Oriented Programming*, 9(7):35-40, settembre 1997.

[Coc97b] Alistair Cockburn, *Surviving Object-Oriented Projects: A Manager's Guide*, Addison Wesley Longman, Reading, MA, 1997.

[Cop92] James O. Coplien, *Advanced C++ Programming Styles and Idioms*, Addison-Wesley, Reading, MA, 1992.

[DL99] Tom Demarco, Timothy Lister, *Peopleware: Productive Projects and Teams*, Dorset House, New York, NY, seconda edizione, 1999.

[FBB+99] Martin Fowler, Kent Beck, John Brant, William Opdyke, Don Roberts, *Refactoring: Improving the Design of Existing Code*, Addison Wesley Longman, Reading, MA, 1999.

[Fow96] Martin Fowler, *Analysis Patterns: Reusable Object Models*, Addison Wesley Longman, Reading, MA, 1996.

[FS99] Martin Fowler, Kendall Scott, *UML Distilled: Applying the Standard Object Modeling Language*, Addison Wesley Longman, Reading, MA, seconda edizione, 1999.

[GHJV95] Erich Gamma, Richard Helm, Ralph Johnson, John Vlissides, *Design Patterns: Elements of Reusable Object-Oriented Software*, Addison-Wesley, Reading, MA, 1995.

[Gla99a] Robert L. Glass, "Inspections - Some surprising findings", in *Communications of the ACM*, 42(4), pp.17-19, aprile 1999.

[Gla99b] Robert L. Glass, "The realities of software technology payoffs", in *Communications of the ACM*, 42(2), pp. 74-79, febbraio 1999.

[Hol78] Michael Holt, *Math Puzzles and Games,* Dorset Press, New York, NY, 1978.

[HT03] Andy Hunt, Dave Thomas. *Pragmatic Unit Testing In Java with Junit,* The Pragmatic Programmers, LLC, Raleigh, NC, e Dallas, TX, 2003.

[Jac94] Ivar Jacobson, *Object-Oriented Software Engineering: A Use-Case Driven Approach,* Addison-Wesley, Reading, MA, 1994.

[KLM+97] Gregor Kiczales, John Lamping, Anurag Mendhekar, Chris Maeda, Cristina Videira Lopes, Jean-Marc Loingtier, John Irwin. "Aspect-oriented programming", in *European Conference on Object-Oriented Programming (ECOOP)*, volume LNCS 1241, Springer-Verlag, giugno 1997.

[Knu97a] Donald Ervin Knuth, *The Art of Computer Programming: Fundamental Algorithms, volume 1*, Addison Wesley Longman, Reading, MA, terza edizione, 1997.

[Knu97b] Donald Ervin Knuth, *The Art of Computer Programming: Seminumerical Algorithms, volume 2*, Addison Wesley Longman, Reading, MA, terza edizione, 1997.

[Knu98] Donald Ervin Knuth, *The Art of Computer Programming: Sorting and Searching, volume 3,* Addison Wesley Longman, Reading, MA, seconda edizione, 1998.

[KP99] Brian W. Kernighan, Rob Pike, *The Practice of Programming, Addison Wesley Longman, Reading,* MA, 1999 [tr. it. *Programmazione nella pratica*, Addison Wesley, Milano, 2000].

[Kru98] Philippe Kruchten, *The Rational Unified Process: An Introduction*, Addison Wesley Longman, Reading, MA, 1998.

[Lak96] John Lakos, *Large-Scale C++ Software Design*, Addison Wesley Longman, Reading, MA, 1996 [dello stesso autore è annunciato per il 2018 un *Large-Scale C++* in più volumi, presso lo stesso editore].

[LH89] Karl J. Lieberherr, Ian Holland, "Assuring good style for object-oriented programs", in *IEEE Software, pp 38-48, settembre1989.*

[Lis88] Barbara Liskov, "Data abstraction and hierarchy", in *SIGPLAN Notices*, 23(5), maggio 1988.

[LMB92] John R. Levine, Tony Mason, Doug Brown. *Lex and Yacc*, O'Reilly & Associates, Inc., Sebastopol, CA, seconda edizione, 1992.

[McC95] Jim McCarthy, *Dynamics of Software Development*, Microsoft Press, Redmond, WA, 1995.

[Mey96] Scott Meyers, *More Effective C++: 35 New Ways to Improve Your Programs and Designs,* Addison-Wesley, Reading, MA, 1996.

[Mey97a] Scott Meyers, *Effective C++: 50 Specific Ways to Improve Your Programs and Designs,* Addison Wesley Longman, Reading, MA, seconda edizione, 1997 [terza edizione 2005].

[Mey97b] Bertrand Meyer, *Object-Oriented Software Construction,* Prentice Hall, Englewood Cliffs, NJ, seconda edizione, 1997.

[Pet98] Charles Petzold, *Programming Windows, The Definitive Guide to the Win32 API,* Microsoft Press, Redmond, WA, quinta edizione, 1998 [sesta edizione 2012, aggiornata a Windows 8, con sottotitolo modificato].

[Sch95] Bruce Schneier, *Applied Cryptography: Protocols, Algorithms, and Source Code in C,* John Wiley & Sons, New York, NY, seconda edizione, 1995.

[Sed83] Robert Sedgewick, *Algorithms*, Addison-Wesley, Reading, MA, 1983 [quarta edizione, con coautore Robert Kevin Wayne, pubblicata nel 2011].

[Sed92] Robert Sedgewick. *Algorithms in C++.* Addison-Wesley, Reading, MA, 1992.

[SF96] Robert Sedgewick, Philippe Flajolet, *An Introduction to the Analysis of Algorithms,* Addison-Wesley, Reading, MA, 1996 [seconda edizione 2013].

[Ste92] W. Richard Stevens, *Advanced Programming in the Unix Environment*, Addison-Wesley, Reading, MA, 1992 [terza edizione 2013, aggiornata da Stephen A. Rago].

[Ste98] W. Richard Stevens. *Unix Network Programming, Volume 1: Networking APIs: Sockets and Xti*, Prentice Hall, Englewood Cliffs, NJ, seconda edizione, 1998 [terza edizione 2004, coautori Bill Fenner e Andrew M. Rudoff].

[Ste99] W. Richard Stevens, *Unix Network Programming, Volume 2: Interprocess Communications*, Prentice Hall, Englewood Cliffs, NJ, seconda edizione, 1999 [terza edizione 2004, coautori Bill Fenner e Andrew M. Rudoff].

[Str35] James Ridley Stroop, "Studies of interference in serial verbal reactions", in *Journal of Experimental Psychology,* 18, pp. 643-662, 1935.

[TFH04] Dave Thomas, Chad Fowler, Andy Hunt, *Programming Ruby, The Pragmatic Programmers' Guide*, The Pragmatic Programmers, LLC, Raleigh, NC e Dallas, TX, 2004 [quarta edizione 2013].

[TH03] Dave Thomas, Andy Hunt, *Pragmatic Version Control Using CVS,* The Pragmatic Programmers, LLC, Raleigh, NC, and Dallas, TX, 2003.

[WK82] James Q. Wilson, George Kelling, "The police and neighborhood safety", in *The Atlantic Monthly*, 249(3), pp. 29-38, marzo 1982.

[YC86] Edward Yourdon, Larry L. Constantine, *Structured Design: Fundamentals of a Discipline of Computer Program and Systems Design,* Prentice Hall, Englewood Cliffs, NJ, seconda edizione, 1986.

[You95] Edward Yourdon, "When good-enough software is best", in *IEEE Software,* maggio 1995.

Risposte agli esercizi

Esercizio 1 State scrivendo una classe che si chiama `Split`, che suddivide le righe di input in campi. Quale delle seguenti due firme di classe Java ha una progettazione più ortogonale?

```
class Split1 {
  public Split1(InputStreamReader rdr) { ...
  public void readNextLine() throws IOException { ...
  public int numFields() { ...
  public String getField(int fieldNo) { ...
}

class Split2 {
  public Split2(String line) { ...
  public int numFields() { ...
  public String getField(int fieldNo) { ...
}
```

Risposta Secondo il nostro modo di pensare, la classe `Split2` è più ortogonale. Si concentra sul proprio compito, dividere le righe, e ignora dettagli come da dove provengono le righe. Questo non solo rende il codice più facile da sviluppare, ma lo rende anche più flessibile. `Split2` può suddividere righe lette da un file, generate da un'altra routine o che le sono passate attraverso l'ambiente.

Esercizio 2 Che cosa porta a una progettazione più ortogonale: le finestre di dialogo senza modalità o quelle modali?

Risposta Se realizzate correttamente, probabilmente quelle senza modalità. Un sistema che usa le finestre di dialogo senza modalità sarà meno preoccupato di quello che succede in ogni particolare momento. Probabilmente avrà una migliore infrastruttura di comunicazione fra i moduli rispetto a un sistema modale, che può avere incorporate ipotesi sullo stato del sistema - ipotesi che portano a un maggiore accoppiamento e a una minore ortogonalità.

Esercizio 3 Che cosa dire dei linguaggi procedurali rispetto alla tecnologia a oggetti? Che cosa dà un sistema più ortogonale?

Risposta Queste sono domande più difficili. La tecnologia a oggetti può dare un sistema più ortogonale, ma ha più caratteristiche di cui si può abusare, perciò di fatto è più facile creare un sistema non ortogonale utilizzando gli oggetti che non in un linguaggio procedurale. Caratteristiche come l'eredità multipla, le eccezioni, il sovraccarico degli operatori e l'aggiramento dei metodi dei genitori (attraverso il subclassing) offrono ampie possibilità di aumentare l'accoppiamento in modi non ovvi. Con la tecnologia a oggetti e un po' di fatica in più, si può ottenere un sistema molto più ortogonale, ma, se si può sempre scrivere "codice a spaghetti" in un linguaggio procedurale, i linguaggi a oggetti, se usati male, possono aggiungere agli spaghetti anche il ragù di carne.

Esercizio 4 Il marketing vuole un incontro e un po' di brainstorming su alcuni progetti di pagine web. Stanno pensando a mappe immagine cliccabili che portino ad altre pagine e cose simili, ma non riescono a decidere un modello per l'immagine - magari un'auto, o un telefono o una casa. Avete un elenco di pagine e di contenuti target; vorrebbero vedere qualche prototipo. Ah, per inciso: avete un quarto d'ora. Che strumenti potreste usare?

Risposta Low tech alla riscossa! Fate qualche disegnino su una lavagna bianca: un'auto, un telefono e una casa. Non devono essere opere d'arte: vanno benissimo figurine disegnate con pochi tratti. Aggiungete dei Post-it che descrivono i contenuti delle pagine bersaglio sulle aree attive (su cui si può fare clic). Con il procedere della riunione, potete perfezionare i disegni e la collocazione dei Post-it.

Esercizio 5 Vogliamo implementare un mini-linguaggio per controllare un semplice pacchetto di disegno (magari un sistema di grafica della tartaruga). Il linguaggio è costituito da comandi di una sola lettera. Alcuni comandi sono seguiti da un singolo numero. Per esempio, questo input disegnerebbe un rettangolo:

```
P 2    # select pen 2
D    # pen down
W 2     # draw west 2cm
N 1        # then north 1
E 2  # then east 2
S 1   # then back south
U         # pen up
```

Implementate il codice che analizza sintatticamente il linguaggio. Deve essere progettato in modo che sia semplice aggiungere nuovi comandi.

Risposta Poiché vogliamo rendere estendibile il linguaggio, creeremo un analizzatore sintattico basato su tabella. Ogni voce nella tabella contiene la lettera di comando, un flag per dire se è necessario un argomento, e il nome della routine da chiamare per gestire quel particolare comando.

```
typedef struct {
  char  cmd;                    /* la lettera del comando */
  int hasArg;                   /* prende un argomento */
  void (*func)(int, int);       /* routine da chiamare */
} Command;
static Command cmds[] = {
  { 'P',  ARG,     doSelectPen },
  { 'U',  NO_ARG,  doPenUp },
  { 'D',  NO_ARG,  doPenDown },
  { 'N',  ARG,     doPenDir },
  { 'E',  ARG,     doPenDir },
  { 'S',  ARG,     doPenDir },
  { 'W',  ARG,     doPenDir }
};
```

Il programma principale è molto semplice: legge una riga, cerca il comando nella tabella, prende l'argomento se richiesto, poi chiama la funzione per gestirlo.

```
while (fgets(buff, sizeof(buff), stdin)) {
  Command *cmd = findCommand(*buff);
  if (cmd) {
    int arg = 0;
    if (cmd->hasArg && !getArg(buff+1, &arg)) {
      fprintf(stderr, "'%c' richiede un argomento\n", *buff);
      continue;
    }
    cmd->func(*buff, arg);
  }
}
```

La funzione che cerca un comando esegue una ricerca lineare nella tabella, restituendo o la voce corrisponde o `NULL`.

```
Command *findCommand(int cmd) {
  int i;
  for (i = 0; i < ARRAY_SIZE(cmds); i++) {
    if (cmds[i].cmd == cmd)
      return cmds + i;
  }
  fprintf(stderr, "Comando sconosciuto '%c'nn", cmd);
  return 0;
}
```

Infine, la lettura dell'argomento numerico è molto semplice con `sscanf`.

```
int getArg(const char *buff, int *result) {
  return sscanf(buff, "%d", result) == 1;
```

```
}
```

Esercizio 6 Progettate una grammatica BNF per analizzare una specifica temporale. Devono essere accettati tutti gli esempi seguenti:

```
4pm, 7:38pm, 23:42, 3:16, 3:16am
```

Risposta Utilizzando la BNF, una specifica temporale potrebbe essere

```
<time>   ::= <hour> <ampm> |
             <hour> : <minute> <ampm> |
             <hour> : <minute>

<ampm>   ::= am | pm

<hour >  ::= <digit> |
             <digit> <digit>
<minute> ::= <digit> <digit>

<digit>  ::= 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9
```

Esercizio 7 Implementate un parser (analizzatore sintattico) per la grammatica BNF dell'Esercizio 6 utilizzando yacc, bison o qualche altro generatore di parser.

Risposta Abbiamo codificato il nostro esempio con bison, la versione GNU di yacc. Per chiarezza, presentiamo qui solo il corpo dell'analizzatore sintattico. Potete esaminare il codice sorgente dell'implementazione completa sul nostro sito web.

```
time:    spec END_TOKEN
          { if ($1 >= 24*60) yyerror("Troppo grande");
            printf("%d minuti dopo mezzanotte\n", $1);
      exit(0);
          }
        ;

spec:     hour ':' minute
          { $$ = $1 + $3;
          }
        | hour ':' minute ampm
          { if ($1 > 11*60) yyerror("Ora fuori intervallo");
            $$ = $1 + $3 + $4;
      }
        | hour ampm
          { if ($1 > 11*60) yyerror("Ora fuori intervallo ");
            $$ = $1 + $2;
          }
        ;

hour:     hour_num
          { if ($1 > 23) yyerror("Ora fuori intervallo ");
        $$ = $1 * 60;
          };

minute:   DIGIT DIGIT
          { $$ = $1*10 + $2;
            if ($$ > 59) yyerror("minuto fuori intervallo");
          };
```

```
ampm:     AM                  { $$ = AM_MINS; }
        | PM                  { $$ = PM_MINS; }
        ;

hour_num: DIGIT               { $$ = $1; }
        | DIGIT DIGIT         { $$ = $1*10 + $2; }
        ;
```

Esercizio 8 Implementate con Perl il parser delle indicazioni temporali. (Suggerimento: le espressioni regolari creano buoni parser.)

Risposta

```
$_ = shift;
/^(ndnd?)(am|pm)$/            && doTime($1, 0, $2, 12);
/^(ndnd?):(ndnd)(am|pm)$/     && doTime($1, $2, $3, 12);
/^(ndnd?):(ndnd)$/            && doTime($1, $2, 0, 24);
die "Orario non valido $_nn";
#
# doTime(hour, min, ampm, maxHour)
#
sub doTime($$$$) {
  my ($hour, $min, $offset, $maxHour) = @_;
  die "Ora non valida: $hour" if ($hour >= $maxHour);
  $hour += 12 if ($offset eq "pm");
  print $hour*60 + $min, " minuti dopo mezzanotte\n";
  exit(0);
}
```

Esercizio 9 Vi viene chiesto: “Che cosa ha una larghezza di banda maggiore, una linea di comunicazione a 1 Mbps o una persona che va da un computer all’altro con un nastro da 4 GB in tasca?”. Quali vincoli indicherete, per essere sicuri che l’ambito della vostra risposta sia corretto? (Per esempio, potreste dire che il tempo di accesso al nastro viene ignorato.)

Risposta La nostra risposta deve fare varie ipotesi.

- Il nastro contiene le informazioni che dobbiamo trasferire.
- Sappiamo la velocità a cui la persona cammina.
- Sappiamo la distanza fra le macchine.
- Non teniamo conto del tempo necessario per trasferire le informazioni dal nastro e al nastro.
- L'overhead della memorizzazione dei dati su nastro è all’incirca uguale all'overhead del loro invio su una linea di comunicazione.

Esercizio 10 Allora, che cosa ha la larghezza di banda maggiore?

Risposta Tenendo presenti le ipotesi fatte nella risposta all’esercizio 9, un nastro da 4 GB contiene 32 × 109 bit, perciò una linea da 1 Mbps

dovrebbe pompare dati per circa 32.000 secondi, ovvero circa 9 ore, per trasferire la quantità equivalente di informazioni. Se la persona cammina a velocità costante di 3,5 chilometri all'ora, le due macchine dovrebbero essere lontane almeno 31 chilometri perché la linea di comunicazione batta il nostro corriere; altrimenti, vince la persona.

Esercizio 11 Il vostro programma in C usa un tipo enumerato per rappresentare uno stato fra 100 possibili. Vorreste poter stampare lo stato come stringa (anziché come numero) a fini di debug. Scrivete uno script che legga dall'input standard un file contenente

```
name
state_a
state_b
 :    :
```

Fornite il file name.h, che contiene

```
extern const char* NAME_names[];
typedef enum {
   state_a,
   state_b,
    :    :
 } NAME;
```

e il file `name.c`, che contiene

```
const char* NAME_names[] = {
   "state_a",
   "state_b",
     :    :
 };
```

Risposta Abbiamo implementato la nostra risposta in Perl.

```
my @consts;
my $name = <>;
die "Formato non valido - manca il nome" unless defined($name);
chomp $name;
# Legge il resto del file
while (<>) {
  chomp;
  s/^ns*//; s/ns*$//;
  die "Invalid line: $_" unless /^(nw+)$/;
  push @consts, $_;
}
# Ora genera il file
open(HDR, ">$name.h") or die "Non posso aprire $name.h: $!";
open(SRC, ">$name.c") or die "Non posso aprire $name.c: $!";
my $uc_name = uc($name);
my $array_name = $uc_name . "_names";
print HDR "/*File generato automaticamente - non modificare */nn";
print HDR "extern const char *$ {array_name}[];";
print HDR "typedef enum {nn ";
print HDR join ",nn ", @consts;
print HDR "nn} $uc_name;nnnn";
print SRC "/*File generato automaticamente - non modificare */nn";
```

```
print SRC "const char *$ {array_name}[] = {nn n"";
print SRC join "n",nn n"", @consts;
print SRC "n"nn};nn";
close(SRC);
close(HDR);
```

Applicando il principio DRY, non procederemo tagliando e incollando il nuovo file nel nostro codice. Effettueremo invece un `#include`: il file piatto è la sorgente di queste costanti. Questo significa che avremo bisogno che un makefile rigeneri l'intestazione quando il file cambia. Il frammento che segue è tratto dal test nel nostro albero sorgente (disponibile sul sito web).

```
etest.c etest.h:  etest.inc enumerated.pl
                  perl enumerated.pl etest.inc
```

Esercizio 12 A metà della scrittura di questo libro, ci siamo resi conto che non avevamo inserito la direttiva `use strict` in molti dei nostri esempi in Perl. Preparate uno script che esamini tutti i file .pl in una directory e aggiunga una `use strict` alla fine del blocco iniziale di commenti a tutti i file che già non la contengono. Ricordare di tenere una copia di riserva di tutti i file che modificate.

Risposta Ecco la nostra risposta, scritta in Perl.

```
my $dir = shift or die "Directory inesistente";
for my $file (glob("$dir/*.pl")) {
  open(IP, "$file") or die "Apro $file: $!";
  undef $/;              # Disattiva il separatore dei record in input --
  my $content = <IP>;    # legge tutto il file come una sola stringa.
  close(IP);
  if ($content !~ /^use strict/m) {
    rename $file, "$file.bak" or die "Rinomino $file: $!";
    open(OP, ">$file") or die "Creo $file: $!";
    # Metti 'use strict' sulla prima riga che
    # non inizia con '#'
    $content =~ s/^(?!#)/nnuse strict;nnnn/m;
    print OP $content;
    close(OP);
    print "Aggiornato $filenn";
  }
  else {
    print "$file già strict\n";
  }
}
```

Esercizio 13 Scrivete un generatore di codice che prenda il file di input della Figura 3.4 e generi output in due linguaggi di vostra scelta. Cercate di fare in modo che sia facile aggiungere altri linguaggi.

Risposta Usiamo Perl per implementare la nostra soluzione, che carica dinamicamente un modulo per generare il linguaggio richiesto, cosicché

aggiungere nuovi linguaggi sarà facile. La routine principale carica il back-end (in base a un parametro da riga di comando), poi legge l'input e chiama le routine di generazione del codice in funzione dei contenuti di ciascuna riga. Non siamo particolarmente fiscali nella gestione degli errori - sapremo comunque rapidamente se le cose non vanno per il verso giusto.

```
my $lang = shift or die "Linguaggio mancante";
$lang .= "_cg.pm";
require "$lang" or die "Impossibile caricare $lang";
# Legge e analizza il file
my $name;
while (<>) {
chomp;
  if (/^ns*$/)                                { CG::blankLine(); }
  elsif (/^n#(.*)/)                           { CG::comment($1); }
  elsif (/^Mns*(.+)/)                         { CG::startMsg($1); $name = $1; }
  elsif (/^E/)                                { CG::endMsg($name); }
  elsif (/^Fns*(nw+)ns+(nw+)$/)               { CG::simpleType($1,$2); }
  elsif (/^Fns*(nw+)ns+(nw+)n[(nd+)n]$/)      { CG::arrayType($1,$2,$3); }
  else {
    die "Riga non valida: $_";
  }
}
```

Scrivere il back-end per un linguaggio è semplice: si fornisce un modulo che implementi i sei necessari punti di ingresso. Ecco il generatore per il C.

```
#!/usr/bin/perl -w
package CG;
use strict;
# Generatore di codice per 'C' (vedi cg_base.pl)
sub blankLine()   { print "nn"; }
sub comment()     { print "/*$_[0] */nn"; }
sub startMsg()    { print "typedef struct {nn"; }
sub endMsg()      { print "} $_[0];nnnn"; }
sub arrayType() {
  my ($name, $type, $size) = @_;
  print " $type $namen[$size];nn";
}
sub simpleType() {
  my ($name, $type) = @_;
  print " $type $name;nn";
}
1;
```

Ed ecco quello per il Pascal:

```
#!/usr/bin/perl -w
package CG;
use strict;
# Generatore di codice per 'Pascal' (vedi cg_base.pl)
sub blankLine()   { print "\n"; }
sub comment()     { print "{$_[0] }\n"; }
```

```
sub startMsg()   { print "$_[0] = packed record\n"; }
sub endMsg()     { print "end;\n\n"; }
sub arrayType()  {
  my ($name, $type, $size) = @_;
  $size--;
  print " $name: array[0..$size] of $type;\n";
}
sub simpleType() {
  my ($name, $type) = @_;
  print " $name: $type;\n";
}
1;
```

Esercizio 14 Che cosa fa di un contratto un buon contratto? Chiunque può aggiungere precondizioni e postcondizioni, ma serviranno a qualcosa? Peggio ancora, faranno più male che bene? Per l'esempio sotto e per quelli degli Esercizi 15 e 16, stabilite se il contratto specificato è buono, brutto o cattivo e spiegate perché.

Come prima cosa, vediamo un esempio in Eiffel. Qui abbiamo una routine per aggiungere una `STRING` a una lista circolare doppiamente concatenata (ricordate che le precondizioni sono indicate con require, le postcondizioni con ensure).

```
-- Aggiunge un elemento unico a una lista doppiamente concatenata
-- e restituisce il NODE appena creato.

add_item (item : STRING) : NODE is
   require
      item /= Void                -- '/=' significa 'diverso da'.
      find_item(item) = Void      -- Deve essere unico
   deferred                       -- Classe base astratta.
   ensure
      result.next.previous = result -- Controlla i link del nodo
      result.previous.next = result -- appena aggiunto.
      find_item(item) = result      -- Deve trovarlo.
   end
```

Risposta L'esempio in Eiffel è buono. Richiediamo che vengano passati dati non nulli, e garantiamo che sarà rispettata la semantica di una lista circolare doppiamente linkata. Ci aiuta anche a trovare la stringa che abbiamo memorizzato. Poiché questa è una classe deferred, la classe effettiva che la implementa è libera di usare qualsiasi meccanismo sottostante desideri. Può scegliere di usare puntatori, o un array o qualsiasi altra cosa: purché adempia al contratto, non ci interessa.

Esercizio 15 Proviamo un esempio in Java, qualcosa di simile all'esempio dell'Esercizio 14. `insertNumber` inserisce un intero in una lista ordinata. Le precondizioni e postcondizioni sono etichettate come in iContract (vedi [URL17]).

```
private int data[];
/**
  *@post data[index-1] < data[index] &&
  *data[index] == aValue
  */
public Node insertNumber (final int aValue)
{
  int index = findPlaceToInsert(aValue);
  ...
```

Risposta Questo è un cattivo contratto. L'operazione matematica nella clausola indice (`index-1`) non funzionerà in condizioni limite come la prima voce. La postcondizione assume una particolare implementazione: vogliamo che i contratti siano molto più astratti.

Esercizio 16 Ecco un frammento di una classe stack in Java. Questo è un buon contratto?

```
/**
  *@pre anItem != null   // Richiede dati reali
  *@post pop() == anItem // Verifica che siano
  *                     // sullo stack
  */
public void push(final String anItem)
```

Risposta È un buon contratto, ma con una cattiva implementazione. Qui mostra la sua brutta testa il famigerato "Heisenbug" [URL52]. Il programmatore probabilmente ha solo commesso un errore di battitura, ha scritto *pop* invece di *top*. Questo è un esempio semplice e di portata limitata, ma gli effetti collaterali nelle asserzioni (o in qualsiasi punto imprevisto del codice) possono essere molto difficili da diagnosticare.

Esercizio 17 Gli esempi classici di DBC (come negli Esercizi 14-16) mostrano una implementazione di un tipo di dati astratto - normalmente uno stack o una coda. Non molti però scrivono realmente classi di basso livello di questo tipo.

Perciò, per questo esercizio, progettiamo un'interfaccia per un miscelatore da cucina. Alla fine sarà un miscelatore basato su Web, abilitato a Internet e basato su CORBA, ma per il momento ci serve solo l'interfaccia per controllarlo. Ha dieci velocità (0 significa spento). Non si può usarlo vuoto e si può cambiare la velocità solo di una unità alla volta (si può passare cioè da 0 a 1 e da 1 a 2, ma non da 0 a 2).

Ecco i metodi. Aggiungete le opportune precondizioni e postcondizioni e una invariante.

```
int getSpeed()
void setSpeed(int x)
boolean isFull()
```

```
void fill()
void empty()
```

Risposta Mostreremo le "firme" delle funzioni in Java, con le precondizioni e postcondizioni etichettate come in iContract.

Prima, l'invariante per la classe:

```
/**
  *@invariant getSpeed() > 0
          *implies isFull()                // Non eseguita se vuoto
  *@invariant getSpeed() >= 0 &&
          *getSpeed() < 10                 // Controllo di intervallo
*/
```

Poi, le precondizioni e postcondizioni:

```
/**
  *@pre Math.abs(getSpeed() - x) <= 1    // Cambia solo di uno
  *@pre x >= 0 && x < 10                 // Controllo dell'intervallo
  *@post getSpeed() == x                 // Rispetta la velocità richiesta
*/
public void setSpeed(final int x)
/**
  *@pre !isFull()                        // Non lo riempie due volte
  *@post isFull()                        // Garantisce che è stato fatto
*/
void fill()
/**
  *@pre isFull()                         // Non lo svuota due volte
  *@post !isFull()                       // Garantisce che è stato fatto
*/
void empty()
```

Esercizio 18 Quanti numeri ci sono nella successione 0, 5, 10, 15, …, 100?

Risposta Nella successione ci sono 21 termini. Se avete detto 20, avete appena commesso un "errore di fencepost".

Esercizio 19 Una veloce "verifica di realtà". Quali di queste cose "impossibili" possono succedere?

1. Un mese con meno di 28 giorni.
2. `stat(".",&sb) == -1` (cioè, non è possibile accedere alla directory corrente).
3. In C++: `a = 2; b = 3; if (a + b != 5) exit(1);`
4. Un triangolo la somma dei cui angoli interni è diversa da 180°.
5. Un minuto che non ha 60 secondi.
6. In Java: `(a + 1) <= a`.

Risposta

1. Settembre 1752 ha avuto solo 19 giorni, per sincronizzare il calendario nell'ambito della riforma gregoriana.
2. La directory potrebbe essere stata eliminata da un altro processo, potreste non avere autorizzazioni sufficienti per leggerla, `&sb` potrebbe non essere valido…
3. Furbescamente, non abbiamo specificato i tipi di `a` e `b`. Il sovraccarico degli operatori potrebbe avere definito un comportamento imprevisto per `+`, `=` o `!=`. Inoltre `a` e `b` potrebbero essere alias della stessa variabile, perciò la seconda assegnazione sovrascriverebbe il valore memorizzato nella prima.
4. Nella geometria non euclidea, la somma degli angoli di un triangolo non è pari a 180°. Pensate a un triangolo riportato sulla superficie di una sfera.
5. Minuti con secondi intercalari (per allineare tempo coordinato universale e giorno solare medio)
6. L'overflow può provocare un risultato negativo di `a + 1` (può succedere anche in C e C++).

Esercizio 20 Sviluppate una semplice classe di controllo delle asserzioni per Java.

Risposta Abbiamo scelto di implementare una classe molto semplice con un unico metodo statico, `TEST`, che stampa un messaggio e una traccia di stack se il parametro `condition` passato è falso.

```
package com.pragprog.util;

import java.lang.System;        // per exit()
import java.lang.Thread;        // per dumpStack()

public class Assert {

  /** Scrive un messaggio, stampa una traccia dello stack
    * ed esce se il nostro parametro è falso.
    */
  public static void TEST(boolean condition) {
    if (!condition) {
      System.out.println("==== Asserzione fallita ====");
      Thread.dumpStack();
      System.exit(1);
    }
  }

  // Banco di prova. Se il nostro argomento è 'okay', prova una asserzione
  // che ha successo, se 'fallito' prova una asserzione che fallisce

  public static final void main(String args[]) {
    if (args[0].compareTo("okay") == 0) {
```

```
      TEST(1 == 1);
    }
    else if (args[0].compareTo("fallito") == 0) {
      TEST(1 == 2);
    }
    else {
      throw new RuntimeException("Argomento errato");
    }
  }
}
```

Esercizio 21 Nel progettare una nuova classe contenitore, identificate queste possibili condizioni di errore:

1. Mancata disponibilità di memoria per un nuovo elemento nella routine `add`.
2. Mancato reperimento dell'elemento richiesto nella routine `fetch`.
3. Passaggio di un puntatore `null` alla routine `add`.

Come gestirle? Generare un errore, sollevare un'eccezione o ignorare la condizione?

Risposta L'esaurimento della memoria è una condizione eccezionale, perciò secondo noi il caso (1) deve sollevare un'eccezione.

La mancata individuazione di un valore in ingresso probabilmente è un evento normale. L'applicazione che chiama la nostra classe collezione può scrivere codice che verifica se è presente un valore, prima di aggiungere un potenziale duplicato. Secondo noi il caso (2) deve solo restituire un errore.

Il caso (3) è più problematico: se il valore `null` è significativo per l'applicazione, può essere giustificato aggiungerlo al contenitore. Se però non ha senso memorizzare valori nulli, probabilmente sarebbe giusto lanciare un'eccezione.

Esercizio 22 Qualche sviluppatore in C e C++ si impone di fissare un puntatore a `NULL` dopo aver liberato la memoria a cui fa riferimento. Perché è una buona idea?

Risposta Nella maggior parte delle implementazioni di C e C++, non c'è modo per verificare se un puntatore punta effettivamente a una posizione valida in memoria. Un errore comune è liberare un blocco di memoria e poi far riferimento a quella memoria nel seguito del programma. A quel punto, il blocco di memoria a cui si punta può essere stato assegnato di nuovo per qualche altro fine. Impostando il puntatore a

NULL, i programmatori sperano di impedire questi riferimenti scorretti: nella maggior parte dei casi, il dereferenziamento di un puntatore NULL genererà un errore di runtime.

Esercizio 23 Qualche sviluppatore in Java si impone di fissare una variabile di oggetto a NULL dopo aver finito di usare l'oggetto. Perché è una buona idea?

Risposta Impostando il riferimento a NULL, si riduce di uno il numero dei puntatori all'oggetto referenziato. Quando il conteggio arriva a zero, l'oggetto è pronto per la "raccolta della spazzatura". Impostare i riferimenti a NULL può essere significativo per programmi che girano a lungo, dove i programmatori devono assicurarsi che l'utilizzo della memoria non cresca nel tempo.

Esercizio 24 Abbiamo esaminato il concetto di disaccoppiamento fisico nella Figura 5.1. Quale dei seguenti file di intestazione C++ è accoppiato più strettamente al resto del sistema?

```
person1.h:     person2.h:
  #include "date.h"       class Date;
  class Person1 {     class  Person2 {
  private:       private:
    Date myBirthdate;       Date *myBirthdate;
  public:     public:
    Person1(Date &birthDate);            Person2(Date &birthDate
    // ...      // ...
```

Risposta Si suppone che un file di intestazione definisca l'interfaccia fra l'implementazione corrispondente e il resto del mondo. Il file di intestazione stesso non ha bisogno di sapere nulla di come è fatta internamente la classe Date; deve solo dire al compilatore che il costruttore prende un Date come parametro. Perciò, a meno che il file di intestazione usi i Date in funzioni inline, il secondo frammento funzionerà bene.

Che cosa non va nel primo frammento? In un progetto di piccole dimensioni, nulla, tranne che si fa una cosa non necessaria, cioè costringere qualsiasi cosa usi una classe Person1 a includere anche il file di intestazione per Date. Una volta che questo tipo di uso diventa comune in un progetto, l'inclusione di un file di intestazione finisce per includere la maggior parte del resto del sistema, con un forte allungamento dei tempi di compilazione.

Esercizio 25 Per l'esempio qui sotto e per quelli negli Esercizi 26 e 27, stabilite se le chiamate di metodi mostrate sono consentite, in base alla

legge di Demetra. Questo primo esempio è in Java.

```
public void showBalance(BankAccount acct) {
  Money amt = acct.getBalance();
  printToScreen(amt.printFormat());
}
```

Risposta La variabile acct viene passata come parametro, perciò è consentita la chiamata `getBalance`. Non è consentito però chiamare `amt.printFormat()`. Non siamo i "proprietari" di `amt` e non ci è stata passata. Potremmo eliminare l'accoppiamento tra `showBalance` e `Money` con qualcosa di questo genere:

```
void showBalance(BankAccount b) {
  b.printBalance();
}
```

Esercizio 26 Anche questo esempio è in Java.

```
public class Colada {
  private Blender myBlender;
  private Vector myStuff;
  public Colada() {
    myBlender = new Blender();
    myStuff = new Vector();
  }
  private void doSomething() {
    myBlender.addIngredients(myStuff.elements());
  }
}
```

Risposta Poiché `Colada` crea e quindi è proprietaria di `myBlender` e `myStuff`, le chiamate a `addIngredients` ed `elements` sono consentite.

Esercizio 27 Questo esempio è in C++.

```
void processTransaction(BankAccount acct, int) {
  Person *who;
  Money amt;
  amt.setValue(123.45);
  acct.setBalance(amt);
  who = acct.getOwner();
  markWorkflow(who->name(), SET_BALANCE);
}
```

Risposta In questo caso, `processTransaction` è proprietaria di `amt`: viene creata sullo stack. Viene passata `acct`, perciò sono consentite sia `setValue` sia `setBalance`. Ma `processTransaction` non è proprietaria di `who`, perciò la chiamata `who->name()` è una violazione. La legge di Demetra suggerisce di sostituire questa riga con

```
markWorkflow(acct.name(), SET_BALANCE);
```

Il codice in `processTrransaction` non ha bisogno di sapere quale sotto-oggetto in un `BankAccount` contiene il nome dell'intestatario: questa informazione struttura non deve emergere, per il contratto di `BankAccount`.

Noi invece chiediamo a `BankAccount` il nome dell'intestatario del conto. `BankAccount` sa dove conserva il nome (magari in un oggetto `Person`, in un `Business` o in un oggetto polimorfo `Customer`).

Esercizio 28 Quali delle cose seguenti sarebbe meglio rappresentare come codice in un programma, e quali esternamente come metadati?

1. Assegnazioni delle porte di comunicazione.
2. Il supporto dell'editor per l'evidenziazione della sintassi di vari linguaggi.
3. Il supporto dell'editor per dispositivi grafici diversi.
4. Una macchina a stati per un analizzatore sintattico o uno scanner.
5. Valori e risultati campione da usare in unit test.

Risposta Non ci sono risposte definitive. Le domande volevano soprattutto spingervi a riflettere. Ecco comunque quello che pensiamo noi.

1. *Assegnazioni delle porte di comunicazione*. Chiaramente, questa informazione va conservata in forma di metadati. Ma fino a che livello di dettaglio? Alcuni programmi di comunicazione per Windows consentono di selezionare solo baud rate e porta (per esempio, da COM1 a COM4). Forse però avete bisogno di specificare anche dimensione di parola, parità, bit di stop e impostazione duplex. Cercate di consentire, dove è praticabile, il livello di dettaglio più fine.
2. *Il supporto dell'editor per l'evidenziazione della sintassi di vari linguaggi*. Va implementato come metadati. Non vorrete dover intervenire sul codice solo perché una nuova versione di Java ha introdotto una nuova parola chiave.
3. *Il supporto dell'editor per dispositivi grafici diversi*. Probabilmente sarebbe difficile implementarlo strettamente in forma di metadati. Non vorrete appesantire la vostra applicazione con molti driver di dispositivo solo per selezionarne uno al momento dell'esecuzione. Potreste però usare i metadati per specificare il nome del driver e caricare dinamicamente il codice. Questo è un altro buon argomento per conservare i metadati in un formato leggibile da un essere

umano: se usate il programma per impostare un driver video disfunzionale, potreste non essere in grado di usare il programma per reimpostarlo.

4. *Una macchina a stati per un analizzatore sintattico o uno scanner*. Dipende da quello che dovete analizzare o passare allo scanner. Se analizzate dei dati definiti rigidamente da un corpus di standard che è improbabile vengano modificati senza una legge del Congresso, codificarla in modo rigido andrebbe benissimo. Se però dovete trattare una situazione più fluida, sarebbe una buona cosa definire esternamente le tabelle degli stati.
5. *Valori e risultati campione da usare in unit test*. La maggior parte delle applicazione definisce questi valori inline nel test harness, ma si può ottenere una flessibilità maggiore spostando i dati di test (e la definizione dei risultati accettabili) al di fuori del codice stesso.

Esercizio 29 Supponiamo di avere un sistema di prenotazioni aeree che include il concetto di "volo".

```
public interface Flight {
  // Restituisce false se il volo è tutto prenotato.
  public boolean addPassenger(Passenger p);
  public void addToWaitList(Passenger p);
  public int getFlightCapacity();
  public int getNumPassengers();
}
```

Se aggiungiamo un passeggero alla lista d'attesa, verrà automaticamente assegnato al volo se si rende disponibile un posto.

Un enorme sistema di reporting continua a tenere sotto controllo voli completi o in overbooking, per suggerire quando sarebbe opportuno prevedere voli ulteriori. Funziona bene, ma la sua esecuzione richiede ore.

Vorremmo avere un po' più di flessibilità nell'elaborare i passeggeri della lista d'attesa, e dobbiamo fare qualcosa a proposito del sistema di reporting che impiega troppo tempo. Usate le idee viste in questa sezione per riprogettare questa interfaccia.

Risposta Prendiamo `Flight` e aggiungiamo qualche altro metodo per mantenere due liste di listener: una per le notifiche della lista d'attesa, l'altra per le notifiche di volo completo.

```
public interface Passenger {
  public void waitListAvailable();
}
```

```
public interface Flight {
  ...
  public void addWaitListListener(Passenger p);
  public void removeWaitListListener(Passenger p);
  public void addFullListener(FullListener b);
  public void removeFullListener(FullListener b);
  ...
}
public interface BigReport extends FullListener {
  public void FlightFullAlert(Flight f);
}
```

Se tentiamo di aggiungere un `Passenger` e non ci riusciamo perché il volo è completo, possiamo, facoltativamente, mettere quel passeggero nella lista d'attesta. Quando si libera un posto, verrà chiamato `waitListAvailable`. Questo metodo poi può scegliere di aggiungere automaticamente il `Passenger`, oppure chiedere a un portavoce del servizio di chiamare il cliente e chiedergli se è ancora interessato, o quello che si vuole. Ora abbiamo la flessibilità di mettere in atto comportamenti diversi cliente per cliente.

Poi, vogliamo evitare che il troll `BigReport` esamini tonnellate di record alla ricerca dei voli completi. Registrando `BigReport` come un listener su `Flights`, ogni singolo `Flight` può comunicare quando è pieno, o quasi pieno, se vogliamo. Ora gli utenti possono avere istantaneamente comunicazioni aggiornatissime da `BigReport`, senza aspettare che giri per ore come succedeva in precedenza.

Esercizio 30 Per ciascuna delle applicazioni seguenti, un sistema a lavagna sarebbe appropriato oppure no? Perché?

1. *Elaborazione di immagini*. Vorreste avere una serie di processi in parallelo che prendono parti di un'immagine, le elaborano e poi rimettono a posto i pezzi completati.
2. *Calendari di gruppo*. Avete persone sparse per il pianeta, sotto fusi orari diversi e che parlano lingue diverse, e vogliono pianificare una riunione.
3. *Strumento di monitoraggio della rete*. Il sistema raccoglie statistiche sulle prestazioni e segnalazioni di guasti. Vorreste implementare qualche agente che usi queste informazioni per verificare i problemi nel sistema.

Risposta

1. *Elaborazione di immagini*. Per la semplice pianificazione di un carico di lavoro da assegnare a processi paralleli, una coda di lavoro condivisa può essere più che adeguata. Potete prendere in considerazione un sistema a lavagna se è coinvolto un feedback - cioè se i risultati di un blocco elaborato influenzano altri blocchi, come nelle applicazioni di visione automatica, o in complesse trasformazioni di immagini 3D.
2. *Calendari di gruppo*. Qui il sistema a lavagna può essere una buona soluzione. Potete inviare alla lavagna le riunioni programmate e la disponibilità. Avete entità che funzionano automaticamente, i feedback delle decisioni sono importanti e i partecipanti possono andare e venire. Potete prendere in considerazione l'idea di partizionare questo tipo di sistema a lavagna in funzione di chi effettua la ricerca: il personale di livello più basso può essere interessato solo all'ufficio a cui appartiene, le risorse umane possono volere solo gli uffici in cui si parla inglese in tutto il mondo, e l'amministratore delegato può volere tutto. Esiste anche una certa flessibilità nei formati dei dati: siamo liberi di ignorare formati o lingue che non comprendiamo. Dobbiamo capire i formati diversi solo per quegli uffici che hanno riunioni comuni, e non dobbiamo esporre tutti i partecipanti a una chiusura transitiva completa di tutti i formati possibili. Questo riduce l'accoppiamento alle situazioni in cui è necessario e non ci vincola in modo artificiale.
3. *Strumento di monitoraggio della rete*. Questo caso è molto simile a quello del programma di richiesta di mutuo/prestito descritto alla fine del Capitolo 5. Avete segnalazioni di problemi inviate dagli utenti e statistiche comunicate automaticamente, tutte inviate alla lavagna. Una persona o un agente software può analizzare la lavagna per diagnosticare guasti di rete: due errori su una linea possono essere solo colpa dei raggi cosmici, ma se sono 20.000 c'è un problema hardware. Come l'investigatore risolve un caso di omicidio, potete avere più entità che effettuano analisi e forniscono idee per risolvere i problemi di rete.

Esercizio 31 Potete identificare qualche coincidenza nel seguente frammento di codice C? Immaginate che questo codice sia sepolto in

profondità in una routine di libreria.

```
fprintf(stderr,"Error, continue?");
gets(buf);
```

Risposta Questo codice presenta vari potenziali problemi. In primo luogo, assume un ambiente tty. Va bene se l'assunto è corretto, ma che cosa succede se questo codice viene chiamato da un ambiente GUI in cui non sono aperti né `stderr` né `stdin`?

In secondo luogo, c'è il problematico `gets`, che scriverà nel buffer tanti caratteri quanti ne riceve. Utenti malintenzionati hanno usato il fallimento di questa funzione per creare falle di sicurezza di buffer overrun in molti sistemi diversi. Non usate mai `gets()`.

In terzo luogo, il codice assume che l'utente capisca l'inglese.

Infine, nessuna persona sana di mente seppellirebbe un'interazione con l'utente di questo tipo in una routine di libreria.

Esercizio 32 Questo pezzo di codice C può funzionare qualche volta su qualche macchina, ma anche no. Che cosa c'è di sbagliato?

```
/*Tronca una stringa ai suoi ultimi maxlen caratteri */
void string_tail(char *string, int maxlen) {
  int len = strlen(string);
  if (len > maxlen) {
    strcpy(string, string + (len - maxlen));
  }
}
```

Risposta Non è garantito che `strcpy` di POSIX funzioni per stringhe che si sovrappongono. Può darsi che funzioni su qualche architettura, ma solo per coincidenza.

Esercizio 33 Questo codice viene da una suite di traccia Java di uso generale. La funzione scrive una stringa in un file di log. Supera il suo unit test, ma fallisce quando la usa uno degli sviluppatori web. Su quale coincidenza si basa?

```
public static void debug(String s) throws IOException {
  FileWriter fw = new FileWriter("debug.log", true);
  fw.write(s);
  fw.flush();
  fw.close();
}
```

Risposta Non funzionerà nel contesto di una applet con limitazioni di sicurezza relativamente alla scrittura sul disco locale. Quando si ha la scelta fra esecuzione in contesti GUI o meno, si può controllare dinamicamente per capire come sia fatto l'ambiente corrente. In questo

caso, potreste inserire un file di log da qualche parte che non sia il disco locale se non è accessibile.

Esercizio 34 Abbiamo codificato un insieme di semplici routine di ordinamento, che possono essere scaricate dal nostro sito web (https://pragprog.com/the-pragmatic-programmer/source-code). Eseguitele sulle diverse macchine che avete a disposizione. I valori che ottenete seguono le curve previste? Che cosa potete dedurne circa la velocità relativa delle vostre macchine? Quali sono gli effetti di varie impostazioni di ottimizzazione del compilatore? Il radix sort è effettivamente lineare?

Risposta Chiaramente, non possiamo dare una risposta assoluta per questo esercizio. Possiamo però darvi un paio di indicazioni.

Se scoprite che i vostri risultati non seguono una curva continua, potete controllare e vedere se qualche altra attività sta usando parte della potenza del processore. Probabilmente non avrete buoni valori in un sistema multiutente e, anche se siete l'unico utente, potreste trovare che i processi in background periodicamente sottraggono cicli ai vostri programmi. Potete anche controllare la memoria: se l'applicazione comincia a utilizzare spazio di swap su disco, le prestazioni scenderanno a picco.

È interessante sperimentare con compilatori diversi e impostazioni di ottimizzazione diverse. Abbiamo trovato che, consentendo un'ottimizzazione aggressiva, erano possibili accelerazioni davvero notevoli. Abbiamo anche scoperto che sulle architetture RISC più ampie i compilatori dei costruttori spesso avevano prestazioni nettamente superiori ai GCC più portabili. Presumibilmente, i costruttori conoscono i segreti della generazione efficiente di codice su queste macchine.

Esercizio 35 La routine qui sotto stampa i contenuti di un albero binario. Ipotizzando che l'albero sia bilanciato, all'incirca quanto spazio di stack userà la routine nello stampare un albero di 1.000.000 di elementi? (Ipotizzate che le chiamate di routine non impongano un overhead significativo allo stack.)

```
void printTree(const Node *node) {
  char buffer[1000];
  if (node) {
    printTree(node->left);
    getNodeAsString(node, buffer);
    puts(buffer);
    printTree(node->right);
  }
}
```

Risposta La routine `printTree` usa circa 1000 byte di spazio sullo stack per la variabile `buffer`. Chiama se stessa ricorsivamente per scendere lungo l'albero e ogni chiamata annidata aggiunge allo stack altri 1000 byte. Chiama se stessa anche quando arriva ai nodi foglia, ma esce immediatamente quando scopre che il puntatore che le viene passato è `NULL`. Se la profondità dell'albero è *D*, il massimo stack richiesto è perciò circa 1000 × (*D* + 1).

Un albero binario bilanciato contiene il doppio di elementi a ogni livello. Un albero di profondità D contiene 1 + 2 + 4 + 8 + … + 2(*D*-1), ovvero 2*D* - 1 elementi. Il nostro albero da un milione di elementi avrà perciò bisogno di lg(1.000.001), ovvero 20 livelli. Ci aspettiamo quindi che la nostra routine usi circa 21.000 byte di stack.

Esercizio 36 Vedete qualche modo per ridurre i requisiti di stack della routine nell'Esercizio 35 (a parte ridurre le dimensioni del buffer)?

Risposta Vengono in mente un paio di ottimizzazioni. In primo luogo, la routine printTree chiama se stessa sui nodi foglia, ma esce immediatamente perché non ci sono figli. Questa chiamata aumenta la profondità massima dello stack di circa 1000 byte. Possiamo anche eliminare la ricorsione in coda (la seconda chiamata ricorsiva), anche se questo non ha conseguenze per l'uso dello stack nel caso peggiore.

```
while (node) {
  if (node->left) printTree(node->left);
  getNodeAsString(node, buffer);
  puts(buffer);
  node = node->right;
}
```

Si ha il vantaggio maggiore, però, allocando un buffer unico, condiviso da tutte le invocazioni di `printTree`. Passiamo questo buffer come parametro alle chiamate ricorsive, e verranno allocati soltanto 1000 byte, indipendentemente dalla profondità a cui arriva la ricorsione.

```
void printTreePrivate(const Node *node, char *buffer) {
  if (node) {
    printTreePrivate(node->left, buffer);
    getNodeAsString(node, buffer);
    puts(buffer);
    printTreePrivate(node->right, buffer);
  }
}
void newPrintTree(const Node *node) {
  char buffer[1000];
  printTreePrivate(node, buffer);
}
```

Esercizio 37 Abbiamo sostenuto che un processo dicotomico è $O(\lg(n))$. Potete dimostrarlo?

Risposta Esistono un paio di possibilità. Una è ribaltare il problema. Se l'array ha solo un elemento, non dobbiamo iterare nel ciclo. Ogni ulteriore iterazione fa raddoppiare le dimensioni dell'array su cui possiamo ricercare. La formula generale per le dimensioni dell'array è perciò $n = 2m$, dove m è il numero delle iterazioni. Se si prendono i logaritmi in base 2 di entrambi i membri, si ottiene $\lg(n) = \lg(2m)$, che per la definizione di logaritmo diventa $\lg(n) = m$).

Esercizio 38 Il codice seguente è stato ovviamente aggiornato varie volte nel corso degli anni, ma i cambiamenti non ne hanno certo migliorato la struttura. Sottoponetelo a refactoring.

```
if (state == TEXAS) {
  rate = TX_RATE;
  amt = base * TX_RATE;
  calc = 2*basis(amt) + extra(amt)*1.05;
}
else if ((state == OHIO) || (state == MAINE)) {
  rate = (state == OHIO) ? OH_RATE : ME_RATE;
  amt  = base * rate;
  calc = 2*basis(amt) + extra(amt)*1.05;
  if (state == OHIO)
    points = 2;
}
else {
  rate = 1;
  amt = base;
  calc = 2*basis(amt) + extra(amt)*1.05;
}
```

Risposta Qui possiamo suggerire una ristrutturazione leggera: verificare che ogni test venga eseguito solamente una volta, e rendere comuni di tutti i calcoli. Se l'espressione `2*basis(…)*1.05` appare in altri punti del programma, probabilmente dovremmo farne una funzione. Qui non ce ne siamo preoccupati.

Abbiamo aggiunto un array `rate_lookup`, inizializzato in modo che le voci diverse da Texas, Ohio e Maine abbiano un valore `1`. Questa impostazione rende facile aggiungere valori per altri stati in futuro. A seconda dello schema d'uso atteso, potremmo rendere anche il campo `points` un array di consultazione.

```
rate = rate_lookup[state];
amt  = base * rate;
calc = 2*basis(amt) + extra(amt)*1.05;
if (state == OHIO)
  points = 2;
```

Esercizio 39 La classe Java che segue deve supportare qualche altra forma. Effettuate il refactoring della classe per prepararla all'aggiunta.

```
public class Shape {
  public static final int SQUARE = 1;
  public static final int CIRCLE = 2;
  public static final int RIGHT_TRIANGLE = 3;
  private int shapeType;
  private double size;
  public Shape(int shapeType, double size) {
    this.shapeType = shapeType;
    this.size = size;
}
// ... altri metodi ...

  public double area() {
    switch (shapeType) {
    case SQUARE:   return size*size;
    case CIRCLE:   return Math.PI*size*size/4.0;
    case RIGHT_TRIANGLE: return size*size/2.0;
    }
    return 0;
  }
}
```

Risposta Quando vedete qualcuno che usa tipi enumerati (o il loro equivalente in Java) per distinguere tra varianti di uno stesso tipo, potete spesso migliorare il codice mediante il subclassing:

```
public class Shape {
  private double size;
  public Shape(double size) {
    this.size = size;
  }
  public double getSize() { return size; }
}
public class Square extends Shape {
  public Square(double size) {
    super(size);
}
  public double area() {
    double size = getSize();
    return size*size;
  }
}
public class Circle extends Shape {
  public Circle(double size) {
    super(size);
  }
  public double area() {
    double size = getSize();
    return Math.PI*size*size/4.0;
  }
}
// ecc...
```

Esercizio 40 Questo codice Java fa parte di un framework che verrà usato in tutto il vostro progetto. Sottoponetelo a refactoring in modo che sia più generale e che sia più facile estenderlo in futuro.

```
public class Window {
  public Window(int width, int height) { ... }
  public void setSize(int width, int height) { ... }
  public boolean overlaps(Window w) { ... }
  public int getArea() { ... }
}
```

Risposta Questo è un caso interessante. A prima vista, sembra ragionevole che una finestra abbia una larghezza e un'altezza. Pensate però al futuro. Immaginiamo di voler supportare finestre di forma arbitraria: sarà difficile, se la classe `Window` sa tutto sui rettangoli e le loro proprietà. Suggeriamo di astrarre la forma della finestra dalla classe `Window` stessa.

```
public abstract class Shape {
  // ...
  public abstract boolean overlaps(Shape s);
  public abstract int getArea();
}
public class Window {
  private Shape shape;
  public Window(Shape shape) {
    this.shape = shape;
    ...
  }
  public void setShape(Shape shape) {
    this.shape = shape;
    ...
  }
  public boolean overlaps(Window w) {
    return shape.overlaps(w.shape);
  }
  public int getArea() {
    return shape.getArea();
  }
}
```

Notate che con questa impostazione abbiamo usato la delega anziché il subclassing: una finestra non è un "tipo di" forma - una finestra "ha una" forma. Usa una forma per svolgere il suo compito. Spesso nel refactoring la delega si rivelerà utile.

Avremmo anche potuto estendere questo esempio introducendo un'interfaccia Java che specificasse i metodi che una classe deve supportare per supportare le funzioni della forma. È una buona idea. Significa che, quando si estende il concetto di una forma, il compilatore vi avvertirà di quali classi sono state influenzate. Consigliamo di usare le

interfacce in questo modo quando si delegano tutte le funzioni di qualche altra classe.

Esercizio 41 Progettate una sessione di test per l'interfaccia del miscelatore descritta nella risposta all'Esercizio 17. Scrivete uno script di shell che esegua un test di regressione per il miscelatore. Dovete sottoporre a test funzionalità base, condizioni di errore e al limite e tutti gli obblighi contrattuali. Quali restrizioni sono imposte ai cambiamenti di velocità? Sono rispettate?

Risposta Innanzitutto, aggiungiamo una `main` che funga da pilota dello unit test. Accetterà come argomento un linguaggio molto piccolo e molto semplice: `E` per svuotare il miscelatore, `F` per riempirlo, le cifre `0-9` per stabilire la velocità e così via.

```
public static void main(String args[]) {
  // Crea il miscelatore da sottoporre a test
  dbc_ex blender = new dbc_ex();
  // E lo testa in base alla stringa nell'input standard
  try {
    int a;
    char c;
    while ((a = System.in.read()) != -1) {
      c = (char)a;
      if (Character.isWhitespace(c)) {
        continue;
      }
      if (Character.isDigit(c)) {
        blender.setSpeed(Character.digit(c, 10));
      }
      else {
        switch (c) {
          case 'F':  blender.fill();
                     break;
          case 'E':  blender.empty();
                     break;
          case 's':  System.out.println("SPEED: " + blender.getSpeed());
                     break;
          case 'f':  System.out.println("FULL " + blender.isFull());
                     break;
          default:   throw new RuntimeException(
                       "direttiva di test sconosciuta");
          }
        }
      }
  }
  catch (java.io.IOException e) {
    System.err.println("Test jig fallito: " + e.getMessage());
  }
  System.err.println("Miscelazione completata\nn");
  System.exit(0);
}
```

Poi c'è lo script di shell per pilotare i test.

```
#!/bin/sh
CMD="java dbc.dbc_ex"
failcount=0
expect_okay() {
    if echo "$*" | $CMD #>/dev/null 2>&1
    then
      :
    else
      echo "FAILED! $*"
      failcount='expr $failcount + 1'
    fi
}
expect_fail() {
    if echo "$*" | $CMD >/dev/null 2>&1
    then
      echo "FALLITO! (Deve essere fallito): $*"
      failcount='expr $failcount + 1'
    fi
}
report() {
  if [ $failcount -gt 0 ]
  then
    echo -e "nnnn*** HA FALLITO $failcount TESTSnn"
    exit 1 # Nel caso siamo parte di qualcosa di più grande
  else
    exit 0 # Nel caso siamo parte di qualcosa di più grande
  fi
}
#
# Inizia I test
#
expect_okay F123456789876543210E # Dovrebbe andare fino in fondo
expect_fail F5       # Fallisce, velocità troppo alta
expect_fail 1        # Fallisce, vuoto
expect_fail F10E1    # Fallisce, vuoto
expect_fail F1238    # Fallisce, salta
expect_okay FE       # Mai attivare
expect_fail F1E      # Svuota in esecuzione
expect_okay F10E     # Dovrebbe essere ok
report               # Comunica i risultati
```

I test controllano se ci sono cambiamenti di velocità non leciti, se si tenta di svuotare il miscelatore mentre è acceso e così via. Mettiamo tutto nel makefile in modo da poter compilare ed eseguire il test di regressione semplicemente scrivendo

```
% make
% make test
```

Notate che facciamo uscire i test con `0` o `1` in modo da poter usare tutto questo anche nell'ambito di un test più grande.

Non c'era niente nei requisiti che parlava di pilotare questo componente attraverso uno script, o addirittura di usare un linguaggio. Gli utenti finali non lo vedranno mai. Ma noi abbiamo uno strumento

potente che possiamo usare per sottoporre il nostro codice a test in modo rapido ed esaustivo.

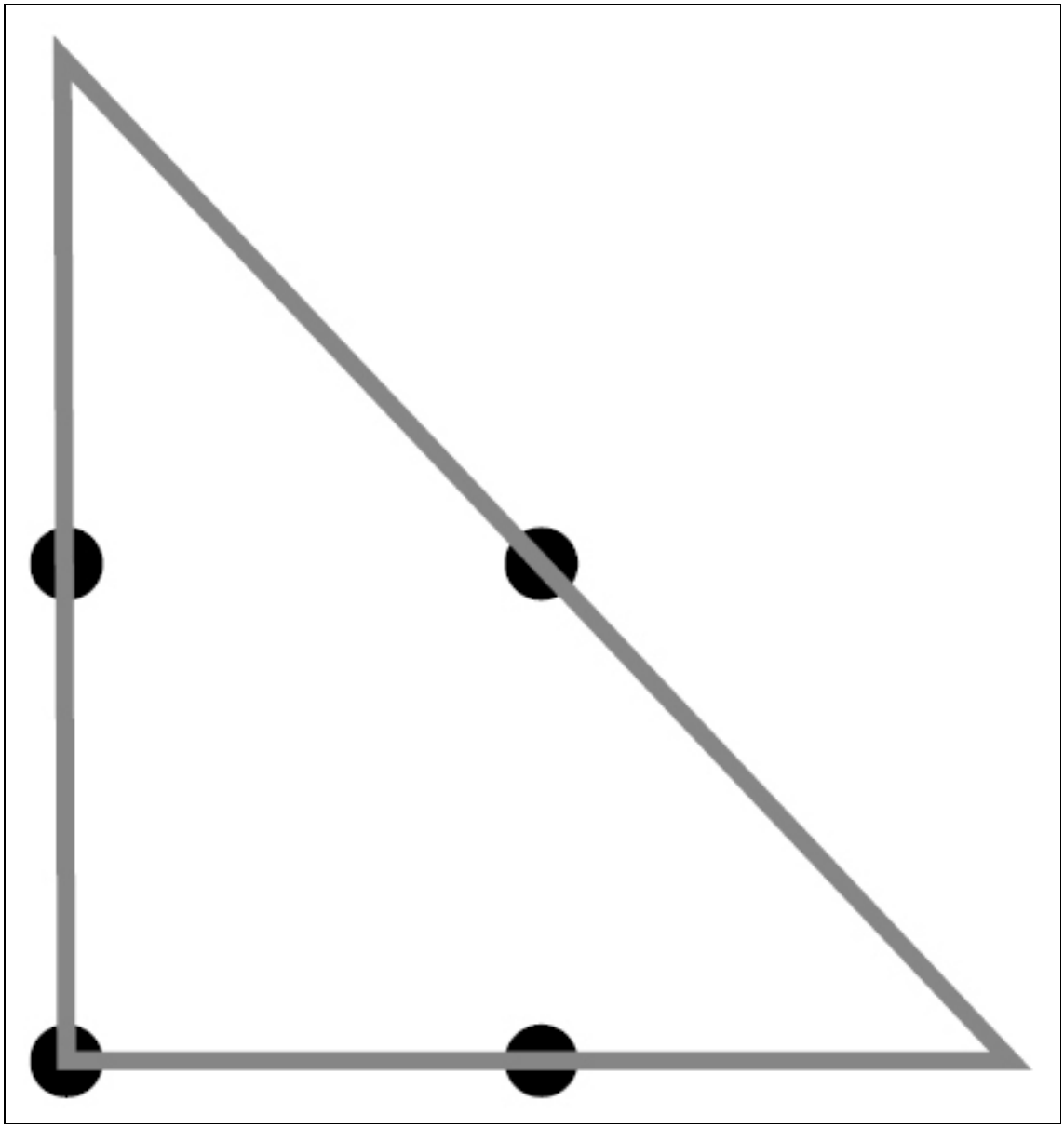
Esercizio 42 Quali fra i seguenti sono probabilmente requisiti genuini? Riformulate quelli che non lo sono in modo da renderli più utili (se possibile).

1. Il tempo di risposta deve essere inferiore a 500 millisecondi.
2. Le finestre di dialogo avranno uno sfondo grigio.
3. L'applicazione sarà organizzata sotto forma di una serie di processi di front-end e un server di back-end.
4. Se un utente inserisce caratteri non numerici in un campo numerico, il sistema emetterà un avviso acustico e non li accetterà.
5. Il codice dell'applicazione e i dati devono stare in 256 kB,

Risposta

1. Questo enunciato sembra un vero requisito: è possibile che l'ambiente determini dei vincoli per l'applicazione.
2. Anche questo può essere uno standard aziendale, non è un requisito. Sarebbe stato formulato meglio come "Lo sfondo delle finestre di dialogo deve essere configurabile dall'utente finale. Senza configurazione, il colore sarà grigio". Ancora migliore sarebbe stato un enunciato ancora più generale: "Tutti gli elementi visivi dell'applicazione (colori, tipi di caratteri e lingue) devono essere configurabili dall'utente finale".
3. Questo enunciato non è un requisito, è architettura. Di fronte a qualcosa del genere, bisogna scavare a fondo per scoprire che cosa sta pensando l'utente.
4. Il requisito sottostante probabilmente è qualcosa di più vicino a "Il sistema impedirà all'utente di inserire valori non validi nei campi, e avvertirà l'utente quando vengono inseriti valori non validi".
5. Questo enunciato è probabilmente un requisito rigido.

Soluzione al rompicapo dei quattro punti di Risolvere rompicapo impossibili, Capitolo 7:



Indice