Progettazione orientata agli oggetti

Obiettivi del capitolo

- Conoscere il ciclo di vita del software
- Imparare a individuare nuove classi e metodi
- Capire l'utilizzo delle schede CRC per l'identificazione di classi
- Saper individuare le relazioni di ereditarietà, aggregazione e dipendenza tra le classi
- Usare al meglio i diagrammi UML per descrivere le relazioni tra classi
- Imparare a usare la progettazione orientata agli oggetti per programmi complessi

Per realizzare con successo un sistema software, che si tratti di una cosa semplice come la prossima esercitazione che svolgerete a casa o di un progetto complesso come un nuovo sistema per il controllo del traffico aereo, è indispensabile un certo impegno per la pianificazione, la progettazione e il collaudo. In effetti, per i progetti di maggiori dimensioni, la quantità di tempo che si dedica alla pianificazione supera di gran lunga il tempo che si impiega per la programmazione e il collaudo.

Se vi accorgete che trascorrete davanti al computer la maggior parte del tempo che dedicate alle esercitazioni, digitando codice sorgente e correggendo errori, vuol dire che state probabilmente impiegando per le vostre esercitazioni più tempo di quello che dovreste: potreste ridurre l'impegno complessivo dedicando più tempo alla fase di progettazione e pianificazione. Questo capitolo vi spiega come affrontare questi compiti in modo sistematico.

1 Il ciclo di vita del software

Il ciclo di vita del software comprende tutte le fasi, dall'analisi iniziale all'obsolescenza.

In questo paragrafo ci occuperemo del *ciclo di vita del software*: le attività che si svolgono dal momento in cui un programma software viene concepito per la prima volta a quello in cui viene ritirato definitivamente dalla produzione.

Un procedimento formale per lo sviluppo del software descrive le fasi del processo di sviluppo e fornisce linee guida su come portare a termine ciascuna fase.

Di solito un progetto software inizia perché un cliente ha qualche problema ed è disposto a pagare per farselo risolvere. Il Dipartimento della Difesa degli Stati Uniti, cliente di molti progetti di programmazione, fu uno dei primi a proporre un *procedimento formale* per lo sviluppo del software. Un procedimento formale identifica e descrive diverse fasi e fornisce linee guida su come procedere in queste fasi e quando passare da una fase alla successiva.

Molti ingegneri del software scompongono il processo di sviluppo nelle seguenti cinque fasi:

- Analisi
- Progettazione
- Implementazione (o realizzazione)
- Collaudo
- Installazione

Nella fase di *analisi* si decide *che cosa* il progetto dovrebbe realizzare; non si pensa ancora a *come* il programma realizzerà i suoi obiettivi. Il prodotto della fase di analisi è un *elenco di requisiti*, che descrive in tutti i particolari che cosa il programma sarà in grado di fare una volta che sarà stato portato a termine. Tale elenco di requisiti potrà essere in parte costituito da un manuale per l'utente, che dice in che modo l'utente utilizzerà il programma per fare quanto preannunciato. Un'altra parte del documento imposterà criteri per le prestazioni: quanti dati in ingresso il programma dovrà essere in grado di gestire e in quali tempi, oppure quali saranno i suoi fabbisogni massimi di memoria e di spazio su disco.

Nella fase di *progettazione* si sviluppa un piano che descrive in che modo verrà realizzato il sistema e si identificano le strutture che sottendono il problema da risolvere. Se ricorrete alla progettazione orientata agli oggetti, stabilite di quali classi avrete bisogno e quali saranno i loro metodi più importanti. Il risultato prodotto da questa fase è una descrizione delle classi e dei metodi, completa di diagrammi che mostrano le relazioni fra le classi.

Nella fase di *implementazione* (o *realizzazione*) si scrive e si compila il codice sorgente per realizzare le classi e i metodi che sono stati individuati durante la fase di progettazione. Il risultato di questa fase è il programma finito.

Nella fase di *collaudo* si eseguono prove per verificare che il programma funzioni correttamente. Il risultato di questa fase è un documento scritto che descrive i collaudi che sono stati eseguiti e i loro risultati.

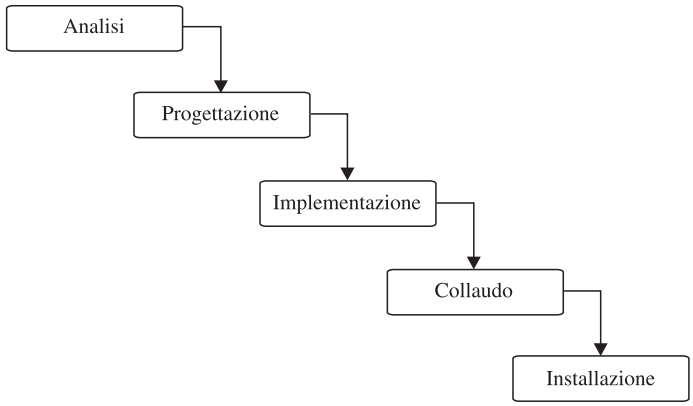
Nella fase di *installazione* gli utenti del programma lo installano e lo utilizzano per lo scopo per il quale è stato creato.

Il modello a cascata per lo sviluppo del software descrive un procedimento sequenziale di analisi, progettazione, realizzazione, collaudo e installazione.

Quando, agli inizi degli anni Settanta, vennero definiti per la prima volta i procedimenti formali di sviluppo, gli ingegneri del software avevano un modello visivo molto semplice per queste fasi. Davano per scontato che, una volta portata a termine una fase, il suo risultato si sarebbe riversato sulla fase successiva, che a quel punto si sarebbe avviata. Questo modello dello sviluppo del software, rappresentato nella Figura 1, prese il nome di *modello a cascata*.

In un mondo ideale il modello a cascata è molto attraente: prima pensate che cosa fare. poi pensate a come farlo, poi lo fate, poi verificate di averlo fatto giusto, quindi passate il prodotto al cliente. Però, quando veniva applicato rigidamente, il modello a cascata semplicemente non funzionava. Era sempre molto difficile ottenere una specifica perfetta dei requisiti. Capitava molto spesso di scoprire nella fase di progettazione che i requisiti non erano coerenti o che una leggera modifica dei requisiti avrebbe portato a un sistema che sarebbe stato al tempo stesso più facile da progettare e più utile per il cliente, ma siccome la fase di analisi era già terminata, i progettisti non avevano scelta: dovevano arrangiarsi con i requisiti che avevano ricevuto, errori compresi. Il problema si ripresentava durante l'implementazione. I progettisti erano magari convinti di conoscere il modo migliore per risolvere il problema, ma, quando il progetto veniva concretamente realizzato, il programma che ne risultava non era veloce come l'avevano pensato i progettisti. Il passo successivo è qualcosa che tutti ben conoscono. Quando il programma veniva passato al reparto "controllo qualità" per il collaudo, venivano scoperti molti bug che si sarebbero eliminati

Figura 1
Il modello a cascata

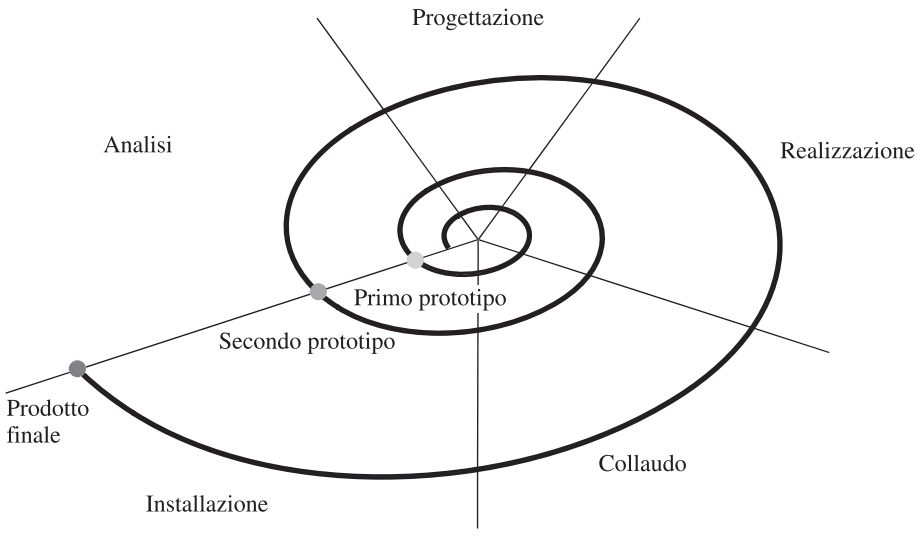


meglio rifacendo l'implementazione, o addirittura riprogettando il programma, però il modello a cascata non lo consentiva. Infine, quando i clienti ricevevano il prodotto finito, molto spesso non ne erano affatto soddisfatti. Sebbene i clienti di norma venissero molto coinvolti nella fase di analisi, loro stessi per primi non sapevano esattamente di che cosa avessero bisogno. Dopo tutto, può essere molto difficile descrivere come utilizzare un prodotto che non si è mai visto prima. Però, quando i clienti cominciavano a usare il programma, proprio allora iniziavano a rendersi conto di quello che gli sarebbe piaciuto avere. Naturalmente, arrivati a quel punto ormai era troppo tardi e dovevano arrangiarsi con quello che avevano.

Il modello a spirale per lo sviluppo del software descrive un processo iterativo in cui vengono ripetute fasi di progettazione e di realizzazione.

È chiaro che è necessaria qualche forma di iterazione: ci deve essere un meccanismo che consenta di affrontare errori che derivano dalla fase precedente. Il *modello a spirale*, proposto da Barry Boehm nel 1988, scompone il processo di sviluppo in fasi multiple (osservate la Figura 2). Le prime fasi si concentrano sulla costruzione di *prototipi*: un prototipo è un piccolo sistema che illustra alcuni aspetti del sistema finale. Dato che i prototipi costituiscono un modello soltanto di una parte di un sistema e non devono resistere agli abusi degli utilizzatori, possono essere realizzati velocemente. Molto spesso si costruisce un *prototipo dell'interfaccia utente* che mostra l'interfaccia utente in azione:

Figura 2
Il modello a spirale



questo fornisce ai clienti una prima possibilità per acquisire familiarità con il sistema e per suggerire miglioramenti prima che l'analisi sia portata a termine. Altri prototipi possono avere il compito di mettere a punto le interfacce con sistemi esterni, verificare le prestazioni, e così via. Ciò che si impara dallo sviluppo di un prototipo può essere applicato nella successiva iterazione all'interno della spirale.

Poiché costruisce il prodotto finale mediante ripetuti tentativi, un procedimento di sviluppo che segua il modello a spirale ha maggiori possibilità di mettere a punto un sistema soddisfacente. Tuttavia, c'è anche un pericolo: se i progettisti sono convinti che non sia necessario fare un buon lavoro perché possono sempre eseguire un'altra iterazione, le iterazioni diventano molto numerose e ci vuole molto tempo per portare a termine l'intero processo.

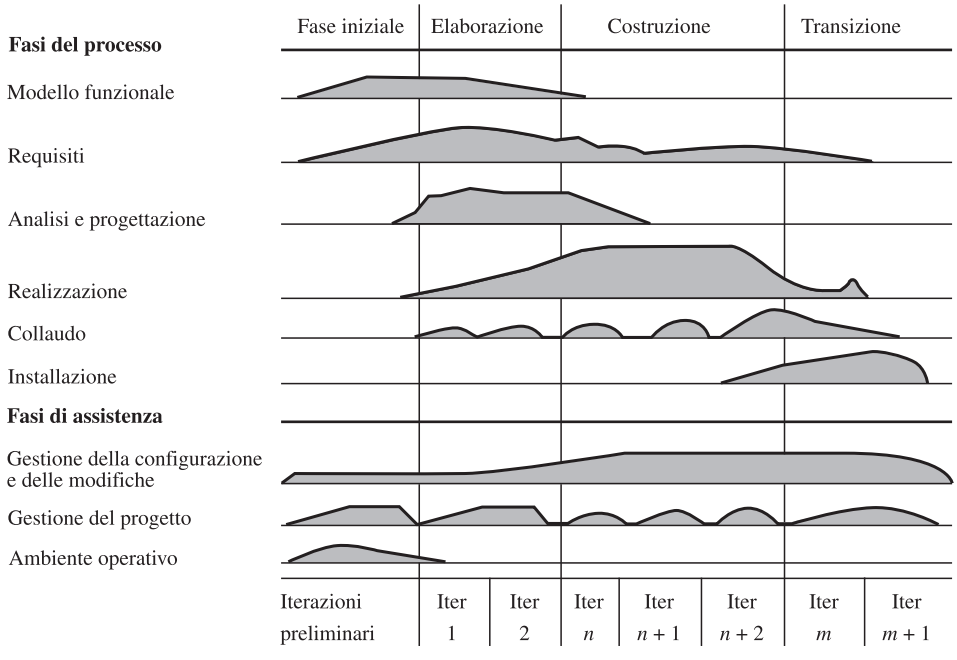
La Programmazione Estremizzata è una metodologia di sviluppo che insegue la semplicità eliminando la struttura formale e concentrandosi sulle migliori regole pratiche.

La Figura 3 mostra i livelli di attività nel "procedimento razionale unificato" ("Rational Unified Process"), una metodologia per il processo di sviluppo proposta dagli ideatori di UML. Senza soffermarvi sui dettagli, potete vedere come questo sia un procedimento complesso che coinvolge più iterazioni.

Procedimenti di sviluppo ancora più complessi, con molte iterazioni, non hanno ancora avuto successo. Nel 1999, Kent Beck pubblicò un testo [2], che ebbe molta risonanza, sulla Programmazione Estremizzata (*Extreme Programming*), una metodologia di sviluppo che insegue la semplicità eliminando la maggior parte dei vincoli formali di una metodologia di sviluppo tradizionale, concentrandosi invece su un insieme di *regole pratiche*:

- *Pianificazione realistica*. I clienti devono prendere le decisioni sulla funzionalità, i programmatori devono prendere le decisioni tecniche. Aggiornate il piano di sviluppo quando è in conflitto con la realtà.
- *Piccoli stati di avanzamento*. Fornite velocemente un sistema utilizzabile, poi fornite aggiornamenti in tempi brevi.
- *Metafora*. Tutti i programmatori dovrebbero condividere un semplice racconto che illustri metaforicamente il sistema in fase di sviluppo.

Figura 3
Livelli di attività nella metodologia del procedimento razionale unificato [1]



- *Semplicità*. Progettate ogni cosa in modo che sia la più semplice possibile, invece di predisporre tutto per future complessità.
- *Collaudo*. Sia i programmatori sia i clienti devono preparare casi di prova. Il sistema deve essere collaudato continuamente.
- *Riprogettazione*. I programmatori devono continuamente ristrutturare il sistema per migliorare il codice ed eliminare parti duplicate.
- *Programmazione a coppie*. I programmatori devono lavorare a coppie e ciascuna coppia deve scrivere codice su un unico calcolatore.
- *Proprietà collettiva*. Tutti i programmatori devono poter modificare qualsiasi porzione di codice quando ne hanno bisogno.
- *Integrazione continua*. Non appena un problema è risolto, mettete insieme l'intero sistema e collaudatelo.
- *Settimana di 40 ore*. Non usate piani di lavoro poco realistici, riempiendoli di sforzi eroici.
- *Cliente a disposizione*. Un vero utilizzatore del sistema deve essere disponibile in qualsiasi momento per la squadra di progettazione.
- *Standard per la scrittura del codice*. I programmatori devono seguire degli standard di codifica che pongano l'accento sul codice auto-documentato.

Molte di queste regole pratiche vengono dal senso comune, mentre altre, come il requisito di programmare a coppie, sono sorprendenti. Beck afferma che la potenza della Programmazione Estremizzata sta nella sinergia fra queste regole pratiche: la somma è maggiore degli addendi.

Nel vostro primo corso di programmazione, non svilupperete ancora sistemi così complessi da richiedere una vera e propria metodologia per risolvere le esercitazioni che vi vengono assegnate, ma questa presentazione del procedimento di sviluppo dovrebbe farvi capire che un procedimento vincente per lo sviluppo del software non consiste soltanto di scrittura del codice. Nella restante parte di questo capitolo entreremo in maggiore dettaglio nella *fase di progettazione* del procedimento di sviluppo del software.

Auto-valutazione

1. Supponete di aver firmato un contratto che vi impegni, per un prezzo concordato preventivamente, a progettare, realizzare e collaudare un pacchetto software che risponda in modo preciso alle specifiche contenute in un documento. Qual è il principale rischio corso da voi e dal vostro cliente in seguito a tale accordo?
2. La Programmazione Estremizzata segue un modello a cascata o a spirale?
3. Qual è l'utilità del "cliente a disposizione" nella Programmazione Estremizzata?

Note di cronaca 1

Produttività dei programmatori

Se parlate con i vostri colleghi di questo corso di programmazione, noterete che alcuni di loro portano sempre a termine le loro esercitazioni più rapidamente di altri. Forse hanno più esperienza. Tuttavia, persino quando si mettono a confronto programmatori con la stessa preparazione professionale e la stessa esperienza, si osservano e si possono misurare ampie variazioni nelle competenze. Non è insolito che, in un gruppo di programmatori, il migliore sia da cinque a dieci volte più produttivo del peggiore, quale che sia lo strumento utilizzato per misurare la produttività [3].

Un ventaglio di prestazioni così ampio fra professionisti qualificati è qualcosa di sconcertante. Dopo tutto, chi arriva primo in una maratona non corre da cinque a dieci volte più velocemente dell'atleta più lento. I responsabili dei prodotti software sono ben consapevoli di queste disparità. La soluzione ovvia, naturalmente, consiste nell'assumere soltanto i programmatori migliori, ma persino in periodi di ristagno dell'economia la domanda di buoni programmatori ha di gran lunga superato l'offerta.

Fortunatamente per tutti noi, rientrare nei ranghi dei migliori non è necessariamente una questione di mera potenza intellettuale. Il buon senso, l'esperienza, l'ampiezza delle conoscenze, la cura per i particolari e la capacità di pianificare sono importanti almeno quanto la brillantezza mentale. Le persone che sono genuinamente interessate a migliorare se stesse sono in grado di acquisire queste competenze.

Persino il più dotato fra i programmatori riesce ad affrontare soltanto un numero limitato di problemi in un determinato periodo di tempo. Supponiamo che un programmatore sia in grado di realizzare e collaudare un metodo ogni due ore, ovvero cento metodi al mese (questa è una stima parecchio generosa: sono pochi i programmatori così produttivi). Se un progetto richiede 10 000 metodi (il che è normale per un programma di medie dimensioni), allora un solo programmatore avrebbe bisogno di 100 mesi per portare a termine il lavoro. In casi del genere si suole parlare di un progetto da "100 mesi/uomo". Però, come spiega Fred Brooks nel suo famoso libro [4], il concetto di "mese/uomo" è un mito. Mesi e programmatori non sono intercambiabili: cento programmatori non possono finire il lavoro in un solo mese, e 10 programmatori probabilmente non riuscirebbero a completarlo in 10 mesi. Per prima cosa, i dieci programmatori hanno bisogno di conoscere il progetto prima di diventare produttivi. Tutte le volte che un metodo presenta qualche problema, sia il suo autore sia i suoi utenti hanno bisogno di riunirsi e di discuterne, il che porta via tempo a tutti. Un errore in un solo metodo può costringere tutti i suoi utenti a girare i pollici nell'attesa che venga corretto.

È difficile stimare questi inevitabili contrattempi. Questa è una delle ragioni per le quali il software viene spesso messo sul mercato in ritardo rispetto alle date promesse. Che cosa può fare un dirigente quando i ritardi si accumulano? Come sottolinea Brooks, aggiungendo altro personale si fa ritardare ulteriormente un progetto che è già in ritardo, perché le persone produttive devono smettere di lavorare per addestrare i nuovi arrivati.

Avrete modo di sperimentare questi problemi quando lavorerete con altri studenti al vostro primo progetto di gruppo. Siate preparati a una vistosa caduta della produttività e ricordatevi di riservare una quota importante del tempo per le comunicazioni all'interno del gruppo.

Non esiste, comunque, alternativa al lavoro di gruppo. La maggior parte dei progetti importanti e che vale la pena di realizzare trascende le capacità di qualunque singola persona. Imparare a lavorare bene in gruppo è importante per la vostra crescita culturale come lo è essere un programmatore competente.

2 Identificare le classi

Nella fase di progettazione dello sviluppo del software, il vostro compito consiste nell'identificazione delle strutture che rendono possibile implementare una serie di operazioni su un computer. Quando utilizzate il processo di progettazione orientato agli oggetti, svolgete le seguenti operazioni:

Nella progettazione orientata agli oggetti dovete identificare le classi, determinare i loro compiti e descriverne le relazioni.

1. Identificare le classi
2. Determinare il comportamento di ciascuna classe
3. Descrivere le relazioni fra le classi

Una classe rappresenta un concetto utile: avete visto classi per entità concrete quali conti correnti bancari, ellissi e prodotti; altre classi rappresentano concetti astratti, come flussi e finestre.

Una semplice regola per trovare le classi consiste nel cercare i *sostantivi* nella descrizione dell'attività. Per esempio, supponete di dover stampare una fattura come quella riprodotta nella Figura 4. Le classi ovvie che vengono subito in mente sono `Invoice` (fattura), `LineItem` (articolo) e `Customer` (cliente). È bene creare un elenco di *classi possibili* su una lavagna o su un foglio di carta. Mentre fate questo esercizio mentale, inserite nell'elenco tutte le idee che vi vengono sulle classi possibili: potrete sempre cancellare in un secondo tempo quelle che non si sono dimostrate utili.

Durante l'identificazione delle classi, ripassate mentalmente i seguenti punti:

- Una classe rappresenta un insieme di oggetti aventi lo stesso comportamento. Le entità che ricorrono più volte nella descrizione del problema, come clienti o prodotti, sono buoni candidati per divenire oggetti: scoprite cosa hanno in comune e progettate classi che racchiudano tali elementi comuni.
- Alcune entità potrebbero essere rappresentate da oggetti, altre da tipi primitivi. Ad esempio, un indirizzo deve essere un esemplare di una classe `Address` oppure può essere, semplicemente, una stringa? Non esiste un'unica risposta: dipende dal problema che si deve risolvere. Se il software ha il compito di analizzare indirizzi (per determinare, ad esempio, i costi di spedizione), allora la scelta progettuale migliore è quella di utilizzare una classe `Address`, ma se il vostro programma non dovrà mai avere tali potenzialità, non dovreste perdere tempo in una progettazione inutilmente complessa. Il vostro compito consiste nella messa a punto di un progetto equilibrato, che non sia troppo vincolato, ma nemmeno eccessivamente generico.
- Non tutte le classi possono essere identificate nella fase di analisi. La maggior parte dei programmi più complessi necessita di classi con compiti ausiliari, come l'accesso a file o a basi di dati, interfacce utente, meccanismi di controllo e così via.
- Alcune delle classi di cui avete bisogno potrebbero già esistere nella libreria standard o in un programma che avete sviluppato in precedenza. Può anche darsi che possiate utilizzare l'ereditarietà per estendere classi preesistenti, creando le classi che soddisfano le vostre esigenze.

Figura 4
Una fattura

```
F A T T U R A

Piccoli Elettrodomestici Aldo
via Nuova, 100
Turbigo, MI 20029

===============================================

Articolo              Q.tà    Prezzo    Totale

Tostapane              3      € 29,95    € 89,85
Asciugacapelli         1      € 24,95    € 24,95
Spazzola elettrica     2      € 19,99    € 39,98

===============================================

IMPORTO DOVUTO:   € 154,78
```

Una volta che avrete identificato un insieme di classi, dovrete definire il comportamento di ciascuna di esse, decidendo quali metodi ciascun oggetto debba eseguire per risolvere il problema di programmazione. Una semplice regola per trovare questi metodi consiste nel cercare i *verbi* nella descrizione dell'attività e quindi mettere i verbi in corrispondenza con gli oggetti appropriati. Per esempio, nel programma che gestisce le fatture, qualche classe deve calcolare l'importo dovuto dal cliente. A questo punto dovete stabilire *quale classe* sia responsabile di questo metodo. Sono i clienti a calcolare quello che devono pagare? Sono le fatture a totalizzare l'importo dovuto? Sono gli articoli che si totalizzano da soli? La scelta migliore consiste nell'assegnare alla classe `Invoice` la responsabilità di "calcolare l'importo dovuto".

Una scheda CRC descrive una classe, le sue responsabilità e le classi che collaborano con essa.

Un ottimo modo per svolgere questo compito è il cosiddetto metodo delle schede *CRC* (CRC sta per *c*lassi, *r*esponsabilità, *c*ollaboratori). Nella sua forma più semplice, il metodo funziona nel modo seguente: utilizzate una scheda di cartoncino per ciascuna *classe* (osservate la Figura 5); quando riflettete, nella descrizione dell'attività, sui verbi che indicano metodi, prendete la scheda della classe che secondo voi dovrebbe averne la responsabilità e scrivete quella *responsabilità* sulla scheda; per ciascuna responsabilità, annotate quali altre classi sono necessarie per portarla a termine, diventando così *collaboratori*.

Per esempio, supponete di aver stabilito che una fattura deve calcolare l'importo dovuto. Scrivete allora "`calcola importo dovuto`" sulla colonna di sinistra della scheda che ha `Invoice` come titolo.

Se una classe svolge quel compito senza ricorrere ad altre classi, non dovete scrivere altro, ma se la classe ha bisogno dell'aiuto di altre classi, scrivete il nome di questi collaboratori sul lato destro della scheda.

Per calcolare l'importo dovuto, la fattura deve chiedere a ciascun articolo il suo prezzo. Di conseguenza, la classe `LineItem` è un suo collaboratore.

È arrivato il momento di esaminare la scheda della classe `LineItem`. Ha un metodo "`fornisci il prezzo totale`"? Se non lo contiene, aggiungetelo.

Come fate a sapere che siete sulla strada giusta? Per ciascuna responsabilità, chiedete a voi stessi come potrebbe essere materialmente svolta utilizzando soltanto le responsabilità che avete scritto sulle varie schede. Molti trovano comodo sistemare sul tavolo tutte le schede, in modo che i collaboratori siano uno vicino all'altro, e poi simulare le attività spostando un contrassegno (una moneta va benissimo) da una scheda alla successiva per indicare quale oggetto è attivo in un certo istante.

Ricordatevi sempre che le responsabilità che avete elencato sulla scheda sono *ad alto livello*: a volte una singola responsabilità dovrà essere realizzata con due o più metodi Java. Alcuni ricercatori sostengono che una scheda CRC non dovrebbe avere più di tre diverse responsabilità.

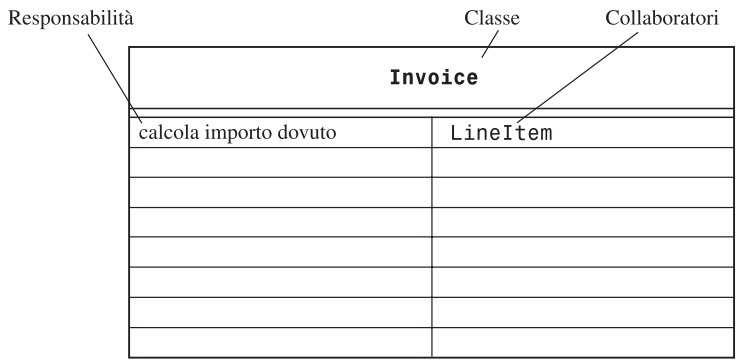


Figura 5
Una scheda CRC

Il metodo delle schede CRC è deliberatamente informale, per consentirvi di essere creativi mentre identificate le classi e le loro proprietà. Quando avrete messo insieme un buon gruppo di classi, vi converrà capire in che modo sono correlate l'una con l'altra. Siete in grado di trovare classi che hanno proprietà comuni, per cui alcune responsabilità possono essere svolte da una superclasse comune? Riuscite a organizzare le classi in aggregati indipendenti l'uno dall'altro? Trovare queste relazioni fra le classi e documentarle con diagrammi è il tema del prossimo paragrafo.

Auto-valutazione

4. Indicate un potenziale collaboratore di una fattura che debba essere memorizzata in un file.
5. In relazione alla fattura rappresentata nella Figura 4, qual è una plausibile responsabilità della classe `Customer`?
6. Cosa potete fare se una scheda CRC viene ad avere dieci responsabilità?

3 Relazioni fra classi

Quando si progetta un programma, torna utile documentare le relazioni fra le classi; così facendo, otterrete una serie di vantaggi. Per esempio, se trovate classi caratterizzate da un comportamento comune, potete risparmiarvi un po' di fatica collocando in una superclasse il loro comportamento comune. Se sapete che certe classi *non* sono correlate tra loro, potete assegnare a programmatori diversi il compito di implementare ciascuna di esse, senza avere la preoccupazione che uno di loro debba aspettare l'altro.

In questo libro avete visto molte volte la relazione di ereditarietà fra classi. L'ereditarietà è una relazione fra classi molto importante, ma, come si vedrà, non è l'unica relazione utile e si può anche correre il rischio di abusarne.

L'ereditarietà è una relazione fra una classe più generale (la superclasse) e una classe più specializzata (la sottoclasse). In questi casi si usa dire che si tratta di una relazione *è-un*. Ogni autocarro *è un* veicolo. Ogni conto corrente *è un* conto bancario. Ogni cerchio *è una* ellisse (con altezza e larghezza uguali).

L'ereditarietà (la relazione *è-un*) viene a volte usata a sproposito, quando invece una relazione *ha-un* sarebbe più opportuna.

Tuttavia, spesso si abusa dell'ereditarietà. Per esempio, prendete in considerazione la classe `Tire`, che descrive un pneumatico di automobile. Dovremmo considerare la classe `Tire` come una sottoclasse di `Circle`? A prima vista sembra comodo, perché vi sono diversi metodi utili nella classe `Circle`: per esempio, la classe `Tire` erediterebbe i metodi che calcolano il raggio, la circonferenza e il punto centrale. Tutte cose che potrebbero venir buone quando si disegnano forme di pneumatici. Eppure, anche se potrebbe essere comodo per il programmatore, questa impostazione non ha senso dal punto di vista concettuale. Non è vero che ogni pneumatico è un cerchio. I pneumatici sono componenti delle automobili, mentre i cerchi sono oggetti geometrici. Esiste tuttavia una relazione fra pneumatici e cerchi: un pneumatico *ha* un cerchio come suo perimetro esterno. Java ci consente di fare un modello anche di tale relazione, utilizzando una variabile di esemplare:

```
class Tire
{
   ...
   private String rating;
   private Circle boundary;
}
```

Il termine tecnico per questa relazione è *aggregazione*: ogni oggetto di tipo `Tire` aggrega un oggetto di tipo `Circle`. In generale, una classe aggrega un'altra classe se i suoi esemplari contengono esemplari di tale altra classe.

L'aggregazione (la relazione *ha-un*) indica che gli esemplari di una classe contengono riferimenti a esemplari di un'altra classe.

Ecco un altro esempio: ogni automobile *è un* veicolo e ogni automobile *ha un* pneumatico (in realtà ne ha quattro, cinque se contate anche la ruota di scorta). Di conseguenza, utilizzerete l'ereditarietà dalla classe `Vehicle` e l'aggregazione con esemplari di `Tire`:

```
class Car extends Vehicle
{
   ...
   private Tire[] tires;
}
```

In questo libro utilizzeremo la notazione UML per i diagrammi delle classi. Avete già visto svariati esempi della notazione UML per l'ereditarietà: una freccia con un triangolo che punta alla superclasse. Nella notazione UML, l'aggregazione viene indicata mediante una linea a tratto continuo con un simbolo simile a un diamante vicino alla classe che aggrega l'altra. La Figura 6 mostra un diagramma di classi con una relazione di ereditarietà e una relazione di aggregazione.

La dipendenza è un altro nome della relazione *usa*.

La relazione di aggregazione si ricollega alla relazione di *dipendenza*, che avete visto nel Capitolo 9. Ricordate che una classe dipende da un'altra se uno dei suoi metodi *usa* in qualche modo un esemplare di tale classe.

Per esempio, molte delle nostre applicazioni dipendono dalla classe `Scanner`, perché utilizzano un esemplare di `Scanner` per ricevere dati in ingresso.

L'aggregazione è una forma più forte di dipendenza: se una classe contiene esemplari di un'altra classe, certamente utilizza esemplari di tale altra classe. Tuttavia, il contrario non è vero. Ad esempio, una classe può utilizzare la classe `Scanner` senza definire un campo di esemplare di tipo `Scanner`: la classe può avere semplicemente una variabile locale di tipo `Scanner` oppure alcuni suoi metodi possono ricevere esemplari di `Scanner` come parametri. In questo caso non si tratta di aggregazione, perché gli esemplari della classe non contengono esemplari di `Scanner`: ne creano o ne ricevono esemplari che vengono utilizzati all'interno di un singolo metodo.

In generale, c'è bisogno di aggregazione quando un oggetto deve memorizzare altri oggetti *tra diverse invocazioni di propri metodi*.

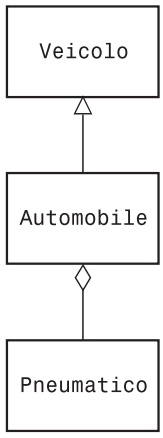


Figura 6
Notazione UML per ereditarietà e aggregazione

Nella notazione UML, occorre poter distinguere le diverse relazioni: ereditarietà, implementazione di interfaccia, aggregazione e dipendenza.

Come avete visto nel Capitolo 9, nella notazione UML la dipendenza è rappresentata mediante una linea tratteggiata con una freccia aperta che punta alla classe dipendente.

Le frecce usate nella notazione UML possono confondere; la tabella seguente riassume i quattro simboli usati nella notazione UML per rappresentare le relazioni che usiamo in questo libro.

Tabella 1
Notazione UML per le diverse relazioni

Relazione	Simbolo	Tratto	Punta della freccia
Ereditarietà		Continuo	Triangolare
Implementazione di interfaccia		Tratteggio	Triangolare
Aggregazione		Continuo	A diamante
Dipendenza		Tratteggio	Aperta

Auto-valutazione

7. Considerate le classi `Bank` e `BankAccount` del Capitolo 7. Che relazione esiste tra loro?
8. Considerate le classi `BankAccount` e `SavingsAccount` del Capitolo 11. Che relazione esiste tra loro?
9. Considerate la classe `BankAccountTester` del Capitolo 3. Da quali classi dipende?

Consigli pratici 1

Schede CRC e diagrammi UML

Prima di scrivere il codice per un problema complesso, è necessario averne progettato una soluzione. La metodologia presentata in questo capitolo suggerisce di seguire un procedimento di progettazione composto dalle seguenti fasi:

1. Identificate le classi.
2. Determinate le responsabilità di ciascuna classe.
3. Descrivete le relazioni tra le classi.

Le schede CRC e i diagrammi UML vi aiutano a identificare e a registrare tali informazioni.

Fase 1. Identificate le classi

Evidenziate i sostantivi nella descrizione del problema. Costruite un elenco dei sostantivi. Cancellate quelli che non sembrano essere ragionevoli candidati per diventare classi.

Fase 2. Determinate le responsabilità di ciascuna classe

Elencate i compiti principali che devono essere svolti dal vostro sistema. Tra di essi, sceglietene uno che non sia banale ma che vi sembri intuitivo, e identificate una classe che abbia la responsabilità di portarlo a termine. Costruite una scheda, scrivendo il nome della classe e tale sua responsabilità. Ora, chiedetevi se un oggetto di quella classe può svolgere completamente tale compito: probabilmente avrà bisogno di aiuto da altri oggetti, quindi costruite le schede CRC per le classi a cui appartengono tali oggetti e scrivetene le responsabilità.

Non abbiate timore di fare cancellature, spostamenti, suddivisioni o fusioni di responsabilità. Sistemate le schede se diventano confuse, si tratta di un procedimento poco formale.

Avrete terminato quando avrete esaminato tutti i compiti più importanti e sarete convinti di poterli risolvere con le classi e le responsabilità che avete identificato.

Fase 3. Descrivete le relazioni tra le classi

Costruite un diagramma di classi che mostri le relazioni tra le classi che avete identificato.

Iniziate con l'ereditarietà, la relazione "è-un" tra classi. Un certa classe è una forma specializzata di un'altra classe? Se è così, tracciate una freccia di ereditarietà. Tenete presente che molti progetti, soprattutto in caso di programmi semplici, non usano molto l'ereditarietà.

Le colonne "collaboratori" sulle schede CRC vi dicono quali classi ne usano un'altra. Tracciate le frecce di dipendenza per i collaboratori presenti sulle schede CRC.

Alcune relazioni di dipendenza danno luogo a un'aggregazione. Per ogni relazione di dipendenza, chiedetevi: come può l'oggetto raggiungere i propri collaboratori? Vi giunge direttamente perché ne conserva un riferimento? Oppure il collaboratore viene passato come parametro di un metodo o come valore restituito da un metodo? Soltanto nel primo caso la classe che collabora è una classe aggregata: in tali casi, tracciate una freccia di aggregazione.

Argomenti avanzati 1

Attributi e metodi nei diagrammi UML

Talvolta è utile indicare *attributi* e *metodi* in un diagramma di classe. Un *attributo* è una proprietà osservabile dall'esterno che hanno gli oggetti di una classe. Per esempio, `name` e `price` potrebbero essere attributi della classe `Product`. Di solito, gli attributi corrispondono a variabili di esemplare, però non è obbligatorio che sia così: una classe potrebbe avere un modo diverso per organizzare i suoi dati. Ad esempio, un esemplare della classe `GregorianCalendar` della libreria Java ha gli attributi `day`, `month` e `year`, ma non ha campi di esemplare che memorizzino tali valori: tutte le date vengono internamente rappresentate mediante il numero di millisecondi trascorsi dal primo gennaio del 1970, un dettaglio realizzativo che gli utenti della classe non hanno certamente bisogno di conoscere.

Metodi e attributi vengono indicati in un diagramma di classe suddividendo il rettangolo di una classe in tre settori, con il nome della classe in alto, gli attributi nel mezzo e i metodi in basso (osservate la Figura 7). Non siete obbligati a elencare proprio *tutti* gli attributi e i metodi in un diagramma: vi basta elencare quelli che vi possono essere utili per capire ciò che volete rappresentare con un tale diagramma.

Inoltre, non elencate come attributo ciò che rappresentate graficamente come aggregazione. Se indicate con un'aggregazione il fatto che un oggetto di tipo `Car` contiene oggetti di tipo `Tire`, non aggiungete l'attributo `tires` alla classe `Car`.

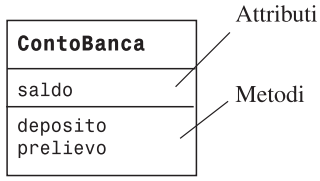


Figura 7
Attributi e metodi in un diagramma di classe.

Argomenti avanzati 2

Molteplicità

Alcuni progettisti hanno l'abitudine di indicare la *molteplicità* di ciascuna relazione di aggregazione, per rappresentare il numero di oggetti che vengono aggregati. Le notazioni più comunemente utilizzate per la molteplicità sono:

- qualsiasi numero (zero o più): `*`
- uno o più: `1..*`
- zero o uno: `0..1`
- esattamente uno: `1`

La Figura 8 evidenzia che un cliente ha uno o più conti bancari.

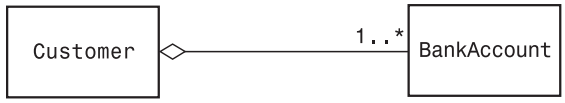


Figura 8
Una relazione di aggregazione con molteplicità

Argomenti avanzati 3

Aggregazione e associazione

Alcuni progettisti trovano che il termine aggregazione o *ha-un* non sia soddisfacente. Consideriamo, ad esempio, i clienti di una banca. La banca "ha" clienti? I clienti "hanno" conti bancari, oppure è la banca ad "averli"? Quali di queste relazioni *ha-un* dovrebbe essere modellata mediante aggregazione? Questo modo di pensare ci può portare a decisioni di implementazione premature.

Nelle prime fasi di progettazione, è sensato utilizzare una relazione tra classi più generica, denominata *associazione*. Una classe è associata a un'altra se è possibile *navigare* da oggetti di una classe a oggetti dell'altra classe. Ad esempio, dato un oggetto di tipo `Bank`, è possibile navigare verso oggetti di tipo `Customer`, accedendovi mediante un campo di esemplare oppure effettuando una ricerca in una base di dati.

La notazione UML utilizzata per una relazione di associazione è costituita da una linea continua che termina con una freccia facoltativa che indica la direzione lungo la quale si può navigare; vicino alla fine della linea è anche possibile scrivere qualcosa che spieghi meglio la natura della relazione. La Figura 9 evidenzia la possibilità di navigare da oggetti di tipo `Bank` verso oggetti di tipo `Customer`, senza poter navigare in senso opposto: in questo particolare progetto, quindi, la classe `Customer` non ha modo di determinare in quale banca siano custoditi i soldi di un cliente.

Sinceramente, le differenze fra associazione e aggregazione possono confondere anche i progettisti esperti. Se pensate che queste distinzioni siano utili, usatele senza dubbio, ma non perdete tempo pensando alle sottili differenze fra questi concetti. Dal punto di

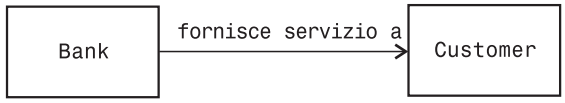


Figura 9
Una relazione di associazione

vista pratico, a un programmatore Java è utile sapere se una classe conserva un riferimento a un'altra classe, e l'aggregazione (o relazione *ha-un*) descrive accuratamente questo fatto.

4 Esempio: stampare una fattura

In questo paragrafo, presentiamo un procedimento di sviluppo articolato in cinque parti, che vi suggeriamo di seguire in quanto è particolarmente adatto a programmatori principianti:

1. Definire i requisiti.
2. Utilizzare schede CRC per identificare classi, responsabilità e collaboratori.
3. Utilizzare diagrammi UML per registrare relazioni fra le classi.
4. Utilizzare `javadoc` per documentare il comportamento dei metodi.
5. Realizzare il programma.

Non ci sono molte notazioni da imparare e i diagrammi delle classi sono facili da tracciare. Quel che si ottiene dalla fase di progettazione è palesemente utile per quella di implementazione: non si deve far altro che prendere i file sorgenti e cominciare ad aggiungere il codice per i metodi. Naturalmente, quando i vostri progetti si faranno più complessi, vi converrà imparare qualcosa di più sui metodi formali di progettazione. Esistono molte tecniche per descrivere gli scenari degli oggetti, la sequenza delle chiamate, la struttura generale dei programmi e quant'altro, che sono molto vantaggiose persino per progetti relativamente semplici. In *Unified Modeling Language User Guide* [1] potete trovare un'ottima rassegna di queste tecniche.

In questo paragrafo, esamineremo la tecnica di progettazione orientata agli oggetti applicata a un esempio molto semplice. In un caso di tale semplicità, l'uso di questa metodologia apparirà certamente un po' eccessivo, però costituisce comunque una buona introduzione alla meccanica di ciascun passaggio. Vi troverete quindi meglio preparati per l'esempio più complesso che segue.

4.1 Requisiti

Questo programma ha lo scopo di stampare una fattura. Una fattura descrive la somma dovuta per un insieme di prodotti, con le rispettive quantità (trascuriamo aspetti complessi quali le date, le imposte, i codici della fattura e dei clienti). Il programma non fa altro che stampare l'indirizzo di chi deve pagare, l'elenco degli articoli e l'importo da pagare. Ciascuna linea che descrive un articolo ne contiene la descrizione e il prezzo unitario, nonché la quantità ordinata e il prezzo totale.

```
                          I N V O I C E

Sam's Small Appliances
100 Main Street
Anytown, CA 98765

Description                    Price  Qty   Total
Toaster                        29,95   3    89,85
Hair dryer                     24,95   1    24,95
Car vacuum                     19,99   2    39,98

AMOUNT DUE: $154,78
```

Inoltre, per semplicità, non forniremo alcuna interfaccia utente. Predisporremo un programma di prova che aggiunge articoli alla fattura e la stampa.

4.2 Schede CRC

Come prima cosa dovrete individuare le classi. Le classi corrispondono a sostantivi presenti nella descrizione del problema. In questo problema, è del tutto ovvio quali siano i sostantivi:

```
Invoice
Address
LineItem
Product
Description
Price
Quantity
Total
AmountDue
```

Naturalmente, "tostapane" non conta: è il nome di un oggetto di tipo `LineItem` ed è quindi un valore, non il nome di una classe.

La descrizione di un prodotto e il suo prezzo sono campi della classe `Product`. E cosa possiamo dire della quantità? La quantità non è un attributo di un oggetto di tipo `Product`. Proprio come nella fattura stampata, è meglio avere una classe `LineItem` che memorizzi un prodotto e la relativa quantità (come, ad esempio, "tre tostapane").

Il prezzo totale e l'importo dovuto vengono calcolati, non c'è bisogno di memorizzarli, per cui non danno origine ad alcuna classe.

Dopo questa procedura di eliminazione, ci restano quattro candidati per le classi:

```
Invoice
Address
LineItem
Product
```

Ciascuno di loro rappresenta un concetto utile, quindi facciamoli diventare tutti classi.

Il programma ha lo scopo di stampare una fattura. Tuttavia, la classe `Invoice` non sa se deve visualizzare i suoi dati nel flusso `System.out`, in un campo di testo di un'interfaccia grafica, oppure in un file. Di conseguenza, è meglio semplificare leggermente l'obiettivo e assegnare alla fattura la responsabilità di *impaginare* i suoi dati. Il risultato di questa attività è una stringa (contenente più righe), che può essere stampata o visualizzata. Annotiamo questa responsabilità in una scheda CRC:

Invoice	
impaginare la fattura	

Come fa una fattura a impaginare i suoi dati? Deve impaginare l'indirizzo di chi deve pagare, poi tutti gli articoli e quindi l'importo dovuto.

Come può una fattura impaginare un indirizzo? Non può: questa è davvero una responsabilità che dobbiamo assegnare alla classe `Address`. Ciò genera una seconda scheda CRC:

Address	
impaginare l'indirizzo	

In modo simile, l'impaginazione di un articolo deve essere una responsabilità della classe `LineItem`.

Il metodo `format` della classe `Invoice` invoca i metodi `format` delle classi `Address` e `LineItem`. Ogni volta che un metodo usa un'altra classe, elencate tale classe come collaboratore. In altre parole, `Address` e `LineItem` sono collaboratori di `Invoice`:

Invoice	
impaginare la fattura	`Address`
	`LineItem`

Per impaginare la fattura, bisogna anche calcolare l'importo totale dovuto: per farlo, la fattura deve chiedere a ciascuna riga il suo prezzo totale.

Come fa una riga a sapere il proprio totale? Deve chiedere al suo prodotto il prezzo unitario e moltiplicarlo per la propria quantità. A questo scopo, la classe `Product` deve rendere accessibile il proprio prezzo unitario ed essere un collaboratore della classe `LineItem`.

Infine, deve essere possibile aggiungere alla fattura prodotti e rispettive quantità, in modo che abbia senso impaginare il risultato. Anche questa è una responsabilità della classe `Invoice`.

A questo punto abbiamo un insieme di schede CRC che conclude questa fase del procedimento.

Product	
fornire la descrizione	
fornire il prezzo unitario	

LineItem	
impaginare l'articolo	Product
fornire il prezzo totale	

Invoice	
impaginare la fattura	Address
aggiungere un prodotto e una quantità	LineItem
	Product

4.3 Diagrammi UML

Le relazioni di dipendenza si ricavano dalla colonna delle collaborazioni nelle schede CRC. Ciascuna classe dipende dalle classi con le quali collabora. Nel nostro esempio, la classe `Invoice` collabora con le classi `Address`, `LineItem` e `Product`. La classe `LineItem` collabora con la classe `Product`.

Ora, chiedetevi se queste dipendenze siano, in realtà, aggregazioni. Come fa una fattura a conoscere l'indirizzo, le righe e i prodotti con cui collabora? Un oggetto che rappresenta una fattura deve avere i riferimenti all'indirizzo e alle righe, per usarli quando impagina la fattura stessa, ma non ha bisogno di memorizzare un riferimento a un oggetto che rappresenta un prodotto, quando tale prodotto viene aggiunto alla fattura. Il prodotto viene inserito in una riga, dopodiché la responsabilità di memorizzare un riferimento al prodotto è della riga.

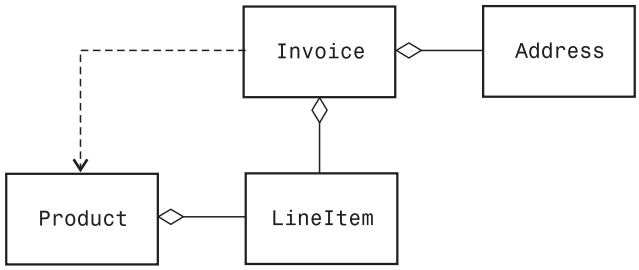


Figura 10
Le relazioni tra le classi per la gestione di fatture

Quindi, la classe `Invoice` aggrega le classi `Address` e `LineItem`, e la classe `LineItem` aggrega la classe `Product`. Tuttavia, non una relazione *ha-un* tra una fattura e un prodotto: una fattura non memorizza direttamente prodotti, che, invece, si trovano all'interno di oggetti di tipo `LineItem`.

In questo esempio non usiamo l'ereditarietà.

La Figura 10 mostra le relazioni che abbiamo individuato tra le classi.

4.4 Documentazione dei metodi

Potete usare i commenti di documentazione di `javadoc` (con i corpi dei metodi in bianco) per annotare formalmente il comportamento delle classi che avete individuato.

Il passo finale della fase di progettazione consiste nella scrittura della documentazione delle classi e dei metodi che sono stati individuati. Scrivete semplicemente un file sorgente Java per ciascuna classe, scrivendo il commento per ciascun metodo che avete identificato e lasciando in bianco i corpi dei metodi.

File Invoice.java

```
/**
   Descrive una fattura per un insieme di articoli acquistati.
*/
public class Invoice
{
   /**
      Aggiunge alla fattura l'addebito per un prodotto.
      @param aProduct il prodotto ordinato dal cliente
      @param quantity la quantità del prodotto
   */
   public void add(Product aProduct, int quantity)
   {
   }

   /**
      Impagina la fattura.
      @return la fattura impaginata
   */
   public String format()
   {
   }
}
```

File LineItem.java

```
/**
   Descrive in una riga la quantità di un articolo
   acquistato e il suo prezzo.
*/
```

```
public class LineItem
{
   /**
      Calcola il prezzo totale di questo articolo.
      @return il prezzo totale
   */
   public double getTotalPrice()
   {
   }

   /**
      Impagina la riga.
      @return la riga impaginata
   */
   public String format()
   {
   }
}
```

File Product.java

```
/**
   Descrive un prodotto avente una descrizione e un prezzo.
*/
public class Product
{
   /**
      Fornisce la descrizione del prodotto.
      @return la descrizione
   */
   public String getDescription()
   {
      return description;
   }

   /**
      Fornisce il prezzo del prodotto.
      @return il prezzo unitario
   */
   public double getPrice()
   {
      return price;
   }
}
```

File Address.java

```
/**
   Descrive un indirizzo postale.
*/
public class Address
{
   /**
      Impagina l'indirizzo.
      @return l'indirizzo sotto forma di stringa di tre righe
   */
```

```
    public String format()
    {
    }
}
```

Fatto questo, eseguite il programma `javadoc` per ottenere una versione della vostra documentazione gradevolmente impaginata in HTML, come si può vedere nella Figura 11.

Questo modo di affrontare il problema della documentazione delle classi presenta numerosi vantaggi. Potete condividere la documentazione HTML con altre persone, se lavorate in gruppo. Vi servite di un formato che è immediatamente utilizzabile: file sorgenti Java che potete trasferire alla fase di implementazione. E, quel che è più importante, avete già creato i commenti per i metodi basilari: un compito che i programmatori meno preparati tendono a rinviare e che poi spesso trascurano per mancanza di tempo.

4.5 Implementazione

Finalmente siete pronti per implementare le classi.

Grazie al passo precedente avete già a disposizione le firme e i commenti dei metodi. Ora esaminate il diagramma UML per aggiungere le variabili di esemplare, che non sono

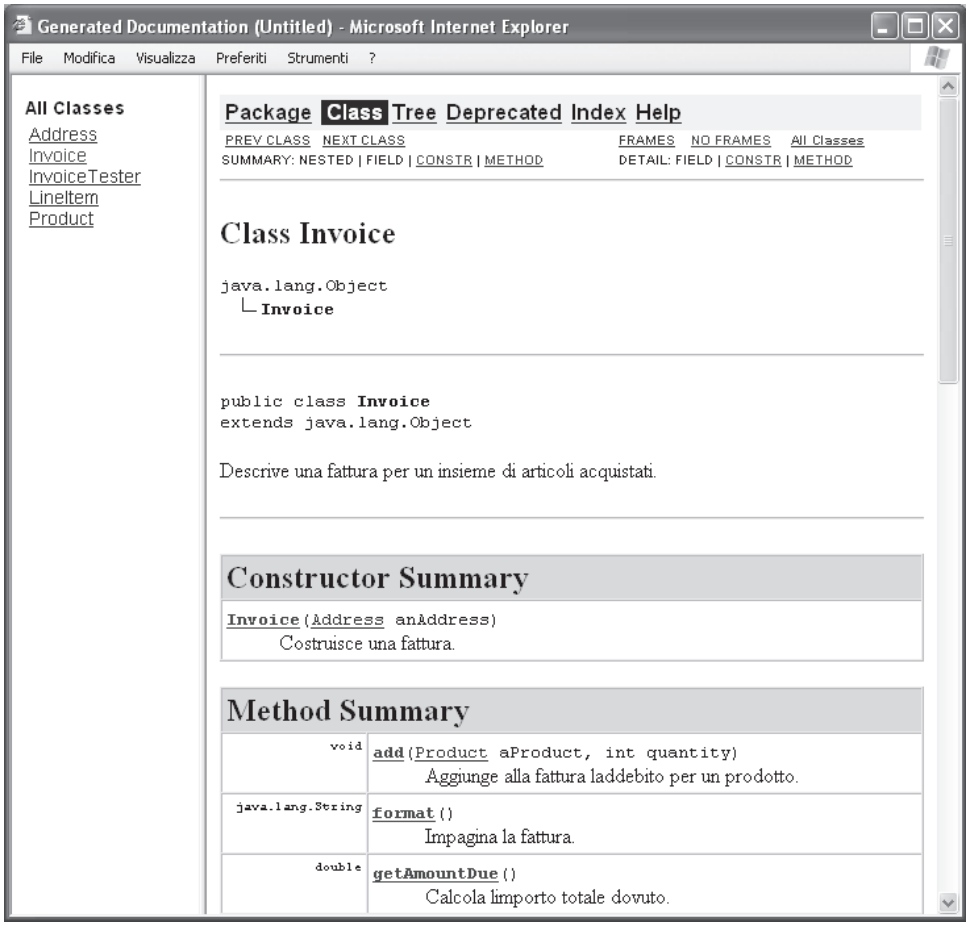


Figura 11
Documentazione di classi in formato HTML

altro che le classi aggregate. Cominciate con la classe `Invoice`, che aggrega le classi `Address` e `LineItem`. Ciascuna fattura ha un solo indirizzo di chi deve pagare, ma può avere molti articoli, che potete memorizzare in un vettore. Ecco quindi le variabili di esemplare della classe `Invoice`:

```
public class Invoice
{
   ...
   private Address billingAddress;
   private ArrayList<LineItem> items;
}
```

Una riga di articolo ha bisogno di memorizzare un oggetto di tipo `Product` e la relativa quantità, dando così origine alle seguenti variabili di esemplare:

```
public class LineItem
{
   ...
   private int quantity;
   private Product theProduct;
}
```

I metodi veri e propri, a questo punto, sono davvero semplici. Ecco un tipico esempio. Già sapete cosa deve fare il metodo `getTotalPrice` della classe `LineItem`: deve leggere il prezzo unitario del prodotto e moltiplicarlo per la quantità.

```
/**
   Calcola il prezzo totale di questo articolo.
   @return il prezzo totale
*/
public double getTotalPrice()
{
   return theProduct.getPrice() * quantity;
}
```

Non discuteremo nei particolari gli altri metodi: sono ugualmente immediati.

Infine, dovrete scrivere i costruttori, un altro compito molto semplice.

Ecco qui tutto il programma. Farete bene a scorrerlo esaminando tutti i particolari, mettendo in corrispondenza le classi e i metodi con le schede CRC e il diagramma UML.

File InvoiceTester.java

```
/**
   Questo programma collauda le classi che gestiscono
   le fatture stampando una fattura di prova.
*/
public class InvoiceTester
{
   public static void main(String[] args)
   {
      Address samAddress
            = new Address("Sam's Small Appliances",
            "100 Main Street", "Anytown", "CA", "98765");
```

```
      Invoice samsInvoice = new Invoice(samAddress);
      samsInvoice.add(new Product("Toaster", 29.95), 3);
      samsInvoice.add(new Product("Hair dryer", 24.95), 1);
      samsInvoice.add(new Product("Car vacuum", 19.99), 2);

      System.out.println(samsInvoice.format());
   }
}
```

File Invoice.java

```
import java.util.ArrayList;

/**
   Descrive una fattura per un insieme di articoli acquistati.
*/
public class Invoice
{
   /**
      Costruisce una fattura.
      @param anAddress l'indirizzo di chi deve pagare
   */
   public Invoice(Address anAddress)
   {
      items = new ArrayList<LineItem>();
      billingAddress = anAddress;
   }

   /**
      Aggiunge alla fattura l'addebito per un prodotto.
      @param aProduct il prodotto ordinato dal cliente
      @param quantity la quantità del prodotto
   */
   public void add(Product aProduct, int quantity)
   {
      LineItem anItem = new LineItem(aProduct, quantity);
      items.add(anItem);
   }

   /**
      Impagina la fattura.
      @return la fattura impaginata
   */
   public String format()
   {
      String r = "              I N V O I C E\n\n"
            + billingAddress.format()
            + String.format("\n\n%-30s%8s%5s%8s\n",
            "Description", "Price", "Qty", "Total");

      for (LineItem i : items)
      {
         r = r + i.format() + "\n";
      }

      r = r + String.format("\nAMOUNT DUE: $%8.2f", getAmountDue());
```

```
      return r;
   }

   /**
      Calcola l'importo totale dovuto.
      @return l'importo dovuto
   */
   public double getAmountDue()
   {
      double amountDue = 0;
      for (LineItem i : items)
      {
         amountDue = amountDue + i.getTotalPrice();
      }
      return amountDue;
   }

   private Address billingAddress;
   private ArrayList<LineItem> items;
}
```

File LineItem.java

```
/**
   Descrive in una riga la quantità di un articolo
   acquistato e il suo prezzo.
*/
public class LineItem
{
   /**
      Costruisce un articolo con un prodotto e una quantità.
      @param aProduct il prodotto
      @param aQuantity la quantità dell'articolo
   */
   public LineItem(Product aProduct, int aQuantity)
   {
      theProduct = aProduct;
      quantity = aQuantity;
   }

   /**
      Calcola il prezzo totale di questo articolo.
      @return il prezzo totale
   */
   public double getTotalPrice()
   {
      return theProduct.getPrice() * quantity;
   }

   /**
      Impagina la riga.
      @return la riga impaginata
   */
   public String format()
   {
      return String.format("%-30s%8.2f%5d%8.2f",
```

```
            theProduct.getDescription(), theProduct.getPrice(),
            quantity, getTotalPrice());
      }

      private int quantity;
      private Product theProduct;
   }
```

File Product.java

```
/**
   Descrive un prodotto avente una descrizione e un prezzo.
*/
public class Product
{
   /**
      Costruisce un prodotto con una descrizione e un prezzo.
      @param aDescription la descrizione del prodotto
      @param aPrice il prezzo del prodotto
   */
   public Product(String aDescription, double aPrice)
   {
      description = aDescription;
      price = aPrice;
   }
   /**
      Fornisce la descrizione del prodotto.
      @return la descrizione
   */
   public String getDescription()
   {
      return description;
   }

   /**
      Fornisce il prezzo del prodotto.
      @return il prezzo unitario
   */
   public double getPrice()
   {
      return price;
   }

   private String description;
   private double price;
}
```

File Address.java

```
/**
   Descrive un indirizzo postale.
*/
public class Address
{
   /**
      Costruisce un indirizzo postale.
```

```
      @param aName il nome del destinario
      @param aStreet la via
      @param aCity la città
      @param aState la sigla a due lettere dello stato
      @param aZip il codice postale
   */
   public Address(String aName, String aStreet,
         String aCity, String aState, String aZip)
   {
      name = aName;
      street = aStreet;
      city = aCity;
      state = aState;
      zip = aZip;
   }

   /**
      Impagina l'indirizzo.
      @return l'indirizzo sotto forma di stringa di tre righe
   */
   public String format()
   {
      return name + "\n" + street + "\n"
            + city + ", " + state + " " + zip;
   }

   private String name;
   private String street;
   private String city;
   private String state;
   private String zip;
}
```

Auto-valutazione

10. Quale classe ha la responsabilità di calcolare l'importo dovuto? Quali sono i suoi collaboratori per tale compito?
11. Per quale motivo i metodi restituiscono oggetti di tipo `String` anziché visualizzare direttamente le informazioni sul flusso `System.out`?

5 Esempio: uno sportello bancario automatico

5.1 Requisiti

Questo progetto ha lo scopo di simulare il funzionamento di uno sportello bancario automatico, basato su un terminale specializzato noto come Automatic Teller Machine (ATM). L'ATM viene usato dai clienti di una banca. Ciascun cliente ha due conti: un conto corrente e un conto di risparmio. Ciascun cliente ha anche un numero cliente e un numero di identificazione personale (Personal Identification Number, PIN): per accedere ai conti è necessario fornire entrambi i numeri (nei veri ATM, il numero cliente viene registrato nella striscia magnetica della carta bancaria; in questa simulazione il cliente dovrà digitarlo). Utilizzando l'ATM il cliente può selezionare un conto (corrente o di risparmio), dopo di che gli viene mostrato il saldo del conto che ha selezionato. Il cliente può, poi, versare o prelevare denaro. Il processo viene ripetuto fino a quando il cliente decide di terminare.

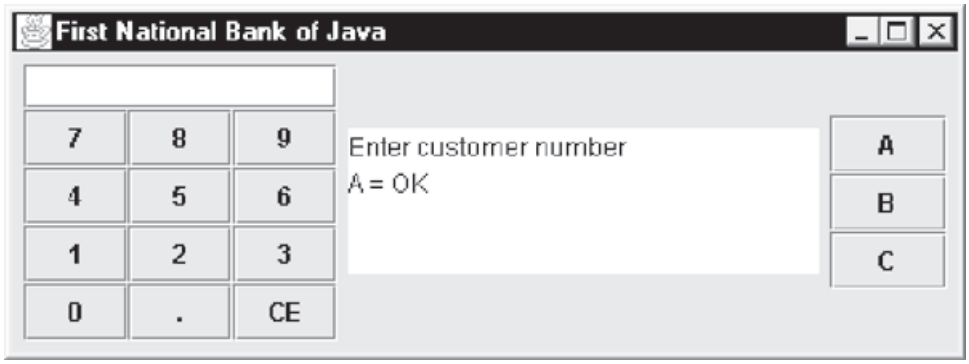


Figura 12
Interfaccia utente di un ATM

I dettagli dell'interazione dell'utente con l'ATM dipendono dall'interfaccia utente scelta per la simulazione. Svilupperemo due diverse interfacce: un'interfaccia grafica molto simile a quella di un vero ATM (vedi Figura 12) e un'interfaccia di solo testo che consentirà di collaudare l'ATM e le classi della banca senza essere distratti dalla programmazione grafica.

Nell'interfaccia grafica, l'ATM ha un tastierino per immettere numeri, uno schermo per visualizzare messaggi e un gruppo di pulsanti, etichettati A, B e C, le cui funzioni dipendono dallo stato della macchina.

In concreto, l'utente interagisce in questo modo. Quando l'ATM viene avviato, rimane in attesa che un utente immetta un numero cliente. Lo schermo presenta il seguente messaggio:

```
Enter customer number
A = OK
```

L'utente digita il numero cliente sul tastierino e preme il pulsante A. Il messaggio visualizzato diventa:

```
Enter PIN
A = OK
```

Quindi, l'utente digita il PIN e preme di nuovo il pulsante A. Se il numero cliente e il PIN corrispondono a uno dei clienti della banca, l'utente è autorizzato a procedere. In caso contrario, gli viene chiesto di nuovo il numero cliente.

Se il cliente è stato autorizzato a usare il sistema, il messaggio visualizzato diventa:

```
Select Account
A = Checking
B = Savings
C = Exit
```

Se l'utente preme il pulsante C, l'ATM torna al suo stato di partenza e chiede al successivo utente di immettere un numero cliente.

Se l'utente preme i pulsanti A o B, l'ATM ricorda il conto selezionato e il messaggio che compare assume questa forma:

```
Balance = saldo del conto selezionato
Enter amount and select transaction
```

```
A = Withdraw
B = Deposit
C = Cancel
```

Se l'utente preme il pulsante A o il pulsante B, il valore digitato sul tastierino viene prelevato dal conto selezionato o, rispettivamente, vi viene versato (questa è soltanto una simulazione e quindi non c'è movimento fisico di denaro). Successivamente, l'ATM torna al suo stato precedente, consentendo all'utente di selezionare un altro conto o di uscire.

Se l'utente preme il pulsante C, l'ATM torna al suo stato precedente senza eseguire alcuna transazione.

Con l'interazione basato soltanto su testo, leggiamo i dati in ingresso dal flusso `System.in` anziché usare pulsanti grafici. Ecco un tipico dialogo dell'applicazione:

```
Enter account number: 1
Enter PIN: 1234
A=Checking, B=Savings, C=Quit: A
Balance=0.0
A=Deposit, B=Withdrawal, C=Cancel: A
Amount: 1000
A=Checking, B=Savings, C=Quit: C
```

Con la soluzione che svilupperemo, la scelta dell'interfaccia utente ha effetti solamente sulle classi relativa all'interfaccia utente stessa. Le altre classi dell'applicazione possono essere utilizzate in entrambe le situazioni: sono, quindi, disaccoppiate dall'interfaccia utente. Trattandosi di una simulazione, l'ATM non comunica materialmente con una banca, ma si limita a caricare un gruppo di numeri cliente e di PIN da un file che li contiene. A tutti i conti viene assegnato un saldo iniziale uguale a zero.

5.2 Schede CRC

Seguiremo ancora una volta lo schema del Paragrafo 2 per mostrare come si individuano classi, responsabilità e relazioni, e come si crea una struttura particolareggiata per il programma ATM.

Ricordate che la prima regola per trovare le classi è "cercare i sostantivi nella descrizione del problema". Ecco un elenco dei sostantivi:

```
ATM
User
KeyPad
Display
Display message
Button
State
Bank account
Checking account
Savings account
Customer
Customer number
PIN
Bank
```

Naturalmente, non tutti questi sostantivi diventeranno nomi di classi e potremmo anche scoprire che ci occorrono classi che non stanno in questo elenco, ma questo è comunque un buon modo per iniziare.

In questo programma utenti e clienti rappresentano il medesimo concetto. Usiamo dunque una classe `Customer`: un cliente ha due conti bancari e un oggetto che rappresenta un cliente deve poterci dire quali sono i suoi conti (un progetto alternativo potrebbe prevedere che la classe `Bank` abbia la responsabilità di reperire i conti di un certo cliente, come vedrete nell'Esercizio P13).

Un cliente ha anche un numero cliente e un PIN. Possiamo, naturalmente, esigere che un oggetto di tipo `Customer` ci fornisca il numero cliente e il PIN, però forse in questo modo si comprometterebbe la sicurezza. Invece, limitiamoci a richiedere che un oggetto di tipo `Customer`, una volta che abbia ricevuto un numero cliente e un PIN, ci dica se tali dati corrispondono alle informazioni di cui dispone.

Customer	
fornisci i conti	
confronta numero e PIN	

Una banca contiene un insieme di clienti. Quando un utente si porta davanti all'ATM e digita numero cliente e PIN, è compito della banca trovare il cliente corrispondente. In che modo? Bisogna controllare ciascun cliente per verificare se il suo numero cliente e il PIN corrispondono. La banca ha bisogno, quindi, di invocare il metodo della classe `Customer` che *confronta numero e PIN*, quello che abbiamo appena individuato. Siccome il metodo *trova cliente* invoca un metodo di `Customer`, la classe `Bank` collabora con la classe `Customer`. Annotiamo questa circostanza sulla colonna destra della scheda CRC.

Quando la simulazione inizia, la banca deve anche poter leggere un insieme di clienti e dei relativi dati.

Bank	
trova cliente	`Customer`
leggi i clienti	

La classe `BankAccount` è quella che già conosciamo, dotata di metodi per leggere il saldo, nonché per versare e prelevare denaro.

In questo programma non c'è nulla che distingua i conti correnti da quelli di risparmio: l'ATM non aggiunge interessi né sottrae commissioni, di conseguenza decidiamo di non usare classi separate per i conti correnti e i conti di risparmio.

Infine, ci resta da analizzare la classe ATM vera e propria. Un concetto importante per l'ATM è lo *stato*: lo stato attuale della macchina determina il testo visualizzato nei mes-

saggi per l'utente e la funzione dei pulsanti. Ad esempio, dopo essere stati autenticati, si usano i pulsanti A e B per selezionare uno dei due conti; poi, gli stessi pulsanti vengono utilizzati per scegliere tra l'operazione di versamento e di prelievo. L'ATM deve, quindi, memorizzare il proprio stato attuale, in modo da poter interpretare correttamente le scelte dell'utente.

Ecco i quattro stati:

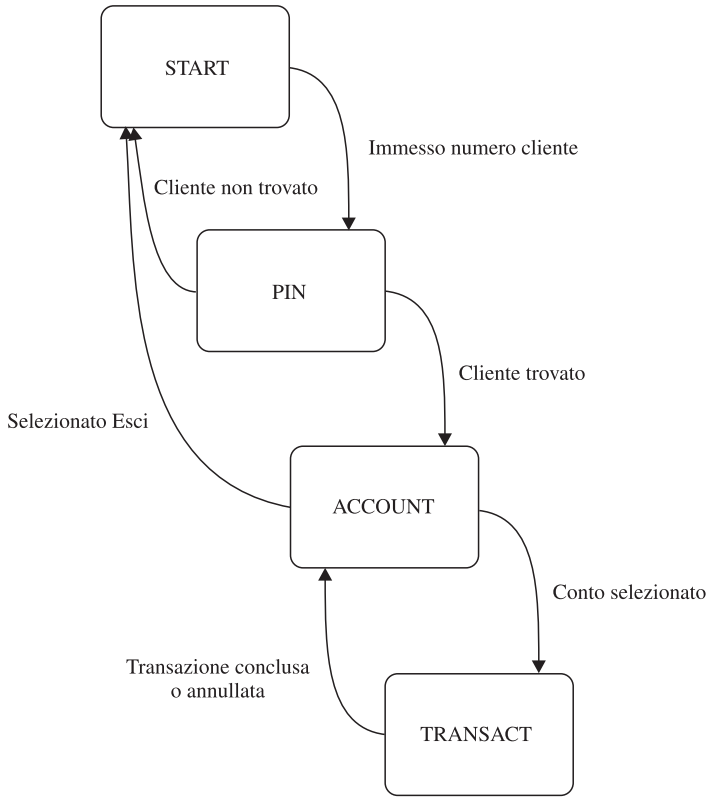
1. START: Immettere numero cliente
2. PIN: Immettere PIN
3. ACCOUNT: Selezionare conto
4. TRANSACT: Selezionare transazione

Per capire come ci si sposta da uno stato al successivo è utile tracciare un *diagramma degli stati* (osservate la Figura 13). La notazione UML utilizza forme standard per i diagrammi degli stati: gli stati sono rappresentati da rettangoli con gli angoli arrotondati; i cambiamenti di stato si indicano con frecce, associate a etichette che spiegano la ragione del cambiamento.

Implementeremo un metodo `setState` che modifica lo stato del sistema e aggiorna lo schermo dell'ATM.

L'utente deve digitare un numero cliente e un PIN che siano corretti. Successivamente l'ATM può chiedere alla banca di trovare il cliente. Per questo serve un metodo *seleziona cliente*, che collabora con la banca, chiedendo quale sia il cliente che corrisponde al numero cliente e al PIN digitati. In seguito, deve esserci un metodo *seleziona conto*, che chiede

Figura 13
Diagramma degli stati per la classe ATM



all'utente che opera sulla macchina se vuole agire sul suo conto corrente o sul suo conto di risparmio. Infine, i metodi `deposit` e `withdraw` eseguono la transazione selezionata sul conto selezionato.

ATM	
imposta lo stato	`Customer`
seleziona il cliente	`Bank`
seleziona il conto	`BankAccount`
esegui la transazione	

Naturalmente, l'individuazione di queste classi e di questi metodi non è stata pulita e ordinata come appare da questa presentazione. Quando ho progettato queste classi per questo libro, ho dovuto fare parecchi tentativi e stracciare un discreto numero di schede prima di arrivare a una struttura soddisfacente. È inoltre importante ricordare che raramente esiste un unico schema che sia il migliore in assoluto.

Questo progetto presenta svariati vantaggi. Le classi descrivono concetti chiari. I metodi sono sufficienti per portare a termine tutti i compiti necessari (ho percorso mentalmente ogni possibile scenario di utilizzo dell'ATM per verificarlo). Non vi sono troppe dipendenze di collaborazione fra le classi. Quindi, mi sono considerato soddisfatto di questa struttura e sono passato al passo successivo.

5.3 Diagramma UML

La Figura 14 mostra le relazioni fra queste classi, usando l'interfaccia grafica per l'utente (l'applicazione con interfaccia utente di solo testo usa un'unica classe `ATMTester` invece delle classi `ATMFrame` e `KeyPad`).

Per tracciare le dipendenze, utilizzate le colonne "collaboratore" delle schede CRC. Esaminando quelle colonne, troverete che le dipendenze sono le seguenti:

- `ATM` usa `Bank`, `Customer` e `BankAccount`.
- `Bank` usa `Customer`.

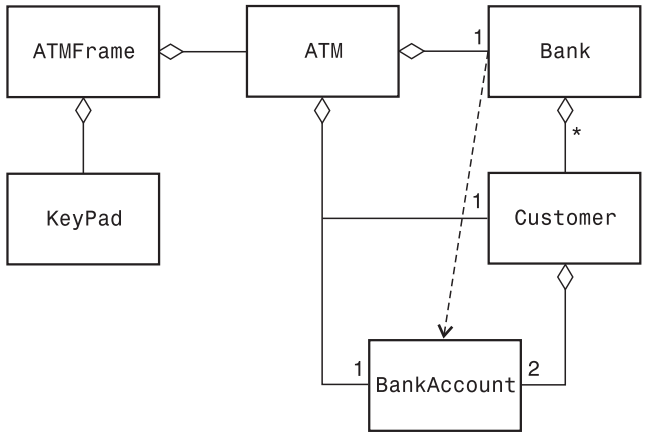


Figura 14
Relazioni fra le classi per il programma dell'ATM

Le relazioni di aggregazione sono facili da individuare. Una banca ha clienti e ogni cliente da due conti bancari.

La classe `ATM` aggrega la classe `Bank`? Per rispondere a questa domanda, chiedetevi se un oggetto di tipo `ATM` abbia bisogno di memorizzare un riferimento a un oggetto di tipo `Bank`: ha bisogno di utilizzare il medesimo esemplare di `Bank` per diverse invocazioni di metodi? Certamente sì, per cui la relazione più appropriata è l'aggregazione.

Un esemplare di `ATM` aggrega clienti? Ovviamente, un ATM non può avere il compito di memorizzare al proprio interno tutti i clienti di una banca: quello è proprio il compito della banca stessa. Nel nostro progetto, però, un esemplare di `ATM` memorizza il cliente *attuale*: dopo che un cliente è stato autorizzato a operare, i comandi successivi sono sempre riferiti al medesimo cliente e l'ATM deve memorizzare un riferimento a esso, per non dover richiedere l'oggetto corrispondete alla banca ogni volta che serve. Si tratta di una scelta progettuale: o memorizziamo l'oggetto, o lo richiediamo ogni volta che ci serve. Abbiamo deciso di memorizzare nell'ATM l'oggetto corrispondente al cliente attuale, per cui usiamo l'aggregazione. Notate che tale scelta di aggregazione non è una conseguenza automatica della descrizione del problema: si tratta di una scelta di progetto.

Analogamente, decidiamo di memorizzare nell'ATM un riferimento al conto bancario (corrente o di risparmio) attualmente selezionato dall'utente, per cui abbiamo una relazione di aggregazione tra `ATM` e `BankAccount`.

Il diagramma delle classi è un ottimo strumento per visualizzare le dipendenze. Osservate le classi relative all'interfaccia grafica: sono del tutto indipendenti dal resto del sistema ATM. Potete sostituire l'interfaccia grafica con un'interfaccia di solo testo, e potete anche estrarre la classe `KeyPad` e utilizzarla in un'altra applicazione. Anche le classi `Bank`, `BankAccount` e `Customer`, sebbene dipendenti l'una dall'altra, non sanno nulla della classe `ATM`. La cosa ha veramente senso: esistono banche che non hanno sportelli automatici. Come potete vedere, quando analizzate le relazioni di dipendenza, andate in cerca più dell'assenza di relazioni che della loro presenza.

5.4 Documentazione dei metodi

Ora siete pronti per il passo finale della fase di progettazione: documentare le classi e i metodi che avete identificato. Ecco una parte della documentazione per la classe `ATM`:

```
/**
   Un ATM che accede a una banca.
*/
public class ATM
{
   /**
      Costruisce un ATM per una certa banca.
      @param aBank la banca a cui si connette questo ATM
   */
   public ATM(Bank aBank) { }

   /**
      Imposta il numero di cliente attuale e imposta lo stato a PIN.
      (precondizione: lo stato è START)
      @param number il numero di cliente
   */
   public void setCustomerNumber(int number) { }

   /**
      Cerca il cliente nella banca.
```

```
      Se lo trova imposta lo stato a ACCOUNT, altrimenti a START.
      (precondizione: lo stato è PIN)
      @param pin il PIN del cliente attuale
   */
   public void selectCustomer(int pin) { }

   /**
      Imposta il conto attuale al conto corrente o di risparmio.
      Imposta lo stato a TRANSACT.
      (precondizione: lo stato è ACCOUNT o TRANSACT)
      @param account CHECKING oppure SAVINGS
   */
   public void selectAccount(int account) { }

   /**
      Preleva dal conto attuale.
      (precondizione: lo stato è TRANSACT)
      @param value la somma da prelevare
   */
   public void withdraw(double value) { }
   ...
}
```

Eseguite poi il programma di utilità `javadoc` per ottenere la documentazione in formato HTML.

Per brevità, omettiamo la documentazione delle altre classi.

5.5 Implementazione

Finalmente è arrivato il momento di implementare l'emulatore dell'ATM. Troverete che la fase di implementazione è quasi immediata e dovrebbe richiedere *molto meno tempo della fase di progettazione*.

Una valida strategia per realizzare le classi consiste nel procedere "dal basso verso l'alto" (*bottom-up*): iniziate con le classi che non dipendono da altre classi, come `KeyPad` e `BankAccount`; poi, realizzate una classe, come `Customer`, che dipende soltanto dalla classe `BankAccount`. Questa metodologia "bottom-up" vi consente di collaudare le classi singolarmente. Alla fine del paragrafo troverete il codice di queste classi.

La classe più complessa è la classe `ATM`. Per realizzarne i metodi, dovete definire le variabili di esemplare necessarie. Dal diagramma delle classi, potete dedurre che l'ATM contiene un riferimento a una banca, che diviene quindi variabile di esemplare della classe:

```
public class ATM
{  ...
   private Bank theBank;
}
```

Dalla descrizione degli stati dell'ATM, è evidente che abbiamo bisogno di ulteriori variabili per memorizzare lo stato, il cliente e il conto attuali.

```
public class ATM
{  ...
   private int state;
   private Customer currentCustomer;
```

```
   private BankAccount currentAccount;
   ...
}
```

La maggior parte dei metodi si realizza molto semplicemente. Considerate il metodo `selectCustomer`. Dalla documentazione di progetto abbiamo la seguente descrizione :

```
/**
   Cerca il cliente nella banca.
   Se lo trova imposta lo stato a ACCOUNT, altrimenti a START.
   (precondizione: lo stato è PIN)
   @param pin il PIN del cliente attuale
*/
```

Questa descrizione si può tradurre quasi letteralmente in istruzioni Java:

```
public void selectCustomer(int pin)
{
   assert state == PIN;
   currentCustomer = theBank.findCustomer(customerNumber, pin);
   if (currentCustomer == null)
      state = START;
   else
      state = ACCOUNT;
}
```

Non ci dedicheremo a una descrizione del programma ATM metodo per metodo, anche se dovreste dedicare un po' di tempo a confrontare la realizzazione effettiva con le schede CRC e il diagramma UML.

File ATM.java

```
import java.io.IOException;

/**
   Un ATM che accede a una banca.
*/
class ATM extends JFrame
{
   /**
      Costruisce un ATM per una certa banca.
      @param aBank la banca a cui si connette questo ATM
   */
   public ATM(Bank aBank)
   {
      theBank = aBank;
      reset();
   }

   /**
      Porta l'ATM nello stato iniziale.
   */
   public void reset()
   {
      customerNumber = -1;
      currentAccount = null;
```

```
   state = START;
}

/**
   Imposta il numero di cliente attuale e imposta lo stato a PIN.
   (precondizione: lo stato è START)
   @param number il numero di cliente
*/
public void setCustomerNumber(int number)
{
   assert state == START;
   customerNumber = number;
   state = PIN;
}

/**
   Cerca il cliente nella banca.
   Se lo trova imposta lo stato a ACCOUNT, altrimenti a START.
   (precondizione: lo stato è PIN)
   @param pin il PIN del cliente attuale
*/
public void selectCustomer(int pin)
{
   assert state == PIN;
   currentCustomer = theBank. findCustomer(customerNumber, pin);
   if (currentCustomer == null)
      state = START;
   else
      state = ACCOUNT;
}

/**
   Imposta il conto attuale al conto corrente o di risparmio.
   Imposta lo stato a TRANSACT.
   (precondizione: lo stato è ACCOUNT o TRANSACT)
   @param account CHECKING oppure SAVINGS
*/
public void selectAccount(int account)
{
   assert state == ACCOUNT ¦¦ state == TRANSACT;<
   if (account == CHECKING)
      currentAccount = currentCustomer.getCheckingAccount();
   else
      currentAccount = currentCustomer.getSavingsAccount();
   state = TRANSACT;
}

/**
   Preleva dal conto attuale.
   (precondizione: lo stato è TRANSACT)
   @param value la somma da prelevare
*/
public void withdraw(double value)
{
   assert state == TRANSACT;
   currentAccount.withdraw(value);
}
```

```
   /**
      Versa nel conto attuale.
      (precondizione: lo stato è TRANSACT)
      @param value la somma da versare
   */
   public void deposit(double value)
   {
      assert state == TRANSACT;
      currentAccount.deposit(value);
   }

   /**
      Ispeziona il saldo del conto attuale.
      (precondizione: lo stato è TRANSACT)
      @return il saldo
   */
   public double getBalance()
   {
      assert state == TRANSACT;
      return currentAccount.getBalance();
   }

   /**
      Torna allo stato precedente.
   */
   public void back()
   {
      if (state == TRANSACT)
         state = ACCOUNT;
      else if (state == ACCOUNT)
         state = PIN;
      else if (state == PIN)
         state = START;
   }

   /**
      Ispeziona lo stato attuale dell'ATM.
      @return lo stato attuale
   */
   public int getState()
   {
      return state;
   }

   private int state;
   private int customerNumber;
   private Customer currentCustomer;
   private BankAccount currentAccount;
   private Bank theBank;

   private static final int START = 1;
   private static final int PIN = 2;
   private static final int ACCOUNT = 3;
   private static final int TRANSACT = 4;

   private static final int CHECKING = 1;
   private static final int SAVINGS = 2;
}
```

File Bank.java

```
import java.io.FileReader;
import java.io.IOException;
import java.util.ArrayList;
import java.util.Scanner;
/**
   Una banca contiene clienti e i loro conti bancari.
*/
public class Bank
{
   /**
      Costruisce una banca senza clienti.
   */
   public Bank()
   {
      customers = new ArrayList<Customer>();
   }

   /**
      Legge i numeri cliente e i PIN e inizializza i conti bancari.
      @param filename il nome del file con i clienti
   */
   public void readCustomers(String filename)
         throws IOException
   {
      Scanner in = new Scanner(new FileReader(filename));
      while (in.hasNext())
      {
         int number = in.nextInt();
         int pin = in.nextInt();
         Customer c = new Customer(number, pin);
         addCustomer(c);
      }
      in.close();
   }

   /**
      Aggiunge un cliente alla banca.
      @param c il cliente da aggiungere
   */
   public void addCustomer(Customer c)
   {
      customers.add(c);
   }

   /**
      Cerca un cliente nella banca.
      @param aNumber un numero cliente
      @param aPin un numero di identificazione personale
      @return il cliente corrispondente
              oppure null se nessun cliente corrisponde
   */
   public Customer findCustomer(int aNumber, int aPin)
   {
      for (Customer c : customers)
      {
```

```
            if (c.match(aNumber, aPin))
               return c;
         }
         return null;
      }

      private ArrayList<Customer> customers;
   }
```

File Customer.java

```
/**
   Un cliente della banca con un conto corrente e un conto
   di risparmio.
*/
public class Customer
{
   /**
      Costruisce un cliente con un numero e un PIN assegnati.
      @param aNumber il numero cliente
      @param aPin il numero di identificazione personale
   */
   public Customer(int aNumber, int aPin)
   {
      customerNumber = aNumber;
      pin = aPin;
      checkingAccount = new BankAccount();
      savingsAccount = new BankAccount();
   }

   /**
      Verifica se questo cliente corrisponde al numero e al PIN.
      @param aNumber il numero cliente
      @param aPin il numero di identificazione personale
      @return true se il numero e il PIN corrispondono
   */
   public boolean match(int aNumber, int aPin)
   {
      return customerNumber == aNumber && pin == aPin;
   }
   /**
      Fornisce il conto corrente del cliente.
      @return il conto corrente
   */
   public BankAccount getCheckingAccount()
   {
      return checkingAccount;
   }

   /**
      Fornisce il conto di risparmio del cliente.
      @return il conto di risparmio
   */
   public BankAccount getSavingsAccount()
   {
      return savingsAccount;
   }
```

```
   private int customerNumber;
   private int pin;
   private BankAccount checkingAccount;
   private BankAccount savingsAccount;
}
```

Ecco, poi, le classi che realizzano l'ATM nella sua versione dotata di interfaccia grafica.

File ATMViewer.java

```
import java.io.IOException;
import javax.swing.JFrame;
import javax.swing.JOptionPane;

/**
   Simulazione grafica di uno sportello bancario automatico.
*/
public class ATMViewer
{
   public static void main(String[] args)
   {
      ATM theATM;

      try
      {
         Bank theBank = new Bank();
         theBank.readCustomers("customers.txt");
         theATM = new ATM(theBank);
      }
      catch (IOException e)
      {
         JOptionPane.showMessageDialog(null,
               "Error opening accounts file.");
      }

      JFrame frame = new ATM();
      frame.setTitle("First National Bank of Java");
      frame.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
      frame.setVisible(true);
   }
}
```

File ATMFrame.java

```
import java.awt.FlowLayout;
import java.awt.GridLayout;
import java.awt.event.ActionEvent;
import java.awt.event.ActionListener;
import javax.swing.JButton;
import javax.swing.JFrame;
import javax.swing.JPanel;
import javax.swing.JTextArea;

/**
   Un frame che visualizza i componenti di un ATM.
*/
```

```
public class ATMFrame extends JFrame
{
   /**
      Costruisce l'interfaccia utente dell'applicazione ATM.
   */
   public ATMFrame(ATM anATM)
   {
      theATM = anATM;

      // costruisci i componenti
      pad = new KeyPad();

      display = new JTextArea(4, 20);

      aButton = new JButton("  A  ");
      aButton.addActionListener(new AButtonListener());

      bButton = new JButton("  B  ");
      bButton.addActionListener(new BButtonListener());

      cButton = new JButton("  C  ");
      cButton.addActionListener(new CButtonListener());

      // aggiungi i componenti

      JPanel buttonPanel = new JPanel();
      buttonPanel.setLayout(new GridLayout(3, 1));
      buttonPanel.add(aButton);
      buttonPanel.add(bButton);
      buttonPanel.add(cButton);

      setLayout(new FlowLayout());
      add(pad);
      add(display);
      add(buttonPanel);
      showState();

      setSize(FRAME_WIDTH, FRAME_HEIGHT);
   }

   /**
      Aggiorna lo schermo dell'ATM.
   */
   public void showState()
   {
      int state = theATM.getState();
      pad.clear();
      if (state == ATM.START)
         display.setText("Enter customer number\nA = OK");
      else if (state == ATM.PIN)
         display.setText("Enter PIN\nA = OK");
      else if (state == ATM.ACCOUNT)
         display.setText("Select Account\n"
               + "A = Checking\nB = Savings\nC = Exit");
      else if (state == ATM.TRANSACT)
         display.setText("Balance = "
               + theATM.getBalance()
```

```
            + "\nEnter amount and select transaction\n"
            + "A = Withdraw\nB = Deposit\nC = Cancel");
}

private class AButtonListener implements ActionListener
{
   public void actionPerformed(ActionEvent event)
   {
      int state = theATM.getState();
      if (state == ATM.START)
         theATM.setCustomerNumber((int) pad.getValue());
      else if (state == ATM.PIN)
         theATM.selectCustomer((int) pad.getValue());
      else if (state == ATM.ACCOUNT)
         theATM.selectAccount(ATM.CHECKING);
      else if (state == ATM.TRANSACT)
      {
         theATM.withdraw(pad.getValue());
         theATM.back();
      }
      showState();
   }
}

private class BButtonListener implements ActionListener
{
   public void actionPerformed(ActionEvent event)
   {
      int state = theATM.getState();
      if (state == ATM.ACCOUNT)
         theATM.selectAccount(ATM.SAVINGS);
      else if (state == ATM.TRANSACT)
      {
         theATM.deposit(pad.getValue());
         theATM.back();
      }
      showState();
   }
}

private class CButtonListener implements ActionListener
{
   public void actionPerformed(ActionEvent event)
   {
      int state = theATM.getState();
      if (state == ATM.ACCOUNT)
         theATM.reset();
      else if (state == ATM.TRANSACT)
         theATM.back();
      showState();
   }
}

private JButton aButton;
private JButton bButton;
private JButton cButton;
```

```
   private KeyPad pad;
   private JTextArea display;

   private ATM theATM;

   private static final int FRAME_WIDTH = 300;
   private static final int FRAME_HEIGHT = 400;
}
```

File KeyPad.java

```
import java.awt.BorderLayout;
import java.awt.GridLayout;
import java.awt.event.ActionEvent;
import java.awt.event.ActionListener;
import javax.swing.JButton;
import javax.swing.JPanel;
import javax.swing.JTextField;

/**
   Un componente che consente all'utente di digitare un numero,
   usando un tastierino con pulsanti etichettati con cifre.
*/
public class KeyPad extends JPanel
{
   /**
      Costruisce il pannello del tastierino.
   */
   public KeyPad()
   {
      setLayout(new BorderLayout());

      // aggiungi il campo di visualizzazione

      display = new JTextField();
      add(display, "North");

      // costruisci il pannello dei pulsanti

      buttonPanel = new JPanel();
      buttonPanel.setLayout(new GridLayout(4, 3));

      // aggiungi i pulsanti per le cifre

      addButton("7");
      addButton("8");
      addButton("9");
      addButton("4");
      addButton("5");
      addButton("6");
      addButton("1");
      addButton("2");
      addButton("3");
      addButton("0");
      addButton(".");
```

```
      // aggiungi il pulsante di cancellazione

      clearButton = new JButton("CE");
      buttonPanel.add(clearButton);

      class ClearButtonListener implements ActionListener
      {
         public void actionPerformed(ActionEvent event)
         {
            display.setText("");
         }
      }
      ActionListener listener = new ClearButtonListener();

      clearButton.addActionListener(listener);

      add(buttonPanel, "Center");
   }

   /**
      Aggiunge un pulsante al pannello dei pulsanti.
      @param label l'etichetta del pulsante
   */
   private void addButton(final String label)
   {
      class DigitButtonListener implements ActionListener
      {
         public void actionPerformed(ActionEvent event)
         {
            // non aggiungere due punti decimali
            if (label.equals(".")
                  && display.getText().indexOf(".") != -1)
               return;

            // aggiungi l'etichetta alla fine del testo già presente
            display.setText(display.getText() + label);
         }
      }

      JButton button = new JButton(label);
      buttonPanel.add(button);
      ActionListener listener = new DigitButtonListener();
      button.addActionListener(listener);
   }

   /**
      Fornisce il valore digitato dall'utente.
      @return il valore presente nel campo di testo del tastierino
   */
   public double getValue()
   {
      return Double.parseDouble(display.getText());
   }

   /**
      Azzera il visualizzatore.
   */
```

```
      public void clear()
      {
         display.setText("");
      }

      private JPanel buttonPanel;
      private JButton clearButton;
      private JTextField display;
   }
```

La classe seguente realizza l'interfaccia di solo testo per l'ATM.

File ATMTester.java

```
import java.io.IOException;
import java.util.Scanner;

/**
   Simulazione di solo testo di uno sportello bancario automatico.
*/
public class ATMTester
{
   public static void main(String[] args)
   {
      ATM theATM;

      try
      {
         Bank theBank = new Bank();
         theBank.readCustomers("customers.txt");
         theATM = new ATM(theBank);
      }
      catch (IOException e)
      {
         System.out.println("Error opening accounts file.");
         return;
      }

      Scanner in = new Scanner(System.in);

      while (true)
      {
         int state = theATM.getState();
         if (state == ATM.START)
         {
            System.out.print("Enter account number: ");
            int number = in.nextInt();
            theATM.setCustomerNumber(number);
         }
         else if (state == ATM.PIN)
         {
            System.out.print("Enter PIN: ");
            int pin = in.nextInt();
            theATM.selectCustomer(pin);
         }
         else if (state == ATM.ACCOUNT)
```

```
            {
               System.out.print("A=Checking, B=Savings, C=Quit: ");
               String command = in.next();
               if (command.equalsIgnoreCase("A"))
                  theATM.selectAccount(ATM.CHECKING);
               else if (command.equalsIgnoreCase("B"))
                  theATM.selectAccount(ATM.SAVINGS);
               else if (command.equalsIgnoreCase("C"))
                  theATM.reset();
               else
                  System.out.println("Illegal input!");
            }
            else if (state == ATM.TRANSACT)
            {
               System.out.println("Balance=" + theATM.getBalance());
               System.out.print("A=Deposit, B=Withdraw, C=Cancel: ");
               String command = in.next();
               if (command.equalsIgnoreCase("A"))
               {
                  System.out.print("Amount: ");
                  double amount = in.nextDouble();
                  theATM.deposit(amount);
                  theATM.back();
               }
               else if (command.equalsIgnoreCase("B"))
               {
                  System.out.print("Amount: ");
                  double amount = in.nextDouble();
                  theATM.withdraw(amount);
                  theATM.back();
               }
               else if (command.equalsIgnoreCase("C"))
                  theATM.back();
               else
                  System.out.println("Illegal input!");
            }
         }
      }
   }
```

Visualizza:

```
Enter account number: 1
Enter PIN: 1234
A=Checking, B=Savings, C=Quit: A
Balance=0.0
A=Deposit, B=Withdrawal, C=Cancel: A
Amount: 1000
A=Checking, B=Savings, C=Quit: C
...
```

In questo capitolo avete appreso un approccio sistematico per la costruzione di programmi relativamente complessi. Tuttavia, la progettazione orientata agli oggetti non è decisamente uno sport da spettatori: per imparare davvero come progettare e implementare programmi, dovete farvi un'esperienza ripetendo questo processo con i vostri progetti. È del tutto possibile che non vi riesca di individuare subito una buona soluzione e che vi tocchi torna-

re indietro e riorganizzare le classi e le responsabilità. Questo è normale e c'è solo da aspettarselo. Il procedimento che si compie con la progettazione orientata agli oggetti ha proprio lo scopo di scoprire questi problemi mentre ci si trova ancora nella fase di progettazione, quando sono ancora facili da risolvere, invece di scoprirli durante la fase di implementazione, quando un'eventuale riorganizzazione su larga scala è molto più ardua e fa perdere molto tempo.

Auto-valutazione

12. Perché la classe `Bank` di questo esempio non memorizza i conti bancari in un vettore?
13. Immaginate che vengano modificati i requisiti del programma: dopo ogni transazione dovete memorizzare in un file i saldi di tutti i conti, per poi rileggerli all'inizio del programma. Che impatto ha tale modifica sul progetto?

Note di cronaca 2

Sviluppo di software: arte o scienza?

Si è dibattuto a lungo se la disciplina dell'informatica sia una scienza oppure no. Negli Stati Uniti si parla di "computer science", ma questo non vuol dire molto: eccettuati forse i bibliotecari e i sociologi, poche persone sono davvero convinte che la scienza della biblioteconomia e le scienze sociali siano attività scientifiche.

Una disciplina scientifica aspira a scoprire certi principi fondamentali imposti dalle leggi della natura. Si avvale del *metodo scientifico*: formula ipotesi e le mette alla prova con esperimenti che altri ricercatori di quel settore possono ripetere. Per esempio, un fisico potrebbe avere una sua teoria sulla composizione delle particelle nucleari e provare a verificarla o a contraddirla eseguendo esperimenti con un acceleratore di particelle. Se un esperimento non può essere verificato, come nel caso delle ricerche sulla "fusione a freddo" all'inizio degli anni Novanta, quella teoria è destinata a estinguersi rapidamente.

Vi sono programmatori che fanno davvero esperimenti. Provano vari metodi per calcolare certi risultati o per configurare sistemi di computer e misurano le differenze nelle prestazioni. Tuttavia, il loro obiettivo non è quello di scoprire leggi della natura.

Alcuni scienziati informatici scoprono principi fondamentali. Una classe di risultati fondamentali, per esempio, stabilisce che è impossibile scrivere determinati tipi di programmi per computer, non importa quanto sia potente la macchina per elaborarli. Per esempio, è impossibile scrivere un programma che prenda come dati di ingresso due programmi qualunque in codice sorgente Java e verifichi se questi due programmi forniscono oppure no sempre lo stesso risultato. Un programma di questo genere tornerebbe molto comodo per valutare i compiti a casa degli studenti, ma nessuno, per brillante che sia, sarà mai in grado di scriverne uno che funzioni per tutti i possibili file in ingresso. I programmatori, però, nella gran maggioranza, scrivono programmi, invece di indagare sui limiti teorici della programmazione.

Alcune persone considerano la programmazione un'*arte* o una forma di *artigianato*. Un programmatore che scriva un codice elegante, facile da capire e che venga eseguito con efficienza ottimale, può davvero essere considerato un buon artigiano. Parlare di arte è forse un po' esagerato, perché un oggetto d'arte esige un pubblico di fruitori che lo apprezzi, mentre il codice di un programma di solito non è visibile a chi usa il programma.

Altri considerano l'informatica una *disciplina dell'ingegneria*. Proprio come l'ingegneria meccanica è basata sui principi fondamentali della statica, espressi in forma matematica, l'informatica ha alcune fondamenta matematiche. L'ingegneria meccanica, però, non si riduce alla sola matematica, c'è molto di più, per esempio la conoscenza dei mate-

riali e la pianificazione dei progetti. Lo stesso vale per l'informatica. Un *ingegnere del software* deve avere conoscenze di pianificazione, progettazione, automazione del collaudo, documentazione e controllo del codice sorgente, oltre ad argomenti specifici dell'informatica, come la programmazione, la progettazione di algoritmi e le tecnologie delle basi di dati.

Uno degli aspetti che un po' turbano dell'informatica è il fatto che non ha la stessa autorevolezza di altre discipline dell'ingegneria. Non c'è unanimità di consensi in merito a ciò che costituisce un comportamento professionale nel campo della programmazione. Diversamente dagli scienziati, la cui principale responsabilità è la ricerca della verità, gli ingegneri devono affrontare esigenze conflittuali in termini di qualità, sicurezza ed economia. Le discipline dell'ingegneria hanno organizzazioni professionali che vincolano i loro associati a determinati standard di condotta. Il campo dei computer è ancora così nuovo che in molti casi noi semplicemente non sappiamo quale sia il metodo corretto per ottenere determinati risultati. E questo rende davvero difficile stabilire standard professionali.

Che cosa ne pensate? Sulla base della vostra limitata esperienza, pensate che lo sviluppo del software sia un'arte, un attività artigianale, una scienza o un'attività nel campo dell'ingegneria?

Riepilogo del capitolo

1. Il ciclo di vita del software comprende tutte le fasi, dall'analisi iniziale all'obsolescenza.
2. Un procedimento formale per lo sviluppo del software descrive le fasi del processo di sviluppo e fornisce linee guida su come portare a termine ciascuna fase.
3. Il modello a cascata per lo sviluppo del software descrive un procedimento sequenziale di analisi, progettazione, realizzazione collaudo e installazione.
4. Il modello a spirale per lo sviluppo del software descrive un processo iterativo in cui vengono ripetute fasi di progettazione e di realizzazione.
5. La Programmazione Estremizzata è una metodologia di sviluppo che insegue la semplicità eliminando la struttura formale e concentrandosi sulle migliori regole pratiche.
6. Nella progettazione orientata agli oggetti dovete identificare le classi, determinare i loro compiti e descriverne le relazioni.
7. Una scheda CRC descrive una classe, le sue responsabilità e le classi che collaborano con essa.
8. L'ereditarietà (la relazione *è-un*) viene a volte usata a sproposito, quando invece una relazione *ha-un* sarebbe più opportuna.
9. L'aggregazione (la relazione *ha-un*) indica che gli esemplari di una classe contengono riferimenti a esemplari di un'altra classe.
10. La dipendenza è un altro nome per la relazione *usa*.
11. Nella notazione UML, occorre poter distinguere le diverse relazioni: ereditarietà, implementazione di interfaccia, aggregazione e dipendenza.
12. Potete usare i commenti di documentazione di `javadoc` (con i corpi dei metodi in bianco) per annotare formalmente il comportamento delle classi che avete individuato.

Ulteriori letture

[1] Grady Booch, James Rumbaugh e Ivar Jacobson, *The Unified Modeling Language User Guide*, Addison-Wesley, 1999.
[2] Kent Beck, *Extreme Programming Explained*, Addison-Wesley, 1999.
[3] W.H. Sackmann, W.J. Erikson e E.E. Grant, "Exploratory Experimental Studies Comparing Online and Offline Programming Performance", *Communications of the ACM*, vol 11, n. 1, (Gennaio 1968), pagg. 3-11.
[4] F. Brooks, *The Mythical Man-Month*, Addison-Wesley, 1975.