

# Collaudo e correzione di errori

## **Obiettivi del capitolo**

- Imparare a eseguire collaudi di unità
- Comprendere i principi della selezione e valutazione dei casi di prova
- Apprendere l'utilizzo delle asserzioni e della registrazione di eventi
- Prendere confidenza con il debugger
- Imparare le strategie per un collaudo efficace

## 2 Collaudo e correzione di errori

Un programma complesso non potrà mai funzionare bene al primo colpo, perché conterrà errori, chiamati comunemente *bug*, e bisognerà collaudarlo. È più facile collaudare un programma se è stato progettato tenendo presente il problema del collaudo. Questa è una diffusa pratica di progettazione: nelle schede elettroniche di un televisore, o fra i cavi di un'automobile, troverete spie luminose e connettori che non servono direttamente al televisore o all'automobile, ma che sono stati collocati lì per il personale delle riparazioni, nel caso che qualcosa non funzioni. Nella prima parte di questo capitolo, imparerete come attrezzare in modo analogo i vostri programmi. Si tratta di fare un po' più di lavoro all'inizio, che però verrà ampiamente ripagato dalla riduzione dei tempi di individuazione e correzione degli errori.

Nella seconda parte del capitolo, imparerete come attivare il *debugger*, per risolvere i problemi con i programmi che non fanno le cose giuste.

# 1 Collaudo di unità

Usate il collaudo di unità per collaudare singole classi, separatamente l'una dall'altra.

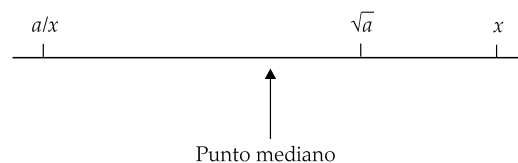
Per eseguire un collaudo, scrivete una *infrastruttura di collaudo*.

Il più importante e semplice strumento di prova è il *collaudo di una singola unità*, di un metodo o di un gruppo di metodi che cooperano fra loro.

Per questo collaudo bisogna compilare le classi al di fuori del programma in cui verranno usate, insieme a un semplice metodo, detto *infrastruttura di collaudo*, che fornisce i parametri ai metodi.

Gli argomenti per il collaudo possono provenire da diverse fonti: dati inseriti dall'utente oppure esecuzione di un ciclo con un insieme di valori, che possono essere valori casuali o valori memorizzati in un file o in una base di dati.

Nei paragrafi seguenti, utilizzeremo come semplice esempio il collaudo di un metodo che realizza un algoritmo di approssimazione per calcolare le radici quadrate, che era già noto agli antichi greci. L'algoritmo, già menzionato nell'Esercizio P6.12, inizia ipotizzando un valore  $x$  che sia abbastanza prossimo alla radice quadrata richiesta,  $\sqrt{a}$ . Non è necessario che il valore iniziale sia molto vicino al risultato:  $x = a$  è un'ottima scelta. A questo punto, esaminate le quantità  $x$  e  $a/x$ . Se  $x < \sqrt{a}$ , allora  $a/x > a/\sqrt{a} = \sqrt{a}$ . Analogamente, se  $x > \sqrt{a}$ , allora  $a/x < a/\sqrt{a} = \sqrt{a}$ . Di conseguenza,  $\sqrt{a}$  si trova fra  $x$  e  $a/x$ . Assumete che il *punto mediano* di questo intervallo sia la nostra nuova ipotesi di radice quadrata (osservate la Figura 1). Quindi, impostate  $x_{\text{new}} = (x + a/x)/2$ , e ripetete il procedimento, ovvero calcolate il punto mediano fra  $x_{\text{new}}$  e  $a/x_{\text{new}}$ . Fermatevi quando due approssimazioni consecutive differiscono per un valore molto piccolo.



**Figura 1**  
Calcolo approssimato  
della radice quadrata

Questo metodo arriva alla sua conclusione molto rapidamente. Per calcolare  $\sqrt{100}$  sono necessari solo otto passaggi:

```

Guess #1: 50.5
Guess #2: 26.24009900990099
Guess #3: 15.025530119986813
Guess #4: 10.840434673026925
Guess #5: 10.032578510960604
Guess #6: 10.000052895642693
Guess #7: 10.000000000139897
Guess #8: 10.0
Guess #9: 10.0
Guess #10: 10.0

```

Ecco una classe che realizza l'algoritmo di approssimazione della radice quadrata. Costruite un oggetto di tipo `RootApproximator` per estrarre la radice quadrata di un numero assegnato. Il metodo `nextGuess` calcola il tentativo successivo, mentre il metodo `getRoot` continua a invocare `nextGuess` finché due tentativi successivi sono sufficientemente vicini.

#### File `RootApproximator.java`

```

/**
 * Calcola successive approssimazioni della radice quadrata
 * di un numero, usando l'algoritmo di Heron.
 */
public class RootApproximator
{
    /**
     * Costruisce un oggetto per calcolare l'approssimazione
     * della radice di un numero assegnato.
     * @param aNumber il numero di cui estrarre la radice quadrata
     * (Pre-condizione: aNumber >= 0)
     */
    public RootApproximator(double aNumber)
    {
        a = aNumber;
        xold = 1;
        xnew = a;
    }

    /**
     * Calcola una migliore approssimazione a partire dalla
     * approssimazione attuale.
     * @return la successiva approssimazione
     */
    public double nextGuess()
    {
        xold = xnew;
        if (xold != 0)
            xnew = (xold + a / xold) / 2;
        return xnew;
    }
}

```

## 4 Collaudo e correzione di errori

```
/**
 * Calcola la radice migliorando ripetutamente l'approssimazione
 * attuale finché due approssimazioni successive sono quasi uguali.
 * @return il valore calcolato per la radice quadrata
 */
public double getRoot()
{
    while (!Numeric.approxEqual(xnew, xold))
        nextGuess();
    return xnew;
}

private double a; // il numero di cui si calcola la radice quadrata
private double xnew; // l'approssimazione attuale
private double xold; // l'approssimazione precedente
}
```

Le approssimazioni per il calcolo di  $\sqrt{100}$  vengono calcolate da questo programma di prova.

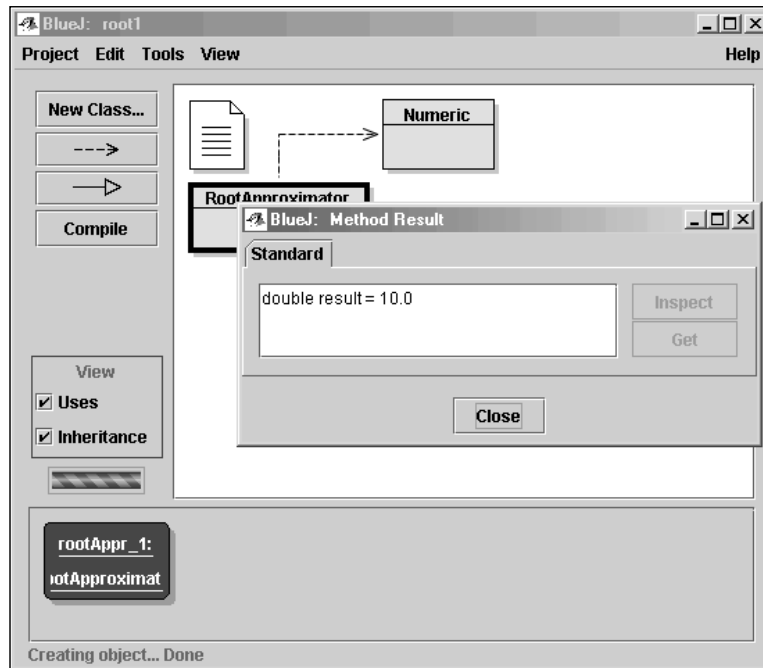
### File `RootApproximatorTest.java`

```
import javax.swing.JOptionPane;

/**
 * Questo programma stampa dieci approssimazioni per il calcolo
 * di una radice quadrata.
 */
public class RootApproximatorTest
{
    public static void main(String[] args)
    {
        String input
            = JOptionPane.showInputDialog("Enter a number");
        double x = Double.parseDouble(input);
        RootApproximator r = new RootApproximator(x);
        final int MAX_TRIES = 10;
        for (int tries = 1; tries <= MAX_TRIES; tries++)
        {
            double y = r.nextGuess();
            System.out.println("Guess #" + tries + ": " + y);
        }
        System.exit(0);
    }
}
```

Funziona correttamente la classe `RootApproximator`? Vediamo di affrontare questo problema in modo sistematico.

Se usate un ambiente come BlueJ, che consente la creazione di oggetti e l'invocazione di loro metodi, potete eseguire facilmente alcune prove costruendo oggetti e invocandone metodi (osservate la Figura 2).



**Figura 2**  
Collaudare una classe  
con BlueJ

In alternativa, se non avete a disposizione tale ambiente di sviluppo, è facile scrivere un'infrastruttura di collaudo che fornisca singoli valori di prova. Ecco un esempio.

### File `RootApproximatorTest2.java`

```
import javax.swing.JOptionPane;

/**
 * Questo programma calcola le radici quadrate di valori forniti
 * dall'utente.
 */
public class RootApproximatorTest2
{
    public static void main(String[] args)
    {
        boolean done = false;
        while (!done)
        {
            String input = JOptionPane.showInputDialog(
                "Enter a number, Cancel to quit");

            if (input == null)
                done = true;
            else
            {
                double x = Double.parseDouble(input);
                RootApproximator r = new RootApproximator(x);
            }
        }
    }
}
```

## 6 Collaudo e correzione di errori

```
        double y = r.getRoot();

        System.out.println("square root of " + x
                           + " = " + y);
    }
}
System.exit(0);
}
```

Se leggete i valori di prova in ingresso da un file, potete facilmente ripetere il collaudo.

Ora potete digitare i valori in ingresso e verificare che il metodo `getRoot` calcoli i valori corretti per la radice quadrata.

Questo approccio al collaudo presenta, però, un problema. Supponete di individuare e correggere un errore nel codice: ovviamente, vorrete collaudare la classe di nuovo. L'infrastruttura di collaudo può essere utilizzata nuovamente, il che è una buona cosa, ma dovete digitare di nuovo i valori in ingresso.

Leggere i valori in ingresso da un file è un'idea migliore: in tal modo, potete semplicemente inserire i valori in un file una volta sola ed eseguire il programma di collaudo ogni volta che volete per mettere alla prova una nuova versione del vostro codice. Il modo più semplice per leggere dati in ingresso da un file è il reindirizzamento del flusso `System.in` (consultate Consigli per la produttività 6.1). Per prima cosa, dovete apportare una modifica marginale all'infrastruttura di collaudo, per leggere i dati da un oggetto di tipo `BufferedReader` (consultate Argomenti avanzati 3.6). Il programma che segue mostra tale modifica.

### File `RootApproximatorTest3.java`

```
import java.io.BufferedReader;
import java.io.InputStreamReader;
import java.io.IOException;

/**
 * Questo programma calcola le radici quadrate di valori forniti
 * dall'utente tramite il flusso System.in.
 */
public class RootApproximatorTest3
{
    public static void main(String[] args)
    {
        BufferedReader console = new BufferedReader(
            new InputStreamReader(System.in));
        boolean done = false;
        while (!done)
        {
            String input = console.readLine();
            if (input == null)
                done = true;
            else
            {
                double x = Double.parseDouble(input);
                RootApproximator r = new RootApproximator(x);
                double y = r.getRoot();
            }
        }
    }
}
```



## 8 Collaudo e correzione di errori

```
    }  
  }  
}
```

Sull'uscita viene visualizzato:

```
square root of 1.0 = 1.0  
square root of 1.5 = 1.224744871391589  
square root of 2.0 = 1.414213562373095  
...  
square root of 9.0 = 3.0  
square root of 9.5 = 3.0822070014844885  
square root of 10.0 = 3.162277660168379
```

Sfortunatamente, questo collaudo è limitato a un piccolo sottoinsieme di valori. Per superare questa limitazione, può essere utile la generazione casuale di casi di prova:

### File `RootApproximatorTest5.java`

```
import java.util.Random;  
  
/**  
 * Questo programma calcola le radici quadrate di valori casuali.  
 */  
public class RootApproximatorTest5  
{  
    public static void main(String[] args)  
    {  
        final double SAMPLES = 100;  
        Random generator = new Random();  
        for (int i = 1; i <= SAMPLES; i++)  
        {  
            double x = 1.0E6 * generator.nextDouble();  
            RootApproximator r = new RootApproximator(x);  
            double y = r.getRoot();  
            System.out.println("square root of " + x  
                               + " = " + y);  
        }  
    }  
}
```

In una possibile esecuzione, sull'uscita viene visualizzato:

```
square root of 298042.3906807571 = 545.932588036982  
square root of 552836.0182932373 = 742.5294333738493  
square root of 751687.182520626 = 866.9989518567056  
square root of 872344.1056077272 = 933.9936325306116  
...
```

Essere in grado di selezionare buoni casi di prova è un'abilità importante per il collaudo dei programmi. Naturalmente, vorreste collaudare il vostro programma inserendo i valori di ingresso che potrebbe fornire un utente tipico.

Dovete provare tutte le caratteristiche del programma. Nel programma che calcola la radice quadrata, dovrete controllare casi di prova tipici quali `100`, `1/4`, `0.01`, `2`,

$10E12$ , e così via. Questi sono casi di prova *positivi*, costituiti da valori di ingresso validi per i quali vi aspettate una gestione corretta da parte del programma.

I casi limite sono casi di prova che si collocano al confine tra i valori accettabili e quelli non validi.

In seguito, dovete aggiungere *casi limite*, valori che si collocano al confine dell'insieme di valori accettabili per il vostro problema. Nel calcolo della radice quadrata, verificate che cosa succede se il valore di ingresso è uguale a 0. I casi limite sono comunque dati validi, quindi ci si aspetta che il programma li elabori correttamente, generalmente in qualche maniera banale o gestendoli come casi speciali. Collaudare casi limite è importante, perché spesso i programmatori compiono errori nella gestione di tali casi: divisioni per zero, estrazione di caratteri da stringhe vuote e utilizzo di riferimenti null sono le cause più comuni di errori.

Infine, usate casi di prova *negativi*. Si tratta di valori di ingresso che, prevedibilmente, verranno rifiutati dal programma, come, nel nostro esempio, la radice quadrata di  $-2$ . Tuttavia, in questo caso dovete stare un po' attenti: se la pre-condizione di un metodo non consente un particolare valore di ingresso, non è necessario che il metodo stesso produca un risultato in tal caso. In effetti, se provate a calcolare la radice quadrata di  $-2$  con il metodo appena visto, il ciclo presente nel metodo `getRoot` non termina mai. Potete capirne il motivo, invocando alcune volte `nextGuess`:

```

Guess #1: -0.5
Guess #2: 1.75
Guess #3: 0.3035714285714286
Guess #4: -3.142331932773109
Guess #5: -1.2529309672222557
Guess #6: 0.1716630854488237
Guess #7: -5.739532701343778
Guess #8: -2.6955361385562107
Guess #9: -0.976784358209916
Guess #10: 0.5353752385394334

```

Indipendentemente da come generate i casi di prova, ciò che importa è che collaudiate a fondo le singole classi prima di metterle insieme in un programma. Se vi è mai capitato di montare un calcolatore o riparare un'automobile, avete probabilmente seguito un procedimento simile. Invece di mettere semplicemente insieme tutti i pezzi sperando per il meglio, probabilmente avete prima verificato ciascun componente separatamente. All'inizio questo richiede un po' di tempo in più, ma riduce fortemente la possibilità di fallimenti totali e misteriosi una volta che le parti sono state assemblate.

## 2 Valutazione dei casi di prova

Nell'ultimo paragrafo, ci siamo preoccupati di come procurarci *valori di ingresso* per i collaudi. Ora, esaminiamo che cosa fare con *i valori prodotti in uscita*. Come sapere se sono corretti?

In qualche caso, potete verificare i valori prodotti in uscita calcolando manualmente i valori giusti. Per esempio, per un programma di elaborazione degli stipendi, potete calcolare le imposte manualmente.

## 10 Collaudo e correzione di errori

Talvolta, un calcolo richiede un sacco di lavoro e non conviene farlo manualmente. È il caso di molti algoritmi di approssimazione, che possono passare attraverso decine o centinaia di iterazioni, prima di arrivare al risultato finale. Il metodo della radice quadrata del paragrafo precedente è un esempio di questo genere di approssimazioni.

In che modo potete verificare che il metodo della radice quadrata funzioni correttamente? Potete fornire valori di prova per i quali conoscete la risposta, per esempio 4 e 100, o anche 1/4 e 0.01, in modo da non limitarvi ai numeri interi.

In alternativa, potete preparare un'infrastruttura di collaudo per verificare che i valori calcolati soddisfino certe proprietà. Per il programma che calcola la radice quadrata, potete calcolare la radice quadrata, quindi elevare al quadrato il risultato e verificare che corrisponda al valore originale fornito in ingresso:

### File `RootApproximatorTest6.java`

```
import java.util.Random;

/**
 * Questo programma verifica il calcolo di valori di radici quadrate
 * controllando una proprietà matematica della radice quadrata.
 */
public class RootApproximatorTest6
{
    public static void main(String[] args)
    {
        final double SAMPLES = 100;
        int passcount = 0;
        int failcount = 0;
        Random generator = new Random();
        for (int i = 1; i <= SAMPLES; i++)
        {
            // genera valori di prova casuali

            double x = 1.0E6 * generator.nextDouble();
            RootApproximator r = new RootApproximator(x);
            double y = r.getRoot();
            System.out.println("square root of " + x
                               + " = " + y);

            // verifica che il valore di prova soddisfi
            // la proprietà della radice quadrata

            if (Numeric.approxEqual(y * y, x))
            {
                System.out.println("Test passed.");
                passcount++;
            }
            else
            {
                System.out.println("Test failed.");
                failcount++;
            }
        }
    }
}
```

```

        System.out.println("Pass: " + passcount);
        System.out.println("Fail: " + failcount);
    }
}

```

Un oracolo è un metodo lento ma affidabile per calcolare un risultato a scopi di collaudo.

Infine, può esistere un sistema meno efficiente per calcolare lo stesso valore prodotto da un metodo. Potete quindi eseguire un'infrastruttura di collaudo che invochi sia il metodo da collaudare, sia un metodo che usa il sistema più lento, e che confronti i risultati. Per esempio,  $\sqrt{x} = x^{1/2}$ , pertanto potete usare il metodo `Math.pow`, più lento, per generare lo stesso valore. Un metodo di questo tipo, più lento ma affidabile, è detto *oracolo*. Il programma che segue mostra come confrontare i risultati del metodo da collaudare con il risultato prodotto da un oracolo. In alternativa, potete scrivere un programma a se stante che scrive in un file i valori calcolati dall'oracolo e li confronta con quelli calcolati dal metodo in esame.

#### File `RootApproximatorTest7.java`

```

import java.util.Random;

/**
 * Questo programma verifica il calcolo di valori di radici quadrate
 * usando un oracolo.
 */
public class RootApproximatorTest7
{
    public static void main(String[] args)
    {
        final double SAMPLES = 100;
        int passcount = 0;
        int failcount = 0;
        Random generator = new Random();
        for (int i = 1; i <= SAMPLES; i++)
        {
            // genera valori di prova casuali

            double x = 1.0E6 * generator.nextDouble();
            RootApproximator r = new RootApproximator(x);
            double y = r.getRoot();
            System.out.println("square root of " + x
                               + " = " + y);

            double oracleValue = Math.pow(x, 0.5);

            // verifica che il valore di prova sia
            // quasi uguale al valore prodotto dall'oracolo

            if (Numeric.approxEqual(y * y, x))
            {
                System.out.println("Test passed.");
                passcount++;
            }
        }
    }
}

```

## 12 Collaudo e correzione di errori

```
        else
        {
            System.out.println("Test failed.");
            failcount++;
        }
    }
    System.out.println("Pass: " + passcount);
    System.out.println("Fail: " + failcount);
}
}
```

## 3 Collaudo regressivo e copertura del collaudo

---

Come si raccolgono i casi di prova? Questa operazione è facile se i programmi ricevono tutti i loro valori dall'ingresso standard: inserite ciascun caso di prova in un file, per esempio chiamando i vari file `test1.in`, `test2.in`, `test3.in`. Questi file conterranno le stesse sequenze di tasti che digitereste normalmente alla tastiera durante l'esecuzione del programma. Fornite al programma da collaudare i file, mediante il reindirizzamento:

```
java Program < test1.in > test1.out
java Program < test2.in > test2.out
java Program < test3.in > test3.out
```

Un pacchetto di prova è un insieme di prove da ripetere per il collaudo.

Al termine, analizzate i risultati prodotti in uscita e verificate se sono corretti.

Conservare un caso di prova in un file è un buon accorgimento, perché potete usarlo per collaudare qualsiasi versione del programma. Di fatto, è una pratica utile e diffusa creare un file di prova quando si riscontra un errore nel programma (*bug*), e il file si può usare per verificare che la correzione dell'errore funzioni veramente. Non eliminate il file, ma usatelo per collaudare la versione successiva del programma e tutte le versioni seguenti. Una simile raccolta di casi di prova è detta *pacchetto di prova* (*test suite*).

Il collaudo regressivo contempla l'esecuzione ripetuta di prove già eseguite in precedenza, per essere certi che guasti noti delle versioni precedenti non compaiano nelle nuove versioni del programma.

Vi stupirete della frequenza con cui un bug, già corretto, riappare in una versione successiva: si tratta di un fenomeno chiamato *ciclicità*. Talvolta, non capirete completamente il motivo di un errore e inserirete una rapida correzione che sembrerà funzionare. Più tardi, applicherete un'altra rapida correzione per risolvere un secondo problema, che, però, farà riemergere il primo errore. Naturalmente, è sempre meglio ragionare a fondo sulle cause di un errore e risolverlo alla radice, anziché produrre una serie di soluzioni "tampone". Tuttavia, se non riuscite ad afferrare il problema, conviene almeno avere una schietta valutazione sulla qualità del funzionamento del programma. Se si conservano a portata di mano tutti i vecchi casi di prova, e se ciascuna nuova versione viene collaudata con tutti tali casi, si avrà questa risposta. L'operazione di verifica di una serie di errori passati si chiama collaudo regressivo (*regression testing*).

Il collaudo a scatola chiusa (*black-box testing*) descrive una metodologia di collaudo che non prende in considerazione la struttura dell'implementazione.

Il collaudo trasparente (*white-box testing*) usa informazioni sulla struttura del programma.

La copertura di un collaudo è una misura di quante parti di un programma siano state collaudate.

Collaudare le funzionalità del programma senza tenere conto della sua struttura interna costituisce un *black-box testing*, ovvero un collaudo a scatola chiusa. È una fase importante del collaudo, perché, dopotutto, gli utenti di un programma non conoscono la sua struttura interna. Se un programma funziona perfettamente in tutti i casi positivi e rifiuta elegantemente tutti quelli negativi, significa che fa bene il suo lavoro.

Tuttavia, è impossibile avere la sicurezza assoluta che un programma funzioni correttamente con tutti i valori di ingresso, se ci si limita a proporre un numero finito di casi di prova. Come osservò il famoso ricercatore informatico Edsger Dijkstra, il collaudo può evidenziare solamente la presenza di errori, non la loro assenza. Per poter arrivare a una maggiore fiducia sulla correttezza di un programma, è utile tenere conto della sua struttura interna. Le tecniche di collaudo che guardano all'interno di un programma sono chiamate *white-box testing* (collaudo trasparente), ed eseguire collaudi di unità su ciascun metodo fa parte di questa tecnica.

Volete essere certi che ogni porzione del programma venga collaudata da almeno uno dei casi di prova. Questo indica la *copertura del collaudo*. Se qualche porzione di codice non viene mai eseguita da qualcuno dei vostri casi di prova, non avrete modo di sapere se il codice funzionerà correttamente nel caso venga attivato dai dati forniti in ingresso dall'utente. Pertanto, dovete esaminare ogni diramazione *if/else*, per controllare che ciascuna venga raggiunta da qualche caso di prova. Molte diramazioni condizionali sono inserite nel codice solamente per gestire valori strani o anomali, ma eseguono comunque qualche operazione. Accade di frequente che finiscano per fare qualcosa di scorretto, ma questi errori non vengono mai scoperti durante il collaudo, perché nessuno ha fornito valori strani o anomali. Naturalmente, questi difetti diventano subito visibili appena si mette all'opera il programma, quando il primo utente inserisce un valore sbagliato e si inferocisce per il fallimento del programma. Un pacchetto di prova dovrebbe garantire che ciascuna parte di codice venga coperta da qualche valore di ingresso.

Per esempio, per collaudare il metodo `getTax`, nel programma per il calcolo delle tasse visto nel Capitolo 5, conviene assicurarsi che ciascun enunciato *if* venga eseguito da almeno un caso di prova. Dovreste provare entrambi i casi di contribuenti coniugati e non coniugati, con redditi in ciascuno dei tre scaglioni fiscali.

È una buona idea scrivere il primo gruppo di casi di prova *prima* di terminare la stesura del codice. Pianificare un po' di casi di prova può costituire un approfondimento su quanto dovrebbe fare il programma, fornendo un aiuto prezioso per l'implementazione. Avrete così anche qualcosa con cui provare il programma appena terminata la compilazione. Naturalmente, la serie iniziale di casi di prova aumenterà mano a mano che procede il processo di collaudo e correzione degli errori.

I programmi attuali possono essere piuttosto ardui da collaudare. In un programma con un'interfaccia utente grafica, l'utente può premere i pulsanti a caso con il mouse, e fornire dati in ordine casuale. I programmi che ricevono i loro dati attraverso una connessione di rete, vanno collaudati mediante la simulazione di ritardi e di cadute di rete occasionali. Sono prove molto più difficili, perché non potete semplicemente inserire combinazioni di tasti in un file. Mentre studiate questo libro, non occorre che vi preoccupiate di queste complicazioni, tuttavia esistono strumenti per au-

tomatizzare il collaudo in queste situazioni e i principi base del collaudo regressivo (non eliminare mai un caso di prova) e della copertura completa del collaudo (eseguire tutto il codice almeno una volta) rimangono validi.



## Suggerimenti per la produttività 1

### File batch e script di shell

Se dovete eseguire ripetutamente le stesse attività sulla riga comandi, vale la pena di imparare le caratteristiche di automazione offerte dal vostro sistema operativo.

In ambiente DOS, potete usare *file batch* per eseguire una serie di comandi automaticamente. Per esempio, immaginate di collaudare un programma che riceve tre file di ingresso:

```
java Program < test1.in
java Program < test2.in
java Program < test3.in
```

In seguito, scoprite un errore, lo correggete ed eseguite nuovamente le prove. A questo punto, avete bisogno di digitare ancora una volta i tre comandi. Ovviamente, deve esistere un sistema migliore per farlo. In ambiente DOS, inserite i comandi in un file di testo, che chiamerete `test.bat`:

#### File test.bat

```
java Program < test1.in
java Program < test2.in
java Program < test3.in
```

Quindi, per eseguire automaticamente i tre comandi contenuti nel file batch, è sufficiente digitare il comando seguente:

```
test
```

È facile migliorare l'utilizzo del file batch. Se avete finito con `Program` e iniziate a lavorare con `Program2`, naturalmente potete scrivere un file batch `test2.bat`, ma si può fare di meglio. Fornite al file un *parametro*, ovvero, chiamate il file batch mediante questo comando:

```
test Program
```

oppure:

```
test Program2
```

Per far funzionare il parametro, dovete modificare il file. In un file batch, `%1` rappresenta la prima stringa che digitate dopo il nome del file stesso, `%2` la seconda stringa, e così via.

**File test.bat**

```
java %1 < test1.in
java %1 < test2.in
java %1 < test3.in
```

Che cosa succede se si hanno più di tre file da collaudare? I file batch DOS offrono un ciclo for molto rudimentale:

**File test.bat**

```
for %%f in (test*.in) do java %1 < %%f
```

Se lavorate in un laboratorio di informatica, al momento di andare a casa vi conviene avere un file batch per copiare tutti i vostri file in un dischetto floppy. Inserire le righe seguenti nel file gohome.bat:

**File gohome.bat**

```
copy *.java a:
copy *.txt a:
copy *.in a:
```

Esistono infiniti utilizzi per i file batch, e vale davvero la pena di conoscerli meglio.

I file batch sono una caratteristica del sistema operativo DOS, non di Java. Nel sistema UNIX, per lo stesso scopo si usano gli *script di shell*.

## 4 Tracciamento, *logging* e asserzioni

Il tracciamento di un programma consiste di messaggi di tracciamento che mostrano il flusso dell'esecuzione.

Talvolta eseguite un programma e non siete sicuri di dove stia perdendo tempo. Per avere un prospetto del flusso di esecuzione del programma, potete inserire nel programma messaggi di tracciamento, come questo:

```
public double getTax()
{
    ...
    if (status == SINGLE)
    {
        System.out.println("status is SINGLE");
        ...
    }
    ...
}
```

Può anche essere utile stampare una traccia della pila di esecuzione (*stack trace*), che vi dice come il programma sia giunto in un certo punto. Usate queste istruzioni:

```
Throwable t = new Throwable();
t.printStackTrace(System.out);
```

## 16 Collaudo e correzione di errori

Una traccia della pila di esecuzione assomiglia a questa:

```
java.lang.Throwable
  at TaxReturn.getTax(TaxReturn.java:26)
  at TaxReturnTest.main(TaxReturnTest.java:30)
```

Una traccia della pila di esecuzione è formata da un elenco di tutte le invocazioni di metodi in attesa in un particolare istante di tempo.

Questa informazione è molto utile: il messaggio di tracciamento è stato generato all'interno del metodo `getTax` della classe `TaxReturn` (più precisamente, alla riga 26 del file `TaxReturn.java`) e tale metodo era stato invocato alla riga 30 del file `TaxReturnTest.java`.

Tuttavia, i messaggi di tracciamento pongono un problema: quando avete terminato il collaudo del programma, dovete eliminare tutti gli enunciati di visualizzazione che producono messaggi di tracciamento. Se, però, trovate un ulteriore errore, dovete inserirli nuovamente.

Per risolvere questo problema, potete usare la classe `Logger`, che consente di zittire i messaggi di tracciamento. Il supporto per tale registrazione di eventi (*logging*) è incluso nella libreria standard di Java a partire dalla versione 1.4.

Invece di stampare direttamente sul flusso `System.out`, usate l'oggetto di logging globale

```
Logger logger = Logger.getLogger("global");
```

e invocate

```
logger.info("status is SINGLE");
```

Per impostazione predefinita, il messaggio viene visualizzato. Ma se chiamate

```
logger.setLevel(Level.OFF);
```

la visualizzazione di tutti i messaggi di logging viene soppressa. In questo modo, potete sopprimere i messaggi di logging quando il programma funziona correttamente, e riabilitarli se trovate un nuovo errore.

Quando state seguendo il flusso di esecuzione di un programma con il tracciamento, gli eventi più importanti sono l'ingresso e l'uscita da un metodo. All'inizio di un metodo, visualizzate i parametri:

```
public TaxReturn(double anIncome, int aStatus)
{
    Logger logger = Logger.getLogger("global");
    logger.info("Parameters: anIncome = " + anIncome
              + " aStatus = " + aStatus);
    ...
}
```

Alla fine di un metodo, visualizzate il valore che verrà restituito:

```
public double getTax()
{
    ...
```

```

    logger.info("Return value = " + tax);
    return tax;
}

```

Per ottenere un tracciamento appropriato, dovete individuare *ciascun* punto di uscita di un metodo. Inserite un messaggio di tracciamento prima di ciascun enunciato `return` e alla fine del metodo.

Ovviamente, non siete costretti a limitarvi a messaggi di entrata e di uscita, ma potete documentare lo stato di avanzamento dell'esecuzione all'interno di un metodo. La classe `Logger` ha molte altre opzioni, per eseguire un'attività di logging adatta ad ambienti di programmazione professionale. Potete controllare la documentazione della classe nella libreria standard per avere un maggiore controllo sui messaggi di tracciamento.

Il tracciamento di un programma può servire per analizzarne il comportamento, ma esistono alcuni suoi precisi svantaggi. Capire quali messaggi di tracciamento bisogna inserire può fare perdere parecchio tempo. Se inserite troppi messaggi, produce una raffica di visualizzazioni che è difficile da analizzare, mentre, se ne inserite troppo pochi, potreste avere informazioni insufficienti per localizzare la causa dell'errore. Se vi sembra una scocciatura, non siete gli unici. La maggior parte dei programmatori professionisti, per trovare gli errori nel codice, usa un *debugger*, anziché usare i messaggi di tracciamento. Il debugger verrà descritto nel prossimo paragrafo.

Spesso i programmi contengono ipotesi implicite. Ad esempio, un tasso di interesse non dovrebbe essere negativo. Sicuramente, nessuno vorrebbe mai depositare denaro in un conto che frutta interessi negativi, ma un meccanismo simile potrebbe insinuarsi in un programma, a causa di un errore di elaborazione o nei valori di ingresso. A volte, è utile verificare che non sia accaduto uno di tali errori, prima di procedere con l'esecuzione del programma. Ad esempio, prima di estrarre la radice quadrata di un numero che sapete non essere negativo, potete voler verificare tale vostra ipotesi.

Un'asserzione è una condizione logica in un programma che ritenete essere vera.

Un'ipotesi ritenuta vera è detta *asserzione*. La verifica di un'asserzione controlla se essa è vera e termina il programma con una segnalazione d'errore in caso contrario. Ecco un tipico controllo di un'asserzione:

```

public void computeIntersection()
{
    ...
    double y = r * r - (x - a) * (x - a);
    assert y >= 0;
    root = Math.sqrt(y);
    ...
}

```

In questo stralcio di programma, il programmatore prevede che la quantità  $y$  non possa mai essere negativa. Se l'asserzione è corretta, la sua verifica non produce alcun danno e il programma funziona normalmente. Se, per qualche motivo, l'asserzione fallisce, al programmatore conviene che il programma termini, piuttosto che continuare, calcolare la radice quadrata di un numero negativo e causare un danno maggiore più avanti.

## 18 Collaudo e correzione di errori

Un comune utilizzo delle asserzioni consiste nel controllo delle pre-condizioni e delle post-condizioni. Ad esempio, ecco come controllare la pre-condizione del metodo `deposit` della classe `BankAccount`:

```
public double deposit(double amount)
{
    assert amount >= 0;
    balance = balance + amount;
}
```

Le asserzioni fanno parte del linguaggio Java a partire dalla versione 1.4; per compilare un programma con le asserzioni, eseguite il compilatore Java in questo modo

```
javac -source 1.4 MyProg.java
```

Per eseguire un programma con le asserzioni abilitate, usate questo comando:

```
java -enableassertions MyProg
```

## 5 Il debugger

Un debugger è un programma che potete usare per eseguire un altro programma e analizzare il suo comportamento durante l'esecuzione.

Come avrete certamente capito a questo punto, i programmi informativi raramente funzionano perfettamente la prima volta. A volte, può essere molto frustrante scovare gli errori. Naturalmente, potete inserire messaggi di tracciamento per evidenziare il flusso del programma e i valori delle variabili fondamentali, eseguire il programma e tentare di analizzarne il prospetto ottenuto. Se tale analisi non indica chiaramente il problema, potrebbe essere necessario aggiungere o eliminare comandi di stampa ed eseguire nuovamente il programma. È un meccanismo che può richiedere tempo. I moderni ambienti di sviluppo contengono programmi speciali, detti *debugger*, che vi aiutano a localizzare gli errori, permettendovi di seguire l'esecuzione di un programma. Potete interrompere e far proseguire il vostro programma, e vedere il contenuto delle variabili quando arrestate temporaneamente l'esecuzione. A ciascuna pausa, potete scegliere quali variabili esaminare e quanti passi di programma eseguire prima dell'interruzione successiva.

Alcuni credono che i debugger siano solo uno strumento per rendere pigri i programmatori. In verità, qualcuno scrive programmi sciatti e poi li sistema alla meglio grazie al debugger, ma la maggioranza tenta onestamente di scrivere il miglior programma possibile, prima di provare a eseguirlo mediante il debugger. Questi programmatori sanno che il debugger, sebbene sia più comodo degli enunciati di stampa, non è esente da costi, perché occorre tempo per impostare e per eseguire una vera sessione di debug.

Nella vera programmazione, non potete evitare di usare il debugger. Più grandi sono i vostri programmi, più difficile sarà correggerli inserendo semplicemente enunciati di stampa. Scoprirete che il tempo investito per imparare a usare il debugger verrà ampiamente ripagato nella vostra carriera di programmatori.

Come per i compilatori, i debugger differiscono ampiamente da un sistema all'altro. In alcuni sistemi, sono alquanto rudimentali e obbligano a ricordarsi una piccola

serie di comandi misteriosi; in altri, hanno un'intuitiva interfaccia a finestre. Le schermate presentate in questo capitolo mostrano il debugger dell'ambiente di sviluppo Forte Community Edition, che si può attualmente ottenere gratuitamente dal sito di Sun Microsystems.

Dovrete scoprire come preparare un programma per il debug e come avviare il debugger nel vostro sistema. Se usate un ambiente di sviluppo integrato, che contiene un editor, un compilatore e un debugger, normalmente questa operazione è molto facile. Semplicemente, costruite il programma nel modo abituale e selezionate un comando di menu per avviare il debugger. In molti sistemi UNIX, dovete costruire manualmente una versione del vostro programma per il debug e quindi invocare il debugger.

Potete usare efficacemente il debugger anche solo comprendendo a fondo tre concetti: punti di arresto (*breakpoint*), esecuzione del programma una riga alla volta (*single step*) e ispezione del valore di singole variabili.

Una volta avviato il debugger, potete fare molto lavoro con tre soli comandi di debug: "imposta un punto di arresto" (*breakpoint*), "esegui la sola riga di codice successiva" (*single step*) ed "esamina la variabile". I nomi da digitare sulla tastiera o le selezioni del mouse che servono per eseguire questi comandi differiscono molto per diversi debugger, ma tutti mettono a disposizione queste operazioni fondamentali. Dovete scoprire il modo di utilizzo, dalla documentazione o dal manuale del laboratorio, oppure chiedendo a qualcuno che ha già usato il debugger.

Quando il debugger esegue un programma, l'esecuzione viene interrotta ogni volta che viene raggiunto un punto di arresto.

Quando fate partire il debugger, il programma viene eseguito a velocità piena fino al raggiungimento di un punto di arresto, nel quale l'esecuzione viene sospesa e viene visualizzato il breakpoint che ha provocato l'arresto (osservate la Figura 3). A questo punto potete ispezionare variabili ed eseguire il programma una riga alla volta, oppure con-

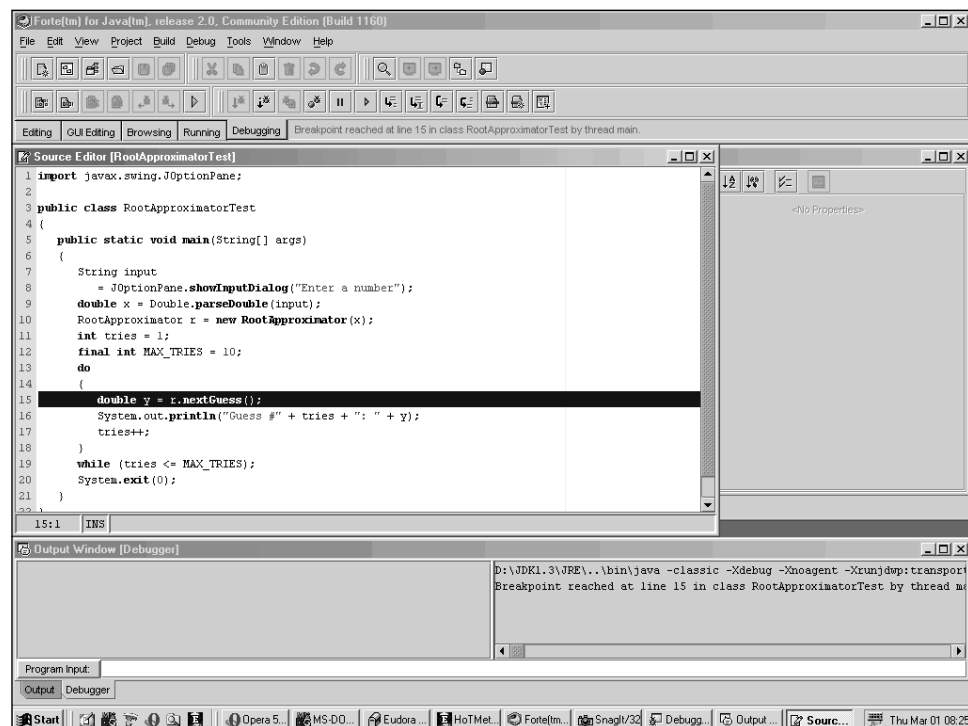
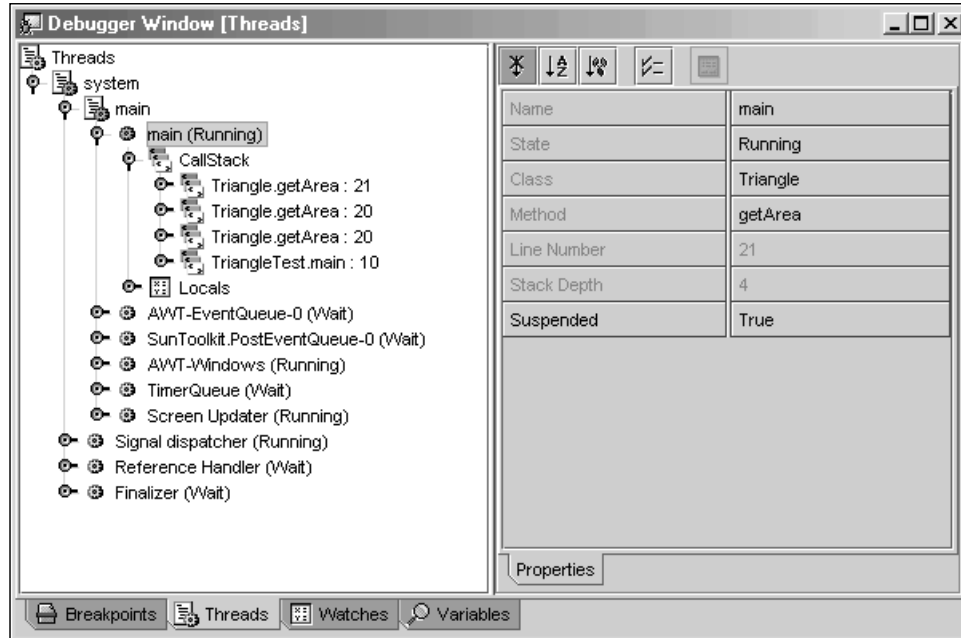


Figura 3

Il debugger fermo a un punto di arresto



**Figura 4**  
Ispezionare  
una variabile

tinuare l'esecuzione del programma a velocità piena fino al raggiungimento del breakpoint successivo. Quando il programma termina, termina anche l'esecuzione del debugger.

Quando il programma si è fermato, potete osservare i valori attuali delle variabili. Ancora una volta, il metodo per selezionare le variabili è diverso per ciascun debugger: alcuni visualizzano sempre una finestra che contiene i valori delle variabili locali (osservate la Figura 4). In altri, dovete eseguire un comando del tipo "esamina la variabile" e digitare o selezionare il nome della variabile, dopodiché il debugger visualizzerà il contenuto della variabile. Se tutte le variabili contengono quello che vi aspettate, potete eseguire il programma fino al punto successivo in cui volete fermarlo.

Quando esaminate oggetti, spesso avrete bisogno di fornire un comando per "aprire" l'oggetto, per esempio selezionando un nodo di un albero. Una volta aperto l'oggetto, potete vedere le sue variabili istanza (osservate la Figura 4).

Il comando di debugging *single step* esegue il programma una riga alla volta.

Eseguire il programma fino a un punto di arresto vi porta velocemente fino al punto che vi interessa, ma non sapete come il programma vi sia arrivato. Come alternativa, potete eseguire il programma una riga alla volta: in questo modo sapete quale sia il flusso di esecuzione del programma, ma può darsi che ci voglia molto tempo per arrivare a un certo punto. Il comando "single step" esegue la linea di codice attuale e si interrompe alla linea successiva. La maggior parte dei debugger hanno due tipi di comandi "single step", uno chiamato "step into", cioè "fai un passo all'interno", che fa procedere l'esecuzione di una riga per volta all'interno dei metodi invocati, e uno chiamato "step over", cioè "fai un passo scavalcando", che esegue le invocazioni di metodi senza arrestarsi al loro interno.

Ad esempio, supponete che la linea attuale sia

```
String token = tokenizer.nextToken();
Word w = new Word(token);
int syllables = w.countSyllables();
System.out.println("Syllables in " + w.getText()
    + ": " + syllables);
```

Quando eseguite un comando di tipo “step over” in un punto in cui esiste l’invocazione di un metodo, procedete fino alla riga successiva:

```
String token = tokenizer.nextToken();
Word w = new Word(token);
int syllables = w.countSyllables();
System.out.println("Syllables in " + w.getText()
    + ": " + syllables);
```

Tuttavia, se eseguite un comando di tipo “step into”, vi trovate nella prima linea del metodo `countSyllables`.

```
public int countSyllables()
{
    int count = 0;
    int end = text.length() - 1;
    ...
}
```

Dovreste entrare in un metodo per verificare che svolga correttamente il suo compito, mentre dovrete scavalcarlo se sapete già che funziona bene.

Infine, quando il programma termina l’esecuzione completamente, si conclude anche la sessione di debug e non potete più esaminare le variabili. Per eseguire il programma nuovamente, dovrete essere in grado di far ripartire il debugger, oppure può darsi che dobbiate uscire dal debugger stesso e iniziare da capo. I particolari dipendono dal tipo di debugger.

## 6 Un esempio di sessione di debugging

Per avere a disposizione un esempio realistico sul quale eseguire il debugger, mettiamo a punto una classe `Word` il cui compito principale è quello di contare le sillabe presenti in una parola, usando la seguente regola:

Ogni *gruppo* di vocali (a, e, i, o, u, y) adiacenti fa parte di una sillaba (ad esempio, il gruppo “ea” in “real” forma una sillaba, mentre “e..a” in “regal” forma due sillabe). Tuttavia, una “e” alla fine di una parola non forma una sillaba. Ancora, ogni parola ha almeno una sillaba, anche se le regole precedenti forniscono un conteggio nullo. Infine, quando viene costruita la parola a partire da una stringa, eventuali caratteri all’inizio o alla fine della stringa che non siano lettere vengono eliminati. Questo è utile quando si usa un oggetto di tipo `StringTokenizer` per spez-

zare una frase in token: i token possono ancora contenere segni di punteggiatura, ma non vogliamo che facciano parte della parola.

Ecco il codice sorgente per la classe, contenente un paio di errori.

### File Word.java

```
public class Word
{
    /**
     * Costruisce una parola eliminando caratteri iniziali e finali
     * che non siano lettere, come i segni di punteggiatura.
     * @param s la stringa di ingresso
     */
    public Word(String s)
    {
        int i = 0;
        while (i < s.length()
            && !Character.isLetter(s.charAt(i)))
            i++;
        int j = s.length() - 1;
        while (j > i
            && !Character.isLetter(s.charAt(j)))
            j--;
        text = s.substring(i, j);
    }

    /**
     * Restituisce il testo della parola, dopo aver rimosso
     * i caratteri iniziali e finali che non siano lettere.
     * @return il testo della parola
     */
    public String getText()
    {
        return text;
    }

    /**
     * Conta le sillabe nella parola.
     * @return il numero di sillabe
     */
    public int countSyllables()
    {
        int count = 0;
        int end = text.length() - 1;
        if (end < 0) return 0; // la stringa vuota non ha sillabe

        // una e alla fine della parola non conta come vocale
        char ch = Character.toLowerCase(text.charAt(end));
        if (ch == 'e') end--;

        boolean insideVowelGroup = false;
        for (int i = 0; i <= end; i++)
        {
```

```

        ch = Character.toLowerCase(text.charAt(i));
        if ("aeiouy".indexOf(ch) >= 0)
        {
            // ch è una vocale
            if (!insideVowelGroup)
            {
                // inizia un nuovo gruppo di vocali
                count++;
                insideVowelGroup = true;
            }
        }
    }

    // ogni parola ha almeno una sillaba
    if (count == 0)
        count = 1;

    return count;
}

private String text;
}

```

Ecco una semplice classe di prova. Inserite una o più parole nella finestra di dialogo, e verrà visualizzato il conteggio delle sillabe di tutte le parole.

#### File WordTest.java

```

import java.util.StringTokenizer;
import javax.swing.JOptionPane;

public class WordTest
{
    public static void main(String[] args)
    {
        String input = JOptionPane.showInputDialog(
            "Enter a sentence");
        StringTokenizer tokenizer =
            new StringTokenizer(input);
        while (tokenizer.hasMoreTokens())
        {
            String token = tokenizer.nextToken();
            Word w = new Word(token);
            int syllables = w.countSyllables();
            System.out.println("Syllables in " + token
                + ": " + syllables);
        }
        System.exit(0);
    }
}

```

Quando eseguite questo programma con l'ingresso `hello regal real`, viene visualizzato quanto segue:

```
Syllables in hello: 1
Syllables in regal: 1
Syllables in real: 1
```

Non è molto promettente.

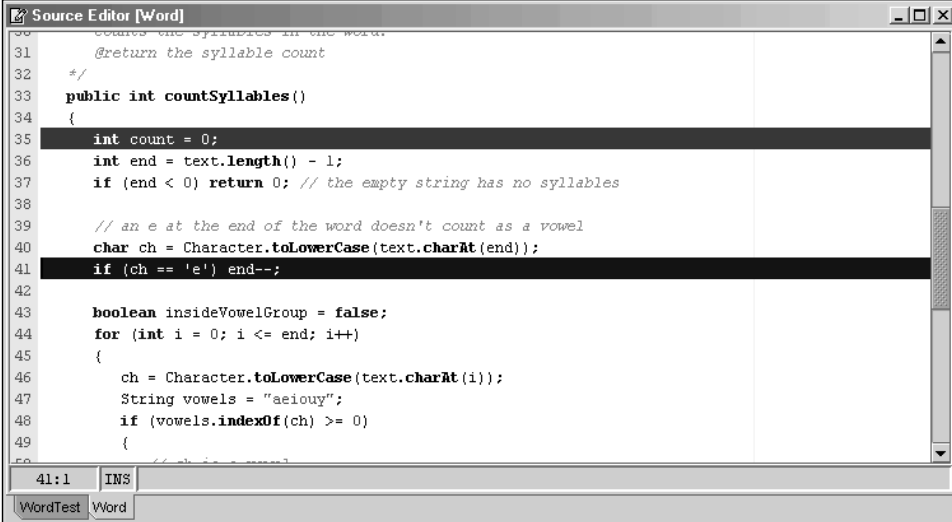
Prima di tutto, impostate un punto di arresto nella prima linea del metodo `countSyllables` della classe `Word`, nella linea 35 del file `Word.java`, poi fate partire il programma, che chiederà i dati in ingresso. Per ora, fornite soltanto il valore d'ingresso `hello`. Il programma si arresterà al breakpoint che avete impostato.

Come prima cosa, il metodo `countSyllables` controlla l'ultimo carattere della parola per vedere se è una lettera 'e'. Vediamo se questo funziona correttamente: eseguite il programma fino alla linea 41 (osservate la Figura 5).

Ora ispezionate la variabile `ch`. Questo particolare debugger ha un comodo visualizzatore per le variabili locali e di istanza (osservate la Figura 6); se il vostro non lo ha, può darsi che dobbiate ispezionare `ch` manualmente. Potere verificare che `ch` contiene il valore 'l', cosa un po' strana. Osservate il codice sorgente: la variabile `end` è stata impostata al valore `text.length() - 1`, l'ultima posizione nella stringa `text`, e `ch` è il carattere che si trova in tale posizione.

Proseguendo nell'analisi, troverete che `end` vale 3, non 4, come vi aspettate. E `text` contiene la stringa "hell", non "hello". Quindi, non c'è da meravigliarsi che `countSyllables` fornisca la risposta 1. Dobbiamo guardare altrove per risolvere il problema: sembra che l'errore sia nel costruttore di `Word`.

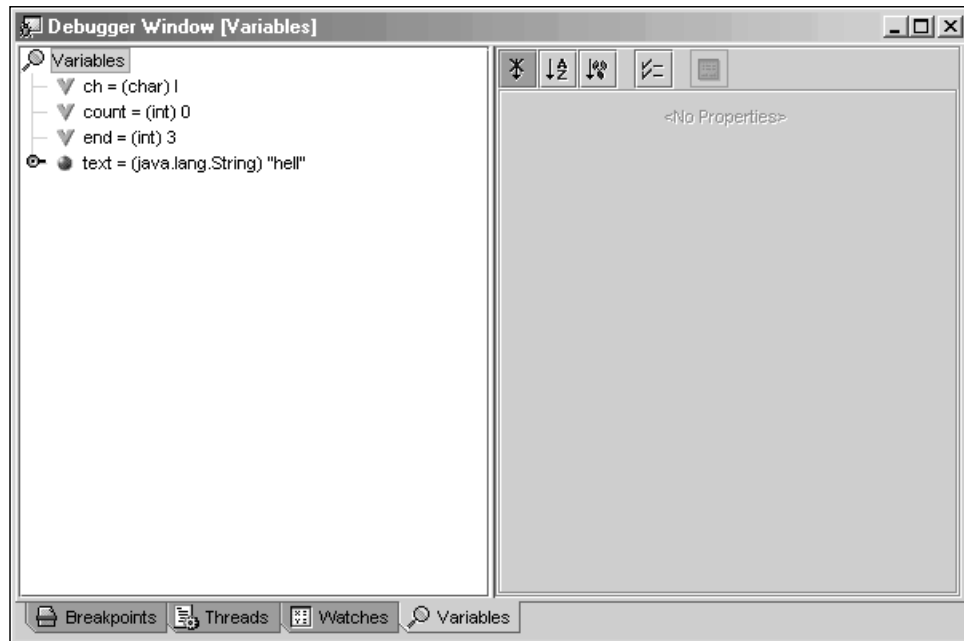
Sfortunatamente, un debugger non può tornare indietro nel tempo, per cui dovete interrompere il programma, impostare un breakpoint nel costruttore di `Word` e far ripartire il debugger. Fornite di nuovo "hello" come dato in ingresso. Il debugger si arresterà all'inizio del costruttore di `Word`, che imposta due variabili `i` e `j`, trascurando tutti i caratteri che non siano lettere all'inizio e alla fine della stringa di ingresso. Impostate un breakpoint dopo la fine del secondo ciclo (osservate la Figura 7) in modo da poter ispezionare i valori di `i` e `j`.



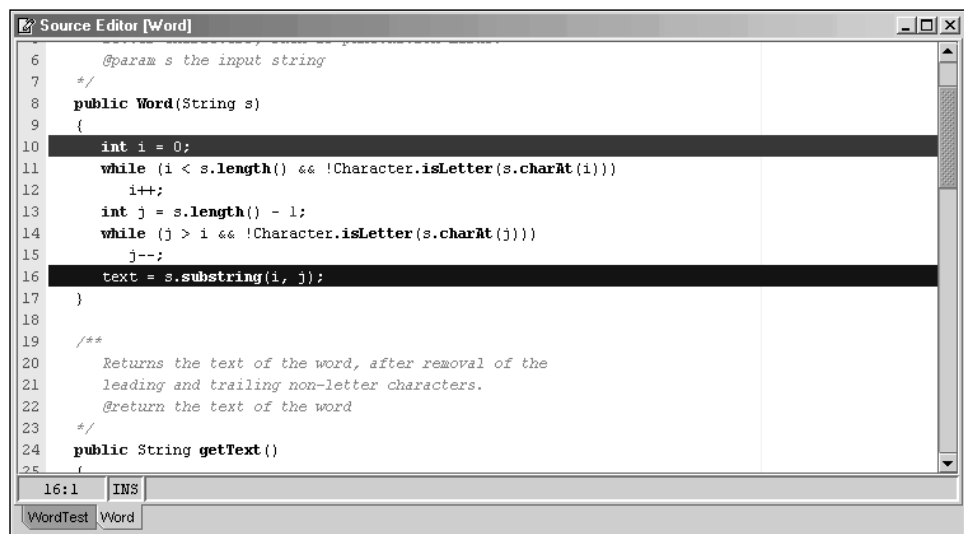
```
Source Editor [Word]
30  counts the syllables in the word.
31  @return the syllable count
32  */
33  public int countSyllables()
34  {
35  int count = 0;
36  int end = text.length() - 1;
37  if (end < 0) return 0; // the empty string has no syllables
38
39  // an e at the end of the word doesn't count as a vowel
40  char ch = Character.toLowerCase(text.charAt(end));
41  if (ch == 'e') end--;
42
43  boolean insideVowelGroup = false;
44  for (int i = 0; i <= end; i++)
45  {
46      ch = Character.toLowerCase(text.charAt(i));
47      String vowels = "aeiouy";
48      if (vowels.indexOf(ch) >= 0)
49      {
50          // ch is a vowel
51          if (!insideVowelGroup)
52              count++;
53          insideVowelGroup = true;
54      }
55  }
56  return count;
57  }
```

**Figura 5**

Verificare il metodo `countSyllables` con il debugger



**Figura 6**  
I valori attuali  
delle variabili locali  
e di istanza



**Figura 7**  
Verificare  
il costruttore di Word  
con il debugger

A questo punto, l'ispezione di `i` e `j` mostra che `i` vale 0 e che `j` vale 4. Ciò ha senso, perché non ci sono segni di punteggiatura da ignorare. Quindi, perché `text` viene impostata a "hell"? Ricordate che il metodo `substring` considera le posizioni fino al secondo carattere *senza includerlo*, per cui l'invocazione corretta dovrebbe essere

```
text = s.substring(i, j + 1);
```

Questo è un tipico errore “per scarto di uno”.

Correggete questo errore, ricompilate il programma e provate di nuovo i tre casi di prova. Otterrete la seguente visualizzazione in uscita:

```
Syllables in hello: 2
Syllables in regal: 1
Syllables in real: 1
```

Va meglio, ma c'è ancora un problema. Eliminate tutti i punti di arresto e impostate un nuovo breakpoint nel metodo `countSyllables`. Fate partire il debugger e fornite in ingresso la stringa "regal". Quando il debugger si arresta al punto prestabilito, iniziate a eseguire il programma passo passo (cioè un'istruzione per volta, in modalità *single step*) all'interno del metodo. Ecco il codice del ciclo che conta le sillabe:

```
boolean insideVowelGroup = false;
for (int i = 0; i <= end; i++)
{
    ch = Character.toLowerCase(text.charAt(i));
    if ("aeiouy".indexOf(ch) >= 0)
    {
        // ch è una vocale
        if (!insideVowelGroup)
        {
            // inizia un nuovo gruppo di vocali
            count++;
            insideVowelGroup = true;
        }
    }
}
```

Nella prima iterazione del ciclo, il debugger non esegue l'enunciato `if`: ciò è corretto, perché la prima lettera, 'r', non è una vocale. Nella seconda iterazione, il debugger entra nell'enunciato `if`, come dovrebbe, perché la seconda lettera, 'e', è una vocale. La variabile `insideVowelGroup` viene impostata a `true` e il contatore delle vocali viene incrementato. Nella terza iterazione l'enunciato `if` viene ignorato nuovamente, perché la lettera 'g' non è una vocale, ma nella quarta iterazione accade qualcosa di strano. La lettera 'a' è una vocale e l'enunciato `if` viene eseguito, ma il secondo enunciato `if` viene ignorato e la variabile `count` non viene nuovamente incrementata.

Perché? La variabile `insideVowelGroup` è ancora `true`, anche se il primo gruppo di vocali era terminato quando è stata esaminata la consonante 'g'. La lettura di una consonante dovrebbe riportare `insideVowelGroup` al valore `false`. Questo è un errore logico più subdolo, ma non inusuale quando si progetta un ciclo che tiene traccia dello stato di un processo. Per correggerlo, interrompete il debugger e aggiungete la clausola seguente:

```
if ("aeiouy".indexOf(ch) >= 0)
{
    ...
}
else insideVowelGroup = false;
```

Ora ricompilate ed eseguite di nuovo il collaudo, ottenendo:

```
Syllables in hello: 2
Syllables in regal: 2
Syllables in real: 1
```

Il debugger può essere usato soltanto per analizzare la presenza di errori, non per mostrare che un programma ne è privo.

Il programma è ora privo di errori? Questa non è una domanda alla quale il debugger possa rispondere. Ricordate: il collaudo può soltanto evidenziare la presenza di errori, non la loro assenza.

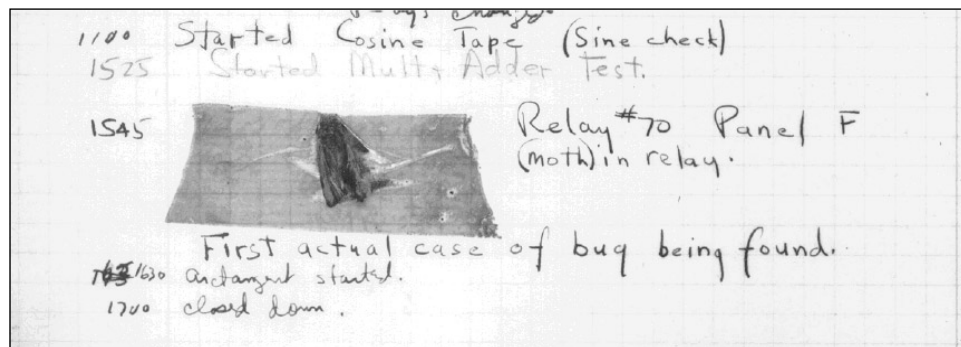


## Note di cronaca 1

### Il primo bug

Secondo la leggenda, il primo bug fu quello rinvenuto nel 1947 nel Mark II, un enorme computer elettromeccanico presso la Harvard University. Fu veramente causato da un insetto (*bug*), una falena intrappolata in un interruttore a relè. In realtà, dalla nota che l'operatore lasciò nel registro, accanto alla falena (osservate la Figura 8), sembra che all'epoca il termine "bug" venisse già impiegato nell'uso quotidiano.

Maurice Wilkes, un pioniere della ricerca informatica, scrisse: "Per qualche motivo, alla Moore School e in seguito, si è sempre pensato che non vi sarebbero state particolari difficoltà a scrivere programmi corretti. Posso ricordarmi il momento esatto in cui mi fu chiaro che gran parte della mia vita futura sarebbe trascorsa nel trovare gli errori nei miei programmi".



**Figura 8**  
Il primo "bug"



## Consigli pratici 1

### Il collaudo

Ora conoscete i meccanismi del collaudo, ma tutta questa conoscenza può ancora lasciarvi disarmati quando eseguite il debugger per esaminare un programma

difettoso. Ecco alcune strategie che potete utilizzare per individuare gli errori e le loro cause.

#### Passo 1. Riproducete l'errore

Mentre controllate il vostro programma, notate che talvolta il codice fa qualcosa di sbagliato. Per esempio, fornisce un valore errato in uscita, sembra stampare qualcosa completamente a caso, imbecca un ciclo infinito, o interrompe bruscamente la propria esecuzione. Scoprite esattamente come *riprodurre* questo comportamento: quali numeri inserite? Dove premete il pulsante del mouse?

Eseguite nuovamente il programma; inserite esattamente le stesse risposte e premete il pulsante del mouse negli stessi punti (o il più vicino possibile). Se il programma esibisce lo stesso comportamento, è il momento di eseguire il debugger per studiare questo particolare problema. I debugger vanno bene per analizzare anomalie precise, mentre non sono molto utili per esaminare un programma in generale.

#### Passo 2. Semplificate l'errore

Prima di eseguire il debugger, è utile spendere alcuni minuti cercando di individuare una situazione più semplice che provochi lo stesso errore. Ottenete comunque un comportamento errato del programma, anche inserendo parole più brevi o numeri più semplici? Se ciò accade, usate tali valori nelle vostre sessioni di utilizzo del debugger.

#### Passo 3. Dividete per vincere (*divide and conquer*)

Ora che siete alle prese con una precisa anomalia, è opportuno avvicinarsi il più possibile all'errore. Il punto chiave dell'utilizzo del debugger è la localizzazione del punto preciso nel codice che produce il comportamento errato. Proprio come con i veri insetti nocivi, trovare un bug può essere difficile, ma, una volta che l'avete trovato, eliminarlo è solitamente la parte più facile. Immaginate che il programma termini l'esecuzione con una divisione per zero. Dal momento che un tipico programma contiene molte operazioni di divisione, spesso non è possibile impostare punti di interruzione per tutte. Piuttosto, usate la tecnica del *dividi per vincere*. Eseguite con il debugger i metodi del `main`, senza entrare al loro interno. A un certo punto, l'anomalia apparirà e saprete quale metodo contiene l'errore: è l'ultimo chiamato da `main` prima della brusca terminazione del programma. Eseguite nuovamente il debugger e tornate alla stessa riga in `main`, poi entrate all'interno del metodo, ripetendo il procedimento.

Usate la tecnica del "dividere per vincere" allo scopo di identificare il punto in cui il programma fallisce.

Alla fine, individuerete la riga che contiene la divisione sbagliata. Può darsi che il codice riveli chiaramente perché il denominatore non è corretto, altrimenti dovete trovare il punto dove viene calcolato. Sfortunatamente, nel debugger non potete andare *a ritroso*, ma dovete eseguire nuovamente il programma e portarvi al punto in cui viene calcolato il denominatore.

#### Passo 4. Siate consapevoli di ciò che il programma dovrebbe fare

Durante il debugging, confrontate il contenuto delle variabili con i valori che dovrebbero avere, secondo le vostre conoscenze.

Il debugger vi mostra che cosa *fa* il programma, ma voi dovete sapere che cosa *dovrebbe* fare, altrimenti non sarete in grado di trovare gli errori. Prima di seguire il tracciamento dell'esecuzione attraverso un ciclo, chiedetevi quante iterazioni vi *aspettate* che esegua il programma. Prima di ispezionare il valore di una variabile, domandatevi che cosa prevedete di trovare. Se non ne avete idea, riservatevi un po' di tempo e riflettete prima di agire. Munitevi di una pratica calcolatrice tascabile per eseguire calcoli indipendenti ed esaminate la variabile quando sapete qual è il valore corretto: questo è il momento della verità. Se il programma è ancora sulla strada giusta, il valore sarà quello previsto e potrete cercare l'errore più avanti. Se il valore è diverso, potrete aver trovato qualcosa. Controllate nuovamente il vostro calcolo: se siete sicuri che il valore sia corretto, scoprite perché il programma giunge a una conclusione diversa.

In molti casi, i bug di un programma sono il risultato di semplici errori, come la condizione per l'uscita da un ciclo sbagliata per scarto di uno. Piuttosto spesso, tuttavia, i programmi cadono su errori di calcolo: magari si pensava di sommare due numeri, ma per sbaglio si è scritto il codice per sottrarli. Diversamente dal vostro assistente di matematica, i programmi (come i problemi del mondo reale) non fanno uno sforzo speciale per assicurarsi che tutti i calcoli si svolgano con semplici numeri interi, e avrete bisogno di fare alcune operazioni con numeri elevati, oppure con complicate cifre in virgola mobile. Talvolta, questi calcoli si possono evitare, domandandosi semplicemente se una certa quantità deve essere positiva, oppure se deve superare un certo valore, e quindi esaminando le variabili per verificarlo.

#### **Passo 5.** Controllate tutti i dettagli

Quando fate il debug di un programma, spesso avete già un'idea della natura del problema. Nondimeno, mantenete una mentalità aperta e guardate a tutti i particolari circostanti. Quali strani messaggi vengono visualizzati? Perché il programma intraprende un'azione diversa e inaspettata? Questi dettagli hanno la loro importanza. Quando eseguite una sessione di debug, siete veramente nella posizione di un detective che deve cogliere qualsiasi indizio disponibile.

Se notate un'altra anomalia, mentre state mettendo a punto il problema, non limitatevi a pensare di tornarvi sopra più tardi, perché potrebbe essere la vera causa del problema attuale. È meglio prendere un appunto per il problema principale, correggere l'anomalia appena trovata, e poi tornare al problema originario.

#### **Passo 6.** Siate certi di aver compreso ciascun errore prima di correggerlo

Quando si scopre che un ciclo esegue troppe iterazioni, si ha la grossa tentazione di "mettere un rattoppo", magari sottraendo un'unità da una variabile, in modo che quel particolare problema non si ripresenti più. Una soluzione rapida di questo tipo ha probabilità schiacciante di creare problemi in qualche altro punto del codice. In realtà, dovete capire a fondo come andrebbe scritto il programma, prima di applicare un rimedio.

Talvolta, succede di scoprire un errore dopo l'altro, e di inserire correzioni dopo correzioni, mentre si gira semplicemente intorno al problema. Generalmente, è il sintomo che esiste un problema maggiore in merito alla logica del programma. In questo

caso, con il debugger si può fare poco, e occorre ripensare la struttura del programma per riorganizzarlo.



## Note di cronaca 2

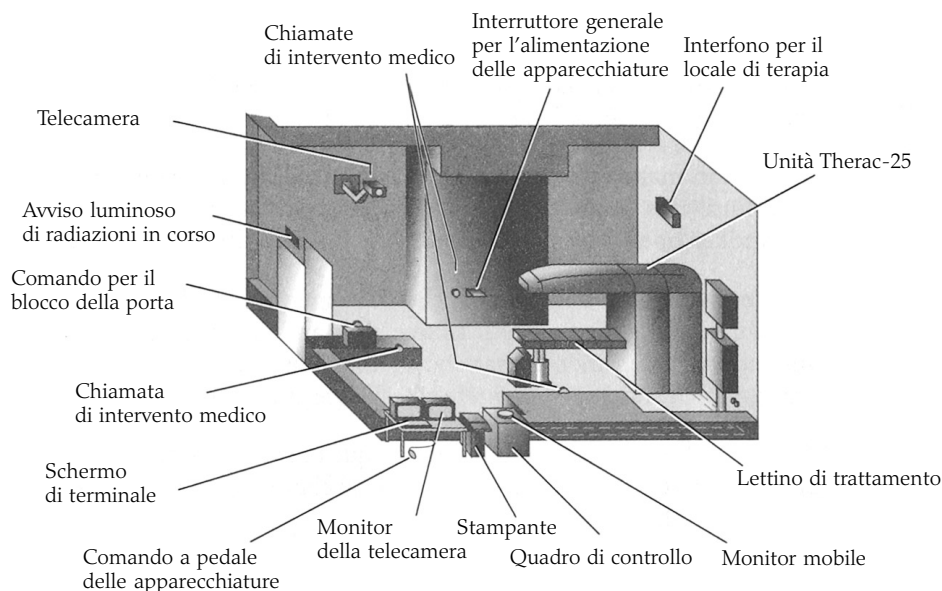
### Le vicende del Therac-25

Il Therac-25 è un dispositivo computerizzato per la terapia a emissione di radiazioni, destinata ai malati di cancro (osservate la Figura 9). Fra il giugno 1985 e il gennaio 1987, alcune di queste macchine rilasciarono dosi eccessive di radiazioni ad almeno sei pazienti, uccidendone alcuni e menomando seriamente gli altri.

Le macchine erano controllate da un programma informatico, i cui errori furono direttamente responsabili delle dosi eccessive. Secondo il libro [1], il programma fu scritto da un unico programmatore, che in seguito lasciò la società costruttrice che produceva l'apparecchio e non fu più possibile rintracciarlo. Nessuno, fra i dipendenti della società che furono interrogati, fu in grado di dire alcunché circa il livello di studi o le qualifiche del programmatore.

L'indagine dell'agenzia federale Food and Drug Administration (FDA) rilevò che il programma era scarsamente documentato e che non esisteva né un elenco di specifiche, né un piano formale di collaudo. (Questo dovrebbe farvi pensare. Avete un piano formale per il collaudo dei vostri programmi?)

Le dosi eccessive erano da imputare alla progettazione dilettantistica del software, che doveva controllare simultaneamente diversi dispositivi, ovvero la tastiera, lo schermo, la stampante e, naturalmente, l'apparecchio per le radiazioni. La sincronizzazio-



**Figura 9**

Una tipica struttura Therac-25

ne e la condivisione dei dati fra le attività erano supportate da una soluzione ad hoc, nonostante fossero già conosciute all'epoca le tecniche di multitasking in sicurezza. Se il programmatore avesse beneficiato di studi regolari in merito a queste tecniche, o se si fosse preso il disturbo di studiare la letteratura sull'argomento, avrebbe potuto costruire una macchina più sicura. Probabilmente, una macchina simile avrebbe comportato l'adozione di un sistema multitasking, scelto fra quelli in commercio, che forse avrebbe richiesto un computer più costoso.

Gli stessi difetti erano presenti nel software che controllava il modello precedente, il Therac-20, ma quella macchina conteneva blocchi hardware che impedivano meccanicamente l'eccesso di radiazioni. Nel Therac-25, i dispositivi di sicurezza hardware vennero eliminati per sostituirli con controlli nel software, presumibilmente per risparmiare.

Frank Houston, della FDA, nel 1985 scrisse [1]: "Una quantità significativa di software, nei sistemi critici per la salute, proviene da piccole aziende, specialmente nell'industria delle apparecchiature mediche. Si tratta di aziende che rientrano nel profilo di quelle refrattarie, o all'oscuro, dei principi sia della sicurezza dei sistemi, sia della progettazione del software."

È difficile individuare il colpevole: il programmatore? Il dirigente, che non solo non si è assicurato che il programmatore fosse all'altezza del compito, ma che pure non ha preteso un collaudo completo? Gli ospedali che hanno installato l'apparecchiatura, o la FDA, che non ha controllato il processo di progettazione? Sfortunatamente, ancora oggi non esistono standard aziendali che definiscano un processo sicuro per la progettazione del software.

## Riepilogo del capitolo

1. Usate il collaudo di unità per collaudare singole classi, separatamente l'una dall'altra.
2. Per eseguire un collaudo, scrivete una *infrastruttura di collaudo*.
3. Se leggete i valori di prova in ingresso da un file, potete facilmente ripetere il collaudo.
4. I casi limite sono casi di prova che si collocano al confine tra i valori accettabili e quelli non validi.
5. Un oracolo è un metodo lento ma affidabile per calcolare un risultato a scopi di collaudo.
6. Un pacchetto di prova è un insieme di prove da ripetere per il collaudo.
7. Il collaudo regressivo contempla l'esecuzione ripetuta di prove già eseguite in precedenza, per essere certi che guasti noti delle versioni precedenti non compaiano nelle nuove versioni del programma.
8. Il collaudo a scatola chiusa (*black-box testing*) descrive una metodologia di collaudo che non prende in considerazione la struttura dell'implementazione.
9. Il collaudo trasparente (*white-box testing*) usa informazioni sulla struttura del programma.

10. La copertura di un collaudo è una misura di quante parti di un programma siano state collaudate.
11. Il tracciamento di un programma consiste di messaggi di tracciamento che mostrano il flusso dell'esecuzione.
12. Una traccia della pila di esecuzione è formata da un elenco di tutte le invocazioni di metodi in attesa in un particolare istante di tempo.
13. Un'asserzione è una condizione logica in un programma che ritenete essere vera.
14. Un debugger è un programma che potete usare per eseguire un altro programma e analizzare il suo comportamento durante l'esecuzione.
15. Potete usare efficacemente il debugger anche solo comprendendo a fondo tre concetti: punti di arresto (*breakpoint*), esecuzione del programma una riga alla volta (*single step*) e ispezione del valore di singole variabili.
16. Quando il debugger esegue un programma, l'esecuzione viene interrotta ogni volta che viene raggiunto un punto di arresto.
17. Il comando di debugging *single step* esegue il programma una riga alla volta.
18. Il debugger può essere usato soltanto per analizzare la presenza di errori, non per mostrare che un programma ne è privo.
19. Usate la tecnica del "dividere per vincere" allo scopo di identificare il punto in cui il programma fallisce.
20. Durante il debugging, confrontate il contenuto delle variabili con i valori che dovrebbero avere, secondo le vostre conoscenze.

## Ulteriori letture

- [1] Nancy G. Leveson e Clark S. Turner, "An Investigation of the Therac-25 Accidents", *IEEE Computer*, luglio 1993, pagg. 18-41.

## Classi, oggetti e metodi presentati nel capitolo

```
java.lang.Throwable  
    printStackTrace  
java.util.logging.Logger  
    getLogger  
    info  
    setLevel
```

## Esercizi di ripasso

**Esercizio R.1.** Definite i termini *collaudo di unità* e *infrastruttura di collaudo*.

**Esercizio R.2.** Che cos'è un oracolo?

**Esercizio R.3.** Definite i termini *collaudo regressivo* e *pacchetto di prova*.

**Esercizio R.4.** Nel collaudo, qual è il fenomeno definito “ciclicità”? Che cosa potete fare per evitarlo?

**Esercizio R.5.** La funzione di arcoseno è l'inverso di quella di seno. Ovvero, se  $x = \sin(y)$ , allora  $y = \arcsin(x)$ . La funzione è definita soltanto se:  $-1 \leq x \leq 1$ . Supponete di dovere scrivere un metodo Java per calcolare la funzione arcoseno. Elencate tre casi di prova positivi e un caso limite, con i rispettivi valori di ritorno previsti, e due casi prova negativi.

**Esercizio R.6.** Che cos'è il tracciamento di un programma? Quando è più indicato usare il tracciamento di un programma, e quando un debugger?

**Esercizio R.7.** Spiegate le differenze fra queste operazioni di debugging:

- Procedere con l'esecuzione all'interno di un metodo
- Procedere con l'esecuzione senza entrare in un metodo

**Esercizio R.8.** Spiegate in dettaglio come esaminare le informazioni contenute in un oggetto di tipo `Point2D.Double` nel vostro debugger.

**Esercizio R.9.** Spiegate in dettaglio come esaminare le informazioni contenute in un oggetto di tipo `String` nel vostro debugger.

**Esercizio R.10.** Spiegate in dettaglio come usare il vostro debugger per esaminare il saldo contenuto in un oggetto di tipo `BankAccount`.

**Esercizio R.11.** Spiegate la strategia “dividi per vincere” per avvicinarsi a un errore usando il debugger.

**Esercizio R.12.** Le affermazioni seguenti sono vere o false?

- Se un programma ha superato tutte le prove di un pacchetto di prove, non ha più errori.
- Se un programma contiene un errore, l'errore emergerà sempre quando si esegue il programma nel debugger.
- Una volta dimostrato che tutti i metodi di un programma sono corretti, il programma è esente da errori.

## Esercizi di programmazione

**Esercizio P.1.** La funzione di arcoseno è l'inverso di quella di seno. Ovvero, se:

$$y = \arcsin(x)$$

allora:

$$x = \sin(y)$$

Per esempio:

$$\begin{aligned} \arcsin(0) &= 0 \\ \arcsin(0.5) &= \pi/6 \\ \arcsin(\sqrt{2}/2) &= \pi/4 \\ \arcsin(\sqrt{3}/2) &= \pi/3 \\ \arcsin(1) &= \pi/2 \\ \arcsin(-1) &= \pi/2 \end{aligned}$$

L'arcoseno si può definire solo per valori compresi fra  $-1$  e  $1$ . Spesso, questa funzione è espressa anche nella forma  $\sin^{-1}(x)$ . Notate, tuttavia, che non è la stessa cosa di  $1/\sin(x)$ . Nella libreria standard di Java, esiste un metodo per calcolare l'arcoseno, ma non dovete usarlo per questo esercizio. Scrivete un metodo Java per calcolare l'arcoseno dalla sua espansione in serie di Taylor:

$$\arcsin(x) = x + x^3/3! + x^5 \cdot 3^2/5! + x^7 \cdot 3^2 \cdot 5^2/7! + x^9 \cdot 3^2 \cdot 5^2 \cdot 7^2/9! + \dots$$

*Suggerimento:* non calcolate le potenze e i fattoriali esplicitamente, ma calcolate ogni termine a partire dal valore del termine precedente

Dovete calcolare le somme fino a quando un nuovo termine è minore di  $10^{-6}$ . Questo metodo verrà usato negli esercizi successivi.

**Esercizio P.2.** Scrivete una semplice infrastruttura di collaudo per la classe `ArcSinApproximator` che legga numeri in virgola mobile dall'ingresso standard e che calcoli il loro arcoseno, finché raggiunge la fine dei dati in ingresso. Quindi, eseguite il programma e confrontate i suoi valori di uscita con i valori di arcoseno forniti da una calcolatrice scientifica.

**Esercizio P.3.** Scrivete un'infrastruttura di collaudo che generi automaticamente casi di prova per la classe `ArcSinApproximator`, ossia numeri compresi fra  $-1$  e  $1$ , separati fra loro da una differenza di  $0.1$ .

**Esercizio P.4.** Scrivete un'infrastruttura di collaudo che generi dieci numeri casuali in virgola mobile, compresi fra  $-1$  e  $1$ , e li fornisca alla classe `ArcSinApproximator`.

**Esercizio P.5.** Scrivete un'infrastruttura di collaudo che verifichi automaticamente la validità della classe `ArcSinApproximator`, controllando che

```
Math.sin(new ArcSinApproximator(x).getArcSin())
```

sia approssimativamente uguale a  $x$ . Eseguite la verifica mediante 100 valori di ingresso casuali.

**Esercizio P.6.** La funzione arcoseno si può ricavare dalla formula dell'arcotangente, mediante questa formula:

$$\arcsin(x) = \arctan\left(\frac{x}{\sqrt{1-x^2}}\right)$$

Usate questa espressione come *oracolo* per verificare se il vostro metodo che calcola l'arcoseno funziona correttamente. Collaudate il metodo mediante 100 valori di ingresso casuali, confrontando i risultati con l'oracolo.

**Esercizio P.7.** Il dominio della funzione arcoseno è definito dalla condizione  $-1 \leq x \leq 1$ . Collaudate la vostra classe calcolando `arcsin(1.1)`: che cosa succede?

**Esercizio P.8.** Inserite messaggi di tracciamento nel ciclo che calcola la serie di potenze nel metodo che calcola l'arcoseno,. Stampate il valore dell'esponente del termine corrente, il valore del termine corrente e l'approssimazione corrente del risultato. Quale tracciamento si ottiene, calcolando `arcsin(0.5)`?

**Esercizio P.9.** Inserite messaggi di tracciamento alla classe `Word` sbagliata. Segnalate i valori significativi, come i valori delle variabili istanza, i valori restituiti dai metodi e i contatori dei cicli. Eseguite il programma con gli stessi valori di ingresso usati per la sessione di debugging. I messaggi sono sufficientemente esplicativi per individuare l'errore?

**Esercizio P.10.** Eseguite, tramite il debugger, un'infrastruttura di collaudo per la classe `ArcSinApproximator`. Eseguite il calcolo di `arcsin(0.5)` entrando nel metodo, fino al punto in cui il termine  $x^7$  è stato calcolato e sommato al totale. Qual è il valore del termine corrente e quale quello del totale?

**Esercizio P.11.** Eseguite, tramite il debugger, un'infrastruttura di collaudo per la classe `ArcSinApproximator`. Eseguite il calcolo di `arcsin(0.5)` entrando nel metodo, fino al punto in cui il termine  $x^n$  è diventato minore di  $10^{-6}$ . Quindi esaminate  $n$ : quale valore contiene?

**Esercizio P.12.** La classe seguente contiene due errori:

```
public class RootApproximator
{
    public RootApproximator(double aNumber)
    {
        a = aNumber;
        x1 = aNumber;
    }

    public double nextGuess()
```

## 36 Collaudo e correzione di errori

```
{
    x1 = x0;
    x0 = (x1 + a / x1) / 2;
    return x1;
}

public double getRoot()
{
    while (!Numeric.approxEqual(x0, x1))
        nextGuess();
    return x1;
}

private double a; // il numero di cui si calcola la radice
private double x0;
private double x1;
}
```

Create una serie di casi di prova per stanare gli errori, quindi eseguite una sessione di debug per individuarli. Quali modifiche avete fatto alla classe per correggere gli errori?