Capitolo 4
Le eccezioni

Introduzione

Dopo aver presentato nel corso dei capitoli precedenti alcune importanti tematiche relative alla programmazione Object Oriented, al multithreading, al miglior utilizzo delle risorse e così via, in questo capitolo l'attenzione è focalizzata su un altro aspetto fondamentale della programmazione che ha un ruolo centrale nella realizzazione di applicazioni affidabili: la gestione delle eccezioni. Indipendentemente dall'impegno profuso nel disegnare e implementare sistemi affidabili, e a tutti gli sforzi compiuti per raggiungere un elevato livello qualitativo, possono verificarsi diversi problemi: per esempio l'impossibilità di instaurare connessioni con il sistema di gestione del database, un network particolarmente lento, servizi non disponibili, e così via. Credeteci: tali problemi si verificano molto più spesso di quanto sarebbe auspicabile. Pertanto, la strategia adottata per gestire queste anomalie fa la differenza tra sistemi affidabili e non affidabili. Chiaramente, la sola strategia di gestione delle eccezioni non è in grado di sopperire a eventuali carenze qualitative: si tratta del necessario completamento per conseguire un elevato livello di qualità.

L'assenza di una ben definita strategia di gestione delle eccezioni, d'altro canto, può generare una serie di problemi. In particolare, ogni sviluppatore si trova costretto, nella migliore delle ipotesi, a utilizzarne una propria, oppure, nei casi peggiori, a decidere, volta per volta, quale metodoogia impiegare. Considerando che i progetti tipici necessitano di decine di sviluppatori, si genera una situazione decisamente caotica. Il risultato finale, pertanto, comprende tutta una serie di scenari, che vanno dalla generazione di non poca confusione per il personale addetto al

supporto del sistema, dovuta, per esempio, a eccezioni notificate in modo incoerente o addirittura rese note nelle parti meno appropriate del sistema, alla generazione di sistemi instabili e/o difficili da monitorare, alla carenza e alla incoerenza nella generazione di log, con conseguente limitata efficacia, o addirittura impossibilità, di utilizzare software forniti da terze parti atti a monitorare il sistema in produzione, e così via.

Una valida strategia di gestione delle eccezioni applicata all'intero sistema è quindi un prerequisito irrinunciabile di qualità per una serie di ragioni:

- realizzare sistemi in grado di esibire un comportamento efficace e coerente al verificarsi di errori e anomalie;
- agevolare il riutilizzo del codice;
- semplificare le attività manuali di investigazione necessarie dopo errori critici;
- agevolare l'utilizzo di applicazioni esterne atte a monitorare il corretto funzionamento del sistema.

Recenti studi hanno dimostrato come lo sviluppo di sistemi robusti sia un compito complesso. A seconda degli studi considerati, la percentuale dei progetti software falliti varia in intervallo compreso tra il 50% e il 70%. Ad "aggravare" la situazione interviene la tendenza moderna di realizzare sistemi sempre più grandi, più complessi, costituiti da un insieme di sottosistemi comunicanti dispiegati in diverse aree geografiche che forniscono servizi *realtime*. Logica conseguenza è che il requisito di affidabilità assume un ruolo di primaria importanza.

Sebbene i moderni linguaggi di programmazione offrano una serie di sofisticati meccanismi di supporto per la gestione delle eccezioni che vanno dalla rilevazione delle anomalie alla loro comunicazione e gestione, non è infrequente imbattersi in codici non adeguati a gestire efficacemente, sistematicamente e consistentemente eventuali condizioni anomale.

Obiettivi

L'obiettivo di questo capitolo, compatibilmente con quanto esposto nei precedenti, è fornire una serie di direttive operative, *best practice* e quant'altro, necessarie per la realizzazione di sistemi robusti e affidabili. La trattazione è focalizzata sul linguaggio di programmazione Java, sebbene molti concetti presentino una validità che prescinde dallo specifico linguaggio di programmazione, e che include concetti architetturali di più ampia portata.

La lettura di questo capitolo dovrebbe fornire ai lettori materiale necessario per la definizione di efficaci politiche di gestione delle eccezioni in grado sia di semplificare il lavoro degli sviluppatori, sia di produrre sistemi robusti e quindi di maggiore qualità.

Dalla lettura di questo capitolo è possibile evidenziare come una corretta gestione delle eccezioni richieda una serie di accorgimenti, che non sempre i programmatori considerano, come ad esempio una consistente struttura dei record di log.

In questo capitolo sono affrontate diverse tematiche come quelle relative alla necessità di non utilizzare direttamente le classi base delle eccezioni, all'efficacia di valutare la durata temporale dei problemi di sistema, presentando un template da utilizzarsi per ritentare una determinata operazione per un numero prefissato di volte prima di propagare l'anomalia, all'esigenza di considerare i problemi, spesso trascurati, che si possono verificare utilizzando sistemi di messagistica.

Un po' di teoria

I concetti di "eccezione" e relativa gestione non sono certo nuovi nella comunità informatica. La formulazione iniziale, infatti, è attribuibile a J.B. Goodenough il quale, nel "lontanissimo" 1975 ([EFFCJA]), propose formalmente di inserire un simile costrutto nei linguaggi di programmazione. Ciò nonostante, fu necessario attendere ben cinque anni per vederne una concreta manifestazione: nel 1980 la Microsoft, infatti, incluse il costrutto `ON ERROR GOTO` nell'allora popolare GWBasic (celebre dialetto del linguaggio BASIC inizialmente studiato per conto della Compaq, il cui nome è un omaggio alle iniziali dal suo ideatore: Greg Whitten).
Il nome eccezione è una contrazione dei termini "evento eccezionale" (*exception = exceptional event*) che, come suggerisce il nome, rappresenta un evento che si verifica durante l'esecuzione del programma e che scompagina il normale flusso di esecuzione delle istruzioni: "An exception is an event that occurs during the execution of a program that disrupts the normal flow of instructions during the execution of a method".
All'iniziale formulazione del concetto di eccezione seguirono accesi dibattiti inerenti il suo utilizzo. In particolare, la comunità informatica si divise in due grandi gruppi: da una parte si schierarono le persone che consideravano le eccezioni come uno strumento demandato esclusivamente a notificare l'insorgere di condizioni di errore, e dall'altra coloro che invece ne proponevano un utilizzo più ampio paragonabile a quello di qualsiasi altro costrutto come i cicli `for`, `while` e `do` ([EXCPRL]). Lo stesso J.B. Goodenough nella formulazione iniziale ne propose un utilizzo alquanto esteso, destinato ad includere le seguenti principali situazioni:

- gestione delle condizioni di errore;
- elaborazione di un costrutto supplementare atto a comunicare, in modo artificioso, informazioni circa il corretto completamento di un'operazione;
- controllo delle operazioni in corso di esecuzione.

La visione attualmente accettata e quindi considerata in questo libro, è chiaramente la prima, che come espresso nel 1987 da Melliar-Smith and Randall asserisce che le "eccezioni sono una proprietà dei linguaggi di programmazione atta a migliorarne la caratteristica di affidabilità alla presenza di condizioni di errori o di eventi inaspettati. Le eccezioni non sono destinate a fornire costrutti di controllo generale. Un utilizzo liberale non dovrebbe essere considerato sufficiente per fornire ai programmi una completa tolleranza degli errori".

Pertanto, come illustrato di seguito, il meccanismo di gestione delle eccezioni permette di risolvere elegantemente il classico problema della gestione degli errori, evitando il continuo passaggio, tra diversi moduli, di codici di errore per via dello stack.

La gerarchia delle eccezioni in Java

Gerarchia

Il linguaggio di programmazione Java prevede una ben definita gerarchia delle eccezioni come mostrato nel diagramma delle classi di figura 4.1.

La gerarchia delle eccezioni ha inizio con la classe java.lang.Throwable (introdotta fin dalla versione JDK 1.0) che quindi rappresenta la classe antenata (superclasse) di tutti gli errori e le eccezioni del linguaggio Java. Pertanto tutti gli oggetti istanze di classi discendenti da questa sono trattati come eccezioni e come tali possono essere lanciati dalla JVM e/o dalle applicazioni Java per mezzo della parola chiave throw. Allo stesso modo, solo le classi che derivano da Throwable possono essere utilizzate come argomenti dei costrutti catch.

Tuttavia, le applicazioni non dovrebbero mai definire delle classi che ereditano direttamente da Throwable né, tantomeno, dovrebbero intercettare (catch) o lanciare (throw) eccezioni di questo tipo.

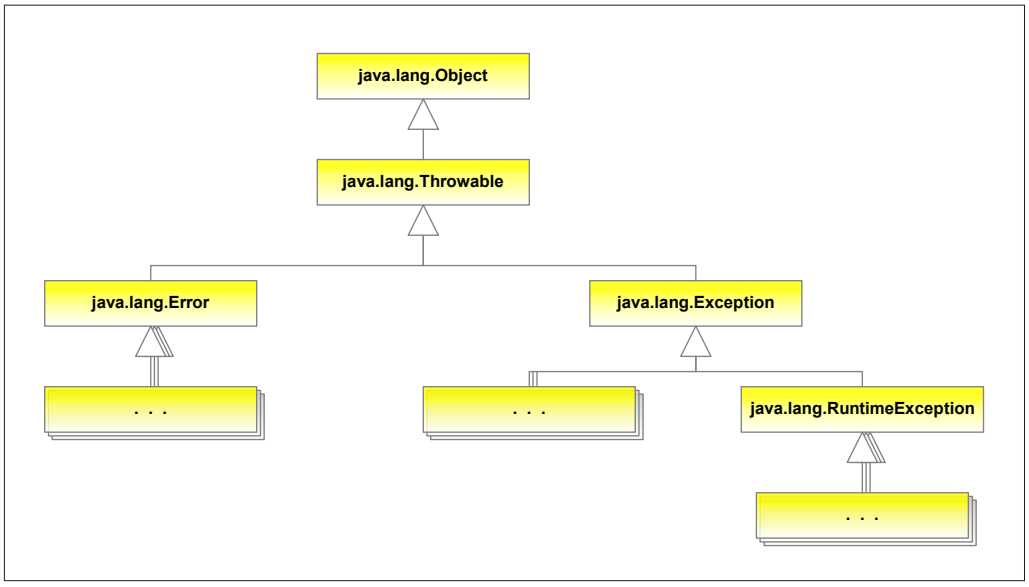

Figura 4.1 – Gerarchia delle eccezioni Java.

La classe java.lang.Error (introdotta anch'essa fin dalla versione JDK 1.0) è utilizzata per comunicare gravi condizioni di errore e pertanto le applicazioni non dovrebbero cercare di intercettarla. Questo è il motivo per il quale Error deriva direttamente da Throwable e non dalla classe Exception.

La classe java.lang.Exception (anch'essa introdotta fin dalla versione JDK 1.0) è utilizzata per comunicare il verificarsi di problemi, corredati da opportune informazioni, che necessitano di essere gestiti ma che non sono così gravi come quelli riportati da oggetti di tipo Error.

La classe Exception a sua volta, prevede due specializzazioni:

1. eccezioni "a tempo di esecuzione" (*runtime exception*), implementate ereditando da RuntimeException;

2. eccezioni controllate (*checked*) dette anche eccezioni "non a tempo di esecuzione" (*non-runtime exception*). Queste sono tutte quelle derivanti da questa e che non includono RuntimeException nella linea di ereditarietà.

Eccezioni runtime e checked

Il primo tipo di eccezioni (*runtime*) è stato ideato per incapsulare errori che occorrono all'interno dell'ambiente di esecuzione Java spesso causati da errori di programmazione. Alcuni esempi famosi sono: NullPointerException, ArithmeticException, IndexOutOfBoundException, etc.

Una delle caratteristiche principali di questo tipo consiste nel non richiedere obbligatoriamente la gestione esplicita, e per questo sono dette anche non controllate (*unchecked*). Pertanto i metodi che possono lanciare eccezioni *runtime* possono ometterne la dichiarazione (clausola throws nella firma del metodo). Allo stesso modo, metodi che ne invocano altri che possono lanciare eccezioni *runtime* non sono obbligati né a gestirle esplicitamente (catch) né a dichiararle sulla linea di comando. Si consideri, per esempio, il metodo statico di conversione di stringhe in numeri: Integer.parseInt(<string>). Quantunque questo possa lanciare un'eccezione NumberFormatException qualora il parametro fornito non rappresenti un valido valore numerico, non è necessario racchiuderla nel ciclo try … catch (NumberFormatException nfe) giacché questa è di tipo *runtime* (eredita da RunTimeException).

Le eccezioni controllate, invece, rappresentano errori che si verificano in parti di codice "al di fuori" dell'ambiente di esecuzione Java, quali errori di I/O. In questo caso, il compilatore Java ne forza una gestione esplicita: è necessario o effettuarne il catch esplicito oppure dichiararle nella firma del metodo, rinviandone la gestione al metodo chiamante.

Listato 4.1 – Esempio di un semplice metodo per la lettura di un file testo.

```
public String readFile(String filePath)
        throws FileNotFoundException, IOException {

  File file = new File(filePath);
```

```
    BufferedReader fileReader    = null;
    StringBuffer strBuffer       = null;

    try {
      fileReader = new BufferedReader(new FileReader(file));

      boolean eof       = false;
      String nextLine   = null;
      strBuffer         = new StringBuffer();

      while (!eof) {

        nextLine = fileReader.readLine();

        eof = (nextLine == null);

        if (!eof) {
          strBuffer.append(nextLine);
        }
      }

    } catch (FileNotFoundException fnfe) {
      throw fnfe;

    } catch (IOException ioe) {
      throw ioe;

    } finally {
      try {
        if (fileReader != null) {
          fileReader.close();
        }
      } catch (IOException ioe) {
        logException(ioe);  // this is an internal method
      }
    }
    return strBuffer.toString();
  }
```

Si consideri, per esempio, di dover leggere delle informazioni presenti in un file di testo (cfr. listato 4.1). A questo punto, la prima cosa da farsi consiste nel crearne una rappresentazione logica del file (File file = **new** File(filePath);) e quindi passarne il riferimento a uno *stream* di lettura

(BufferedReader fileReader = new BufferedReader(new FileReader(file))). Giacché il file potrebbe non esistere, il costruttore della classe java.io.FileReader può lanciare un'eccezione java.io.FileNotFoundException che deriva da java.io.IOException che a sua volta eredita java.lang.Exception. Trattandosi di un'eccezione *checked* è obbligatorio o gestirla attraverso apposito costrutto catch oppure rilanciarla al metodo chiamante (dichiarazione nella definizione del metodo). Quest'ultima soluzione è adottata nel listato 4.1. Qualora ciò non avvenga, il compilatore genera un apposito messaggio di errore.

La tabella 4.1 è una comoda sintesi delle proprietà dei due diversi tipi di eccezione:

La controversia relativa all'uso

Quantunque i principi base che hanno portato all'ideazione delle due eccezioni siano piuttosto chiari, altrettanto chiaro non sembrerebbe esser il loro utilizzo. Non a caso esiste una dichiarata controversia relativa all'utilizzo (http://java.sun.com/docs/books/tutorial/essential/exceptions/runtime.html). Pertanto si è ritenuto opportuno riportare le direttive presenti nel sito della Sun Microsystem.

Il fatto che il linguaggio di programmazione Java non richieda ai metodi di specificare eccezioni *runtime* o errori, potrebbe tentare i programmatori a scrivere codice che lanci esclusivamente eccezioni *runtime* o a definire proprie eccezioni derivanti da java.lang.RuntimeException. Queste "scorciatoie" permettono ai programmatori di scrivere del codice senza preoccuparsi di gestire le eccezioni, bypassando i controlli del compilatore.

Sebbene, a prima vista, possa sembrare una tecnica molto comoda, questa elude gli intenti dei requisiti "cattura o specifica" (catch or specify) alla base del meccanismo Java delle eccezioni e può generare una serie di problemi ai programmatori che dovranno utilizzare classi implementate con questa tecnica.

Perché i progettisti hanno deciso di forzare i metodi a specificare tutte le eccezioni *checked* non gestite che possono essere lanciate nel loro ambito?

	Eccezioni runtime	**Eccezioni checked**
Dichiarazione delle eccezioni lanciate nella firma del metodo	Non necessario	Obbligatorio
Gestione delle eccezioni nel metodo chiamante	Non necessario	Il metodo chiamante deve gestire esplicitamente l'eccezione (catch) oppure rilanciarla (throws)
Classe antenata	RuntimeException o Error	Exception

Tabella 4.1 – Proprietà delle eccezioni runtime *e* checked.

Ogni eccezione che può essere lanciata da un metodo fa parte dell'interfaccia pubblica dello stesso. I metodi chiamanti devono essere messi a conoscenza delle eccezioni che un metodo può lanciare affinché siano in grado di decidere come trattarle. Queste eccezioni sono parte dell'interfaccia programmativa del metodo così come lo sono i parametri e il valore di ritorno.

La prossima domanda potrebbe essere: se è così valido documentare l'API di un metodo, incluse le eccezioni che questo può lanciare, perché non forzare la specifica esplicita anche delle eccezioni *runtime*?

Le eccezioni *runtime* rappresentano anomalie che sono il risultato di un problema di programmazione e, come tale, non si può ragionevolmente richiedere alle API del client di eseguire qualche azione per risolvere il problema e/o di effettuarne una qualsiasi gestione. Questi problemi includono eccezioni aritmetiche (come la divisione per zero), eccezioni di riferimento (come per esempio il tentativo di accedere alle proprietà di un oggetto attraverso un puntatore nullo) ed eccezioni di indicizzazione (come per esempio un tentativo di accedere ad un elemento dell'array attraverso un indice il cui valore sia superiore o inferiore alle posizioni valide dell'array). Le eccezioni *runtime* possono avvenire in ogni parte del programma e tipicamente possono essere numerose. Pertanto l'obbligo di dover aggiungere la dichiarazione di eccezioni *runtime* in ogni metodo ridurrebbe drasticamente la leggibilità del programma. Per questo motivo, il compilatore non richiede obbligatoriamente di gestire o dichiarare eccezioni *runtime*, (sebbene nessuno vieti di farlo).

Un caso in cui è pratica comune lanciare eccezioni a tempo di esecuzione è relativo a invocazioni scorrette di metodi. Per esempio, un metodo può verificare se gli argomenti specificati siano o meno nulli, ed in caso siano nulli, lanciare un'eccezione `NullPointerException` che appunto è di tipo *runtime*.

In generale, non si dovrebbe lanciare un `RuntimeException` or creare una sotto classe per il semplice motivo che si voglia evitare di dover specificare esplicitamente le eccezioni lanciate dai propri metodi.

Direttive

4.1 Utilizzare le eccezioni

Anche se da quanto riportato nei paragrafi precedenti dovrebbe essere ormai evidente l'importanza del corretto utilizzo del meccanismo delle eccezioni, vista l'importanza del concetto, si ritiene che sia il caso del *melius abundare quam deficere*.

Il meccanismo delle eccezioni permette di risolvere una serie di anomalie causate dalle precedenti strategie di gestione degli errori, e in particolare permette di:

1. Dare luogo ad una migliore organizzazione del codice. Questo grazie alla separazione elegante e pulita tra il codice necessario per implementare il particolare servizio e quello richiesto per gestire eventuali errori. Inoltre, non è necessario ritornare esplicitamente codici di errore che riducono la chiarezza delle API con continui passaggi di codici di

controllo e forzano la ripetizione di blocchi di istruzioni necessari per controllare tali valori in tutti i metodi chiamanti.

2. Aumentare il livello di robustezza. Codici di errori sono difficili da mantenere, specialmente a seguito a *refactoring* del codice e possono facilmente generare situazioni inconsistenti come quelle dovute a stessi codici utilizzati per rappresentare anomalie diverse, o a codici non più generati da una classe che comunque persistono nel sistema. Pertanto, a differenza delle eccezioni, i codici di errore possono facilmente sfuggire al controllo.

3. Scrivere codici più leggibili. In particolare, poiché i parametri restituiti dai metodi non devono essere compromessi per via della necessità di restituire codici di errore, è possibile dar luogo a firme dei metodi non artificiosi. Inoltre, il meccanismo delle eccezioni rappresenta la pratica quotidiana che consiste nello specificare una serie di richieste e quindi fornire ulteriori informazioni relative alla situazione in cui non sia possibile soddisfare l'elenco principale. Un esempio classico è quello della spesa "acquista 1 kg di riso e una bottiglia di Barolo; se non trovi il Barolo, prendi pure una bottiglia di Nebbiolo" e così via.

4. Dar luogo a un migliore disegno. Questo grazie al fatto che, spesso, il punto in cui un errore si verifica non è il posto migliore dove gestirlo e, utilizzando i codici di errore, è molto frequente che accada che il percorso necessario per la propagazione a ritroso dell'errore generi la perdita di informazioni relative al contesto dell'errore verificatosi. Le eccezioni, invece, includono tutte le informazioni necessarie dal punto in cui si verificano sino al punto in cui vengono gestite.

5. Migliorare le prestazioni. Ciò essenzialmente per via del fatto che non è necessario verificare continuamente i valori restituiti dai metodi.

4.1.1 Le eccezioni posso verificarsi: e questo è un fatto inevitabile

Nel mondo dell'ideale si sarebbe tentati di pensare che, in presenza di un codice ben scritto, le eccezioni non dovrebbero mai verificarsi, quindi l'infrastruttura dovrebbe sempre funzionare correttamente, nessun processo business dovrebbe mai interrompersi, tutti i messaggi dovrebbero essere corretti e consegnati nell'ordine previsto, il Database Management System dovrebbe essere sempre perfettamente funzionante e così via.

Purtroppo, nella realtà la situazione è decisamente ben diversa e, per quanto accuratamente si tenti di scrivere il codice, le eccezioni comunque si verificano. Chiaramente, se poi il codice è scritto in modo trasandato, allora le eccezioni tendono a presentarsi frequentemente e l'applicazione tende a presentare un elevato livello di instabilità.

Pertanto, sebbene sia necessario produrre il massimo sforzo per aumentare la qualità del codice e per realizzare opportuni meccanismi atti a diminuire la probabilità del verificarsi delle eccezioni, queste comunque si verificano e quindi solo un attento e ben progettato sistema di gestione è grado di fare la differenza tra sistemi affidabili e di qualità e sistemi problematici.

4.1.2 Evitare di comunicare situazioni di errore attraverso i parametri di ritorno

Il meccanismo di gestione delle eccezioni permette di risolvere elegantemente il classico problema della gestione degli errori, evitando il continuo passaggio di codici di errore, tra diversi moduli, attraverso lo *stack*. Quest'ultima strategia genera una serie di problemi.

1. Aumento della complessità. Tutti i moduli chiamanti coinvolti in una sequenza di invocazioni sono costretti ad occuparsi in maniera esplicita degli errori. Questo implica la ripetizione della stessa logica necessaria per verificare il valore del parametro di ritorno e quindi intraprendere le necessarie operazioni a seconda di tale valore.

2. Problemi di firma dei metodi. Questo problema deriva dal fatto che il valore di ritorno di diversi metodi è utilizzato per fornire possibili codici di errore. Pertanto sono necessarie strategie alternative per gestire il normale passaggio dei valori di ritorno.

3. Ridotta capacità informativa. Poiché gli errori sono comunicati esclusivamente attraverso un singolo codice, ne segue che l'elenco dei codici utilizzati deve essere conosciuto da tutti i moduli presenti nella catena delle invocazioni con possibili problemi di incongruenza.

Si consideri il codice (**assolutamente sconsigliato**) riportato nel listato 4.2. Si tratta del metodo già riportato nel listato 4.1, ove le eccezioni sono state sostituite da codici di errore. Come si può notare, la firma del metodo risulta decisamente meno intuitiva e non c'è un modo formale per inserire la lista dei possibili errori che possono verificarsi. Per quanto riguarda il codice restituito è stata utilizzata la seguente convenzione:

- ritorno = 0, non si sono verificati errori; e quindi il parametro di input/output, result, è valido e contiene il contenuto del file;

- ritorno > 0; è analogo al caso precedente con la variazione che il valore numerico indica il verificarsi di problemi marginali (*warning*); nel caso proposto non ci sono warning;

- ritorno < 1 rappresenta l'insorgere di problemi, per esempio -1 rappresenta un errore di "file not found" e -2 problemi di IO; il problema principale di questo codice è relativo alla classe invocante (e di diverse tra quelle presenti nella sequenza di invocazione) che sono costrette a ripetere il frammento di codice riportato nel listato 4.3.

Listato 4.2 – Sconsigliato: il codice del listato 4.1 modificato per far restituire al metodo codici di errore anziché eccezioni.

```
public String readFile(String filePath, String result) {

  int errorCode  = 0;
```

```
    result      = null;
    File file   = new File(filePath);

    BufferedReader fileReader = null;
    StringBuffer   strBuffer  = null;

    try {
      fileReader = new BufferedReader(new FileReader(file));

      boolean eof    = false;
      String nextLine = null;
      strBuffer      = new StringBuffer();

      while (!eof) {

        nextLine = fileReader.readLine();

        eof = (nextLine == null);

        if (!eof) {
          strBuffer.append(nextLine);
        }
      }

      result = strBuffer.toString();

    } catch (FileNotFoundException fnfe) {
      errorCode = -1;

    } catch (IOException ioe) {
      errorCode = -2;

    } finally {
      try {
        if (fileReader != null) {
          fileReader.close();
        }
      } catch (IOException ioe) {
        logException(ioe);  // this is an internal method
      }
    }
    return errorCode;
  }
```

Listato 4.3 – Controlli del codice di ritorno da ripetersi nella lista di metodi inclusi nella sequenza di invocazione.

```
Sting fileContent = "";

int errorCode = readFile(filePath, result);

if (errorCode == 0) {
   // sequenza di istruzioni necessarie per gestire
   // il caso di successo
} else if (errorCode == -1) {
   // sequenza di istruzioni atte a gestire
   // la situazione di file not found
} else if (errorCode == -2) {
   // sequenza di istruzioni atte a gestire
   // la situazione di problemi di IO
}
```

4.1.3 Utilizzare le eccezioni esclusivamente per comunicare condizioni di errore

Come illustrato nei paragrafi precedenti, il meccanismo delle eccezioni così come lo si intende ora è il risultato di una evoluzione relativamente lunga. In particolare, sebbene agli albori fosse stato proposto un utilizzo più ampio non necessariamente relegato alla gestione delle anomalie, questo si è dimostrato causa di diversi problemi e confusione nel codice. Pertanto la versione moderna ne prevede un utilizzo confinato alla gestione degli errori. Quest'ultima strategia è quella divenuta standard e pertanto, onde evitare la generazione di codici confusi e poco leggibili, è necessario utilizzare le eccezioni solo ed esclusivamente per la gestione delle condizioni di errore.

4.2 Utilizzare correttamente il blocco finally

Un tipico modello di gestione delle eccezioni prevede di chiudere accuratamente (tecnicamente *clean-up*, "pulire") le varie risorse prima di passare il controllo a ritroso. Il linguaggio di programmazione Java prevede un apposito costrutto per gestire queste operazioni: il finally. Si tratta di una parte opzionale del costrutto try (così come lo è la parte catch) e fornisce un meccanismo elegante e opportuno per porre il sistema in uno stato adatto indipendentemente da quanto succede nel corpo del try … catch.

Come regola generale, si tenga presente che la clausola finally viene sempre eseguita anche quando potrebbe sembrare il contrario, come per esempio, in presenza di istruzioni di return all'interno del try. Una delle pochissime eccezioni a questa regola è data dalla presenza dell'istruzione System.exit.

Si consideri il seguente frammento del listato 4.1. Questo è stato modificato per includere l'istruzione di return all'interno del costrutto try (listato 4.4).

Listato 4.4 – Sconsigliato: frammento di codice appositamente modificato per aggiungere l'istruzione di ritorno all'interno del costrutto try.

```
try {
   fileReader = new BufferedReader(new FileReader(file));

   boolean eof     = false;
   String nextLine = null;
   strBuffer       = new StringBuffer();

   while (!eof) {

      nextLine = fileReader.readLine();

      eof = (nextLine == null);

      if (!eof) {
         strBuffer.append(nextLine);
      }
   }

   return strBuffer.toString();
} catch (FileNotFoundException fnfe) {
   throw fnfe;

} catch (IOException ioe) {
   throw ioe;

} finally {
   try {
      if (fileReader != null) {
         fileReader.close();
      }
   } catch (IOException ioe) {
      logException(ioe);  // this is an internal method
   }
}
```

Si noti che la clausola finally è eseguita sia quando non intervengono problemi (ossia la JVM esegue il return presente nel costrutto try), sia quando ci sono dei problemi, ossia quando vengono eseguite le clausole catch.

4.2.1 Utilizzare il blocco finally per il clean-up

Utilizzare il costrutto finally ogni qualvolta nel corso del costrutto try ... catch si utilizzano degli oggetti da porre in un determinato stato prima di passare il controllo a ritroso.

Si consideri l'ennesima variazione del listato 4.1 riportata nel listato 4.5.

Listato 4.5 – Sconsigliato: esempio di chiusura di uno stream *senza utilizzare il blocco finally.*

```
public String readFile(String filePath)
        throws FileNotFoundException, IOException {

  File file = new File(filePath);

  BufferedReader fileReader = null;
  StringBuffer strBuffer    = null;

  try {
    fileReader = new BufferedReader(new FileReader(file));

    boolean eof     = false;
    String nextLine = null;
    strBuffer       = new StringBuffer();

    while (!eof) {

      nextLine = fileReader.readLine();

      eof = (nextLine == null);

      if (!eof) {
        strBuffer.append(nextLine);
      }
    }

    try {
      if (fileReader != null) {
        fileReader.close();
      }
    } catch (IOException ioe) {
```

```
            logException(ioe);  // this is an internal method
        }

    } catch (FileNotFoundException fnfe) {

        try {
            if (fileReader != null) {
                fileReader.close();
            }
        } catch (IOException ioe) {
            logException(ioe);  // this is an internal method
        }

        throw fnfe;

    } catch (IOException ioe) {

        try {
            if (fileReader != null) {
                fileReader.close();
            }
        } catch (IOException ioe) {
            logException(ioe);  // this is an internal method
        }
        throw ioe;

    }

    return strBuffer.toString();
}
```

Come si può notare, il mancato utilizzo del blocco finally ha richiesto di ripetere il blocco delle istruzioni di chiusura (fileReader.close()) in diverse parti del codice (all'interno del blocco try e in tutti i vari catch). Ciò, oltre a dar luogo a un'inutile ripetizione di codice (parzialmente risolvibile con l'implementazione di un apposito metodo), potrebbe generare non pochi problemi qualora un aggiornamento del codice richieda di gestire nuove eccezioni e ci si dimentichi di ripetere il blocco di chiusura dello *stream*.

4.2.2 Valutare la necessità di inserire un try … catch all'interno del blocco finally

Spesso le operazioni di pulizia presenti all'interno del blocco finally possono generare delle eccezioni come nei casi presentati in precedenza (cfr. listato 4.1). In questi casi è opportuno

inserire un opportuno costrutto try … catch all'interno del costrutto finally. In questo caso però quasi mai è opportuno riportare, qualora insorgesse, la nuova eccezione: quella che ha generato il problema iniziale è tipicamente più importante. Comunque è buona pratica riportare eventuali ulteriori eccezioni nel file di log.

4.2.3 Utilizzare finally al posto di finalize

Come visto nel precedente capitolo, non è opportuno ricorrere al metodo finalize per effettuare il clean-up degli oggetti. Questo perché non vi è garanzia del momento esatto in cui il GC (Garbage Collector) reclamerà lo spazio di memoria occupato da un oggetto e quindi invocherà il metodo finalize. Nel codice Java funzionante all'interno di un application server, il metodo finalize potrebbe anche venir invocato dopo diverse ore dal momento in cui l'oggetto stesso è stato referenziato. Pertanto, per eseguire la pulizia degli oggetti, è buona pratica includere opportuni blocchi finally all'interno dei metodi come visto in precedenza. Chiaramente il costrutto finally, ha un funzionamento assolutamente diverso dal metodo finalize. Tuttavia, è conveniente utilizzarlo per realizzare un sistema più accurato e affidabile di pulizia delle risorse.

4.3 Non utilizzare i tipi base delle eccezioni

Come visto in precedenza, il linguaggio di programmazione Java dispone di una serie di classi fondamentali utilizzate per notificare anomalie (figura 4.1). Queste sono: Throwable, Error, Exception e RuntimeException. L'idea alla base del meccanismo delle eccezioni è che queste classi (ad eccezione della classe Error che dovrebbe essere utilizzata solo in contesti particolarissimi) rappresentano comodi punti di estensione per permettere la definizione di opportune gerarchie, come quelle dei package Java. Pertanto, bisognerebbe evitare un riferimento diretto a questi tipi.

4.3.1 Non utilizzare direttamente Throwable, Error, Exception, RuntimeException

Le applicazioni non dovrebbero mai utilizzare direttamente le seguenti classi: Throwable, Error, Exception e RuntimeException. In particolare è opportuno che framework e package utilizzino eccezioni derivate da Exception o RuntimeException standard e/o appartenenti a un'opportuna gerarchia.

La comunicazione di eccezioni attraverso le classi Throwable ed Error può severamente compromettere la possibilità di eseguire, all'interno della stessa JVM, librerie, framework e package scritti da terze parti e di riutilizzare il codice.

4.3.2 Non ereditare direttamente dalle classi Throwable o Error

Le applicazioni non dovrebbero mai definire eccezioni derivanti direttamente dalla classe Throwable né cercare di intercettare (catch) o lanciare (throw) eccezioni di questo tipo.

Allo stesso modo, le applicazioni non dovrebbero definire nuove classi di errore derivanti direttamente da java.lang.Error né tantomeno tentare di intercettarle.

Si tratta di classi con un significato ben definito lanciate dalla JVM per segnalare gravi anomalie che non possono essere gestite.

4.3.3 Non utilizzare i tipi base per intercettare le eccezioni

Le eccezioni non dovrebbero mai essere intercettate utilizzando i tipi base come Throwable, Error, Exception e RuntimeException.

Come regola generale è opportuno cercare di intercettare le eccezioni specifiche evitando di definire dei "filtri" troppo ampi che possono facilmente portare a situazioni di errore.

Per esempio, una rifattorizzazione del codice potrebbe portare all'inserimento di istruzioni aggiuntive in grado di generare nuovi tipi di eccezione; vista la mancanza di selettività della clausola catch, tali eccezioni finirebbero comunque per essere accomunate alla gestione prevista dal caso generale, senza fornire al programmatore alcuna indicazione. Ciò, nella maggior parte dei casi sarebbe fonte di problemi non facilmente individuabili.

Si consideri per esempio il seguente frammento di codice:

```
try {

   String content = fileManager.readFile(xmlFile);

} catch (Exception e) {
   // qualche operazione di gestione
}
```

Come si può notare il costrutto catch intercetta, in maniera impropria, istanze della classe java.lang.Exception. Ora, si supponga di estendere il precedente codice invocando un metodo che effettui il parsing del contenuto del file XML, trasformando la stringa in un apposito grafo di oggetti (ConfigVO), come riportato nel seguente frammento:

```
try {

   String content = fileManager.readFile(xmlFile);
   configVO  = (ConfigVO) genericStringUnmarshaller(content);

} catch (Exception e) {
   // qualche operazione di gestione
}
```

Si supponga, come è lecito fare, che il metodo sia in grado di generare un'eccezione. Come si può notare, la presenza del costrutto catch (Exception e) finisce per intercettare la nuova eccezio-

ne senza dare comunicazione al programmatore. Il caso in questione potrebbe sembrare comunque senza troppe conseguenze: in fondo, sono coinvolte solo due istruzioni… Si immagini però il caso tipico di una serie abbastanza lunga di invocazioni, in cui diversi metodi siano composti da circa 10-15 istruzioni, oppure la situazione abbastanza frequente in cui sia necessario cambiare la firma di un metodo, aggiungendo una nuova eccezione, e che questo sia utilizzato in molte parti del sistema. In questi casi si capisce come è facile generare lo scenario in cui la nuova eccezione venga intercettata in un posto dove non dovrebbe esserlo e quindi viene gestita in modo errato. A complicare le cose interviene il fatto che problemi di questo tipo, normalmente, siano difficilmente individuabili.

4.3.4 Valutare l'opportunità di implementare una propria gerarchia di eccezioni

Nel disegno e nella codifica di framework, package e strati logici è spesso opportuno prevedere l'implementazione di una specifica gerarchia di eccezioni. Ciò fa sì che le classi client siano esposte a un insieme di interfacce consistente e minimale. In tali circostanze, tuttavia, è necessario che le nuove eccezioni siano codificate in modo tale da incapsulare quelle originali. Pertanto, sebbene implementare proprie gerarchie di eccezioni per librerie, framework, etc. sia un'ottima pratica per agevolarne la gestione da parte delle classi fruitrici, è comunque consigliabile mantenere le originali nel codice interno del package, framework, etc.

In questi casi, prima di dar luogo a una nuova eccezione, è consigliabile porsi le seguenti domande:

- esiste una eccezione base Java in grado di descrivere il problema che si intende comunicare?
- l'implementazione di nuove eccezioni migliora il codice? In particolare, le classi client ricevono un chiaro beneficio dalla presenza della nuova eccezione?
- la stessa porzione di codice lancia altre eccezioni in qualche modo legate alla nuova?
- la nuova eccezione o quelle fornite da una terza parte, sono accessibili alle classi client?

Qualora una o più delle precedenti domande presenti una risposta negativa, è il caso di verificare opportunamente la necessità di ricorrere all'implementazione di una nuova eccezione.

4.3.5 Non perdere le informazioni relative all'eccezione iniziale

Un'importante miglioria introdotta con la versione JDK 1.4 è la possibilità di annidare eccezioni. In particolare ciò è stato possibile grazie ad un'opportuna revisione della classe `java.lang.Throwable`. Questo perfezionamento permette di lanciare nuove eccezioni senza perdere informazioni relative a quella originale. Si tratta, pertanto, di una caratteristica molto utile in

quando rende possibile sviluppare *framework* che, per comodità delle classi fruitrici, possono lanciare proprie eccezioni con le eccezioni originali incapsulate all'interno.

Quindi, in tutti i casi in cui si decida di procedere con l'implementazione di opportune classi eccezione, è importante includere i costruttori riportati nel listato 4.6.

Listato 4.6 – Costruttori da inserire nelle proprie classi eccezione.

```
/**
 * Constructor method
 * @param error  encapsulated error
 */
public MyException(Throwable error) {
   super(error);
}


/**
 * Constructor method
 * @param excMessage  exception message
 * @param error       encapsulated occurred
 */
public MyException(String excMessage, Throwable error) {
   super(excMessage, error);
}
```

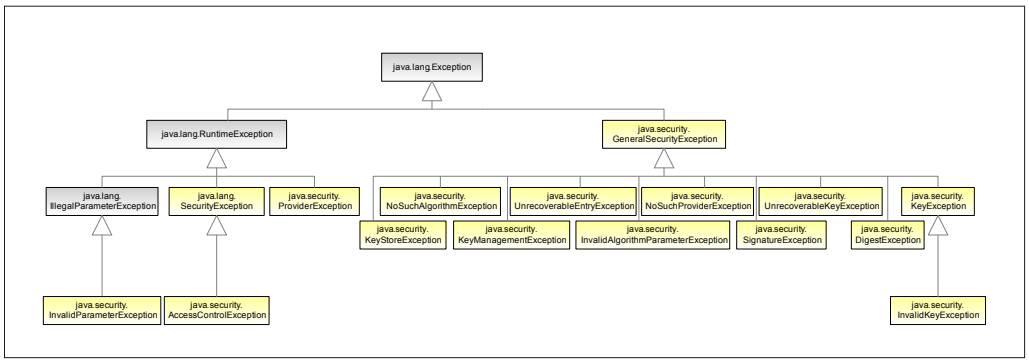

Figura 4.2 – Esempio di gerarchia delle eccezioni definito nel package java.security.

4.4 Fare attenzione alla modalità con cui notificare le eccezioni

Non è infrequente il caso in cui le eccezioni che si presentano in un sistema sono relative a problemi non gestibili. In questo caso è necessario fare in modo che le cause che hanno generato il problema siano opportunamente riportate e illustrate. Qualora, l'eccezione si verifichi durante l'espletamento di un servizio richiesto da un utente, magari collegato con un browser remoto, è necessario renderlo consapevole del problema che si è presentato. In ogni caso, le varie anomalie vanno opportunamente segnalate sia al fine di semplificare eventuali investigazioni manuali, sia per fornire sufficienti informazioni a eventuali meccanismi automatici predisposti per la risoluzione di problemi.

4.4.1 Non assorbire mai le eccezioni silenziosamente

Si dice che un'eccezione è assorbita "silenziosamente" quando il codice incluso nella relativa clausola `catch` non risolve il problema e non esegue alcuna operazione atta a notificare il verificarsi dell'eccezione stessa. Il listato 4.7 mostra un esempio di eccezione assorbita silenziosamente:

Listato 4.7 – Sconsigliato: esempio di eccezione assorbita "silenziosamente".

```
try {
   <istruzione che può causare l'eccezione>
   <instruzione>
   <instruzione>
} catch (<eccezione> e) {
   // eccezione assorbita sileziosamente
}
```

Anche qualora si pensi che l'eccezione non si verificherà mai, è sempre il caso di aggiungere una segnalazione, magari un semplice record di log. In effetti, può sempre succedere (e succede!) che anche scenari di eccezione che inizialmente sembrano innocui e non verificabili, diventinoproblematici a seguito di processi di manutenzione del codice. Individuare eccezioni assorbite silenziosamente può richiede un impegno elevatissimo.

4.4.2 Mantenere l'utente informato

Qualora un'eccezione si verifichi durante la gestione di uno stimolo generato da un utente, è necessario far in modo che opportune informazioni circa l'anomalia siano notificate, in modo opportuno, all'utente.

Chiaramente questo vale per tutte le eccezioni non temporanee. In particolare, è necessario:

1. presentare un messaggio che sia comprensibile all'utente;

2. fornire all'utente diverse opportunità sul da farsi e, in casi specifici, fornire la possibilità di tentare nuovamente la stessa operazione.

Un sistema di qualità dovrebbe sempre fornire all'utente informazioni su quanto accade dietro le quinte, non solo in presenza di problemi, ma anche qualora l'operazione richieda diverso tempo per essere eseguita.

4.4.3 Pianificare e applicare in maniera uniforme e coerente il formato dei log riferiti alle eccezioni

La realizzazione di sistemi ad alta disponibilità/affidabilità, richiede di includere diversi accorgimenti non necessari in altre situazioni. Per esempio, è necessario progettare meccanismi in grado di rilevare tempestivamente eventuali componenti e sistemi malfunzionanti o gravati da stress enorme. Una delle tecniche più largamente impiegate consiste nel ricorrere a sistemi dedicati a monitoraggio e analisi dei file log prodotti dalle varie applicazioni. Ciò al fine di individuare dei pattern predefiniti che identifichino situazioni di malfunzionamento (per esempio una linea di testo iniziante con la stringa "ERROR").

Al fine di rendere possibile l'impiego di questi sistemi di controllo è opportuno assegnare ai log opportuni formati, specialmente a quelli relativi all'insorgere di un'eccezione.

Una struttura abbastanza ricorrente è la lista di attributi separati da caratteri bianchi, in cui le diverse informazioni prevedono i seguenti dati.

- **Data e ora** dell'eccezione secondo il classico formato: `yyyy-mm-dd hh:mm:ss.mmm`. Onde evitare problemi di cambi di orari e di fusi orari si consiglia di utilizzare sempre lo *Universal Time* (per esempio: `2006-07-07 15:13:10.990`).

- **Livello di log**. Si tratta di un'enumerazione che prevede diversi valori identificati da sigle di tre lettere. `TRC`: livello di tracciamento, utilizzato per il debug di eventi molto dettagliati. Tipicamente lo si utilizza per eseguire il log di flussi di esecuzione. `DBG`: livello di debug, anche in questo caso questo livello è utilizzato per informazioni molto dettagliate, solo che al contrario del caso precedente, questo livello è utilizzo per mostrare il valore di attributi, oggetti, etc. `INF`: il livello di informazione è stato disegnato per riportare messaggi di informazione che illustrano la progressione dell'applicazione a un livello abbastanza elevato. `WRN`: il livello di *warning* (avviso) serve per notificare situazioni potenzialmente dannose. `ERR`: questo livello serve per riportare situazioni di errore che però posso ancora permettere all'applicazione di continuare a funzionare. `FTL`: il livello *fatal* (fatale) è stato pensato per situazioni di errore che forzano l'applicazione a una terminazione improvvisa.

- **Identificatore univoco**. Si tratta di un'informazione utile per semplificare il lavoro di possibili agenti automatici demandati all'esecuzione di opportune procedure di gestio-

ne degli errori. Pertanto l'assegnazione di un identificatore è necessario esclusivamente per gli ultimi tre livelli di log (warning, error e fatal).

- **Percorso completo** della classe in cui si è verificato il problema.
- **Messaggio** relativo al problema verificato.
- Eventuale lista con gli **argomenti** del problema.

Ricapitolando:

```
<exception_date_time> <log_level> <unique_id> <class_path> <message> {[arguments]}
```

Per esempio:

```
2006-12-03 12:51:07.103 WRN SDS-CN01
com.mokabyte.tool.umlclassgenerator.service.DiagramParser –'diagram not found' diagram=class34
```

Da quanto riportato risulta evidente che i primi tre attributi sono necessari per una automatizzazione delle procedure di gestione degli errori, mentre i restanti sono utili per un'ispezione manuale.

4.5 Valutare attentamente il ciclo di vita delle eccezioni

Il meccanismo delle eccezioni fornisce un'ottima soluzione per notificare eventuali anomalie. Queste però, oltre a essere opportunamente segnalate devono, ovviamente, essere anche propriamente gestite. Un elemento di particolare importanza nell'implementazione delle procedure di gestione consiste nel selezionare il luogo migliore in cui gestirle. In effetti, si deve evitare una gestione in posti in cui non è chiaro il da farsi e, al tempo stesso, non è opportuno notificare all'infinito un'eccezione qualora sia possibile gestirla.

4.5.1 Non provare a gestire un'eccezione in un posto in cui non è possibile farlo

Un ragionevole principio di programmazione prescrive di non tentare la gestione di un'eccezione in una parte di codice in cui non siano disponibili sufficienti informazioni per farlo. In caso contrario, nella migliore delle ipotesi, si finisce con il rendere il codice molto complesso e difficilmente riutilizzabile, mentre nella peggiore si finisce per assumere alcune informazioni, non sempre valide, generando un insieme di problemi rilevabili solo all'insorgere di casi di eccezione, spesso non facilmente riproducibili e testabili.

Si consideri per esempio il codice riportato nel listato 4.1. Nel caso in cui si verifichi un'eccezione di tipo `FileNotFoundException` non si hanno sufficienti informazioni per selezionare l'opportuna gestione, pertanto la cosa migliore da farsi è rimandare la gestione alla classe chiamante.

Le eccezioni, quindi, vanno intercettate (`catch`) quando si hanno sufficienti informazioni per poterle gestirle correttamente. Questa regola può essere scavalcata nel caso in cui l'eccezione debba essere incapsulata in un'altra più generale. In questo caso si ha un `catch` simbolico, come nel caso del listato 4.1.

4.5.2 Gestire le eccezioni non appena possibile

Questa regola a prima vista potrebbe sembrare in contraddizione con quella precedente. In realtà, le eccezioni dovrebbero essere gestite non appena si giunge, nel percorso a ritroso della successione delle invocazioni, a un metodo che abbia sufficienti informazioni per gestirla. Ciò, in alcuni scenari può coincidere con il comunicare indietro l'eccezione fino al metodo che ha eseguito la chiamata iniziale della catena di invocazioni.

4.6 Considerare la natura delle eccezioni

Un criterio di raggruppamento delle eccezioni consiste nel suddividerle in base alla natura del problema che le ha originate. Ciò porta a individuare due gruppi di eccezioni: *business* e di sistema.

Le prime si riferiscono a eventi che violano una o più *business rules* (le regole alla base della logica applicativa), la cui conseguenza sta nell'impossibilità da parte del sistema di fornire il servizio dove l'eccezione si è verificata.

Le eccezioni di sistema, invece, si riferiscono a eventi che si verificano nel sistema (per esempio, il database diventa indisponibile) e che compromettono la fornitura di un insieme di servizi per un intervallo più o meno lungo. Pertanto problemi di sistema sono potenzialmente più gravi di quelli di business.

4.6.1 Gestire correttamente le eccezioni business

Le eccezioni di tipo business si riferiscono ad eventi che violano una o più regole della logica applicativa (*business rules*) e pertanto non permettono al sistema di portare a termine correttamente il relativo servizio. Per esempio, un sistema di *back office* bancario potrebbe ricevere un messaggio contenente un trade relativo a un cliente (*counter-party*) delle cui necessarie informazioni il sistema stesso non dispone. Il sistema, pertanto, non può che lanciare un'eccezione di business relativa all'impossibilità di processare automaticamente tale trade. Situazioni come questa, come la quasi totalità delle eccezioni di tipo business, richiedono l'intervento umano. Nel caso in questione, un operatore dovrebbe occuparsi di inserire i dati mancanti nel sistema oppure di correggere l'identificativo errato del cliente.

Sistemi medio/grandi atti a gestire importanti business (come nell'esempio precedente) richiedono di progettare meccanismi atti a risolvere prontamente questo tipo di problemi. Nel caso precedente, per esempio, sebbene il sistema non fosse in grado di processare il trade, non è pensabile che questo possa tranquillamente rifiutare o scartare il trade: nei sistemi bancari viaggiano spesso trade relativi a contratti da centinaia di milioni di dollari.

Pertanto, se da un lato è necessario progettare sistemi informatici in grado di processare gran parte degli eventi automaticamente (i famosi sistemi STP, *Straight-Through Processing*), dall'al-

tro è necessario prevedere meccanismi da attuare qualora si verifichino eccezioni durante l'elaborazione automatica. Questi meccanismi, tipicamente, richiedono di realizzare un apposito sottosistema, denominato sistema di gestione delle eccezioni business (BEM, *Business Exception Management*) incaricato appunto di gestire eccezioni di tipo business. In particolare, questo sistema si dovrebbe occupare di:

1. predisporsi a ricevere notifiche relative a eccezioni di tipo business (per esempio sottoscrivendo il canale delle eccezioni);

2. ricevere e analizzare le varie segnalazioni generate dai sottosistemi assistiti;

3. per ogni messaggio ricevuto, valutare, in base a un opportuno sistema di regole, la priorità da assegnare alla corrispondente gestione;

4. creare un record relativo ad ogni eccezione ricevuta, inserirlo in un'apposita coda interna e, contestualmente, inviare una segnalazione agli opportuni operatori circa la sua presenza;

5. verificare continuamente i record presenti nelle varie code al fine di aumentare la priorità di quelli che sono presenti nella coda da un eccessivo intervallo temporale.

Da tener presente che quantunque l'intervento umano sia una soluzione molto flessibile, tipicamente è anche molto costosa, e pertanto è buona pratica minimizzarne l'utilizzo.

4.6.2 Valutare l'estensione temporale delle eccezioni

Le eccezioni di business tendono per loro natura a essere di tipo permanente. Se per esempio l'identificativo di un cliente presente in un messaggio è errato, questo rimarrà tale fin quando non viene intrapresa un'opportuna azione correttiva. Quindi all'interno del servizio non c'è nulla da fare: una volta generata l'eccezione questa non ha alcuna possibilità di risolversi automaticamente.

Un discorso diverso vale per le eccezioni di sistema. In questo caso, le cause che hanno portato alla generazione dell'eccezione potrebbero automaticamente risolversi all'interno dello stesso servizio in cui si è manifestata. Si consideri, per esempio, un servizio che tenti di connettersi al Database Management System e che fallisca per un eccesso temporaneo di traffico presente nella rete. Un altro esempio è relativo a un sistema che tenti di connettersi a un altro e che, per un eccessivo e temporaneo stress di quest'ultimo, non riceva una risposta entro l'intervallo di tempo previsto (timeout).

Eccezioni di sistema presentano spesso una persistenza temporanea e quindi una corretta gestione prevede di ritentare l'istruzione che può generarla, per un numero di predefinito di volte, prima di notificarla a ritroso.

Un esempio di possibile gestione è presentata nel listato 4.8.

Listato 4.8 – Modello per la gestione delle eccezioni di sistema.

```
int     attempts = 0;      // counts the number of failed attempts
boolean success  = false;  // success flag
try {
  while ( (attempts < MAX_FAILED_ATTEMPTS) && (!success) ) {
    try {
      < istruzione che può generare un'eccezione>
      <istruzione>
      <istruzione che può generare un'eccezione>
      <instruction>
      success = true;
    } catch (<eccezione specifica> es) {
      attempts++;
      if (attempts < MAX_FAILED_ATTEMPTS) {
        try {
          Thread.sleep(BASIC_TIME_WAIT*attempts);
        } catch (InterruptedException ie) {
          <effettuare il log dell'eccezione e quindi proseguire>
        }
      }
    }
  }
} finally {
  <parte del codice richiesto dal finally>
}
```

4.6.3 Gestire correttamente il perdurare delle eccezioni di sistema

In tutti quei in cui un'eccezione di sistema non si risolva in un arco temporale abbastanza breve (per esempio il codice del listato 4.8 fallisca le varie iterazioni) nasce il problema intricato di come gestire la situazione. Il sistema, indubbiamente, si trova in una situazione critica in cui, difficilmente, potrà soddisfare altre richieste. Situazioni del genere, tipicamente, richiedono l'intervento umano, magari semplicemente per riavviare uno dei sistemi entrato in un stato di malfunzionamento.

Le alternative disponibili includono: l'avvio della procedura controllata di *shut-down*, far transitare il sistema in un particolare stato di inattività (*idle*). Tale stato dovrebbe essere caratterizzato dal fatto che il sistema inibisca i canali di comunicazione di input (non accetti più stimoli), eccetto quelli provenienti da un'opportuna console di amministrazione e, a intervalli di tempo via via crescenti, tenti di verificare l'effettivo perdurare del problema. Questa seconda alternativa, sebbene più complessa, permette al sistema di riprendere automaticamente il corretto funzionamento non appena il problema di sistema sia risolto.

Quale sia la soluzione da implementare, come al solito, dipende da un insieme di fattori, quali: i requisiti utente, il particolare tipo di eccezione e i meccanismi presenti atti a segnalare tempestivamente, a un operatore umano, l'insorgere del problema.

Indipendentemente dalla soluzione scelta, è necessario far in modo che i processi business colpiti dal problema siano gestiti correttamente, come visto in precedenza. Ciò può limitarsi a inviare una comunicazione all'utente, a cercare di salvare opportune informazioni, e così via.

4.7 Considerare i classici problemi relativi all'impiego di sistemi di messaggistica

Il recente passato ha visto l'affermarsi di moderni sistemi di integrazione software come gli Enterprise Service Bus (ESB, Bus di Servizio Aziendale); ciò nonostante i sistemi di messaggistica sono ancora molto utilizzati. Questi permettono di disegnare e implementare grandi sistemi in termini di un insieme di sottosistemi interconnessi con elevato grado di disaccoppiamento. La loro presenza, tuttavia, nella quasi totalità dei casi, richiede di valutare attentamente e di gestire le seguenti due problematiche: messaggi avvelenati (*poisoned messages*) e ricezione di messaggi fuori ordine.

4.7.1 Gestire correttamente i messaggi "avvelenati" (*poisoned*)

Con il termine di messaggi "avvelenati", ci si riferisci a messaggi viziati da qualche problema che tendono a restare perennemente nel sistema se non opportunamente rimossi. Questo scenario si può verificare in situazioni in cui, per un particolare canale, il Message Oriented Middleware (MOM, infrastruttura orientata al messaggio) è impostato per un funzionamento a consegna garantita (*guaranteed delivery*: il messaggio viene consegnato una ed una sola volta). In particolare, questa situazione si ha quando un sistema riceve un messaggio che viola una o più pre- e/o post-condizioni del sistema ricevente; il ricevente, quindi, non può far altro che scartarlo generando un'opportuna eccezione. Il sistema di messaggistica, non ricevendo la conferma dell'avvenuta ricezione del messaggio (ossia il commit), dopo un leggero intervallo di tempo, presenta nuovamente il medesimo messaggio al sistema. Ciò perché si è impostata la modalità di consegna garantita. Questa successione potrebbe continuare all'infinito qualora nessuna azione sia intrapresa. Alla presenza di diversi messaggi avvelenati, si potrebbe generare una serie di conseguenze indesiderate come una notevole perdita di performance del sistema, perdita delle informazioni contenute nel messaggio stesso giacché questo non viene gestito, e così via.

Una buona tecnica per gestire problemi di questo tipo consiste nel richiedere ai sistemi destinatari di gestire apposite tabelle in cui memorizzare tre campi: l'identificatore univoco dei messaggi, il timestamp dell'ultima ricezione e un contatore di ricezioni. In questo modo, quando questo contatore raggiunge un valore prefissato (per esempio 3), il sistema ricevente è in grado di accorgersi delle presenza di un messaggio avvelenato e quindi procedere alla sua gestione. Questa, invece di generare un'eccezione (che spingerebbe nuovamente il messaggio nel MOM), si deve occupare di spostare il messaggio in un'apposita coda, denominata normalmente "ospedale dei messaggi" e di comunicare, al sistema di messaggistica l'avvenuta ricezione del messaggio, in modo analogo a quando tutto funziona correttamente.

La coda dei messaggi avvelenati dovrebbe poi essere monitorata da un opportuno meccanismo in grado di comunicare ad appositi operatori umani la presenza di messaggi avvelenati che devono essere gestiti.

4.7.2 Messaggi fuori sequenza

Questo scenario, come suggerisce il nome, si riferisce a situazioni in cui un sistema destinatario riceva dei messaggi in ordine diverso da quello di trasmissione. Sebbene non sempre questo scenario costituisca un problema insormontabile, in alcune situazioni potrebbe però generare scenari molto dannosi. Si consideri per esempio il caso in cui i messaggi relativi al prezzo di uno stesso prodotto finanziario siano ricevuti in ordine errato.

Sebbene molti sistemi di messaggistica recenti dispongano di meccanismi interni atti a gestire questo problema, la loro efficacia, in particolari scenari è decisamente ridotta. Si consideri, per esempio, il caso di un sistema a elevato livello di parallelismo; per un qualsiasi problema, uno dei sottosistemi diventa improvvisamente instabile ("va in crash") e necessita di essere riavviato. Si supponga, ulteriormente, che questo sottosistema abbia sottoscritto alcuni canali con consegna garantita dei messaggi. In questo scenario, i messaggi presenti nei relativi buffer passano automaticamente in uno stato di bloccaggio finché i corrispondenti *socket* non vengono chiusi. Ciò può richiedere un intervallo di tempo non trascurabile. Se nel frattempo, un meccanismo disegnato per l'elevata disponibilità riavvia il sottosistema "crashato", magari su un server diverso da quello originario, ecco che si verifica un'elevata probabilità di avere messaggi recapitati con un ordine diverso da quello in cui sono stati immessi nella coda.

Anche se lo scenario presentato può sembrare un caso limite, in sistemi complessi non lo è affatto! Il dato di fatto è che il problema dei messaggi fuori sequenza non solo *può* verificarsi ma *si verifica*. Pertanto, qualora ciò possa arrecare danni, è necessario disegnare ed implementare meccanismi per la gestione di queste anomalie.

Un sistema di intercettazione di questo problema richiede l'utilizzo di un meccanismo simile a quanto visto per i messaggi avvelenati. In particolare è necessario che i sistemi destinatari gestiscano una tabella in cui memorizzare l'identificativo del messaggio ricevuto e il momento in cui è stato ricevuto. Questo identificativo deve essere ottenuto dall'entità incapsulata nel messaggio stesso, e quindi deve avere una valenza per così dire business.

A questo punto, quando il sistema destinatario riceve un messaggio deve verificare se abbia o meno già ricevuto un messaggio con il medesimo identificativo del messaggio. In caso negativo, ovviamente, non ci sono problemi, ed è sufficiente creare un record relativo al nuovo messaggio. In caso positivo, invece è necessario controllare il *timestamp* dell'ultima occorrenza ricevuta con quella del messaggio. Se quest'ultimo si dimostra essere più recente, allora non ci sono problemi ed è sufficiente aggiornare il relativo record nella tabella. Mentre, in caso contrario, il nuovo messaggio è stato ricevuto fuori sequenza e quindi è necessario avviare la relativa gestione. Questa in funzione della situazione in cui si presenta, può richiedere diverse soluzioni. Per esempio, nel caso dei prezzi, qualora si riceva un messaggio fuori sequenza è sufficiente scartarlo: il prezzo attuale è sicuramente più recente di quello ricevuto in ritardo, mentre in altri contesti si può giungere fino alla situazione di dover richiedere l'intervento dell'operatore umano.