Exception Handling in Middlewares: the case of TuCSoN

Progetto di Sistemi Distribuiti

Balducci Mattia : 681211
Papini Alessia : 710524

Introduzione

Nell'ambito di questo progetto ci si è occupati di analizzare e riprogettare alcuni aspetti inerenti alla struttura, all'utilizzo e alla gestione delle eccezioni in TuCSoN.
In particolar modo ci si è concentrati sui seguenti aspetti fondamentali:

- Checked exception vs. Unchecked exception, discriminare un errore di programmazione da un errore non evitabile a priori;
- Utilizzo corretto e coerente delle eccezioni già presenti in TuCSoN e creazione di nuove eccezioni, al fine di aumentarne l'espressività e garantire una migliore correlazione tra causa ed eccezione;
- Throw vs. Catch, utilizzo corretto di questi due costrutti per assegnare le giuste responsabilità alle classi in fatto di gestione delle eccezioni.

Premessa

Nello svolgere questo progetto si è cercato di seguire le linee guida indicate dal manuale Java [1] riguardo all'uso e gestione delle eccezioni.
Si è, inoltre, ritenuta opportuna una fase preliminare di studio di TuCSoN in modo da tenere conto del contesto in cui si operava ed avere così le conoscenze necessarie per applicare al meglio le *best practices* sopra citate. Questa fase è stata svolta consultando la guida di TuCSoN [2] e tramite una prima esplorazione dei vari packages presenti.
Il progetto è stato svolto analizzando classe per classe, affrontando contemporaneamente le problematiche relative agli aspetti fondamentali precedentemente elencati che, per una maggior chiarezza, verranno qui esposti separatamente.

Checked vs. Unchecked

In merito all'utilizzo di eccezioni *checked* ed *unchecked*, il manuale Java [1] riporta: "**use checked exceptions for conditions from which the caller can reasonably be expected to recover**" e "**use runtime exceptions to indicate programming errors**".
Basandosi su queste regole si sono apportate modifiche all'eccezione "*InvalidOperationException"*, precedentemente implementata come *checked* e successivamente convertita ad *unchecked*, facendole quindi estendere la classe *RuntimeException.*
Analizzando questa eccezione, si è notato che veniva utilizzata per segnalare errori evitabili a priori utilizzando correttamente i metodi previsti.

Alcuni esempi si possono trovare all'interno della classe "*JVal"*, metodi come *toInt(), toLiteral(),* etc. effettuano un controllo sul *JArgType* dell'oggetto, lanciando una *InvalidOperationException* nel caso non corrisponda a quello indicato nel nome del metodo. Questa eventualità è evitabile facendo uso dei metodi di test come *isInt(), isLiteral(),* etc. già presenti nella classe.

```
@Override
public int toInt(){
    if (this.type == JArgType.INT) {
        return ((Integer) this.arg).intValue();
    }
    throw new InvalidOperationException("The JVal is not an Int");
}

@Override
public String toLiteral(){
    if (this.type == JArgType.LITERAL) {
        return this.arg.toString();
    }
    throw new InvalidOperationException("The JVal is not a Literal");
}
```

Un altro caso in cui viene utilizzata *InvalidOperationException* è nel metodo *getArg(int i)* di *JTuple,* nel quale si controlla che l'indice *i* non superi il numero di argomenti contenuti dalla tupla. Anche in questo caso è evidente che tale situazione è evitabile utilizzando il metodo *getNArgs()* presente nella classe.

```
@Override
public IJVal getArg(final int i) {
    if (i >= 0 && i < this.args.size()) {
        return this.args.get(i);
    }
    throw new InvalidOperationException("Index out of bounds. Value of the index i: "+i);
}

@Override
public int getNArgs() {
    return this.args.size();
}
```

Un caso in cui la scelta tra *checked* ed *unchecked* si è rivelata più difficoltosa, è quello delle eccezioni *InvalidAgentIdException* e *InvalidTupleCentreIdException*. In questo particolare caso entrambe le scelte avrebbero comportato sia aspetti negativi che positivi.
Il problema degli *Id*, sia per gli *agent* che per i *tuple centre,* nasce dal fatto che possono essere specificati dall'utente e quindi risultare non conformi con quanto richiesto da TuCSoN. Questo tipo di evenienza non rappresenta un esempio di "cattiva programmazione", per cui sembrerebbe corretto gestirla con un'eccezione *checked*. Tuttavia, nel codice di TuCSoN, è comune imbattersi in *Id* specificati direttamente nel codice, e quindi corretti, per i quali risulta inutile e fastidioso l'obbligo di inserire blocchi *try/catch*. La decisione finale è ricaduta sul mantenere le due eccezioni di tipo *checked*, in quanto non è possibile prevenire l'inserimento di *Id* errati da parte dell'utente. Se questa problematica risultasse non essere critica, sarebbe opportuno rivalutare la scelta in favore di eccezioni *unchecked.*

Utilizzo delle eccezioni già presenti e nuove eccezioni

Per quanto riguarda l'utilizzo corretto delle eccezioni già esistenti ci si è concentrati nel cercare di aggiungere informazioni significative, sia per il programmatore che per l'utente, mediante l'utilizzo della tecnica di *exception chaining* e/o l'aggiunta di un *detail message*. Mediante queste due informazioni è possibile, nel momento in cui occorrerà gestire l'eccezione, avere informazioni più approfondite sulle cause dell'eccezione, compresi i valori di eventuali variabili che l'hanno causata.
Si è notato inoltre una certa "confusione" o "incoerenza" nell'utilizzo di alcune eccezioni, in particolar modo quelle riguardanti problemi di rete. Un esempio è dato dalla classe *InspectorContextStub.java*, in cui metodi come *setTupleSet()* o *setProtocol()* catturano eccezioni di tipo *DialogException* per poi lanciare una *IOException*. Questa operazione appare già errata poichè si sta trasformando un' eccezione specifica di TuCSoN con una generica di Java, inoltre esaminando il metodo responsabile della *DialogException* si riscontra che quest'ultima viene generata a partire da una *IOException*. Si ottiene così una catena del tipo *IOException->DialogException->IOException* che conferma l'uso confusionale di queste eccezioni.
Nel tentativo di fare chiarezza, è stata riprogettata la struttura dell'eccezione *DialogException*, che ora rappresenta la *root exception* dalla quale estendono le nuove eccezioni *DialogInitializationException, DialogAcceptException, DialogCloseException, DialogSendException* e *DialogReceiveException*.
Tali eccezioni sono state pensate per essere facilmente associabili alla fase del *life-cycle* di una *dialog* in cui si sono verificate, astraendo dal problema specifico che le ha causate (informazione in ogni caso presente tramite *exception chaining*).

```
@Override
public InspectorContextEvent receiveInspectorEvent() throws DialogReceiveException {
    InspectorContextEvent msg;
    try {
        msg = (InspectorContextEvent) this.inStream.readObject();
    } catch (final IOException e) {
        throw new DialogReceiveException(e);
    } catch (final ClassNotFoundException e) {
        throw new DialogReceiveException(e);
    }
    return msg;
}
```

Oltre a quelle già presentate, sono state aggiunte anche le seguenti eccezioni:

- *InvalidProtocolTypeException*, eccezione *checked* utilizzata nei casi in cui si richieda l'utilizzo di un protocollo non supportato da TuCSoN;
- *IllegalPortNumberException,* eccezione *unchecked* utilizzata nel caso in cui venga specificato un numero di porta che non rientra nell'intervallo consentito.

Entrambe queste eccezioni prima erano gestite con una generica *DialogException*.

Throw vs. Catch

Per ogni singola eccezione lanciata nel codice di TuCSoN è stato necessario scegliere quale classe dovesse assumersi la responsabilità di gestirla. Questo aspetto è stato senza dubbio il più complesso da affrontare, poichè per ogni eccezione incontrata è stato necessario capire il contesto in cui essa si verificava e decidere se fosse giusto gestirla (*catch)* o demandarla ad una classe di livello superiore (*throw).*
Il ragionamento utilizzato prevede che nel caso di errori non risolvibili o di errori risolvibili dove le informazioni necessarie per risolverli sono possedute dalla classe, la responsabilità della gestione sia affidata alla classe stessa. In caso contrario si prevede che la gestione venga assegnata alla classe di livello superiore, nella quale verranno ripetute le stesse considerazioni.

Uno degli esempi più significativi è rappresentato dalla gestione degli *Id*. Nel caso in cui l'utente inserisca un *Id* non valido, l'eccezione generata verrà propagata "verso l'alto" fino ad una classe in grado di interagire con l'utente al fine di ottenere un nuovo *Id* valido.

Conclusioni

In questo progetto ci si è quindi concentrati sulla struttura e sull'utilizzo delle eccezioni in TuCSoN e non sulle azioni da intraprendere per gestirle. Nei primi due punti elencati è stato possibile effettuare valutazioni più obiettive basate quasi esclusivamente sull'applicazione di *best practices,* mentre l'aspetto riguardante la responsabilità della gestione è stato influenzato dal livello generale di conoscenza di TuCSoN, di conseguenza le decisioni prese sono, almeno in parte, di natura soggettiva.

Bibliografia

[1] "*Effective Java – Second Edition*", Joshua Bloch, 2008
[2] "*The TuCSoN Coordination Model & Technology - A Guide"*, Andrea Omicini e Stefano Mariani, 2014, http://www.slideshare.net/andreaomicini/the-tucson-coordination-model-technology-a-guide