TECNICHE SPECIALI DI PROGRAMMAZIONE

Progetto finale: analisi e implementazione del pattern ***Chain of Responsibility*** in ambito AOP

a.a. 2016-2017

GIORGIO AUDINO
886853

Struttura delle classi con eredità implementate

Handler — implementa HandlerI
I\
I \
I *GenericApplicationHandler*
I I- OkButtonHandler
I I- PrintButtonHandler
I I- PrintDialogHandler
I I- SaveDialogHandler
I I- ApplicationHandler
\
GenericUserHandler
I- UserNotEnabledHandler
I- UserWithoutActivityHandler
I- UserBirthdayHandler

Chain
I\
I \
I ApplicationChain
\
UserChain

CORPattern (.aj)
I\
I \
I ApplicationCofR
\
UserCofR

Implementazione astratta

L'implementazione astratta del pattern è contenuta nel package cofr, mentre l'implementazione concreta è nel default package del progetto.

L'implementazione creata del pattern prevede che la chain of responsibility riceva un oggetto di tipo Request e restituisca una Response.

Un Handler è un oggetto in grado di gestire una richiesta delegata alla catena. Contiene un puntatore all'handler successivo nella catena ed espone, oltre a metodi accessori come quello per definire il nome e l'handler successivo, un metodo chiamato handleRequest che riceve come parametro un oggetto di tipo Request. Tra i metodi non esposti, ci sono tre metodi chiave:

- canHandle(Request) è il metodo che dev'essere definito per ciascun handler concreto e che restituisce un booleano che indica se quel particolare handler è in grado di gestire la richiesta
- doHandle(Request) è il metodo invocato se l'handler in questione è quello deputato alla gestione della richiesta ricevuta. Restituisce un oggetto di tipo Response
- mustHandle(Request) è il metodo invocato se l'handler è l'ultimo della catena. In base alle diverse implementazioni può non fare niente oppure fare attività generiche.

Sono stati definiti quattro stati di risposta all'interno dell'interfaccia HandlerI implementata da Handler:

- OK: la richiesta è stata elaborata correttamente
- NOT_HANDLED: la richiesta non è stata elaborata dalla catena
- MUST_HANDLE: la richiesta è stata elaborata dall'ultimo elemento della catena ma nessuno degli handler era in grado di gestirla.
- ERROR: c'è stato un errore nell'elaborazione della richiesta

Le classi Chain e HandlerChainFactory sono le due classi intorno a cui ruota l'implementazione del pattern. La classe Chain, astratta, contiene alcuni metodi statici e una proprietà statica. Tale proprietà, chiamata handlerMap, è una hashMap che lega il nome di una catena alla lista dei nomi delle classi che la compongono, mentre i metodi statici consentono di manipolare tale hashMap. Contiene quindi un metodo non statico, chiamato delegateRequest la cui implementazione restituisce un oggetto Response molto semplice, con il parametro status settato a NOT_HANDLED. Si tratta di fatto della fallback nel caso in cui l'operazione di weaving non funzioni come previsto. I pointcut degli aspetti sono in ascolto sulle invocazioni di questo metodo (sulle sottoclassi) e i piece of advice si sostituiscono alla sua esecuzione: in un flusso normale, la porzione di codice del metodo delegateRequest scritta nella classe Chain non viene mai eseguita.

La classe HandlerChainFactory è deputata alla costruzione concreta della catena, che finora è stata solo definita come una lista di nomi di classi. Contiene un solo metodo statico che, ricevuta come parametro la lista degli handler da creare, ricorsivamente collega ogni Handler a quello successivo e restituisce la testa della lista. In questo modo la catena è creata on-the-fly ogni volta che si rende necessario.

L'aspetto CORPattern è l'aspetto astratto creato per gestire la catena. Contiene il prototipo di un pointcut, chiamato creation, e di un metodo getType che servirà al piece of advice a sapere a quale catena deve fare riferimento. Il piece of advice è già definito nell'aspetto astratto. Esso, con un joinpoint "around" legato all'unico pointcut definito, invoca getType ricevendo come risposta il nome della catena da creare, invoca il metodo createHandlerChain della classe HandlerChainFactory quindi invoca il metodo handleRequest dell'oggetto Handler ricevuto (quindi la testa della linked list) e restituisce l'oggetto Response ricevuto.

Caso di studio: Application

La catena chiamata Application deriva dall'esempio riportato nel Gamma. Il caso d'uso è quello di un sistema di aiuto contestuale per un'interfaccia grafica. L'utente dovrebbe poter ricevere informazioni facendo click su un punto qualsiasi dell'interfaccia e ciò che riceve deve dipendere dal punto in cui ha cliccato e dal contesto.

All'interno della mia implementazione questa interazione è stata ovattata e gli oggetti Request la cui gestione è deputata alla catena Application hanno un parametro "target" in base al quale ogni Handler può sapere se è in grado di gestire o meno tale richiesta. In un'implementazione reale di questo caso d'uso la richiesta dovrebbe contenere dati sull'oggetto su cui l'utente ha cliccato e sul contesto e ogni Handler avrà una propria implementazione del metodo canHandle relativa al caso specifico (come nel secondo caso di studio).

La classe GenericApplicationHandler è una sottoclasse, a sua volta astratta, della classe Handler. Implementa i tre metodi canHandle, doHandle e mustHandle solo definiti nella classe padre e servirà come superclasse per tutti gli handler di questa catena. Tutti gli handler figli (OkButtonHandler, PrintButtonHandler, PrintDialogHandler, SaveDialogHandler, ApplicationHandler) non contengono alcuna implementazione, poiché il criterio di canHandle per questo caso di studio semplificato è che il parametro "target" della richiesta sia uguale al nome dell'handler.

Per testare questo caso ho scritto un metodo generateForApp nella classe RequestGenerator che genera un certo numero di richieste, specificandone il target. Aggiunge infine alla lista di richieste generata una richiesta con un target inesistente, che nessun handler sarà in grado di gestire e che verrà quindi gestita "a forza" dall'ultimo anello della catena, producendo una risposta di tipo MUST_HANDLE. Ciascuna delle richieste generate sarà il parametro di un'invocazione del metodo delegateRequest della classe ApplicationChain.

Caso di studio: User

Consideriamo un servizio web dove gli utenti, per fruire del servizio, devono registrarsi. Ogni giorno un worker interno al sistema analizza tutti gli utenti registrati per inviare loro le email di *nurturing*, cioè tutte quelle per ricordare agli utenti dell'esistenza di tale servizio e/o far notare funzionalità magari non ancora scoperte. La decisione riguardo a ciascun utente verte su diversi fattori, quasi tutti inerenti lo stato dell'utente. Ad esempio inviare email ad un utente troppo frequentemente probabilmente potrebbe irritarlo e portare ad un abbandono o a una segnalazione per spam, così come bisogna inviare a ciascun utente comunicazioni in merito a ciò che potrebbe essere interessato a fare oppure istruzioni riguardo a funzionalità che non ha mai usato.

Per poter implementare questo caso sono state create due classi: User e Email. Un utente, oltre ai dati personali (nome, cognome, data di nascita), ha salvati i dati relativi al momento in cui è stato creato (createdAt), il fatto che sia abilitato e che abbia, quindi, confermato la propria email (si considera un servizio che prevede una registrazione two-step), il momento in cui ha fatto l'ultimo accesso (lastAccess) e l'ultima email che gli è stata inviata (lastEmailSent). Tutte le date sono salvate come istanze di TimeStamp.

La mia implementazione di questo caso d'uso prevede tre semplici casi in cui il fornitore del servizio vuole notificare l'utente, che corrispondono ai tre handler che estendono GenericUserHandler:

- UserNotEnabledHandler: l'utente si è registrato da una settimana ma non ha confermato la propria email
- UserWithoutActivityHandler: l'utente non fa niente da una settimana
- UserBirthdayHandler: è il compleanno di questo utente

L'unico caso in cui viene tenuto in considerazione il fatto di non voler inviare troppe email all'utente è il secondo; nel primo caso l'utente non ha ancora attivato il proprio account, quindi è importante inviarla in ogni caso, mentre nell'ultimo la mail è solo per fare gli auguri di compleanno: nessuno se ne lamenterà mai.

Contrariamente al caso di studio precedente, dove la richiesta già conteneva quanto necessario a discernere tra i vari handler della catena, in questo caso la decisione dipende dallo stato di ciascun utente. La classe GenericUserHandler implementa quindi i metodi doHandle e mustHandle lasciando alle sottoclassi l'onere di implementare canHandle in base ai diversi casi. Qualora si giunga all'ultimo anello della catena senza che nessuno debba gestire la richiesta, nessuna email dovrà essere inviata a tale utente e il metodo sendEmail di EmailService, che simula l'invio di una mail, non verrà invocato.

L'implementazione di doHandle, metodo invocato quando l'handler è in grado di gestire la richiesta, crea una mail del tipo corretto, invoca il metodo sendEmail e restituisce un oggetto Response il cui stato è OK se la mail è stata inviata correttamente oppure ERROR in caso contrario. In questo caso simulato la mail viene sempre "inviata" correttamente.

Per testare questo caso il metodo generateForUser della classe RequestGenerator crea quattro richieste ciascuna contenente un utente differente in modo tale che ci sia una richiesta per ciascuno degli handler della catena più una "ingestibile", cioè di un utente al quale non bisogna mandare email. Nel main, analogamente a quanto avviene nel caso di Application, ciascuna delle richieste generate sarà parametro di un'invocazione del metodo delegateRequest di UserChain.

Considerazioni sull'implementazione AOP del pattern

Durante l'implementazione del pattern utilizzando gli aspetti si sono posti alcuni ostacoli per i quali è stato necessario trovare una soluzione. Uno di questi è sorto scrivendo il pointcut *creation*, che inizialmente era il seguente:

```
public pointcut creation(Request request):
    call( *Chain.delegateRequest(Request)) && args(request);
```

L'idea era di non avere due implementazioni del pointcut solo definito nella classe padre e di non avere le due classi concrete ApplicationChain e UserChain ma di utilizzare solo la classe Chain (non più astratta quindi). Tuttava, il pointcut così scritto avrebbe "intercettato" tutte le invocazioni al metodo delegateRequest, costringendo il piece of advice a discernere, in base all'oggetto Request (o ad un secondo parametro e.g. un int/enum mappato su una proprietà statica), che tipo di invocazione è stata fatta.

L'implementazione fatta rende molto dinamica la gestione della catena che in un'applicazione concreta avrei gestito in maniera simile ma salvando da qualche parte le varie catene e non solamente all'interno di una classe statica (e.g. un database) onde evitare di dover ricreare le catene ogni volta.

A mio parere l'utilizzo dei costrutti propri dell'AOP non dà un reale contributo all'utilizzo del pattern. Per quanto sia vero che l'implementazione suggerita consenta una notevole elasticità riguardo agli elementi che compongono la catena, ad ogni invocazione del metodo delegateRequest di una certa sottoclasse di Chain, il metodo factory deve costruire ogni oggetto della catena, che resteranno in vita per una manciata di istruzioni da eseguire, cioè il percorso della richiesta lungo la catena, per poi essere raccolti dal garbage collector. Se nell'esempio degli utenti la catena fosse composta da 10 classi e la base di dati da analizzare contenesse 1000 utenti diversi, servirebbe creare 10.000 classi. In un'implementazione tradizionale ne basterebbero 10.

Oltre alla costruzione della catena, non ho evidenziato altre funzionalità orizzontali rispetto al tipo di catena tali per cui l'estrazione di porzioni di codice e l'utilizzo di un aspetto sia utile per la manutenibilità del codice. Ogni catena ha caratteristiche proprie, sebbene utilizzando un sistema tipo client-server con richieste e risposte ho potuto generalizzare i parametri di ogni invocazione della catena. Anche in questo caso, però, per poter implementare il secondo caso d'uso è stato necessario aggiungere la proprietà user nella classe Request per evitare di creare una lista di parametri di elementi di tipo Object e dover aggiungere un overhead di istruzioni per il controllo del tipo di ogni elemento, rendendo il codice poco leggibile e meno performante e perdendo anche ogni controllo statico sui tipi.

In conclusione, la mia opinione è che l'implementazione del pattern chain of responsibility non giovi con l'utilizzo di aspetti che, in questo caso, rendono il codice meno performante e più difficilmente manutenibile.

Note

Le istruzioni per poter testare sono:

```
$ ajc -1.8 -sourceroots src
$ java src/AOPCofRTester
```

L'esecuzione del main, nella classe AOPCofRTester, fa 3 test differenti del sistema: due sul primo caso di studio e uno sul secondo. I primi due test sono uguali con l'unica differenza che, prima del secondo, viene aggiunto un handler alla catena. Vengono quindi generate n richieste casuali (con n uguale al numero di elementi che compongono la catena) più un'ultima che nessun handler sarà in grado di gestire. Per il terzo test il generatore di richieste crea prima quattro utenti aventi quattro stati diversi e ne incapsula ciascuno all'interno di una richiesta, come descritto nella sezione riguardante il caso di test User.