Example14

Table of contents

Aspect-Oriented Programming

Active Aspect

Un oggetto si dice **attivo** se i suoi metodi vengono eseguiti in thread che non necessariamente sono quelli del chiamante. Sarebbe una delle ipotesi base della programmazione a oggetti concorrente: mettersi in attesa del risultato e non procedere finche' questo non viene generato, e' uno dei modi possibili.
Altra possibilita' e' quella d'inviare il messaggio e lasciare che quello che ha prodotto il risultato, proceda con il calcolo che vogliamo.
Sarebbero rispettivamente i meccanismi del `Future` e `Callback`, invio del messaggio e niente attesa, *chiamate asincrone* senza aggiunta di thread.

Avremo oggetti riceventi messaggi, con disposizione di un *pool di thread* per elaborare i messaggi (eventualmente aspettando se nessuno e' libero), che potranno fare 2 cose a seconda del tipo di chiamata:

1. asincrona, producente risultato tramite `Future`;
2. chiamata *continuation* dal `Callable`, come se chiedessimo di fare altro dopo aver ottenuto il risultato.

Nel momento in cui applichiamo oggetto attivo, applichiamo anche l'**interfaccia attiva**: se partiamo da una interfaccia con certi metodi, certi messaggi, identificati da tipi di argomento, eccezioni e valori di ritorno, nell'interfaccia attiva mettiamo:

- un valore di ritorno tipo `Future<T>`, messaggio asincrono "mi metto in attesa di quando il risultato arrivera'";
- un metodo per ogni messaggio, con numero argomenti originale+1, di stesso tipo e ultimo essere `Callback<T>`, messaggio "manipolare il risultato".

Estendiamo l'interfaccia T con `Active<T>`. Sono metodi di A:

- `Future<R> m(T1, T2, ..., Tn)`
- `void m(T1, T2, ..., Tn, Callback<R>)`

it.unipr.informatica.examples

DownloadManager.java

Interfaccia passiva pensata per descrivere oggetti i cui messaggi vengono inviati in modo sincrono. Viene restituita, o una URL insieme ai byte scaricati, o un'eccezione.

SimpleDownloadManager.java

Implementa l'interfaccia `DownloadManager.java`.
Aggiungiamo `@Override` sul metodo `ResourceContent`.

```
        // ...
        @Override
        public ResourceContent download(String url) throws IOException {
                // ...
        }
        // ...
```

ActiveDownloadManager.java

Per ogni messaggio dell'interfaccia passiva, ne mettiamo 2.
Queste eccezioni non devono essere le stesse dei metodi originali sincroni, siccome passano per `Future` e per `Callback`.

Leghiamo l'interfaccia attiva con quella passiva usando `Active<DownloadManager>`. L'interfaccia attiva si puo' chiamare in qualunque modo, non importa quale.

```
public interface ActiveDownloadManager
        extends Active<DownloadManager> {
        public Future<ResourceContent> download(String url);
        public void download
                (String url, Callback<ResourceContent> callback);
}
```

Example14.java

Per prima cosa viene costruito il `DownloadManager` passivo.
Appiccichiamo l'aspetto attivo e per farlo ha bisogno di 2 informazioni:

- qual'e' l'oggetto passivo;
- qual'e' l'interfaccia da utilizzare come interfaccia attiva.

La variante dell'oggetto attivo viene ritornata tramite handler, di tipo `ActiveHandler` con parametro generico il nome dell'interfaccia attiva `<ActiveDownloadManager>`.
Facendo `get()` otteniamo l'oggetto attivo che per ogni messaggio alloca l'esecuzione di un

metodo su uno degli n thread del pool.
Avendo l'oggetto attivo possiamo fare le chiamate asincrone:

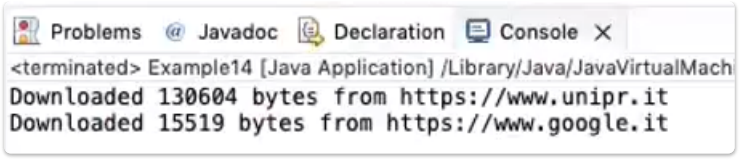
- chiamata con `Callback`
  usando una lambda expression, riceve argomento `ResourceContent` passando a questo la funzione `process()` che ci dice "scaricati tot. byte da URL"

```
private void process(ResourceContent content) {
        System.out.println("Downloaded " + content.getData().length +
        " bytes from " + content.getURL());
}
// ...
```

- chiamata con `Future`
  specifichiamo che stiamo usando la chiamata con il `Callback` in fondo, dove viene ritornato un `Future` per poter andare avanti o fermarci ed elaborare `process()`

Una volta finito di elaborare facciamo `shutdown()`.

```
        // ...
        private void go() {
                DownloadManager downloadManager = new SimpleDownloadManager();

                ActiveHandler<ActiveDownloadManager> downloadManagerHandler =
                ActiveAspect.attach(ActiveDownloadManager.class,
                downloadManager);

                ActiveDownloadManager activeDownloadManager =
                downloadManagerHandler.get();

                activeDownloadManager.download
                ("https://www.unipr.it", (ResourceContent content) →
                process(content));

                Future<ResourceContent> future =
                activeDownloadManager.download("https://www.google.it");

                try {
                        ResourceContent content = future.get();
                        process(content);
                } catch (Throwable throwable) {
                        throwable.printStackTrace();
                }
                downloadManagerHandler.shutdown();
        }

        public static void main(String[] args) {
                new Example14().go();
        }
}
```



Una volta che abbiamo l'oggetto attivo, inviamo i messaggi usando la *dot notation* che usiamo sempre, con la differenza che il tutto e' asincrono. Cambia quello che succede una volta che il messaggio viene recapitato.

`it.unipr.informatica.aspects`

`Active.java`

Interfaccia marcatore che serve unicamente per costruire oggetti con le chiamate asincrone, costruire oggetti attivi. `<T>` e' nome dell'interfaccia passiva.

```
public interface Active<T> {
}
```

`ActiveHandler.java`

Viene ritornato dal nostro aspetto, oggetto che permette l'accesso alla variante attiva dell'oggetto target da cui siamo partiti e che se serve, quando serve, permette di fare `shutdown()`.

```
public interface ActiveHandler<T extends Active<?>> {
        public T get();
        public void shutdown();
}
```

`ActiveAspect.java`

Vediamo che vincoli tra i tipi vengono specificati.
Il metodo `attach()` ha vincoli di tipo, siccome statico vanno specificati all'inizio:

- tipo dell'interfaccia attiva `A`;
- tipo del target `T`.

L'interfaccia attiva di `A` deve estendere `Active<T>`.

```
public class ActiveAspect {
        public static <T, A extends Active<T>> ActiveHandler<A> attach
        (Class<A> activeInterface, T target) {
                return attach(activeInterface, target, 10);
        }
        // ...
```

La seconda `attach()` e' identica con la sola differenza la specifica del numero di thread da mettere nel pool. Viene costruito per davvero il proxy, che dovra' intercettare tutti i messaggi sull'interfaccia attiva, costruire task da svolgere da parte dell'esecutore, chiedere l'esecuzione di questi e infine restituire i risultati che potranno essere o in forma `Future` o nel caso `Callback`, il task sara' quello a eseguire le sue manovre.

Il proxy viene costruito con i soliti argomenti:

- class loader il piu' vicino possibile al target;
- array d'interfacce che andremo a implementare (in questo caso la sola interfaccia attiva);
- invokation handler, contenente il metodo da utilizzare all'arrivo di messaggi.

Ritorniamo l'active handler per ottenere `get()` e `shutdown()` a cui serve:

- l'oggetto attivo;
- invocation handler (avremmo da passare l'executor service ma e' uguale).

```
        // ...
        public static <T, A extends Active<T>> ActiveHandler<A> attach
        (Class<A> activeInterface, T target, int poolSize) {
                if (activeInterface == null)
                        throw new IllegalArgumentException
                        ("activeInterface == null");

                if (target == null)
                        throw new IllegalArgumentException("target == null");

                if (poolSize < 1)
                        throw new IllegalArgumentException("poolSize < 1");

                InnerInvocationHandler invocationHandler =
                new InnerInvocationHandler(target, poolSize);

                Object proxy =
                Proxy.newProxyInstance(target.getClass().getClassLoader(),
                new Class<?>[] { activeInterface }, invocationHandler);

                @SuppressWarnings("unchecked")
                A object = (A) proxy;

                return new InnerActiveHandler<A>(object, invocationHandler);
        }
        // ...
```

```
        // ...
        private static class InnerActiveHandler<A extends Active<?>>
        implements ActiveHandler<A> {
                private A proxy;
                private InnerInvocationHandler handler;

                private InnerActiveHandler
                (A proxy, InnerInvocationHandler handler) {
                        this.proxy = proxy;
                        this.handler = handler;
                }

                @Override
                public A get() {
                        return proxy;
```

```
                }

                @Override
                public void shutdown() {
                        handler.shutdown();
                }
        }
        // ...
```

L'invokation handler riceve un messaggio e sulla base di questo, costruisce il task e lo fornisce all'executor service. Il task costruito prevede l'esecuzione sincrona del metodo passivo e una volta fatto, sara' l'executor service a scegliere la strada `Future` o `Callback`.

```
        // ...
        private static class InnerInvocationHandler implements
        InvocationHandler {
                private ExecutorService executorService;
                private Object target;

                private InnerInvocationHandler(Object target, int poolSize) {
                        this.target = target;
                        this.executorService =
                        Executors.newFixedThreadPool(poolSize);
                }

                private void shutdown() {
                        executorService.shutdown();
                }
                // ...
```

Nel caso del `Future` e' semplice siccome il numero di argomenti deve essere uguale in entrambe le versioni attive e passive. Finiamo quindi nell'`else` che si trova in fondo. Se altrimenti le condizioni non sono soddisfatte (l'utente non ha seguito le convenzioni), non siamo in grado di bloccare l'errore e quindi lo ritorniamo.

Nel caso del `Callback`, se ha piu' di un argomento e l'ultimo e' una callback, per convezione questo messaggio e' uno di quelli asincroni.
Costruiamo copia degli argomenti effettivi, lasciando fuori l'ultimo argomento e mettendoci `Callback` come suo valore.
Possiamo costruire il task che fara' la stessa cosa del caso `Future`.

```
                // ...
                @Override
                public Object invoke
                (Object proxy, Method method, Object[] arguments)
                throws Throwable {
                        Class<?>[] parameterTypes = method.getParameterTypes();
                        int parameterCount = parameterTypes.length;
                        Class<?> targetClass = target.getClass();

                        // controlliamo se siamo nel caso del 'Callback'
                        if (parameterCount > 0 &&
                        parameterTypes[parameterCount - 1] == Callback.class) {
```

```
                                parameterCount--;
                                Class<?>[] newParameterTypes =
                                (Class<?>[]) Arrays.copyOf
                                (parameterTypes, parameterCount);

                                Method passiveMethod =
                                targetClass.getMethod
                                (method.getName(), newParameterTypes);

                                @SuppressWarnings("unchecked")
                                Callback<Object> callback =
                                (Callback<Object>) arguments[parameterCount];

                                Object[] newArguments =
                                Arrays.copyOf(arguments, parameterCount);

                                executorService.submit(() ->
                                invokeMethod(passiveMethod, newArguments),
callback);

                                return null;
                        } else {
                                Method passiveMethod =
                                targetClass.getMethod
                                (method.getName(), parameterTypes);

                                return executorService.submit(() ->
                                invokeMethod(passiveMethod, arguments));
                        }
                }
                // ...
```

Come s'invoca il metodo.

```
                // ...
                private Object invokeMethod
                (Method passiveMethod, Object[] arguments) throws Exception {
                        try {
                                return passiveMethod.invoke(target, arguments);
                        } catch (InvocationTargetException exception) {
                                Throwable cause = exception.getCause();

                                if (cause instanceof RuntimeException)
                                        throw (RuntimeException) cause;

                                if (cause instanceof Exception)
                                        throw (Exception) cause;

                                throw exception;
                        }
                }
        }
}
```