CONTRADER

Guida Java Console PRO

Antonio De Santis,
Vittorio Valent,
Davide Ferretti

Giugno 2019

Indice

1 Introduzione **4**

2 DTO **4**

3 Converter **5**

4 Interfacce e Classi Astratte **6**

4.1 Interfaccia 6

4.2 Classe Astratta 7

1 Introduzione

Questa guida estende la guida Java Console. In questa fase si aggiungono il pattern DTO, il Converter e il concetto di classe astratta. Si raccomanda, come sempre, di leggere i commenti nel codice.

2 DTO

L'Oggetto di Trasferimento Dati o Data Transfer Object in sigla DTO è un design pattern usato per trasferire dati tra sottosistemi di un'applicazione software. I DTO sono spesso usati in congiunzione con gli oggetti di accesso ai dati (DAO) per recuperare i suddetti da una base di dati.

La differenza tra gli oggetti di trasferimento dati e gli oggetti di business o gli oggetti di accesso ai dati è che un DTO non ha alcun comportamento se non di archiviare e recuperare i suoi dati. Il pattern permette di alleggerire il passaggio di dati alla View, di separare logicamente l'accesso ai dati (DAO) e il loro trasferimento (DTO), e di avere una maggior sicurezza (i DTO possono non avere dati sensibili che invece hanno gli oggetti de model). A livello di codice le classi DTO e Model sono pressoché uguali:

```
public class UserDTO {

   private int id;

   private String username;

   private String password;

   private String usertype;


   public UserDTO() {

   }

   public UserDTO (String username, String password, String usertype) {
      this.username = username;
      this.password = password;
      this.usertype = usertype;
   }
```

```
    public UserDTO (int id, String username, String password, String usertype) {
        this.id = id;
        this.username = username;
        this.password = password;
        this.usertype = usertype;
    }
}
```

Naturalmente, come del Model, ci saranno i getter, i setter e il toString.

3 Converter

Il Converter si occupa di trasformare gli oggetti del Model in oggetti di tipo DTO. I suoi metodi saranno pertanto un `DTO toDTO(Entity entity)` e un metodo `Entity toEntity(DTO dto)`. Il converter viene chiamato dal Service in fase di recupero dati dal DAO (vedi codice di AbstractService).

Di seguito vediamo come funziona il Converter dello User: come possiamo osservare il metodo crea un nuovo oggetto (User DTO nel primo, User nel secondo) e , tramite i getter, lo riempie con gli attributi dell'oggetto da convertire. Questo nuovo oggetto convertito viene quindi ritornato da metodo.

```
public class UserConverter implements Converter<User, UserDTO> {

    public UserDTO toDTO(User user) {
        UserDTO userDTO = new UserDTO(user.getId(), user.getUsername(),
                                user.getPassword(), user.getUsertype());
        return userDTO;
    }

    public User toEntity(UserDTO userDTO) {
        User user = new User(userDTO.getId(), userDTO.getUsername(),
                        userDTO.getPassword(), userDTO.getUsertype());
        return user;
    }
}
```

4 Interfacce e Classi Astratte

4.1 Interfaccia

In generale, un interfaccia (`interface`) rappresenta una sorta di "promessa" che una classe si impegna a mantenere. La promessa è quella di implementare determinati metodi di cui viene resa nota soltanto la definizione. Ciò che è importante non è tanto come verranno implementati tali metodi all'interno della classe ma, piuttosto, che la denominazione ed i parametri richiesti (le cosiddette *firme dei metodo*) siano assolutamente rispettati.
Sebbene le interfacce non vengano istanziate, come avviene per le classi, esse conservano determinate caratteristiche che sono simili a quelle viste nelle classi ordinarie. Ad esempio, una volta definita un'interfaccia, è possibile dichiarare un oggetto come se fosse del tipo dichiarato dall'interfaccia stessa utilizzando la medesima notazione utilizzata per la dichiarazione di variabili. Inoltre, allo stesso modo delle classi, è possibile utilizzare l'ereditarietà anche per le interfacce, ovvero definire una interfaccia che estenda le caratteristiche di un'altra, aggiungendo altri metodi all'interfaccia padre.
Infine, una classe può implementare più di una interfaccia. Ovvero, è possibile obbligare una classe ad implementare tutti i metodi definiti nelle interfacce con le quali essa è legata. Questa ultima caratteristica fornisce, indiscutibilmente, la massima flessibilità nella definizione del comportamento che si desidera attribuire ad una classe.
Di seguito troviamo l'interfaccia Service: come possiamo vedere i metodi di CRUD non sono implementati, ma solo **firmati**. Osserviamo inoltre che l'interfaccia da un tipo generico DTO, che servirà poi alla classe astratta che la implementa.

```
public interface Service<DTO> {

public List<DTO> getAll();

public DTO read(int id);

public boolean insert(DTO dto);

public boolean update(DTO dto);

public boolean delete(int id);

}
```

4.2 Classe Astratta

Nella programmazione orientata agli oggetti una classe astratta (`abstract class`) è una classe che definisce una interfaccia senza implementarla completamente. Questo serve come base di partenza per generare una o più classi specializzate aventi tutte la stessa interfaccia di base, le quali potranno poi essere utilizzate indifferentemente (ovvero in modo polimorfico) da applicazioni che conoscono l'interfaccia base della classe astratta.

La classe astratta da sola non può essere istanziata, viene progettata soltanto per svolgere la funzione di classe base (chiamata a volte anche classe genitrice) e da cui le classi derivate (chiamate anche classi figlie) possono ereditare i metodi. Le classi astratte sono usate anche per rappresentare concetti ed entità astratte. Le caratteristiche "incomplete" della classe astratta vengono condivise da un gruppo di sotto-classi figlie, che vi aggiungono caratteristiche diverse, in modo da colmare le "lacune" della classe base astratta.

Le classi astratte possono essere considerate come super-classi che contengono metodi astratti, progettate in modo che le sotto-classi che ereditano da esse ne "estenderanno" le funzionalità implementandone i metodi. Il comportamento definito da queste classi è "generico". Prima che una classe derivata da una classe astratta possa essere istanziata essa ne deve implementare tutti i metodi astratti. Per sintetizzare, quando viene definita una classe astratta il programmatore deve tener presente che si tratta di una classe che non può essere istanziata direttamente; per fare ciò è necessario creare una classe derivata mediante l'ereditarietà. Questo processo di astrazione ha lo scopo di creare una struttura base che semplifica il processo di sviluppo del software o che indirizza la programmazione delle classi figlie. Al contrario, una classe concreta è una classe dalle quali possono essere create ("istanziate").

Vediamo nel dettaglio un esempio: la classe AbstractService. Osserviamo innanzitutto che AbstractService ha due tipi generici: `Entity` e `DTO`. Questi verranno specificati in ciascuna classe concreta (ad esempio UserService). Si osservi che il tipo generico `DTO` è lo stesso della interfaccia implementata.

```
public abstract class AbstractService<Entity,DTO>
                      implements Service<DTO> {
```

Dato che il Service deve chiamare il Converter e il DAO ne dichiara le interfacce. Ogni classe concreta dovrà costruire poi il converter e il DAO della

propria entità (vedi UserService).

```
   protected DAO<Entity> dao;

   protected Converter<Entity,DTO> converter;
```

A questo punto passiamo all'implementazione dei metodi , che useranno i tipi generici come parametri e valori di ritorno. Così facendo, quando la classe concreta erediterà i metodi, essa sostituirà il proprio tipo al posto dei tipi generici. Il vantaggio di usare i tipi generici è chiaro: Invece di copiare e incollare gli stessi metodi da una classe concreta all'altra cambiando solo i tipi, li implemento una volta sola con i tipi generici.

```
   public List<DTO> getAll() {
      return converter.toDTOList(dao.getAll());
   }

   public DTO read(int id) {
      return converter.toDTO(dao.read(id));
   }

   public boolean insert(DTO dto) {
      return dao.insert(converter.toEntity(dto));
   }

   public boolean update(DTO dto) {
      return dao.update(converter.toEntity(dto));
   }

   public boolean delete(int id) {
      return dao.delete(id);
}
```