DAO

O padrão **Data Access Object** (DAO) é um padrão introduzido no ambiente JEE [3] para simplificar e desacoplar a interação das aplicações Java com a API JDBC.

O problema

A maioria (para não dizer todas) das aplicações de nível corporativo usam algum tipo de persistência de dados. Entre eles o mais usado é o Banco de Dados e a linguagem SQL é amplamente utilizada para comunicar com os sistemas gerenciadores de banco de dados. Java suporta esta necessidade desde cedo com o advento da API JDBC (Java Database Connectivity).

Antes do uso de Java em sistemas corporativos as aplicações eram majoritariamente escritas em linguagens orientadas a processo e a ordem e instruções SQL específicas continham regras de negócio. Ao passar esses sistemas para Java, essas regras não se poderiam perder e, ao mesmo tempo, há que se trabalhar com objetos. O padrão DAO visava originalmente encapsular esses conjuntos de códigos SQL que existiam em aplicações legadas[3]. Esse código continha regras tanto na pesquisa dos dados, tanto na edição dos dados. É comum ao inserir um determinado dado, ter que inserir ou atualizar outros.

Para projetos novos, que não dependem de código legado, JDBC é a escolha para comunicar com bancos dados. Contudo o uso de JDBC obriga as aplicações Java, a escrever SQL para comunicar com o gerenciador de bando de dados. Essa comunicação é feita utilizando o padrão Bridge em que a interface é Java e igual para todos os sistema de gerenciamento de bancos de dados e a implementação é específica de cada um encapsulada no conceito de *driver*. O problema é que nem todos os drivers JDBC suportam todos os tipos de instrução SQL, ou nem sempre com o mesmo dialeto. Alguns suportam operações que outros não. Java é orientado a objetos e mapear as propriedades dos objetos para tabelas utilizando SQL é processo chato, demorado, passível de erro e que ninguém quer repetir em diferentes pontos da aplicação.

Finalmente, nem todas as aplicações comunicam apenas com banco de dados. Algumas podem comunicar com servidores LDAP ou serviços externos Business-to-Business (B2B), por exemplo. Esta comunicação é muitas vezes elaborada depois que a aplicação está em produção substituindo o banco como fonte de dados. As API para comunicação com estes sistemas são diferentes do JDBC quem em termos de interface quer em termos de filosofia.

O problema tem na realidades três perspectivas:

- Legado : Queremos encapsular lógicas de persistência legadas escritas com SQL ou outras tecnologias de forma simples. Queremos apenas utilizar as tecnologias java, mas manter as mesmas regras de negocio
- Isolamento : Queremos que a aplicação seja isolada da API com que está comunicando. Queremos poder alterar a API sem alterar a aplicação. No caso de JDBC queremos utilizar a mesma API para comunicar com diferentes gerenciadores de bando de dados sem alterar a aplicação.
- Mapeamento de Objetos : Qeremos utilizar objetos no ambiente Java. Queremos poder utilizar os mesmos objetos independentemente da API de persistencia.ORM (*Object-Relational Mapping*) – Mapeamento Objeto-Relacionamento é o mapeamento mais comum, mais outros mapeamentos são possiveis, como para XML, LDAP ou WebServices. Os mapeamentos ORM podem ainda mudar quando mudamos de gerenciador de banco de dados

A solução

O padrão **DAO** soluciona estes problemas de uma forma simples. Todas as comunicações com o mecanismo de persistência são mediadas por um objeto o DAO. Esse objeto mapeia informações transportadas em objetos (Tranfer Object) para instruções da API de persistência e mapeia resultados obtidos dela de volta para os mesmos objetos de transporte. Toda a lógica de mapeamento e execução das instruções é deixada dentro do objeto DAO desta forma isolando a aplicação da API de persistência por completo.

O objeto DAO é responsável por operar o mecanismo de persistência em nome da aplicação tipicamente executando os quatro tipos de operações – Criar, Recuperar , Alterar, Apagar – conhecidas pela sigla CRUD – do inglês *Create, Retrive, Update , Delete*. As operações de edição são invocadas diretamente passando o objeto com as informações a serem editadas.
As operações de recuperação são normalmente implementadas como métodos específicos. Por exemplo, recuperar o objeto que corresponde com uma certa chave, ou critério de busca. Estes métodos contém regras de negocio diferentes conforme o tipo de dados sendo persistido. Em particular pode não existir nenhuma regra particular, ou a regra só poder ser executada em um tipo especifico de API de persistência.

Interface

É interessante que o DAO seja especificado através de interface ao invés de uma classe concreta. Desta forma, não só facilitamos o trabalho de implementar novos mecanismos de persistência, mas também, impedimos que se criem referencias a classes concretas, diminuindo o acoplamento a um mecanismo em particular.

Se estamos utilizando diferentes DAO conforme o mecanismo de persistência que queremos usar, temos que ter alguma forma de escolher qual utilizar. Se você estiver utilizando algum *framework* de injeção de dependência (DI), basta configurá-lo para injetar a implementação certa. Outra opção (que não é incompatível com a anterior) é o uso do padrão Factory Estes padrões funcionam melhor se utilizarmos interfaces seguindo o Principio de Design por Contrato.

A interface do padrão original é simples. São fornecidos métodos para as operações CRUD, sendo que a operação de recuperação é distribuída em diferentes métodos de pesquisa especializados. São estes métodos especializados que encapsulam facilmente lógicas de pesquisa de sistemas legados.

```
01
02
03 public class Customer {
04 // atributos
05 // acessores e modificadores
06
07 }
08
09 public interface CustomerDAO {
10
11 Customer create () ;
12 void insert ( Customer c ) ;
13 void update ( Customer c ) ;
14 void delete ( Customer c ) ;
15 Customer findByID ( Integer id ) ;
16 Customer finfByCustomerNumber ( String customerNumber ) ;
17 }
18
19 public class JDBCCustomerDAO implements CustomerDAO {
20
21 public Customer create (){
22 return new Customer () ;
23 }
24 public void insert ( Customer c ){
25 // usa JDBC para criar e executar uma frase SQL de insersão.
26 }
27 public void update ( Customer c ){
28 // usa JDBC para criar e executar uma frase SQL de atualização.
29 }
30 public void delete ( Customer c ){
31 // usa JDBC para criar e executar uma frase SQL que remove o cliente do
banco
32 }
33 public Customer findByID ( Integer id ){
```

```
34 // usa JDBC para criar e executar uma frase SQL que pesquisa o cliente com a chave passada.
35 }
36 public Customer finfByCustomerNumber ( String customerNumber ){
37 // usa JDBC para criar e executar uma frase SQL que pesquisa o cliente com o numero passado.
38 }
39 }
```

Código 1:

O mesmo código teria que ser repetido para todos os tipos de objeto persistido.

Coloquei o método create() explicitamente na interface do DAO porque será importante mais à frente. Por agora ele simplesmente cria o objeto algo que poderíamos fazer facilmente fora do DAO. Esta utilização do padrão Factory Method parece fútil, mas como veremos depois é importante para alguns tipos especiais de implementação do padrão **DAO**.

Sabores de DAO

Existem vários “sabores” de DAO. A razão para isto é que o padrão DAO não é muito prático quando o sistema tem muitos objetos persistentes.

DAO padrão

Vimos como seria a implementação padrão[3] do **DAO**. Para cada tipo de objeto de transporte (TO) – Cliente, Pedido , etc… – existe um objeto DAO correspondente. Os objetos DAO formam uma camada na aplicação[1]. A aplicação tem que invocar o DAO certo para trabalhar com o objeto de transporte certo. Cada objeto DAO sabe como ler e popular as propriedades do TO e usá-las na API do mecanismo de persistência que está usando.

Neste sabor do padrão os objetos DAO formam um camada espessa recheada de lógica de mapeamento misturada com lógica de persistência e lógica de negócio. Além disso ele cria a necessidade de um conjunto muito grande de classes distribuídas no seguintes tipos:

- **TO** – Objetos de transporte. Eles são encarregados de manter os dados em memoria na forma de objetos e estruturas de objetos. Exemplo: Customer
- Interface **DAO** – Interface para o DAO de um certo tipo de TO. Exemplo: CustomerDAO
- Implementação **DAO** – Implementação da interface para o DAO de um certo tipo de TO e um certo tipo de API de persistência. Exemplo: JDBCCustomerDAO
- **Factory** de **DAO** – Fabrica que sabe qual implementação utilizar para cada interface. Na realidade a fabrica é uma metáfora para o mecanismo de mapeamento entre a interface e a implementação. Utilizando um mecanismo de DI teríamos de as mapear igualmente, apenas o faríamos de forma diferente.

Se o seu sistema tiver 10 tipos de objeto persistente (10 tabelas) você precisa escrever 30 classes só para começar.

DAO Genérico

Você pode estar pensando que usando tipos genéricos poderíamos diminuir a quantidade de interfaces e implementações necessárias. Na realidade isso não é bem verdade, porque as interfaces do DAO contêm métodos de procura específicos dependentes do objeto de transporte , das regras de persistência e de regras de negocio.

Usando tipos genéricos neste estágio não nos ajuda a diminuir o número de classes ou simplificar a implementação mas no ajuda a diminuir o numero de métodos por interface DAO utilizando interfaces comuns (padrão **Separated Interface** [1]).

```
01
02
03 public interface GenericDAO<T> {
04
05 T create () ;
06 void insert ( <T> obj ) ;
07 void update ( <T> obj ) ;
08 void delete ( <T> obj ) ;
09 T findByID ( Integer id ) ;
10
11 }
12
13 public interface CustomerDAO extends GenericDAO<Customer> {
14
```

```
15 Customer finfByCustomerNumber ( String customerNumber ) ;
16 }
```

Código 2:

Diminuimos a quantidade de métodos na interface DAO especifica de cada objeto de transporte, mas não ganhamos muito. Criamos mais uma classe e a implementação ainda continua específica.

DAO + Bridge

Até agora deixamos os objetos de transporte serem explicitamente classes. Isto é na realidade um problema. Todas as implementações do DAO para as várias tecnologias estão forçadas a utilizar o mesmo objeto. Se o objeto mudar (por exemplo, adicionarmos um campo) todas as implementações de DAO para as várias tecnologias têm que mudar. Para evitar este problema aplicamos o padrão Bridge deixando as implementações e as interfaces serem definidas independentes.Para isso utilizamos interfaces em vez de classes para especificar os nossos objetos de transporte[3].

Como estamos utilizando o padrão Factory Method para obter os objetos de transporte, podemos agora retornar uma implementação especifica da implementação do DAO. Com isto podemos fazer muitos tipos de otimização. Por exemplo, podemos incluir uma lógica que nos permite saber se os valores dos dados mudaram. Isso ajudará no método de atualização. Se não mudou nada simplesmente não faz nada e poupa a comunicação com o banco. Quando a implementação controla todos os fatores é fácil fazer otimizações. Esta técnica é utilizada pela própria API JDBC para desacoplar a aplicação Java do gerenciador de banco de dados permitindo que este faça as otimizações que achar necessárias.

DAO + BRIDGE + PROXY

Como não existe nenhuma outra lógica nos objetos de transporte que não seja ler e escrever os seus atributos podemos utilizar uma implementação genérica baseada num mapa atributo=valor utilizando o padrão Proxy. Basicamente utilizamos um Map e o encasuplamos dinamicamente na interface do objeto de transporte esperado. Isto é relativamente simples de fazer utilizando a classe java.lang.reflect.Proxy da API padrão do JSE.

A utilização do padrão **Bridge** não nos ajuda a diminuir classes, mas ajuda na implementação. Podemos ter controle total sobre como implementamos os objetos DAO e como comunicamos com a API de persistência. Isso permite que utilizemos otimizações, entre as quais o uso de **Proxy** para os objetos de transporte. Retirando efetivamente da aplicação o trabalho de os codificar e manter.

DAO + Metadados

Para reduzir o número de implementações necessárias, ou pelo menos diminuir a implementação de métodos comum é necessário termos a informação descrita de uma forma mais abstrata. Temos que utilizar metadados.

Existem dois tipos de metadados que podem ser usados, não necessariamente excludentes.
Podemos utilizar metadados existentes na própria estrutura das classes. Este uso é normalmente referido como Introspecção (*Introspection* é um tipo particular de *Reflection*) Por exemplo, não usar new na classe e usar os mecanismo de do Java para criar o objeto a partir da sua classe. (se utilizarmos o mecanismo de proxy descrito antes já estaremos fazendo isto implicitamente) Outro exemplo é utilizar introspecção para ler e escrever os atributos dos objetos dinamicamente invocando os métodos get/set dinamicamente. Para que o mecanismo de introspecção funcione a implementação do DAO precisa previamente saber a classe do TO.

Outra forma de metadados são aqueles relacionados à estrutura persistente do objeto. No caso de bancos de dados seriam os metadados das tabelas e seus relacionamentos.

Podemos deixar à responsabilidade da implementação do DAO descobrir e utilizar os metadados. Uma melhor opção é utilizar o padrão **MetadataMapper** deixando essa responsabilidade para outro objeto. Este objeto pode simultaneamente obter informações da estrutura de classes como ser configurado com informações extra sobre relacionamentos e tabelas.

Com metadados a implementação dos métodos básicos é comum a todos os tipos de objeto de transporte, simplificando a implementação do DAO. Essas implementações comuns podem ser encasupladas num classe pai de todos os DAOs. Os DAO específicos podem estender esta classe se necessário para prover mais mecanismos de busca, ou alterar os mecanismos padrão.

```
01
02
03 public class AbstractDAO<T> implements GenericDAO<T> {
```

```
04
05 MetadataProvider metadata;
06 public AbstractDAO ( MetadataProvider metadata ){
07 this .metadata = metadata;
08 }
09
10 public T create (){
11 return metadata.getTOClass () .newInstance () ;
12 }
13
14 public T findOne ( QueryObject<T> q ){
15 // traduz q para SQL usando os metadados
16 }
17
18 …
19 }
```

Código 3:

O número de classes diminuiu. Agora podemos utilizar sempre AbstractDAO apenas constituindo classes especificar quando existem métodos ou regras especificas a serem implementadas. A utilização de fabricas ou DI é vital para o sucesso desta abordagem porque podemos retornar uma instância de AbstractDAO devidamente configurada para um TO particular. Apenas em casos particulares precisaremos retornar alguma classes especial. E quando tivermos que fazer isso poderemos aproveitar a maioria dos métodos via herança.

DAO + QueryObject

Aquilo que separa ainda de implementações ainda mais genéricas para o padrão DAO são os métodos especiais de pesquisa. Criar um método para cada estratégia de pesquisa aumenta rapidamente a interface do DAO. Isso é mau pois estamos aumentando a responsabilidade dos implementadores da interface. Isso obriga a que todos os DAO de todos os mecanismos implementem esses métodos. Inadevertidamente podemos criar métodos que certos tipos de persistencia não podem executar. Mesmo com o padrão Bridge, isso pode ser complicado de gerenciar.

Para solucionar isso podemos invocar o Principio de Separação de Responsabilidade (SoC) e abstrair a logica de pesquisa num objeto à parte.

Estas várias estratégias de pesquisa podem ser utilizadas por DAO de diferentes tipos e até mesmo para diferentes mecanismos de persistencia.

Em particular podemos criar implementações de Query Object que permitam as operações mais comuns. Deixando para os métodos especiais apenas aquelas que não podem ser traduzidas pelo **Query Object**. Sendo que SQL é uma linguagem de pesquisa genérica é normalmente possível explicitar a maioria das opções utilizando uma implementação de **Query Object** O numero de métodos específicos que ainda teriam que ser criados nas interfaces especificas dos DAO diminuem na razão inversa do poder de abstração da sua implementação de **Query Object**.Ou seja, quanto mais poder usar **Query Object** menos usará métodos específicos.

```
01
02
03 public interface GenericDAO<T>> {
04
05 T create () ;
06 void insert ( <T> obj ) ;
07 void update ( <T> obj ) ;
08 void delete ( <T> obj ) ;
09 List<T> findAll ( QueryObject<T> criteria ) ;
10 T findOne ( QueryObject<T> criteria ) ;
11 }
12
13 DAO Genérico com QueryObject
```

Código 4: DAO Genérico com QueryObject

A utilização destes objetos de pesquisa pode facilmente distribuir logicas de pesquisa por toda a aplicação. Para que isso não acontece você pode utilizar o padrão Repository que mantém as pesquisa num só lugar.

O uso de **Query Object** simplifica a interface do DAO diminuindo o numero de métodos de pesquisa especificos.

Fonte:

https://sergiotaborda.wordpress.com/desenvolvimento-de-software/java/patterns/dao/ acessado em 04/10/2018