








# A factory for the package rostDAO

Ramsès TALLA  
 Fachbereich Informatik  
 Technische Universität Darmstadt  
 Karolinenplatz 5, 64289 Darmstadt, Germany  
 ramses.talla@stud.tu-darmstadt.de

**Abstract:** The purpose this paper is devoted to is hiding and abstracting concrete classes as well as some related implementations designed in the last package rostDAO.

**Keywords:** Pattern DAO, Pattern Factory, Pattern Decorator, Abstraction.

## Legends and notations:

- **The plus sign (+):** corresponds to the level of visibility “public”
- **The Minus sign (-):** corresponds to the level of visibility “private”
- **The Tilde (~):** corresponds to the level of visibility “protected”
- **The hashtag (#):** corresponds to the level of visibility “package-private”
-  : public interfaces
-  : package-private abstract classes
-  : package-private abstract classes
-  : public classes
-  : package-private classes
-  : inheritance
-  : implementation

- 1) *The naming convention in the whole project is the standard Java naming convention !*
- 2) *CRUDL methods remains unchanged and are the same as in the last package. They are somewhere omitted on UML diagrams in this paper for the sake of brevity.*

## I. INTRODUCTION

In [1], we recently sketched a strong class architecture to build DAO classes to manage data independently of their storage platform, being `SQL`, `XML`, `CSV`, just to name a few. The classes introduced in that package already fulfil all requirements and expectations of developers.

But one may easily notice that using all these classes may be confusing. The main problems arising in the following situations :

- There were so many classes and each class had two constructors. The second constructor of each class was only a call of the first with some arguments. This was so a duplication of the structure, which we promised to avoid.
- Direct instantiations of the classes `DAO_SQL`, `DAO_CSV`, `DAO_XML`, and `AbstractDAO` do not require the end developer to utilize abstraction or polymorphism by using the general DAO interface as the reference type for their DAO objects.
- One may bear in mind that the DAO pattern also achieves the goal of making the data source interchangeable. This is done by using the polymorphism and more precisely the covariance of objects.

- The DAO interface had two generic types : one standing for the type of the unique identifiers and the other one representing the JavaBean of the entity. But it seems easy to show that not all entities must always be managed with IDs or primary keys. So, the end developer had to specify two generic types, whatever his entities are managed by IDs or not. And this may cause some contradictions in the logic layer.

The purpose of this work is to new-design our recent classes, so that they remains coherent and abstract in all situations, by following the guidelines provided by the three above mentioned issues.

## II. SOME MODIFICATIONS APPLIED ON THE LAST PACKAGE

Modifications applied here include:

- The classes `DAO_SQL`, `DAO_CSV`, `DAO_XML` and `AbstractDAO` are yet **package-private**. This means there can neither be instantiated nor accessed outside the package. But since instances of these classes are necessary to perform data managing, instances of these classes are got through the class `DAOFactory` which return them by hiding their concrete reference types. More explanations in further lines. Turning these classes into package-private allows us to design them in the manner we want.
- All classes have yet exactly **one and only one constructor**. A variable `checkID` is there to provide whether

the DAO class will be managed by identifiers or not.

- The class `DAO0` does not hold the method `getIDsInStringFormat()` anymore since the `String` format of identifiers could easily be obtained through streams and the related `map` function.
- The new method `getProperty(Supplier<R> supp, String propertyName): List<R>` returns a list containing a fixed property of all entities lying on the data storage. One may easily bear in mind that this method has been implemented differently in all DAO classes since implementing it through the `map` function of the class `Stream` after getting all objects represents much more time cost. This implementation is then the default and has been added to the `AbstractDAO` class.
- The interface `DAOPredicate` has been removed and replaced by the standard `Predicate` interface lying in the package `java.util.function`. In fact, this second work is devoted to reducing the lines of code and making the hierarchy as short as possible. Since `DAOPredicate` relayed the thrown exceptions to the runtime, implementing methods in bodies of `test(T obj)` is hence more tedious, since catching all exceptions is impossible to achieve. Instead of trying to catch the being thrown exceptions, we simply throw them during the runtime in the following manner :

```

public <R> List<R> getProperty(Supplier<R> supp, String propertyName) throws Exception {
    check_property_names(propertyName); // to avoid TOCTTOU and SQL injections

    Method method =
objClass.getDeclaredMethod(getter(objClass.getDeclaredField(propertyName)));

    List<R> list = getAll()
        .stream()
        .map(obj -> {
            try {
                return (R) method.invoke(obj);
            } catch (Exception e) {
                throw new RuntimeException(e);
            }
        })
        .toList();

    return new RostArrayList<R>(list.size(), list.toArray(), false);
}

public List<J> getByPattern(String property, String regex) throws Exception {
    check_property_names(property); /* checks if the name of the identifiers is set.
Very meaningful by preventing injections for example*/

    return get(obj -> {
        try {
            return
String.valueOf(objClass.getDeclaredMethod(getter(property)).invoke(obj)).matches(regex);
        } catch (Exception e) {
            throw new RuntimeException(e);
        }
    });
}

```

As shown by the code above, every method throwing an exception that is called in a context that does not take exceptions into account (lambdas of standard functional interfaces like `Predicate`, `Function`, ...) will see its exceptions converted into instances of `RuntimeException` so that they can still be thrown as far as they arise. The main advantage behind this is that the end developer (the user of these classes) will receive all exceptions and threat them himself since we don't take care of them here. In fact, handling these exceptions here will lead to hiding them to the end developer and he will

never the main reasons behind the failures in his potential tests.

- The generics parameters `K` and `T` have been renamed in codes into `I` and `J` respectively, standing respectively for initials of the terms “identifiers” and “JavaBean”. This makes the functional wise of the DAO classes more readable.
- The two methods `int add(Iterable<J> list)` and `int delete(Iterable<J> list)` have been added to the general DAO interface. These methods store or delete many records provided as an implementation of the interface

Iterable being for example  
Collections (List, Set, ...)

- Figure 1 shows the diagram of the classes we recently obtained in the last package. In this new work, in order to return an instance of a DAO class handling with identifiers or not, the general DAO interface has been split in two as shown by the Figure 2. Since the two new interfaces hold the same name, they have been placed in two different packages namely `rost.dao.base` and `rost.dao.extended`. For the sake of brevity, we will call in the rest of this paper “*base DAO*” the DAO lying in the package `rost.dao.base` and “*extended DAO*” the DAO being in the another one.

```
private static <I, J> DAO<I, J> getDAO(Supplier<J> instanceSupplier, Supplier<I> ID_Supplier,
Supplier<List<J>> retriever, Function<Iterable<J>, Integer> persistenceFunction,
Function<Iterable<J>, Integer> removingFunction, String nameOfTheID, boolean checkID) throws
Exception {

    return new AbstractDAO<I, J>(instanceSupplier, nameOfTheID, checkID) {

        @Override
        public List<J> getAll() throws Exception {
            return retriever.get();
        }

        @Override
        public int add(Iterable<J> list) throws Exception {
            return persistenceFunction.apply(list);
        }

        @Override
        public int delete(Iterable<J> list) throws Exception {
            return removingFunction.apply(list);
        }

    };
}
```

This method returns a DAO instance and takes seven parameters whose roles could be easily deduced from their names. The parameter `ID_Supplier` does not appear in the method implementation and is only there to infer the type of identifiers managed by the DAO.

- The abstract class `AbstractDAO` is yet an inner class of the package `rostDAO`. In other words, this class is no longer a public class and can not be accessed outside the package. This have been made is other to stay coherent with our first purpose of hiding all implementations to the end user of the user. But it has been shown in our last work, that this class is very important when reducing the number of implementations of storage platforms. Implementations of these class will be returned as follow in a class we will present further :

`persistenceFuntion` and `removingFunction` are implementations of the interface `java.util.function.function`. They take many beans (instances of a `JavaBean`) and return the number of records matching those beans they store to or respectively remove from the data storage.



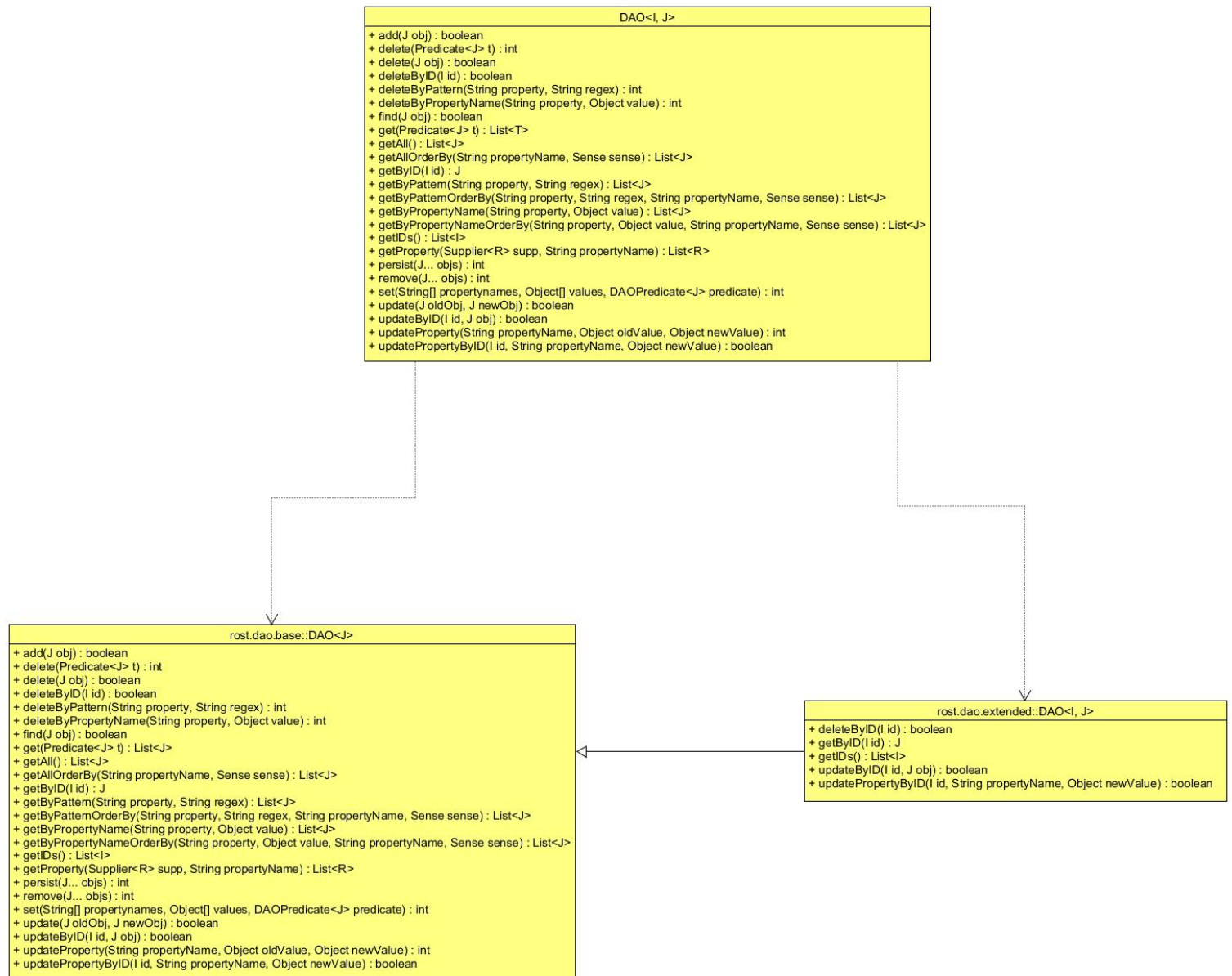


Figure 2 : Splitting the general DAO interface into two

### III. THE PATTERN DECORATOR AND THE CLASS DAO\_BASE

According to the purpose of returning two general types of DAO classes depending on whether identifiers will be managed or not, one may easily think about splitting the last classes into two. Since we had 5 five classes namely `DAO0`, `AbstractDAO`, `DAO_CSV`, `DAO_XML` and `DAO_SQL`, the resulting classes can be named with the terminology -base (for DAO classes implementing the base DAO) or -extended (for classes implementing the extended DAO) leading to a final result of 10 classes. And 10 tclasses without taking other

utility classes into account is very much. But that is not the real problem. The real problem is that implementations lying in “*base DAO classes*” will be repeated in the “*extended DAO classes*”.

Trying to avoid repetitions in the codes, one would think about inheriting the “*extended DAO classes*” from the “*base DAO classes*”. But the problem is that the returned type of these classes will be the base DAO interface instead of the extended one. And that is not want we want. Now one would think about adding the signature “*implements rost.dao.extended.DAO*” in the class



declaration, but this would lead to the error :  
*“The hierarchy of the type A is inconsistent”*.

The situation we are describing is resumed by the following UML diagram, where the components of the classes have been omitted for the sake of brevity :

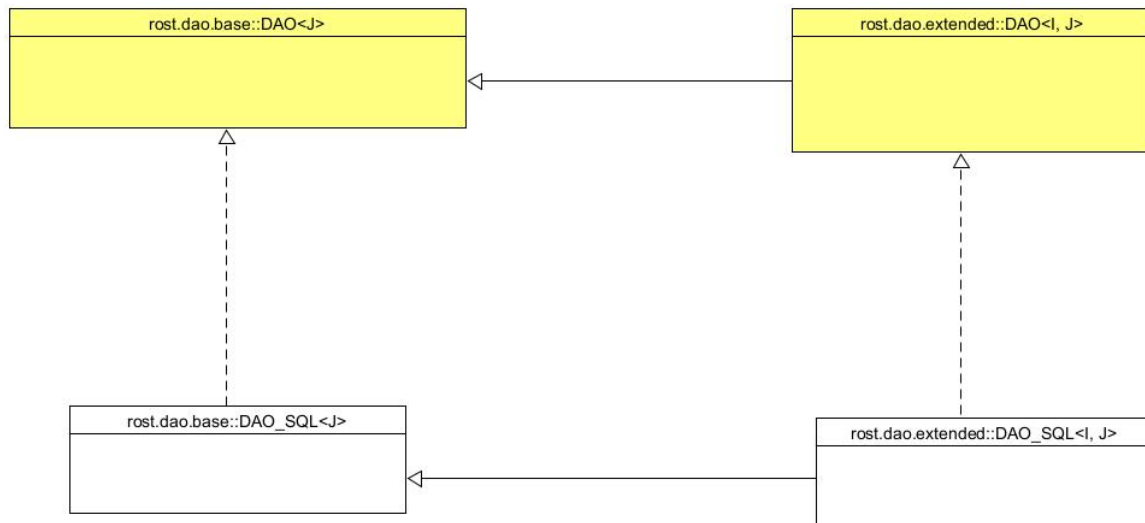


Figure 3 : UML diagram of classes leading to a hierarchy’s inconsistency

So instead of splitting our recent classes, we will keep the structure of the last package and add another class devoted to the building of instances of the “base DAO” : the class `DAO_Base`. The code of this class is rather relatively simple to understand

```
package rost.dao.extended;

import java.util.List;
import java.util.function.Predicate;
import java.util.function.Supplier;

import rost.dao.base.Sense;

@SuppressWarnings("unchecked")
class DAO_Base<J> implements
rost.dao.base.DAO<J> {

    private rost.dao.extended.DAO<?, J>
dao = null;

    DAO_Base(rost.dao.extended.DAO<?, J>
dao) {
        this.dao = dao;
    }

    @Override
    public final int add(Iterable<J> list)
throws Exception {
        return dao.add(list);
    }

    @Override
```

```
    public final boolean add(J obj)
throws Exception {
        return dao.add(obj);
    }

    @Override
    public final int add(J... objs)
throws Exception {
        return dao.add(objs);
    }

    @Override
    public final int delete(Iterable<J>
list) throws Exception {
        return dao.delete(list);
    }

    \\ implements the rest of the methods
here in the same manner
}
```

This package-private class implements the interface `rost.dao.base.DAO` (the base DAO) and has a constructor taking an instance of the interface `rost.dao.extended.DAO` (the base DAO). All methods of the “base DAO” are yet implemented while calling the same methods on the object `dao` of type “extended DAO”. One will notice that the generic parameter `I` has been replaced by the wildcard, since it does not matter anymore. The code of this class could be refined further using the Lombok library. Using Lombok

in order to generate our class yields the following class :

```
package rost.dao.extended;

import lombok.experimental.Delegate;
import lombok.RequiredArgsConstructor;
import rost.dao.base.Sense;

import java.util.List;
import java.util.function.Predicate;
import java.util.function.Supplier;

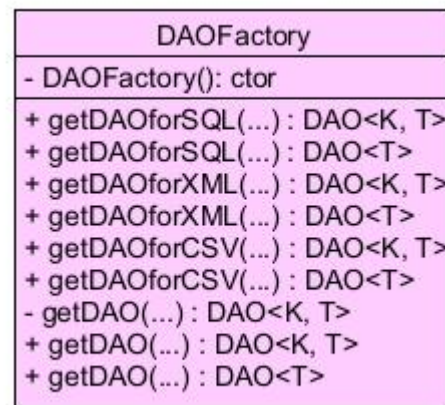
@RequiredArgsConstructor
@SuppressWarnings("unchecked")
class DAO_Base<J> implements
rost.dao.base.DAO<J> {

    @Delegate
    private final rost.dao.extended.DAO<?,
J> dao;
}
```

In this code, Lombok takes care of writing during the compile time, all implementations of the base DAO Interface, by calling the same methods on the object `dao`, without adding or removing anything. This class is much simpler as far as it helps to implement automatically methods in the class `DAO_Base` as soon as new methods appear in the base DAO interface.

#### IV. THE PATTERN FACTORY AND THE CLASS DAOFactory

Since the classes generating instances of DAO classes are all package-private, a class called `DAOFactory` is devoted to this task and takes care of returning the instances with reference types `rost.dao.base.DAO` or `rost.dao.extended.DAO` following the diagram below :



Fi

Figure 4 : UML Diagram of the class

DAOFactory

No need to specify again that the DAO interfaces with one and two generic parameters are respectively the base and the extended one. The private method `getDAO(...)` help to factorize the code of the two other methods holding the same name. And follows the code :

```
package rost.dao.extended;

import java.sql.Connection;
import java.util.List;
import java.util.function.Function;
import java.util.function.Supplier;

import rost.dao.base.Format;

public final class DAOFactory {

    /* Don't let anyone instantiate this
    class ! */
    private DAOFactory() {}

    public static <I, J> DAO<I, J>
getDAOforSQL(Supplier<J> entitySupplier,
Supplier<I> ID_Supplier, Connection connect,
String nameOfTheID) throws Exception {

        return new DAO_SQL<I,
J>(entitySupplier, connect, nameOfTheID,
true);
    }

    public static <J> rost.dao.base.DAO<J>
getDAOforSQL(Supplier<J> entitySupplier,
Connection connect) throws Exception {

        return new DAO_Base<J>(new
DAO_SQL<Object, J>(entitySupplier, connect,
null, false));
    }
}
```



```

    public static <I, J> DAO<I, J>
getDAOforXML(Supplier<J> entitySupplier,
Supplier<I> ID_Supplier, String filePath,
Format format, String nameOfTheID) throws
Exception {

    return new DAO_XML<I,
J>(entitySupplier, filePath, format,
nameOfTheID, true);
}

    public static <J> rost.dao.base.DAO<J>
getDAOforXML(Supplier<J> entitySupplier,
String filePath, Format format) throws
Exception {

    return new DAO_Base<J>(new
DAO_XML<Object, J>(entitySupplier, filePath,
format, null, false));
}

    public static <I, J> DAO<I, J>
getDAOforCSV(Supplier<J> entitySupplier,
Supplier<I> ID_Supplier, String filePath,
String nameOfTheID, char separator) throws
Exception {

    return new DAO_CSV<I,
J>(entitySupplier, filePath, nameOfTheID,
true, separator);
}

    public static <J> rost.dao.base.DAO<J>
getDAOforCSV(Supplier<J> entitySupplier,
String filePath, char separator) throws
Exception {

    return new DAO_Base<J>(new
DAO_CSV<Object, J>(entitySupplier, filePath,
null, false, separator));
}

    private static <I, J> DAO<I, J>
getDAO(Supplier<J> instanceSupplier,
Supplier<I> ID_Supplier, Supplier<List<J>>
retriever, Function<Iterable<J>, Integer>
persistenceFunction, Function<Iterable<J>,
Integer> removingFunction, String
nameOfTheID, boolean checkID) throws
Exception {

    return new AbstractDAO<I,
J>(instanceSupplier, nameOfTheID, checkID) {

        @Override
        public List<J> getAll() throws
Exception {

            return retriever.get();
        }

        @Override

```

```

        public int add(Iterable<J> list)
throws Exception {

            return
persistenceFunction.apply(list);
        }

        @Override
        public int delete(Iterable<J>
list) throws Exception {

            return
removingFunction.apply(list);
        }
    };
}

    public static <I, J> DAO<I, J>
getDAO(Supplier<J> instanceSupplier,
Supplier<I> ID_Supplier, Supplier<List<J>>
retriever, Function<Iterable<J>, Integer>
persistenceFunction, Function<Iterable<J>,
Integer> removingFunction, String
nameOfTheID) throws Exception {

    return getDAO(instanceSupplier,
ID_Supplier, retriever, persistenceFunction,
removingFunction, nameOfTheID, true);
}

    public static <J> rost.dao.base.DAO<J>
getDAO(Supplier<J> instanceSupplier,
Supplier<List<J>> retriever,
Function<Iterable<J>, Integer>
persistenceFunction, Function<Iterable<J>,
Integer> removingFunction) throws Exception
{

    return new
DAO_Base<J>(getDAO(instanceSupplier,
Object::new, retriever, persistenceFunction,
removingFunction, null, false));
}
}

```

On the above code, one will notice that instances of the base DAO are generated from the DAO\_Base class.

## V. REFERENCES

- [1] Ramses TALLA, *A new package for the DAO Pattern in Java*, 2024