A new package for the DAO pattern in Java

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

Abstract: In data management, the DAO (Data Access Objects) pattern is well-famous as a strong architecture to design classes and objects that easily permits the persistence and the manipulation of data wherever there are from. But this pattern is much more ill repute to be very tedious to implement since it requires a very huge quantity of codes to perform all functionalities as well. Implementing this pattern in Java may be frustrating if someone isn't used to build lots of codes. Furthermore, the quality of data must be ensured in order to avoid some awkward situations like hacks or loss of data. This paper is devoted to the introduction of a new package, a set of Java classes which allow the end developer to not write any line of code related to the data management anymore at all. The classes of the pattern enclosed by this package already hold all implementations, so that the end developer will only need to instantiate them rather than designing his own classes which represents much time lost.

Keywords: DAO, security, design pattern, entities, records, JDBC, DOM, business objects, time complexity of a program, generics, JavaBeans, inheritance, implementation.

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"
- 🟨 : interfaces
- 🟧 : package-private abstract classes
- 🟩 : public abstract classes
- ⬜ : public classes
- ⟶ : inheritance
- ⇢ : implementation

The naming convention in the whole project is the standard Java naming convention !

I. INTRODUCTION

Software systems reliant on external data inevitably require storing new, additional data or retrieving existing data already saved in storage. Reading, storing, deleting, and modifying data in the business logic of an application can often be implemented clearly and coherently using the DAO (Data Access Object) design pattern. The primary strengths of this pattern lie in making data access logic more comprehensible, simplifying future changes to the storage format and enabling independence between business objects and the data layer. However, every software developer who has worked with this pattern knows that its use in a project requires a substantial amount of source code, significant time for testing, and numerous classes. This aspect of the DAO pattern contradicts the fundamental principle of avoiding code and structural duplication, especially in the case that one has many entities. In fact, the main problem of this pattern lies in the obligation to code a new different class as soon as one has a new entity to manage in the project. It will be then easy to remark that all these classes obey the same structure and duplication is the first enemy of each programmer.

The article *"DAO Dispatcher Pattern: A Robust Design of the Data Access Layer"* [1] served as a practical starting point for developing the final classes. In their article, authors Pavel Micka and Zdenek Kouba defined two primary classes, established several key principles for a successful and well-structured project, and described the functioning of their approach. Their work already addresses the concerns of developers seeking to create shorter and well-structured classes, which aligns with the ultimate goal of **rostDAO**. In essence, their work provides a strong foundation for effective data management modelling in Java projects. While their previous work shares similar objectives, functions, and principles with this new package, it differs significantly in four fundamental aspects:

- **rostDAO** primarily focuses on the security and quality of data [2]. Additionally, the speed of data processing is optimized to reduce their runtimes [1].

- **rostDAO** includes more access methods, known as "CRUDL methods". It offers a variety of access methods, known as CRUDL methods (Create, Read, Update, Delete, List). A total of 23 such methods are available, providing efficient and flexible data handling.

- **rostDAO** allows the end developer to utilize multiple storage platforms. In fact, with rostDAO, an end developer can easily integrate SQL databases, CSV files, XML files, and other storage options into their final project.

- **rostDAO** is not considered a final package, and anyone can derive or shape their own classes and methods from it as they wish. It allows developers to create their own classes and methods or modify existing ones to meet the specific needs of their projects.

II. PRINCIPLES APPLIED DURING THE WHOLE CONCEPTION OF THIS PROJECT

- **You are not going to need it:** In this project, only the essential and main solutions to problems were written. Variables that are only called once—except for passed parameters—were strictly avoided to save memory space, as a large amount of data needs to be managed.

- **Never trust user inputs:** Cybercrime and security issues in large software systems often arise from user inputs. Since verifying user inputs also takes time during design and testing, this point is often overlooked by many developers. In our project, user inputs were carefully validated, even if it resulted in delays in our programs.

- **Do not repeat yourself:** One of the fundamental rules in programming is the avoidance of code and structural duplication. Clearer and cleaner source code can be produced by identifying and removing repetitive code snippets. This principle is strongly applied in this project, not only because it is highly recommended but also because the package is still under development, and future changes or additions are likely to occur.

- **Security:** Every user data input in the rostDAO package are tested everywhere before being passed to their respective methods. The method `check_property_name(String name)` from the `DAO0` class is used to check whether the name of a property (case-sensitive) is properly declared in a JavaBean. For executing SQL statements, the `PreparedStatement` object is used instead of `Statement` to prevent SQL injections, as it handles query data and user inputs securely. To prevent TOCTTOU (Time-of-Check to Time-of-Use) issues, sensitive data is either hidden or constructed in an immutable way. For this reason, the CRUDL methods in this package only return immutable lists.

- **Testability:** The classes in the package are eventually testable independently of other software components. This allows for efficient and quick fixes in case of bugs or incompleteness.

- **Speed:** Since the package already allows for a strict software structure and handles data access, care is also taken to ensure that the code runs with minimal runtime. Therefore, fast algorithms are preferred, even though this sometimes leads to an increase in the overall size of the package.

- **Encapsulation and Factoring:** Multiple algorithms that require the same sub algorithm lead to the creation of methods to clarify the logic and also adhere to the "Don't repeat yourself" principle. This helps to hide the actual implementations and places more emphasis on understanding the logic.

III. THE CLASSES OF THE PACKAGE rostDAO

First and foremost, rostDAO is not a new additional design pattern for solving data management in software development. Rather, it is a small package that efficiently simplifies the use of the already well-known DAO design pattern.

a) Classic and fundamental approach

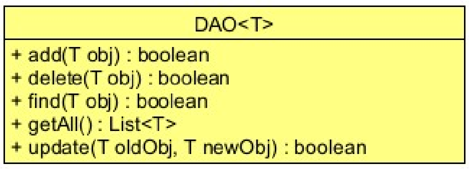
The DAO pattern, also known as Data Access Object, is a design pattern for data management that facilitates communication between business logic and the data stored in the storage medium by handling the storage of additional data and reading existing data. Typically, code related to data access is directly integrated into the business logic. However, this can lead to several issues:

- Changing the storage medium becomes very difficult in the future (e.g., from XML files to SQL databases).

- It will be impossible to test the business logic without also running the data access layer. Conversely, it will not be possible to test the data access code without running the business logic. This makes bug fixing particularly difficult and can result in significant time expenditure.

- The source code clearly reveals which storage model (SQL databases, CSV files, XML files, JSON, YAML, etc.) is being used. This can, for instance, make the job easier for a potential hacker, who would already know where to search for data.

To avoid these issues as much as possible, the use of the DAO pattern is recommended. Each entity in the project is represented by a JavaBean, and for every entity whose records need to be inserted, deleted, read, or modified externally in the storage, a DAO class is defined. This DAO class is called whenever data related to that entity needs to be accessed. A typical DAO class can be summarized on the following diagram:



Picture 1: UML Diagram of the general DAO interface

The generic parameter `T` represents the JavaBean of the entity, which defines data being handled. By way of reminder, a JavaBean is a Java class which follows these standard conventions :

- A JavaBean must have a **public zero-parameter** constructor, which makes the dynamic instantiation (the instantiation without the operator `new`) possible.

- All attributes of a JavaBean must be declared `private`, in order to ensure that the attributes are correctly and securely managed.

- Each attribute of a JavaBean must be associated with a getter and a setter, to retrieve respectively update its value. A getter is a void and zero parameter method that returns the value of the attribute. The name of each getter respectively setter begins with the prefix “get” respectively “set” and continues with the name of the attribute where the first letter of the attribute is capitalized. If one has for example the attribute “name”, one has the getter “getName” and the setter “setName”.

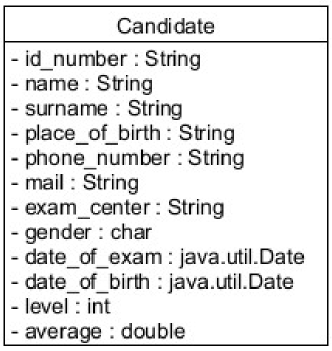
- A JavaBean must implement the interface Serializable, which ensures tat its states can be saved and retrieved later for treatments. Since the DAO pattern stores attribute’s values on data source, this convention can be omitted in this special case.

A typical example of a JavaBean may be this one :

```
package dao.beans;

public class Person {

        private String name;

        private String surname;

        private int age;

        public String getName() {
                return name;
        }

        public void setName(String name) {
                this.name = name;
        }

        public String getSurname() {
                return surname;
        }

        public void setSurname(String surname) {
                this.surname = surname;
        }

        public int getAge() {
                return age;
        }

        public void setAge(int age) {
                this.age = age;
        }
}
```

The user of these classes should bear in mind that a JavaBean which does not follow them will be the source of some being thrown exceptions. The five methods in this class diagram are commonly referred to by the acronym CRUDL. CRUDL stands for Create, Read, Update, Delete, List.

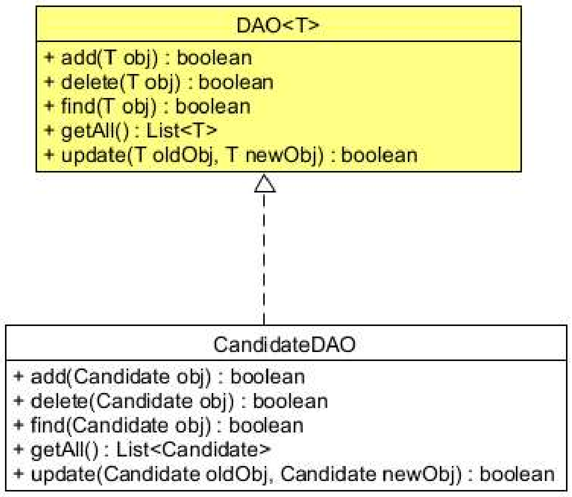
These five methods are also called access methods. Assume now, we are developing a project and have the following entity class, where the getter and setter methods of the JavaBean have been omitted for brevity:



<u>Picture 2:</u> UML diagram of the entity Candidate“

The construction of the corresponding DAO class by implementing the above-mentioned DAO interface results in the CandidateDAO class, as shown in Figure 3. Suppose there is already an SQL table in the database, and now the task is to manage the personal data of candidates for an exam (delete, insert, update) through the CandidateDAO class. This task can be easily performed in the business objects by calling these five CRUDL methods. The only problem lies in the actual implementation of these access methods. All fields of the `Candidate` JavaBean -12 in total- must be correctly considered during object-relational mapping. Furthermore, the security of the passed data must be carefully handled through the `PreparedStatement` object. When considering just these two aspects in the implementation of the `CandidateDAO` class, it becomes clear that the source code for the entire corresponding DAO class consists of more than 150 lines and has at least 100 individual steps.

If there are more entities and, by extension, more JavaBeans in the project, the same number of DAO classes must be created. This implies, having


DAO<T>
+ add(T obj) : boolean
+ delete(T obj) : boolean
+ find(T obj) : boolean
+ getAll() : List<T>
+ update(T oldObj, T newObj) : boolean
CandidateDAO
+ add(Candidate obj) : boolean
+ delete(Candidate obj) : boolean
+ find(Candidate obj) : boolean
+ getAll() : List<Candidate>
+ update(Candidate oldObj, Candidate newObj) : boolean

Picture 3: *UML diagram of the CandidateDAO class*

20 JavaBeans forces to code 20 DAO classes. The problem here arises because the general structure of object-relational mappings and security measures is duplicated in each DAO class. And duplication contradicts the fundamental principles of all developers. It is also important to note that the storage model may change for unknown reasons from SQL to CSV, JSON, XML, or something else. The same concern about the risks of duplication applies to these above-mentioned storage models, once at least two DAO classes share the same storage format.

The aim and core purpose of our work is to introduce a new Java package named *rostDAO* which factorizes the code of the general structure of DAO implementations depending on the storage format taken into account. Traditionally, each entity or JavaBean would correspond to at least one DAO class. But this package ensures that each entity or JavaBean will correspond to instances of classes already holding strong implementations for data management.

b) Presentation of the core classes of rostDAO and their connections one another

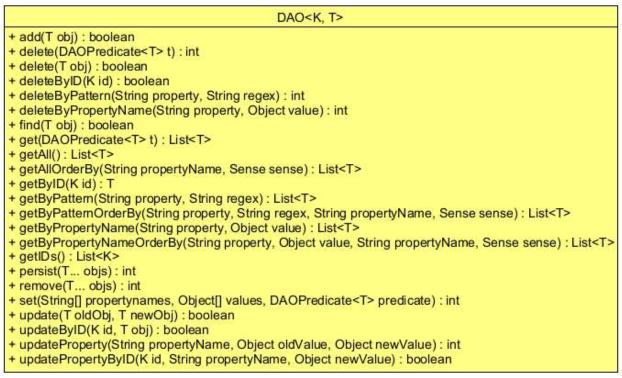
Note: In this section, access methods on some UML diagrams have been omitted for the sake of brevity and simplification!

As previously mentioned, the main disadvantages of the DAO design pattern are that it sometimes forces the repetition of the same structure across classes and that object-relational mapping becomes increasingly challenging as the corresponding JavaBean contains more attributes. Furthermore, the DAO class must be consequently updated as soon as the number of attributes in JavaBeans changes independently on whether it increased or decreased. These two issues also often explain why security measures between code layers were not given significant attention.

The rostDAO package is not intended as a new design pattern to replace DAO. Rather, it represents a collection of classes that addresses the disadvantages of the classic DAO pattern. It is clear and evident that not all disadvantages can be resolved, but a significant portion of them appear solvable through this package. Following lines presents the core classes of the package.

1. The DAO interface

All DAO classes must implement this general DAO interface. A full documentation of its methods is provided in the Java file of the class. This interface does not hold any `default` method and therefore acts rather than a factorizing class, as a signature of all DAO classes derived from this package.



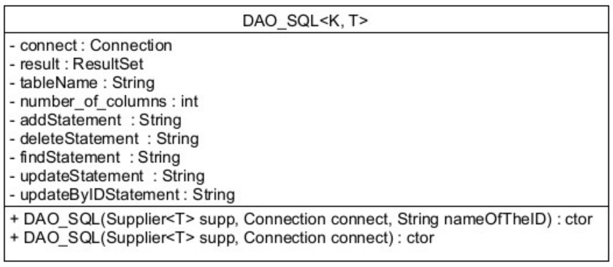
```
DAO<K, T>
+ add(T obj) : boolean
+ delete(DAOPredicate<T> t) : int
+ delete(T obj) : boolean
+ deleteByID(K id) : boolean
+ deleteByPattern(String property, String regex) : int
+ deleteByPropertyName(String property, Object value) : int
+ find(T obj) : boolean
+ get(DAOPredicate<T> t) : List<T>
+ getAll() : List<T>
+ getAllOrderBy(String propertyName, Sense sense) : List<T>
+ getByID(K id) : T
+ getByPattern(String property, String regex) : List<T>
+ getByPatternOrderBy(String property, String regex, String propertyName, Sense sense) : List<T>
+ getByPropertyName(String property, Object value) : List<T>
+ getByPropertyNameOrderBy(String property, Object value, String propertyName, Sense sense) : List<T>
+ getIDs() : List<K>
+ persist(T... objs) : int
+ remove(T... objs) : int
+ set(String[] propertynames, Object[] values, DAOPredicate<T> predicate) : int
+ update(T oldObj, T newObj) : boolean
+ updateByID(K id, T obj) : boolean
+ updateProperty(String propertyName, Object oldValue, Object newValue) : int
+ updatePropertyByID(K id, String propertyName, Object newValue) : boolean
```

Figure 4: UML diagram of the interface DAO

The generic parameter `K` represents the type of the unique keys of entities, also referred to as IDs. This could be a number, a string, or any simple type that uniquely identifies entities. The generic parameter `T` represents the corresponding JavaBean class being managed. More commands are frequently required and used. For this reason, additional access methods have been introduced in the general DAO interface, bringing the total to 23 access methods. While five access methods were previously difficult to implement, 18 more methods to implement now seems unthinkable!

However, there is no need to worry. Implementations of these 23 access methods can be found in the `DAO0`, `DAO_SQL`, `DAO_CSV`, `DAO_XML`, and `AbstractDAO` classes.

2. The DAO_SQL class



```
DAO_SQL<K, T>
- connect : Connection
- result : ResultSet
- tableName : String
- number_of_columns : int
- addStatement  : String
- deleteStatement  : String
- findStatement  : String
- updateStatement  : String
- updateByIDStatement : String
+ DAO_SQL(Supplier<T> supp, Connection connect, String nameOfTheID) : ctor
+ DAO_SQL(Supplier<T> supp, Connection connect) : ctor
```

Figure 5: UML diagram of the class DAO_SQL

The `DAO_SQL` class is the corresponding class for constructing DAO classes whose entities are stored in SQL databases. The constructors are provided with a `Connection` object through which SQL commands are executed.

With this class, developers no longer need to build a fully defined and standalone implementation for their DAO classes. These can, for example, simply be understood as follows through instantiations:

```java
DAO<String, Candidate> candidateDAO = new DAO_SQL<
>(Candidate::new,  getConnection(),  "id_number");
```

Example 1.1

It may be unconceivable to say that only this single instantiation is equivalent to the whole `CandidateDAO` class coded in the previous section. But it is strictly the same. Better. The instantiated object hold more methods than the previous DAO class.. The access methods can then be invoked and executed through the `candidateDAO` object, as if one had previously implemented an entire class themselves. Assume that the `getConnection()` method establishes a connection between the SQL server and the software and returns it. However, a short class must be derived from the `DAO_SQL` class for this purpose.

```java
    package dao.implementations.sql;

    import java.sql.Connection;

    import beans.Candidate;

    import rost.dao.base.DAO_SQL;

    public     class     CandidateDAO     extends
DAO_SQL<String, Candidate> {

       public    CandidateDAO(Connection    conn)
throws Exception {

        super(Candidate::new, conn, "id_number");

       }
    }
```

Example 1.2

In this second example, more source code is required. However, the advantage is that the constructor of this derived class allows other

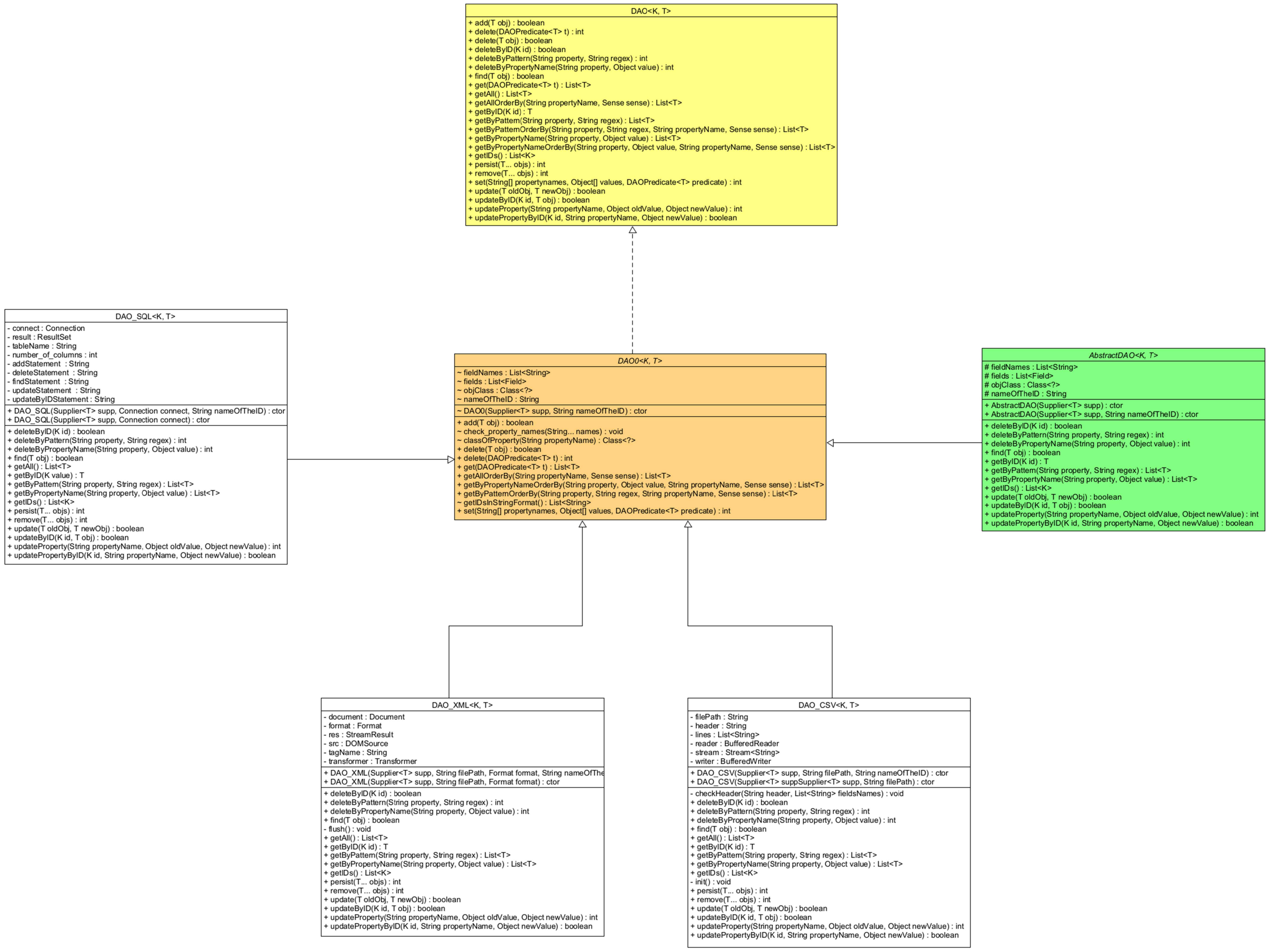

DAO<K, T>
+ add(T obj) : boolean
+ delete(DAOPredicate<T> t) : int
+ delete(T obj) : boolean
+ deleteByID(K id) : boolean
+ deleteByPattern(String property, String regex) : int
+ deleteByPropertyName(String property, Object value) : int
+ find(T obj) : boolean
+ get(DAOPredicate<T> t) : List<T>
+ getAll() : List<T>
+ getAllOrderBy(String propertyName, Sense sense) : List<T>
+ getByID(K id) : T
+ getByPattern(String property, String regex) : List<T>
+ getByPatternOrderBy(String property, String regex, String propertyName, Sense sense) : List<T>
+ getByPropertyName(String property, Object value) : List<T>
+ getByPropertyNameOrderBy(String property, Object value, String propertyName, Sense sense) : List<T>
+ getIDs() : List<K>
+ persist(T... objs) : int
+ remove(T... objs) : int
+ set(String[] propertynames, Object[] values, DAOPredicate<T> predicate) : int
+ update(T oldObj, T newObj) : boolean
+ updateByID(K id, T obj) : boolean
+ updateProperty(String propertyName, Object oldValue, Object newValue) : int
+ updatePropertyByID(K id, String propertyName, Object newValue) : boolean
DAO_SQL<K, T>
- connect : Connection
- result : ResultSet
- tableName : String
- number_of_columns : int
- addStatement : String
- deleteStatement : String
- findStatement : String
- updateStatement : String
- updateByIDStatement : String
+ DAO_SQL(Supplier<T> supp, Connection connect, String nameOfTheID) : ctor
+ DAO_SQL(Supplier<T> supp, Connection connect) : ctor
+ deleteByID(K id) : boolean
+ deleteByPattern(String property, String regex) : int
+ deleteByPropertyName(String property, Object value) : int
+ find(T obj) : boolean
+ getAll() : List<T>
+ getByID(K value) : T
+ getByPattern(String property, String regex) : List<T>
+ getByPropertyName(String property, Object value) : List<T>
+ getIDs() : List<K>
+ persist(T... objs) : int
+ remove(T... objs) : int
+ update(T oldObj, T newObj) : boolean
+ updateByID(K id, T obj) : boolean
+ updateProperty(String propertyName, Object oldValue, Object newValue) : int
+ updatePropertyByID(K id, String propertyName, Object newValue) : boolean
DAO0<K, T>
~ fieldNames : List<String>
~ fields : List<Field>
~ objClass : Class<?>
~ nameOfTheID : String
~ DAO0(Supplier<T> supp, String nameOfTheID) : ctor
+ add(T obj) : boolean
~ check_property_names(String... names) : void
~ classOfProperty(String propertyName) : Class<?>
+ delete(T obj) : boolean
+ delete(DAOPredicate<T> t) : int
+ get(DAOPredicate<T> t) : List<T>
+ getAllOrderBy(String propertyName, Sense sense) : List<T>
+ getByPropertyNameOrderBy(String property, Object value, String propertyName, Sense sense) : List<T>
+ getByPatternOrderBy(String property, String regex, String propertyName, Sense sense) : List<T>
~ getIDsInStringFormat() : List<String>
+ set(String[] propertynames, Object[] values, DAOPredicate<T> predicate) : int
AbstractDAO<K, T>
# fieldNames : List<String>
# fields : List<Field>
# objClass : Class<?>
# nameOfTheID : String
+ AbstractDAO(Supplier<T> supp) : ctor
+ AbstractDAO(Supplier<T> supp, String nameOfTheID) : ctor
+ deleteByID(K id) : boolean
+ deleteByPattern(String property, String regex) : int
+ deleteByPropertyName(String property, Object value) : int
+ find(T obj) : boolean
+ getByID(K id) : T
+ getByPattern(String property, String regex) : List<T>
+ getByPropertyName(String property, Object value) : List<T>
+ getIDs() : List<K>
+ update(T oldObj, T newObj) : boolean
+ updateByID(K id, T obj) : boolean
+ updateProperty(String propertyName, Object oldValue, Object newValue) : int
+ updatePropertyByID(K id, String propertyName, Object newValue) : boolean
DAO_XML<K, T>
- document : Document
- format : Format
- res : StreamResult
- src : DOMSource
- tagName : String
- transformer : Transformer
+ DAO_XML(Supplier<T> supp, String filePath, Format format, String nameOfThe
+ DAO_XML(Supplier<T> supp, String filePath, Format format) : ctor
+ deleteByID(K id) : boolean
+ deleteByPattern(String property, String regex) : int
+ deleteByPropertyName(String property, Object value) : int
+ find(T obj) : boolean
- flush() : void
+ getAll() : List<T>
+ getByID(K id) : T
+ getByPattern(String property, String regex) : List<T>
+ getByPropertyName(String property, Object value) : List<T>
+ getIDs() : List<K>
+ persist(T... objs) : int
+ remove(T... objs) : int
+ update(T oldObj, T newObj) : boolean
+ updateByID(K id, T obj) : boolean
+ updateProperty(String propertyName, Object oldValue, Object newValue) : int
+ updatePropertyByID(K id, String propertyName, Object newValue) : boolean
DAO_CSV<K, T>
- filePath : String
- header : String
- lines : List<String>
- reader : BufferedReader
- stream : Stream<String>
- writer : BufferedWriter
+ DAO_CSV(Supplier<T> supp, String filePath, String nameOfTheID) : ctor
+ DAO_CSV(Supplier<T> suppSupplier<T> supp, String filePath) : ctor
- checkHeader(String header, List<String> fieldsNames) : void
+ deleteByID(K id) : boolean
+ deleteByPattern(String property, String regex) : int
+ deleteByPropertyName(String property, Object value) : int
+ find(T obj) : boolean
+ getAll() : List<T>
+ getByID(K id) : T
+ getByPattern(String property, String regex) : List<T>
+ getByPropertyName(String property, Object value) : List<T>
+ getIDs() : List<K>
- init() : void
+ persist(T... objs) : int
+ remove(T... objs) : int
+ update(T oldObj, T newObj) : boolean
+ updateByID(K id, T obj) : boolean
+ updateProperty(String propertyName, Object oldValue, Object newValue) : int
+ updatePropertyByID(K id, String propertyName, Object newValue) : boolean

Figure 6 : UML diagram of the classes of the package rostDAO

individual steps to be written and executed. These additional steps would include, among others:

- Stronger security measures.
- Creating the SQL table if it does not already exist in the database.
- Deleting unnecessary data from memory.

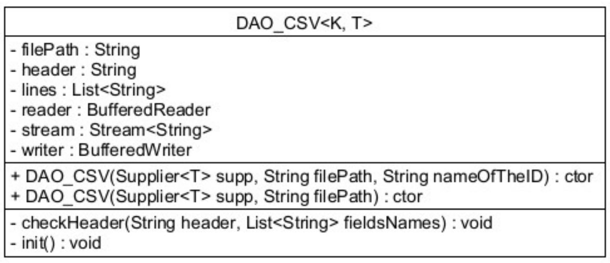
It is particularly important to note with the `DAO_SQL` class that:

- The implementations of the 15 access methods (since the three methods `orderBy()` have already been implemented in the `DAO0` class) were written based on general code as far as possible. This means that commands specific to certain relational database management systems (RDBMS) were strictly avoided. In other words, it is expected that these generally implemented access methods will work with all RDBMS, whether it is Oracle, MySQL, PostgreSQL, DB2, MongoDB, etc.

- Connections between tables using the `inner join` / `outer join` commands could not be implemented. If necessary, the end developer will have to implement their own access methods.

- The name of the table to be queried must be the same as the corresponding JavaBean name. Both can only differ in case sensitivity (uppercase and lowercase).

2. *The class DAO_CSV*



```
DAO_CSV<K, T>
- filePath : String
- header : String
- lines : List<String>
- reader : BufferedReader
- stream : Stream<String>
- writer : BufferedWriter
+ DAO_CSV(Supplier<T> supp, String filePath, String nameOfTheID) : ctor
+ DAO_CSV(Supplier<T> supp, String filePath) : ctor
- checkHeader(String header, List<String> fieldsNames) : void
- init() : void
```

Figure 7: UML Diagram of the class *DAO_CSV*

`DAO_CSV` is the class to be instantiated or implemented when data is stored in CSV files. Its constructors receive the parameter `filePath`, which refers to the path of the CSV file.

The `void` method `checkHeader(String header, List<String> fieldNames)` verifies whether all attributes of the JavaBean are present in the header of the CSV file. In doing so, the attribute names must match exactly, taking case sensitivity into account (i.e., uppercase and lowercase letters must match). However, the order of the attributes in the JavaBean class and their order in the CSV file header is irrelevant and therefore not considered. If a header adheres to these rules, it is considered **valid**.

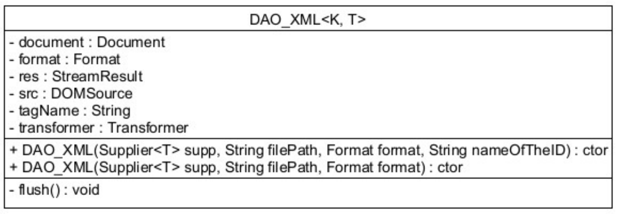
How this class is applied can be understood by referring to examples 1.1 and 1.2 of the `DAO_SQL` class. When using the `DAO_CSV` class, the following points must be strictly observed:

- Every CSV file associated with this class must have a *valid header* as being shown in the standard conventions of CSV files.

- The only valid separator is the *comma*. Other characters commonly used as separators in CSV files cannot be read by this class.

- Each method opens, uses, and ultimately closes its resources. Although it is often discouraged to repeatedly open and close resources, this small rule is violated here because no other security measures against TOCTTOU (Time-of-Check to Time-of-Use vulnerabilities) have yet been found.

3. *The class DAO_XML*



```
DAO_XML<K, T>
- document : Document
- format : Format
- res : StreamResult
- src : DOMSource
- tagName : String
- transformer : Transformer
+ DAO_XML(Supplier<T> supp, String filePath, Format format, String nameOfTheID) : ctor
+ DAO_XML(Supplier<T> supp, String filePath, Format format) : ctor
- flush() : void
```

Figure 8: UML Diagram of the class *DAO_XML*

Very often, datasets are also stored in XML files. In such cases, the end developer can use the `DAO_XML` class. Its constructors, like those of the `DAO_CSV` class, receive the parameter `filePath`, which refers to the path of the XML file. The second parameter, `format`, specifies whether the data in the XML file is written in the form of attributes or XML tags. This parameter must take one of the two valid values, `TAG` or `ATTRIBUTE`, from the `Format` enumeration in the package.

How this class is applied can be understood by referring to examples 1.1 and 1.2 of the `DAO_SQL` class. When using the `DAO_XML` class, the following points must be strictly observed:

- Complex XML tags and types are *not supported*. For instance, the following XML structure cannot be read by the `DAO_XML` class:

  ```
  <price currency="dollar">20.000</price>
  ```

  The readable fields must either all be *attributes* or all be *XML tags*.

- After performing some access methods, the file may no longer follow correct typographic indentation and, therefore, may not be very presentable. However, the shortcut **Ctrl + Shift + F** in Eclipse can help resolve this issue.

4. The class AbstractDAO

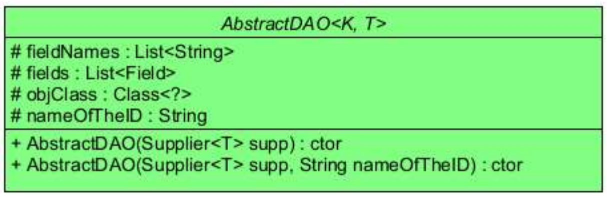

AbstractDAO<K, T>
# fieldNames : List<String>
# fields : List<Field>
# objClass : Class<?>
# nameOfTheID : String
+ AbstractDAO(Supplier<T> supp) : ctor
+ AbstractDAO(Supplier<T> supp, String nameOfTheID) : ctor

Figure 9: UML Diagram of the class *`AbstractDAO`*

The three previous classes, `DAO_SQL`, `DAO_CSV`, and `DAO_XML`, serve as the foundational classes for creating DAO implementations tailored to specific storage formats. However, it is evident that not all possible storage formats can be covered. For instance, if an end developer works with JSON, they are required to implement the general DAO interface themselves and individually override all 18 access methods. As mentioned earlier, implementing these 18 CRUDL methods is a significant challenge and, in many cases, impractical or even inconceivable.

To ease the developer's workload, the `AbstractDAO` class was introduced. This class is abstract, as indicated by its green colouring in the diagram, which clarifies why it cannot be directly applied in the same manner as, for example, the `DAO_SQL` class in Example 1.1. Eight of the 23 CRUDL methods have already been implemented in the `DAO0` class. Furthermore, `AbstractDAO` provides implementations for 12 additional data access methods. The developer is only required to implement the three remaining abstract methods - `persist(T… obj)`, `remove(T… obj)`, and `getAll()`–either through inheritance or by using an anonymous class. Bear in mind that these 3 access methods have been omitted in the UML diagram of the class `AbstractDAO` on Figure 5 because they are already specified in the general `DAO` interface and still remain **`abstract`** in the class `AbstractDAO` which its turn is also abstract. Writing them again in this last class with their abstract signature may appear superfluous.

***First way:* through inheritance**

```
    package dao.implementations.abstract;

    import java.util.List;

    import dao.beans.Candidate;
    import rost.dao.base.AbstractDAO;

    public      class      CandidateDAO      extends
AbstractDAO<String, Candidate> {

        public CandidateDAO() throws Exception {
           super(Candidate::new, "id_number");
        }

        // implement the methods persist, remove
and getAll here !

    }
```

***Second way:* through anonym classes**

```
    public  static  DAO  getCandidateDAO()  throws
Exception {

       return new AbstractDAO<String, Candidate>(
Candidate::new, "id_number") {

    // implements the methods persist, remove and
getAll here
       });
    }
```

How these methods are implemented is entirely up to the end developer. However, these implementations must adhere to one of the storage formats: SQL, CSV, or XML. One could easily note that the other 20 methods implemented in this class depend on the three abstract methods. This implies : as far as the three abstract methods are erroneously implemented, the other 20 methods will fail.

c) Important notes

- Each generic DAO class (`DAO_CSV`, `DAO_SQL`, `DAO_XML`, and `AbstractDAO`) has two constructors. If entities are identified and managed using a key, the second constructor, which requires the name of the attribute serving as the key or ID, must be used. If entities are processed without IDs or keys, the first constructor should be used.

- The two generic parameters, `K` and `T`, must always be specified, even when entities are processed without IDs.

- Each constructor must be called with a non-null reference to a `Supplier` that supplies a non-null instance of the considered JavaBean. Otherwise, a `NullPointerException` is thrown.

- At least one constructor must be invoked. This can be achieved either through inheritance (using the `super` keyword) or—except in the case of the `AbstractDAO` class—through direct instantiation (using the `new` keyword).

- Every attribute of a JavaBean must have a corresponding representation in the persistence layer. If a JavaBean has `n` attributes, the persistence layer must also account for all these `n` attributes.

- Only simple JavaBeans can be correctly processed by the provided classes. "Simple JavaBeans" refer to JavaBeans that do not inherit from other classes and whose attributes are all primitive types (numbers, booleans, strings, dates). JavaBeans with attributes such as lists, arrays, or more complex types cannot be handled by these classes.

- The access methods also ensure that no duplicate entities exist in the persistence layer. If duplication attempts are made or two identical entities are found during a read operation, a `DAOException` is thrown. This means that the generic classes will never store two identical records in the persistence layer.

- Any access method that returns a list will, in practice, return an immutable list. This is to prevent TOCTTOU (Time of Check to Time of Use) vulnerabilities in the project's business objects. End developers are required to implement the `getAll()` method—if inheriting from the `AbstractDAO` class—such that the resulting list is immutable.

- The JavaBeans must comply with the standard definitions of JavaBeans [3, p. 9-10] [7], as well as their accessor methods [4]. Violations of these standards will result in exceptions being thrown.

IV. <u>THE DEVELOPMENT METHOD</u>

Typically—and even necessarily—the attributes of a JavaBean must be transformed into their form on the persistence layer. This conversion is referred to as object-relational mapping. As mentioned earlier, the more attributes a JavaBean has, the more verbose and difficult it becomes to implement this object-relational mapping. The general structure of this task, and therefore the setup of access methods between classes, is often nearly identical in classic DAO implementations. The universal solution to duplication is, obviously, to factorize repetitive elements. However, these methods cannot be factorized, as each JavaBean has neither the same attribute names nor the same number of attributes.

Upon closer examination of these access methods, many patterns become apparent. For example, consider the `add(T obj)` method in SQL-DAO classes, which operates with the `PreparedStatement` object. One can observe that insertion consistently follows five steps:

- First, it is checked whether a record with the values of the passed object already exists in the persistence layer. If such an entity exists, a `DAOException` is thrown.
- Next, a query for insertion is precompiled using the `PreparedStatement` object.
- Then, the values of the passed object are added to the precompiled query. This step corresponds to object-relational mapping.
- Subsequently, the query is executed, and the final result is returned, indicating whether the insertion was successful.
- Finally, the opened resources are closed.

The only problem in creating a single general class that can universally consider all JavaBeans and pre-implement the 23 access methods lies in the fact that neither the number of attributes in the JavaBean nor their names can be predetermined for the purposes of query precompilation or object-relational mapping. Fortunately, a feature of the Java programming language exists that solves this problem: reflection.

Reflection in Java allows us to inspect and modify classes, interfaces, constructors, methods, and fields at runtime, even if the class name is unknown at compile time [5]. Using reflection, methods of any class can be invoked. This also makes it easy to calculate the names and number of fields. A `Class` object is required as an entry point to reflection in Java. This object can be obtained in three different ways:

1. **First way:** By searching with the JVM

   ```
   Class class = Class.forName("String");
   ```

   Java throws a `ClassNotFoundException` if it cannot find a class with that name.

2. **<u>Second way:</u>** Through the `class` variable of each class

   ```
   Class class = String.class;
   ```

   This second method has the great advantage that it never throws an exception, as the accessed class must necessarily exist beforehand.

3. **<u>Third way:</u>** Through a non-null instance of the class

   ```
   String name = "Java Virtual Machine";

   Class class = name.getClass();
   ```

   In this third method, the `Class` object is obtained through its getter, compared to the second method where it is accessed directly as a class variable. The `getClass()` method throws a `NullPointerException` if the calling object is null.

A complete documentation of the `Class` class can be found in the JavaSE platform documentation [6]. However, the methods that were relevant for this project include:

1. `getConstructor(Class... parameterTypes)`: This method returns the constructor of the class whose parameter types match the types specified in the `parameterTypes` variable. A new instance is then created using the `newInstance(Object... initargs)` method of the obtained constructor object, where `initargs` represent the arguments as in a normal call. Setting these arguments and parameters can sometimes be so demanding that a no-argument constructor must be included to define a valid JavaBean. This method is used to construct new beans.
2. `getDeclaredMethod(String name, Class<?>... parameterTypes)`: This method takes the name of a method as input and returns a `Method` object whose

argument types match the types specified in the `parameterTypes` variable. The method is then executed by calling `invoke(Object obj, Object... args)`. Here, `obj` is the object on which the method is invoked, and `args` are the arguments to pass. This is used to call getters and setters of JavaBeans in this project, particularly for object-relational mapping.

3. `getFields()`: This method returns an array containing all fields of a class. It is used to consider all fields of JavaBeans in object-relational mappings.

In fact, our design approach consists of combining reflection with loops.

V. JUSTIFICATION OF SOME CONCEPTIONS

1. Why does the DAO0 class exist, and what is its purpose?

Each more specific generic DAO class (`DAO_SQL`, `DAO_CSV`, `DAO_XML`, `AbstractDAO`) handles datasets and objects through reflection. The variables that support reflection would need to be duplicated in each class. For this first reason, these variables are consolidated into a single class—`DAO0`—so that the more specific DAO classes can access them through inheritance.

Additionally, handling such data can pose significant risks, such as TOCTTOU (Time Of Check To Time Of Use). For this second reason, a class is introduced to either validate or carefully handle certain data-related information. This class also contains methods that can help with factoring out common functionality, implementing security measures, or improving the overall code design.

Figure 9 illustrates how the classes in this package would be interconnected if the `DAO0` class did not exist. In that case, its content would move into the upper-level class. However, since `DAO` is an interface and cannot contain instance objects or concrete methods, it would have to be converted into an abstract class. This would result in two abstract classes serving generally the same purpose.

For the sake of brevity, the `AbstractDAO` class would then be omitted, and its entire content would be transferred into the `DAO` class, leading to a single final abstract DAO class as shown on the Figure 9. The answers to questions 2 and 3 mention fundamental reasons why this particular hierarchy is incompatible with our development principles and was therefore discarded.

2. Why are some access methods especially the sorting one designed in the `DAO0` class?

The three methods `getAllOrderBy`, `getByPropertyNameOrderBy`, and `getByPatternOrderBy` must return lists of objects that are sorted by a property. There are two possible ways to implement them. Either they can be implemented in the respective concrete classes (`DAO_CSV`, `DAO_SQL` and `DAO_XML`) with advanced APIs, or they can be pre-implemented in the `DAO0` class for any persistence layer. Assume there are N records in memory, and let N' be the number of objects that meet any given condition for each method. Let a represent the runtime for reading (access + object-relational mapping) a record/object, and b represent the runtime for testing an object. It is mandatory that: $N' \leq N$, and in any case, all records must be read, thus requiring an additional time of $N \times a$.

In the concrete classes, all records must first be sorted into a list and then, from the resulting list, the objects whose properties meet a precise condition must be selected. The runtime for this task is $N * b$, as all objects must be tested. However, every correct sorting algorithm requires to perform $\Omega(N \times ln(N))$ steps. The runtime for this option is therefore

$$T_1(N) = N.a + \Omega(N.\ln N) + N.b + c$$

$$= N(a+b) + \Omega(N.\ln N) + c$$

However, if one first selects all records that meet the conditions, transforms them, and then performs the sorting algorithm, the runtime is...

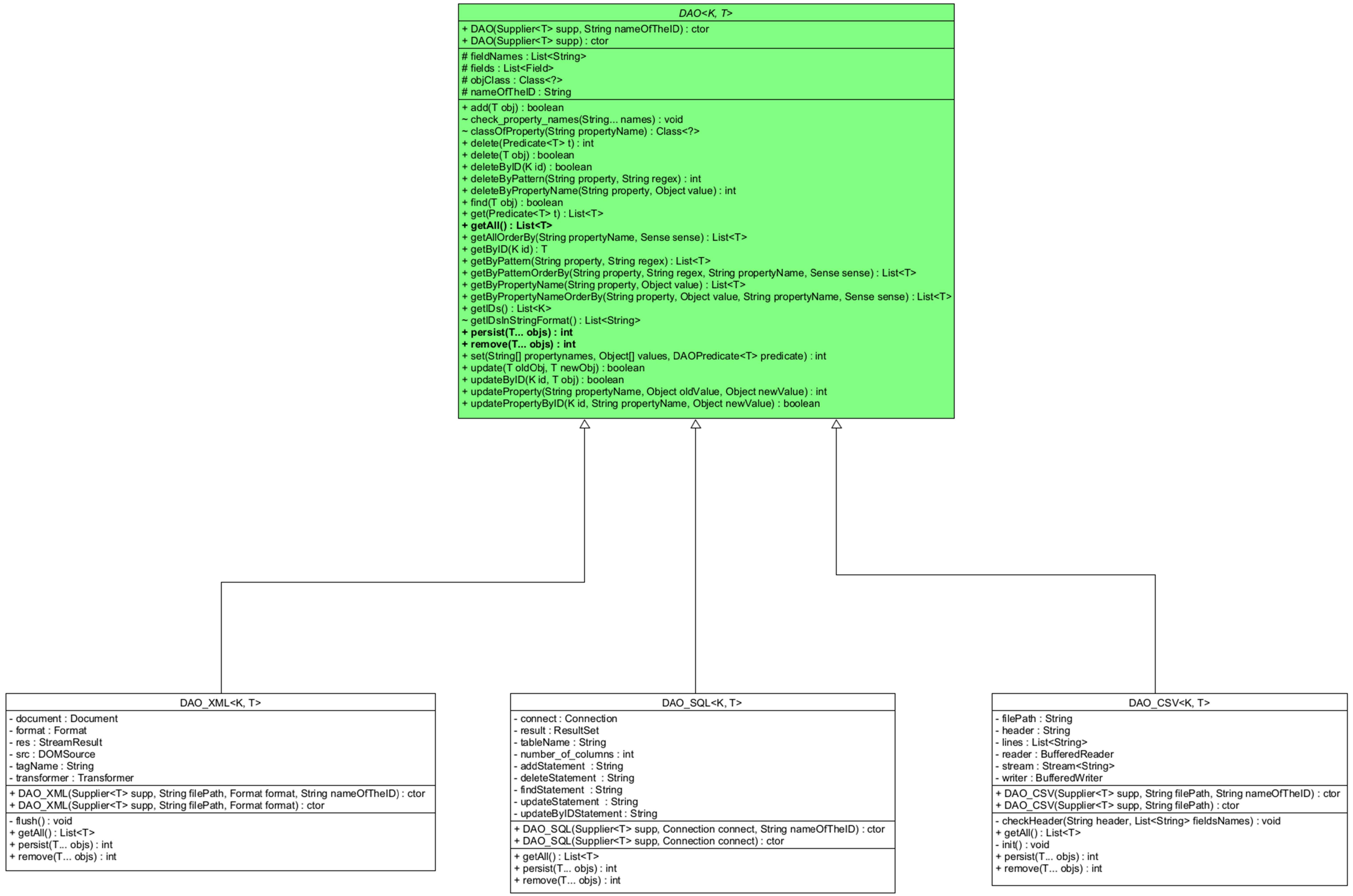

DAO<K, T>
+ DAO(Supplier<T> supp, String nameOfTheID) : ctor
+ DAO(Supplier<T> supp) : ctor
# fieldNames : List<String>
# fields : List<Field>
# objClass : Class<?>
# nameOfTheID : String
+ add(T obj) : boolean
~ check_property_names(String... names) : void
~ classOfProperty(String propertyName) : Class<?>
+ delete(Predicate<T> t) : int
+ delete(T obj) : boolean
+ deleteByID(K id) : boolean
+ deleteByPattern(String property, String regex) : int
+ deleteByPropertyName(String property, Object value) : int
+ find(T obj) : boolean
+ get(Predicate<T> t) : List<T>
+ getAll() : List<T>
+ getAllOrderBy(String propertyName, Sense sense) : List<T>
+ getByID(K id) : T
+ getByPattern(String property, String regex) : List<T>
+ getByPatternOrderBy(String property, String regex, String propertyName, Sense sense) : List<T>
+ getByPropertyName(String property, Object value) : List<T>
+ getByPropertyNameOrderBy(String property, Object value, String propertyName, Sense sense) : List<T>
+ getIDs() : List<K>
~ getIDsInStringFormat() : List<String>
+ persist(T... objs) : int
+ remove(T... objs) : int
+ set(String[] propertynames, Object[] values, DAOPredicate<T> predicate) : int
+ update(T oldObj, T newObj) : boolean
+ updateByID(K id, T obj) : boolean
+ updateProperty(String propertyName, Object oldValue, Object newValue) : int
+ updatePropertyByID(K id, String propertyName, Object newValue) : boolean
DAO_XML<K, T>
- document : Document
- format : Format
- res : StreamResult
- src : DOMSource
- tagName : String
- transformer : Transformer
+ DAO_XML(Supplier<T> supp, String filePath, Format format, String nameOfTheID) : ctor
+ DAO_XML(Supplier<T> supp, String filePath, Format format) : ctor
- flush() : void
+ getAll() : List<T>
+ persist(T... objs) : int
+ remove(T... objs) : int
DAO_SQL<K, T>
- connect : Connection
- result : ResultSet
- tableName : String
- number_of_columns : int
- addStatement : String
- deleteStatement : String
- findStatement : String
- updateStatement : String
- updateByIDStatement : String
+ DAO_SQL(Supplier<T> supp, Connection connect, String nameOfTheID) : ctor
+ DAO_SQL(Supplier<T> supp, Connection connect) : ctor
+ getAll() : List<T>
+ persist(T... objs) : int
+ remove(T... objs) : int
DAO_CSV<K, T>
- filePath : String
- header : String
- lines : List<String>
- reader : BufferedReader
- stream : Stream<String>
- writer : BufferedWriter
+ DAO_CSV(Supplier<T> supp, String filePath, String nameOfTheID) : ctor
+ DAO_CSV(Supplier<T> supp, String filePath) : ctor
- checkHeader(String header, List<String> fieldsNames) : void
+ getAll() : List<T>
- init() : void
+ persist(T... objs) : int
+ remove(T... objs) : int

Figure 10 : Possible conception of the rostDAO package without the `DAO0` class

$$\begin{aligned} T_2(N) &= N.a + N.b + \Omega(N'.\ln N') + c \\ &= N(a+b) + \Omega(N'.\ln N') + c \\ &\leq N(a+b) + \Omega(N.\ln N) + c \\ &\leq T_1(N) \end{aligned}$$

It is therefore clear that the algorithms for the sorting access methods in the `AbstractDAO` class are significantly faster than those in the `DAO_CSV`, `DAO_SQL`, and `DAO_XML` classes when applying APIs. However, since these pre-implementations are generally faster, they must be designed in such a way that each class (`DAO_CSV`, `DAO_SQL`, `DAO_XML`, and `AbstractDAO`) can access them. And only the `DAO0` class is the intended place for this. The other 4 access methods held by this class are -according to the hierarchy- the fastest and most easy to implement for all data storage ever. Since these 8 methods are or will be the best possible, they are all declared as **`final`** in the class `DAO0` and this implies that they can neither be overridden nor overwritten.

3. The AbstractDAO class already correctly implements 12 access methods for arbitrary data sources. Why are these 12 access methods re-implemented in the classes DAO_SQL, DAO_CSV, and DAO_XML?

In the `AbstractDAO` class, access methods are implemented to provide correct outputs and reliably perform tasks for any form of persistence layer (SQL, CSV, JSON, XML, YAML, Properties, etc.). There are three types of methods to note, especially based on the implementation techniques:

1. *Access methods whose implementations neither require reflection nor advanced APIs:* For example, the `find(T obj)` method can be implemented by first retrieving all records using the `getAll()` method and then searching in the returned list to see if the object `obj` is found.

2. *Access methods whose implementations require reflection but do not need advanced APIs:* For example, the `getByPattern(String property, String regex)` method can be implemented by first requesting all records using the `getAll()` method, and then using reflection to check each object in the resulting list to see if the `property` matches the regular expression `regex`. All objects that meet the condition are collected in the returned list.

3. *Access methods whose implementations require both reflection and advanced APIs:* To implement the `getAll()` method, the records in memory must be read and converted into their equivalent objects through object-relational mapping. Reading these records requires advanced APIs depending on how the records are stored: JDBC for SQL, DOM for XML, Input/Output for CSV, and so on. This category also includes the methods `persist(T… obj)` and `remove(T… obj)`, which should be considered as abstract methods.

It is clear that only methods from the first two categories can be pre-implemented. The remaining three methods, `persist`, `remove`, and `getAll`, are abstract and cannot be pre-implemented. As a result, the `DAO_CSV`, `DAO_SQL`, and `DAO_XML` classes can inherit from the `AbstractDAO` class rather than from the `DAO0` class and focus only on implementing abstract access methods. Furthermore, the package can omit the general DAO interface, and therefore, the `DAO`, `DAO0`, and `AbstractDAO` classes can be unified, leading to a single final abstract `DAO` class, as shown on Figure 9. While the final structure of the package is much smaller and requires less code, these pre-implementations take significantly more time.

Assuming there are N objects/records in memory, `a` is the runtime for reading (access + object-relational mapping) a record/object, and `b` is the runtime for testing an object. Let's consider the access method `find(T obj)` as a pre-implementation:

- First, all objects are retrieved using the `getAll()` method, runtime: $N \times a$
- Then, the search is performed, runtime: $N \times b$

The total runtime for this implementation is

$$T(N) = N.a + N.b + c.$$

In the worst case, this implementation must convert all records into objects and then traverse the entire returned list. In the average case, it will convert all records but stop as soon as the passed object is found in the list. In the best case, all records are converted, but only one additional step is performed, meaning the object being searched for is at the beginning of the returned list. Therefore, the overall runtime is generally $\Omega(N)$. However, if we directly test during the reading process whether the properties match the records and, if correct, convert the record into an object, the runtime becomes:

$$T'(N) = N.a + c$$

In the worst case, all records in the persistence layer are visited. In the best case, only one step is performed, meaning the object being searched for is at the beginning of the returned list. In the average case, the implementation stops as soon as a record with the same properties as the passed object is found, and it is then converted into an object. The overall runtime is therefore generally $O(N)$.

It is then clear that pre-implementations—except in exceptional cases—are significantly slower than implementations with advanced APIs, since $T'(N) < T(N)$ and $O(N)$ is much better than $\Omega(N)$.

4. Why is AbstractDAO the only class that contains the attributes `fieldNames`, `fields`, `objClass`, `nameOfTheID` ?

First of all, we recall that the classes `DAO_SQL`, `DAO_CSV` and `DAO_XML` are already complete and fully implemented classes, in that sense that the end does not need to write any line of code related to CRUDL methods when using these classes. He only needs to instantiate them with correct arguments to perform manipulations on data by calling the methods. In other words, leaving these attributes as **`public`** in these classes is nothing than superfluous. In contrast, the class `AbstractDAO` is still abstract since it contains the three abstract methods -`persist(T obj)`, `remove(T obj)`, and `getAll()`. If one extends this class (`AbstractDAO`) to design a DAO class for a special JavaBean, one can leave these above-mentioned attributes out of sight. But if one wishes to build a general generic DAO class for a special saving platform like JSON or YAML, the use of these properties become necessary.

5. Why are immutable lists returned?

All access methods that return lists provide an immutable list, either through the statement `return new RostArrayList<T>(size, list, copy);` or `return Collections.unmodifiableList(list);`

This measure is implemented for security reasons, ensuring that the business logic always receives the correct data from the data source. In reality, lists can easily be modified with TOCTTOU (Time Of Check To Time Of Use), which can lead to various issues. The end developer may also unknowingly alter the data in the list, even though they are not changing the data in memory in a corresponding, reliable, and equivalent way. Returning immutable lists protects against unexpected changes and helps maintain consistency between the persistence layer and the business logic.

6. Why was the `RostArrayList` class introduced?

The JavaSE platform's API collection is considered a robust and well-designed set of classes that simplifies handling multiple data in various data structures. However, the algorithms in these classes are final. The `RostArrayList` class was designed to change algorithms whenever possible based on certain circumstances. By using this class, more control over data is achieved.

7. Why are the elements of a `RostArrayList<E>` ***managed in an array of type*** `Object[]` ***instead of an array of type*** `E[]` ***?***

First of all, Java can not ensure at the level of compilation that all objects held by an array of type `Object[]` are instances of a generic type E. This leads sometimes to some runtime errors which Java developers preferred avoiding. Furthermore, Casting objects while writing them in an array represents a little time cost which become as huge as the storage contain data.

8. Why must all access methods in this package be surrounded by a try-catch block?

Accessing external data often results in throwing multiple exceptions. These exceptions are frequently caused by incorrect property determination, duplicate keys, missing files, unknown formats, etc. None of the possible exceptions are handled within the package. Therefore, all access methods in the DAO interface should be read with the final signature `throws Exception`. There are two reasons for this: First, the end developer would not know why their commands failed if the exceptions were handled within the package. It is also impossible to predict exactly which exceptions might be thrown when designing these methods due to the vast number of possible exceptions. Since all CRUDL methods do not handle exceptions and therefore have the `throws Exception` ending in their signatures, calling these methods must either be within a method that also has the same signature or be enclosed in a try-catch block.

9. Which role does the `Supplier` ***object in the constructor of each class play?***

Three different methods for reflection were introduced in the section "The Development Method." The first method has the major disadvantage that an exception is thrown if no class with the given name exists. The second method has the significant disadvantage that it does not work with generic parameters. For example, if `T` is a generic parameter in a class, the following Java statement is not recognized and will not compile:

```
        Class x = T.class;
```

However, it is possible to request an already existing `Class` object instead of an instance from the class `T`. The third method has the advantage that a non-null instance of an object will always return a valid instance of the corresponding `Class` object so that it will be possible to start the instantiation of objects helping by reflexivity. This non-null instance of an object of type `T` is supplied by a `Supplier<T>` object. And the obligation of this object within the parameters of each constructor is motivated by the fact that one can easily write `T::new` as the argument at the matching place when instantiating a `DAO` class, where `T` denotes the name of the JavaBean class. For instance, show the examples above. This allow the user of the classes to write few and forces him to provide no parameter constructors within each JavaBean class, which in reality is at the top of the requirements of this package.

10. What are these strings **`addStatement`*, *`deleteStatement`*, *`findStatement`*, *`updateStatement`*, and*** **`updateByIDStatement`** ***in the*** **`DAO_SQL`** ***class, and what are they used for?***

The `PreparedStatement` object in the `DAO_SQL` class needs to execute different corresponding SQL commands for each access method. SQL commands for inserting, deleting, finding, and updating are quite verbose and are generated from field or attribute names using loops. The only issue is that looping to generate the required SQL commands on each method call would take more time. Therefore, all these SQL commands are built in the called constructor with just one loop and are eventually made available to the corresponding access methods. To protect the programs from TOCTTOU and SQL injection attacks, these SQL commands are declared as `final` (which is why they are written in uppercase).

11. Why is there no **`updateByProperty(String property, Object value, T obj)`** ***method in the package?***

Upon thorough consideration, this method would search through all the records in the persistence layer where the attribute `property` has the value `value`, and replace them with the object `obj`. However, this would result in duplicates of records in the data source, which contradicts our design principles.

12. Why are the methods persist(T… obj) and remove(T… obj) abstract, since it is possible to implement them by calling in loops respectively the methods add(T obj) and delete(T obj), if one declare these last two as abstract ?

Implementing the methods `persist(T… obj)` and `remove(T… obj)` of the class `AbstractDAO` in loops like above implies to write the following codes as standard implementations in the class `DAO0`:

```
public final int persist(T... objs) throws
Exception {

        int result = 0;

        for(T obj : objs)
                result += add(obj) ? 1 : 0;

        return result;
}

public final int remove(T... objs) throws
Exception {

        int result = 0;

        for(T obj : objs)
                result += delete(obj) ? 1 : 0;

        return result;
}
```

Where the methods `add(T obj)` and `delete(T obj)` are assumed to be **`abstract`**. This conception is correct but it presents the main problem that both the methods `add(T obj)` and `delete(T obj)` open and close a non negligeable quantity of resources only to perform operations on one record. Opening and closing resources one after one may lead to some problems and impacts negatively the runtime by increasing it because of opening and closing the resources.

13. Why could data not be transferred through Java Records like JavaBeans?

This work also had as purpose to make possible the use of Java Records like JavaBeans to transfer data. But the main problem that withstood the reflexions came from the difficulty by calling constructors of Java Records with the right order of parameters, since the call of the method `getConstructor(…)` requires to provide the class of each parameter and in the same order as declared in the corresponding `Record` class. In other words, we had to get the right order of the parameters declared in the constructor. But the best algorithm in disposition right now that performs this goal must check $O(n!)$ permutations of objects, where n is the number of fields / attributes of the record. And this runtime is too much and grows very fast as the record has more fields.

14. Why has been the interface DAOPredicate introduced in the package ?

The interface `DAOPredicate` is strictly the same as the interface `Predicate` lying in the package `java.util.function.`. They perform the same operations and are meant for strictly the same purpose. But the interface `Predicate` showed its limits while we tried to derive the methods `deleteByPattern` and `deleteByPropertyName` from the method `delete(Predicate<T> t)` with lambdas for the sake of brevity of the codes. The same remark has been made on the methods `getByPattern` and `getByPropertyName` while trying to derive them from the method `get(Predicate<T> t)` also for the sake of brevity. The problem lied on the fact that the standard interface does not precise in its signature that some exceptions may be thrown during the runtime. In order words, the declaration **`throws`** `Exception` does not exist in the signature of the method `test` of the standard interface. Yet, our CRUDL methods access the properties of the JavaBean through reflectivity and so many exceptions may be thrown, (such as `IllegalAccessException`, `InvocationTargetException`, `NoSuchFieldException`, just to name a few`)`, that the invocation of methods related to the reflectivity must be surrounded by `try-catch` blocks. But catching these exceptions in our projects implies hiding to the end developer which errors occurred when running the methods. It is even impossible to manage all of them properly and according to all situations.

The interface `DAOPredicate` has been then designed, so that its method `test` has the declaration `throws Exception`, in its signature which allows us, not to catch possible been thrown exceptions that may occur. This permits to thrown them only during the execution of CRUDL methods by the end developer.

VI. FURTHER POINTS OF FUTURE WORK

Although our work seems to be very complete, there is a lot of questions to which we could not provide a right and correct conception. The following key points may be considered as fundamental starting points for deep future improvements to the package:

- Test as frequently as possible, in the shortest time, to ensure there are no duplicate records.
- Provide the end developer with a message in the exception being thrown that specifies exactly where duplicate records exist in CSV or XML files (for example by showing the lines where duplicate records lie in these files)
- Design other concrete generic classes for other data sources such as JSON, Properties, and YAML.
- Add new access methods to the general DAO interface.
- Enable JavaBeans to inherit properties from another JavaBean and account for this in object-relational mapping.
- Reduce the length of the code, thereby significantly increasing the speed of the algorithms.
- Improve the conversion of complex values into SQL.

VII. CONCLUSION

In the literature, it is strongly recommended to use the DAO (Data Access Object) pattern in data management for data-driven software. However, since this pattern is very demanding to implement, this could be one of the reasons why either security measures or strict structure—and sometimes both—are neglected. This paper introduces the new package *rostDAO*, which addresses and implements both of these programming aspects. DAO classes, which previously required multiple lines to design, can be easily completed with this package in fewer than 20 lines. However, the package has the minor drawback that the user must adhere to strict rules. This work has only been undertaken in the Java programming language. Therefore, the structure of the project suggests that the same approach could be achieved in other programming languages, such as Python.

VIII. REFERENCES

[1] Pavel Micka, Zdenek Kouba, *DAO Dispatcher Pattern: A Robust Design of the Data Access Layer*, Copyright (c) IARIA, 2013. ISBN: 978-1-61208-276-9

[2] Chapman, A. D. 2005, *Principles of Data Quality*, version 1.0. Report for the Global Biodiversity Information Facility, Copenhagen

[3] Sun Microsystems, *JavaBeans^{TM}*, Version 1.01-A, 8 August 1997

[4] https://de.wikipedia.org/wiki/Zugriffsfunktion

[5] https://www.delftstack.com/de/howto/java/reflection-in-java/

[6] https://docs.oracle.com/javase/8/docs/api/java/lang/Class.html

[7] https://de.wikipedia.org/wiki/JavaBeans

[8] https://en.wikipedia.org/wiki/Data_access_object#Disadvantages