UNIVERSITY OF TARTU
Institute of Computer Science
Computer Science Curriculum

Gregor Suurvarik

Generating Code for Database Classes in Java at Compile Time

Bachelor's Thesis (9 ECTS)

Supervisor:
Stefan Hermann Kuhn, PhD

Tartu 2025

Generating Code for Database Classes in Java at Compile Time

Abstract:

An increasing number of Java applications use databases. To make working with them more convenient, libraries are used that simplify communication between the database and the Java application. One of the most common solutions for this is object-relational mapping (ORM). Many ORM solutions perform their tasks entirely at runtime, including the resolution of mappings. This work involves creating an application that generates all the code necessary for database interaction at compile time. One of its goals is to reduce the amount of required input data (such as annotated objects and database model descriptions).

Keywords: Code generator, Java, Database

CERCS: P175 Informatics, systems theory

Andmebaasi klasside koodi genereerimine Javas komileerimis ajal

Lühikokkuvõte:

Üha rohkem Java rakendusi kasutab andmebaase. Nendega mugavamaks töötamiseks kasutatakse teeke, mis lihtsustavad suhtlust andmebaasi ja Java rakenduse vahel. Üks levinumaid lahendusi selleks on objekt-relatsiooniline vastendus (ingl *object-relational mapping*, lühendatult ORM). Paljud ORM-lahendused teevad oma töö täielikult jooksuajal, sealhulgas ka vastenduste otsimise. Selles töös luuakse rakendus, mis genereerib kogu andmebaasiga suhtlemiseks vajaliku koodi juba kompileerimise ajal. Üheks eesmärgiks on vähendada vajalike algandmete hulka (nagu annoteeritud objektid ja andmebaasi mudeli kirjeldus).

Võtmesõnad: Koodi generaator, Java, Andmebaas

CERCS: P175 Informaatika, süsteemiteooria

Contents

1. Introduction 4
   1.1 Background 4
   1.2 Goals 5
2. Selection of Technologies 7
   2.1 Language 7
   2.2 Build tool 7
   2.3 Annotation processor 7
   2.4 Code generation 8
3. Structure 9
4. Implementation 11
   4.1 Code generator 11
   4.2 Usage 13
   4.3 Generated code during runtime 17
5. Results 20
   5.1 Difficulties 20
   5.2 Future development 20
   5.3 Comparing to other similar applications 21
6. Conclusion 24

References 25

Appendices 26

License 31

1. Introduction

1.1 Background

Working with relational databases is a core functionality or at least an important part of many applications. In Java applications, frameworks are often used to simplify communication with databases. One of the most commonly used approaches is Object-Relational Mapping (ORM), which allows data in a relational database to be stored and accessed via Java objects. However, implementing such solutions typically requires use of annotations on Java objects and often does not support reading data directly into random objects without the developer having to write mappers for those objects.

One of the most widely used ORM standards is the Java Persistence API (JPA), which defines how data should be mapped using annotations [1]. A popular implementation of this standard is Hibernate ORM. Hibernate aims to simplify data handling, but because much of the SQL and object mapping happens at runtime, this can lead to problems. For instance, it may not know how to map JDBC Timestamp to LocalDateTime if the object does not have needed annotations. Additionally, since the Reflection API can not enforce strict type checking at compile time, it is possible to encounter type mismatch errors at runtime. For example, in the record `Pair<Long, Long>`, defined as `record Pair<L, R>(L left, R right) {}`, the left component could inadvertently be assigned an Integer, leading to an error when the value is accessed due to type incompatibility (ClassCastException in Java).

Furthermore, Hibernate determines the object type based on the SQL type and does not attempt to convert the value to match the object type e.g., it does not convert a Long object to Integer.

Currently, many frameworks use the Reflection API for dynamic programming, enabling runtime access to classes that may not be known at compile time, including their constructors, methods, and fields. However, this comes with risks, as type checking and other validations cannot be performed at compile time.

In the article by Ovidiu Mihai Tacu [2], the advantages and disadvantages of the Java Reflection API are discussed. The advantages mentioned include:

- Enables dynamic programming in Java
- Supports modular program design
- Allows runtime analysis of the program

However, the article also outlines several disadvantages to reflection:

- Increased performance overhead, as the Java Virtual Machine (JVM) cannot perform certain optimizations
- Reflection operations are slower than their lower-level counterparts
- Reduced code readability, which in turn lowers maintainability
- No compile-time type checking
- Enables access to private fields and methods, which may have unintended consequences

The article concludes that the Reflection API should be used cautiously and only when necessary, for instance, in scenarios that require dynamic programming or when the source code cannot be modified.

1.2 Goals

The goal of this bachelor's thesis is to develop an application capable of generating, at compile time, the classes necessary for database interaction based on annotated interfaces. This enables type safety checks during compilation and improves runtime performance by minimizing the use of Java Reflection API, which can block some Just-In-Time (JIT)[1] compiler optimizations.

The application should be capable of taking annotated Java interfaces and generating their implementations. The interface is annotated and contains queries that can be re-executed upon each application startup, such as queries for creating or updating database tables.

The application must be able to preprocess user-defined SQL queries by replacing placeholders with corresponding Java object values. Examples of such queries include:

- `SELECT [(paramName)];`
- `SELECT [(paramName.varName)];`
- `SELECT [(paramName.methodName)];`
- `SELECT [(paramName.toString())];`

[1] https://www.ibm.com/docs/en/sdk-java-technology/8?topic=reference-jit-compiler

In these examples, the placeholder is the part enclosed within `[(` and `)]`. The name of the placeholder must correspond to the name of the parameter in the method signature. From the corresponding object, it must be possible to read fields and invoke methods, e.g last three examples.

The application must be able to generate code to execute queries defined in the interface as statement-type queries (inserts, procedure calls) using JDBC [3], including the preprocessing described above. In addition to statement-type queries, the application should also be capable of generating code for application-level query methods. When generating automatic query mappings, the application should support primitive types, their boxed counterparts, simpler types such as String, and various date and time-related objects. These types should be readable as single values, as Optional values, and as lists.

Beyond simple types, the application must also be able to generate mappings for more complex objects based on SQL results, assuming that the object's fields are of simple types or generics[2]. Users should be able to define support for custom types by specifying them in a separate class when defining the interface.

Application must display all errors and warnings to the user via correct channels so that the user would not need to check extra logs or compilation process could be ended prematurely if validation error occurs, in that case application needs to inform the user where and what happened.

[2]https://docs.oracle.com/javase/tutorial/java/generics/types.html

2. Selection of Technologies

This chapter presents the rationale for the selection of technologies used in the application.

2.1 Language

For this application Java 21 was chosen, because it is statically-typed language, meaning variable and method types are known at compile time. Java also has automatic garbage collection, meaning the developer does not have to manually manage memory. Additionally, it provides platform independence, as the code is executed by the runtime environment rather than directly by the operating system. Java 21 has long-term support, so it will receive updates until the year 2028 [4]. Java 21 introduced pattern matching for the switch statements, making it more convenient to structure control logic [5]. For these reasons, the Java programming language, version 21, was chosen for this application.

2.2 Build tool

To create an runnable application from the source code, a build tool is required. For this purpose, Maven and Gradle were considered, those are two popular build tools for building Java applications, as both offer similar functionalities, including:

- dependency management,
- defining and executing tasks,
- building and compiling the application.

According to Maven's official website [6], the initial setup and module creation for an application can be done within seconds. However, this feature is not particularly important for this project. On the other hand, according to Gradle's official website [7], Gradle is faster and provides better options for managing the build process. Therefore, Gradle was chosen for this application.

2.3 Annotation processor

As this application uses annotation as its primary entrypoint, their processing is a big part of the application. Here the best choice was to use Java's built-in annotation processor; third-party libraries were considered, but they did not provide all functionalities that JavaX annotation processors would provide. By examining the Java 21 documentation [8], it was found that the JavaX annotation processor is sufficiently powerful for this application. According to the documentation, it provides access to all elements annotated with the corresponding annotation

during the compilation cycle. Since the program that uses this application has not yet been compiled at this stage, it is not possible to access classes or fields as with the Reflection API. Instead, elements are treated as non-compiled program components. These non-compiled program components make comparing types easier and make analysis of classes more accessible, so discovering methods with specific signatures is simpler. Additionally, based on the author's testing, the JavaX annotation processor allows for easier transmission of error messages to the compiler, ensuring that users can see exactly where an issue occurred in the source code. While the annotation processor cannot modify existing source code, it can generate new code that will be compiled in the next step.

Processor needs to be registered in `META-INF/services/javax.annotation.proccessing.Proc` file, this file contains list of classes that implement processor. It needs to be done for the compiler to find correct entry points to call the processor. For easier processor registration, Google's Auto Service is used, which automatically generates the required metadata file [9].

2.4 Code generation

To simplify the generation of Java source code, Palantir Technologies' library JavaPoet is used. It makes code generation more programmatic and modular. According to the JavaPoet documentation [10], the library provides several conveniences for code generation, including:

- automatic importing of libraries and types,
- programmatic creation and combination of classes, methods, and fields,
- automatic code formatting,
- writing generated code to files.

By using this library, the application does not need to check whether the syntax is correct or if anything needs to be imported, as JavaPoet's structure eliminates the need for such checks.

3. Structure

To achieve the required functionality of the application, a significant amount of code must be generated. Therefore, it was decided to create a set of base classes and abstractions on which the code generator can rely. One of the most commonly used abstractions is the database. The database is defined as an interface, as its precise implementation partially depends on the specific DBMS used. This choice promotes modularity and allows support for multiple database backends. This database interface allows the program and the end user to:

- perform queries (database statements that return value),
- execute statements (database statements that do not have return value),
- obtain a database connection,
- close database connections.

By relying on the database interface, the code can be generated without needing to know the exact method of connection.

To facilitate the addition of metadata necessary for user-defined interfaces, custom annotations are used, annotations in Java store metadata, they can not handle anything dynamic. In order to discover the repository interface during compilation, it must be annotated with the `@Repository` annotation, which can include SQL statements that should be executed upon initialization, such as table creation statements. Additionally, the annotation may reference external classes containing type definitions for custom column mappings. These must be public static methods that accept a ResultSet and an int as parameters, may throw only SQLException, and whose return type represents a custom column type for the generator. This mechanism allows extending the type system of the generator in a declarative and reusable manner.

For query metadata, the `@Query` annotation is used. This annotation defines the SQL query string and, in the case of primitive return types, also specifies a default value. It also allows the configuration of behavior in the event of no returned rows—whether to return a null or throw a NoSuchElementException, so that the developer could decide what situation occurred and what control flow they could expect to use.

For statement metadata, the `@Statement` annotation is provided, containing only the relevant SQL statement. Both `@Statement` and `@Query` support placeholders in SQL with format `[(placeholder)]`. The generator must be able to distinguish whether a placeholder

refers to a method parameter or a specific field within a method parameter object (e.g., `[(param.field)]`). Furthermore methods annotated with `@Statement` or `@Query` can but do not have to throw SQLException, when it is not implicitly thrown it still will be thrown as RuntimeException.

In addition to regular database operations, transactional support is often required. As this involves additional logic, the user's repository interface can be extended with the Transactional interface, which provides methods to:

- Begin new transactions,
- Execute queries and statements within a transaction,
- Join existing transactions,
- Roll back and commit transactions, consistent with SQL behavior.

Once the user has annotated their repository interface, they must also be able to access its implementation. However, direct access should be avoided, as during the initial analysis phase of the code, the generator may not yet have run, and the implementation classes may not yet exist—potentially causing issues for the analyzer. To resolve this, a static method is provided that allows users to instantiate the implementation of the repository interface by passing in a database object.

Since an application may need to support multiple database engines simultaneously, it is essential that the system can work with various DBMS implementations. This framework supports two commonly used database engines: MariaDB[3] and PostgreSQL[4]. Factory methods have been provided for both, enabling the creation of database objects corresponding to each engine, which can then be used to instantiate repository implementations.

To ensure the code generator functions correctly, it must be included not only as a runtime dependency but also as a compile-time annotation processor.

[3]https://mariadb.org/
[4]https://www.postgresql.org/

4. Implementation

Source code for the developed application can be found on GitHub: https://github.com/GregorSomething/CompileBaseConnector/tree/v1.

4.1 Code generator

A code generator for an application is implemented using AbstractProcessor from the `javax.annotation.processing` package, which is part of the Java compiler module. The processor is configured to recognize only elements annotated with a custom `@Repository` annotation and is set up to generate code compatible with Java version 21. Additionally, the processor is registered with the compiler using the `@AutoService` annotation from Google's AutoService library, which ensures that the appropriate metadata is generated.

Once the processor is invoked during compilation, it scans the source code for elements annotated with `@Repository`. These elements, represented by instances of the Element class (a part of Java's abstract syntax tree), are first validated to ensure they conform to the expected structure. Upon successful validation, the processor proceeds to generate the corresponding code.

The first step in code generation is the resolution of helper types. These types are specified via class references within the `@Repository` annotation. The processor inspects each of these classes to locate methods that match a specific signature: they must be public static, accept parameters of types ResultSet and int, and declare SQLException in their throws clause. Valid methods are cached for use in later stages, particularly for mapping SQL results.

With the helper methods resolved, the processor then generates a constructor for the repository implementation. This constructor accepts a database object and includes initialization logic that executes any SQL statements defined directly in the annotation. This ensures that necessary database operations are carried out at the time of object creation.

Following constructor generation, the processor creates implementations for methods associated with SQL statements. This phase begins with parsing the SQL to identify placeholders, which are then mapped to corresponding method parameters. If a placeholder name matches a parameter exactly, it is directly referenced in the generated code. If a placeholder includes dot notation (e.g., user.id), the processor attempts to resolve it by accessing a field, calling a getter method, or referencing a record component of the matching parameter. Successfully resolved placeholders are embedded as accessors in the output code.

Placeholders that cannot be resolved—often due to the unavailability of Lombok-generated methods at compile time—are left as-is. Once all placeholders are processed, the processor completes the method implementation by generating code that invokes the relevant database method, note that database uses ParamaterizedStatments to prevent SQL injection and for some database engine it is more performant.

After generating the statement methods, the processor proceeds to generate query methods. These methods undergo a similar preprocessing step where placeholders are parsed and removed from the SQL statements. Following this, the processor validates whether the result of the SQL query can be mapped to a corresponding Java object. The system supports mapping results into Optional and List types, provided a suitable mapper exists for their generic type parameter.

By default, the following column types are supported for mapping:

- Java primitive types,
- boxed primitives (e.g., Integer, Double),
- String,
- selection of Java time types: Instant, LocalDate, LocalTime, LocalDateTime.

Users can extend this set by defining their own mappers for additional types. Alternatively, advanced users may choose to access the ResultSet directly and perform manual mapping. However, in such cases, they are fully responsible for correctly managing resource lifecycles, including properly closing the ResultSet.

When a type is not recognized by the system, the processor attempts to treat it as a complex type. In this case, the processor will try to derive a mapper from the SQL query structure itself.

To construct such a mapper, the system uses JSQLParser to analyze the select statements in queries and extract the names or aliases of the selected columns [11]. It then searches for public constructors or static factory methods in the target type that accept the same number of arguments as there are selected columns. Each constructor or method parameter is matched by name to the effective name of the SQL column. If a match is found for every selected element, the processor proceeds to generate a method within the repository implementation that maps a ResultSet into an instance of the target object.

Due to the variability in column ordering and aliases across different queries, these mapping methods are not easily reusable. Therefore, the processor generates a dedicated mapping method for each use case to ensure accuracy and maintainability.

Type validation is not performed against the SQL schema directly. Instead, Java types dictate how data is read from the ResultSet. This design choice avoids the need for full schema introspection at code generation time, which would require the generator to have prior knowledge of all table definitions. Since type conversion is based on Java types, the generated code avoids certain runtime type errors that may occur in other frameworks. For example, in ORM tools like Hibernate, it is possible for a Long object to be incorrectly mapped to a field of type Integer in a generic structure, leading to a ClassCastException at runtime. This exact scenario was encountered by the author in a separate project, motivating the type-safe approach used here.

With most of the repository class now implemented, the processor optionally adds transaction support. This is indicated by having the repository interface extend a special interface called `Transactional<T>`, where `T` is the repository's own type. This generic structure allows the repository to return a transactional version of itself.

A transaction can be initiated from any repository that supports transactions. Furthermore, it is possible to share an active transaction across multiple repositories by explicitly passing the existing transactional context. Transactions in this system are limited to commit and rollback operations.

To implement transactions, a new repository instance is created and bound to the transactional database object. As a result, all operations performed through this instance are subject to the active transaction. When a transaction is closed, meaning its inner database is closed, to release resources like connections, all repositories created with this transaction are now also closed.

4.2 Usage

The developed application is designed to be user-friendly, making the setup process straightforward for end users. The easiest way to understand its usage and output is through an

example. The complete source code for the following example is available in the application's GitHub repository[5].

First, the user must add the developed application as both an annotation processor and a runtime dependency using their build tool of choice, such as Gradle or Maven. The artifact is hosted on the JitPack[6] repository:

- Repository: `https://jitpack.io`
- Group: `com.github.GregorSomething`
- Artifact: `CompileBaseConnector`
- Version: `v1`

Once added, the compiler will automatically invoke the annotation processor during the build process. The processor generates specific repository implementations. However, it must also be included as a runtime dependency because some components—such as internal database classes that use JDBC to communicate with a specific database engine—are not generated. These classes are shared abstractions designed to avoid code duplication across repositories.

To illustrate this, assume the user has two domain objects: User, defined as follows:

```java
public record User(int id, String name, Email email) {}
```

and Email, defined as follows:

```java
public record Email(String name, String domain) {
    @Override
    public String toString() {
        return name + "@" + domain;
    }
}
```

[5] https://github.com/GregorSomething/CompileBaseConnector/tree/1330206ad3c10a34e9af0adfa803bba95214a75f/src/test/java/me/gregorsomething/example
[6] https://jitpack.io/

In this case, one can define a corresponding repository interface named UserRepository. This interface also includes a reference to the EmailTypeReader class, which is used to locate the method responsible for mapping the email field, as the email is represented here as a column-type object. The header of such a repository interface would therefore be as follows:

```
@Repository(value = """
        CREATE TABLE IF NOT EXISTS users (
            id SERIAL PRIMARY KEY,
            name VARCHAR(255),
            email VARCHAR(255)
        );
        """, additionalTypes = {EmailTypeReader.class})
public interface UserRepository extends Transactional<UserRepository> {}
```

To specify the column-type mapper, one can implement an arbitrary class that contains public static methods. These methods return an object of the desired type and take as input a ResultSet (representing the current row) and an integer indicating which column to read from. The previously referenced code would resemble the following:

```
public class EmailTypeReader {
    public static Email from(ResultSet rs, int column)
                                throws SQLException {
        // Implementation
        return ...
    }
}
```

Once the repository is defined, methods for executing database queries can be added using the `@Query` annotation. These methods perform read operations on the database. For example, a method can be added that retrieves and maps a User object based on its integer ID. If the corresponding record is not found, the method throws an exception. The signature of such a method in the interface would be as follows:

```
@Query(value = "SELECT id, name, email FROM users WHERE id = [( id )];",
        onNoResultThrow = true)
User byId(int id);
```

Additionally, a method can be added to retrieve all users from the database:

```
@Query("SELECT id, name, email FROM users;")
List<User> getAll();
```

In addition to read operations, the repository must also support write operations. For this purpose, user can define several methods that do not return a value, those method should be annotated with `@Statement` annotation. For instance, a method for updating a User object can be created. In this example, it is explicitly designed to throw an SQLException as a checked exception, rather than wrapping it in a RuntimeException, generator automatically checks what the user wanted. The method also takes advantage of convenient parameter processing, which allows updates to be performed using fewer parameters on the Java side. The implementation of such a method would be as follows:

```
@Statement(value = """UPDATE users SET name = [( u.name )], email =
                   [( u.email().toString() )] WHERE id = [( u.id )];
                   """)
void updateUser(User u);
```

Once the repository is defined, it must be instantiated to enable access. This requires creating the database instance and then using the RepositoryProvider class to instantiate the desired repository. The corresponding code would appear as follows:

```
void main() {
    DatabaseDetails details = DatabaseDetails.builder()
            .dbURL("127.0.0.1")
            .user("postgres")
            .password("postgres")
            .dbName("test")
            .build();
    Database database = PostgresDatabaseProvider.of(details);
    UserRepository userRepository = RepositoryProvider
            .create(UserRepository.class, database);
    // ...
}
```

The transaction examples, as mentioned earlier, can be found in the application's GitHub repository[7], in the me.gregorsomething.example package within the tests folder.

4.3 Generated code during runtime

The extended version of the example presented in the previous subsection is shown in Appendix 1, and the implementation generated by the generator is in Appendix 2. The application partially uses its own classes to function, one of which is a database abstraction. In Figure 1, the interaction between the generated code and the user is illustrated.

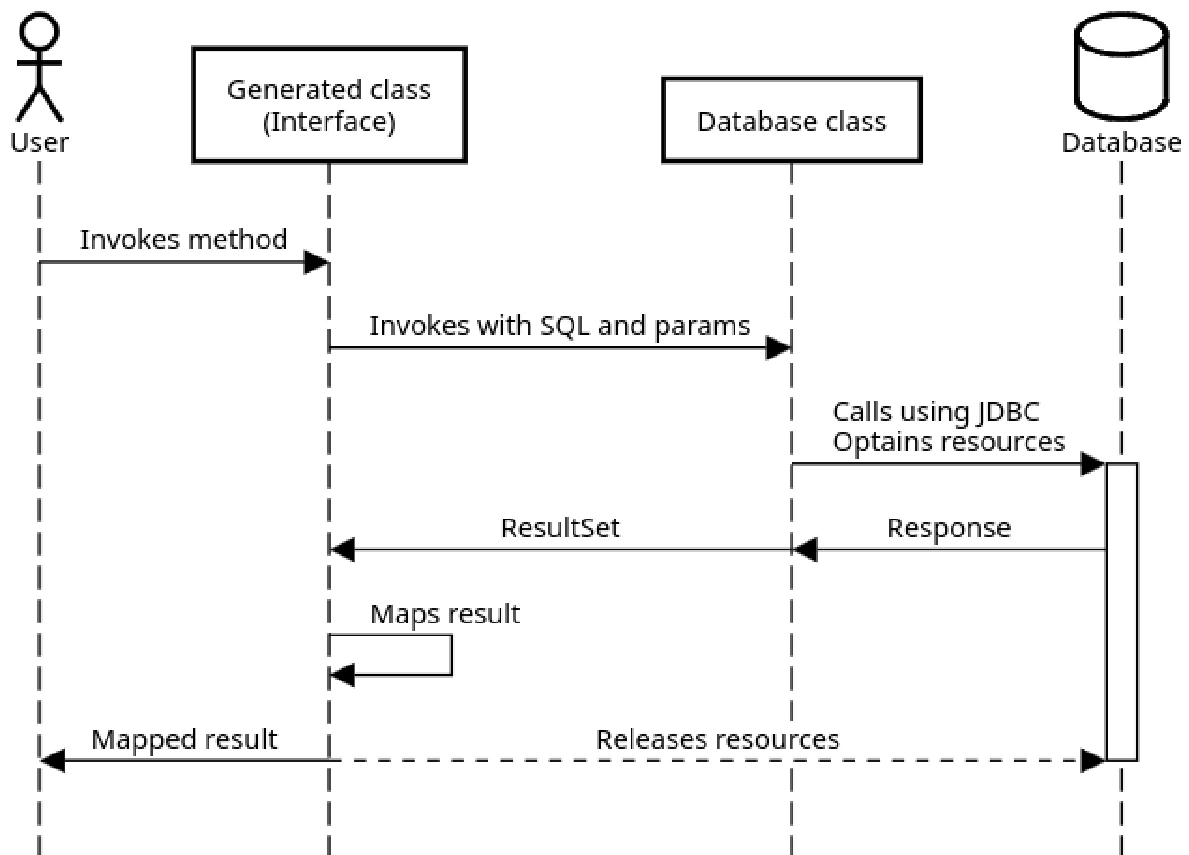

Figure 1. User database interaction in generated repository.

In the generated repository class, it can be seen that when the user calls a method, the generated method delegates the call to the database abstraction with the correct parameters. The database class then communicates with the database using JDBC and returns the obtained result (a

[7] https://github.com/GregorSomething/CompileBaseConnector/blob/1330206ad3c10a34e9af0adfa803bba95214a75f/src/test/java/me/gregorsomething/example/Main.java

ResultSet) back to the generated method. The generated method then maps this result into the desired format. No analytical checks are performed during this process, as the entire mapping is defined at compile time. When the data is returned to the user, the database resources — such as the query and the connection — are released simultaneously.

Using transactions is slightly more complex. To participate in a transaction, a new instance of the repository class is created with a transactional database implementation. All interactions made through this instance are part of that transaction. To manage the transaction's lifecycle, the user is given an object through which the transaction can be controlled (Figure 2). After the transaction is closed — which the user is responsible for, typically using a try-with-resources block — the created repositories can be safely ignored, as the transaction has already released their resources.

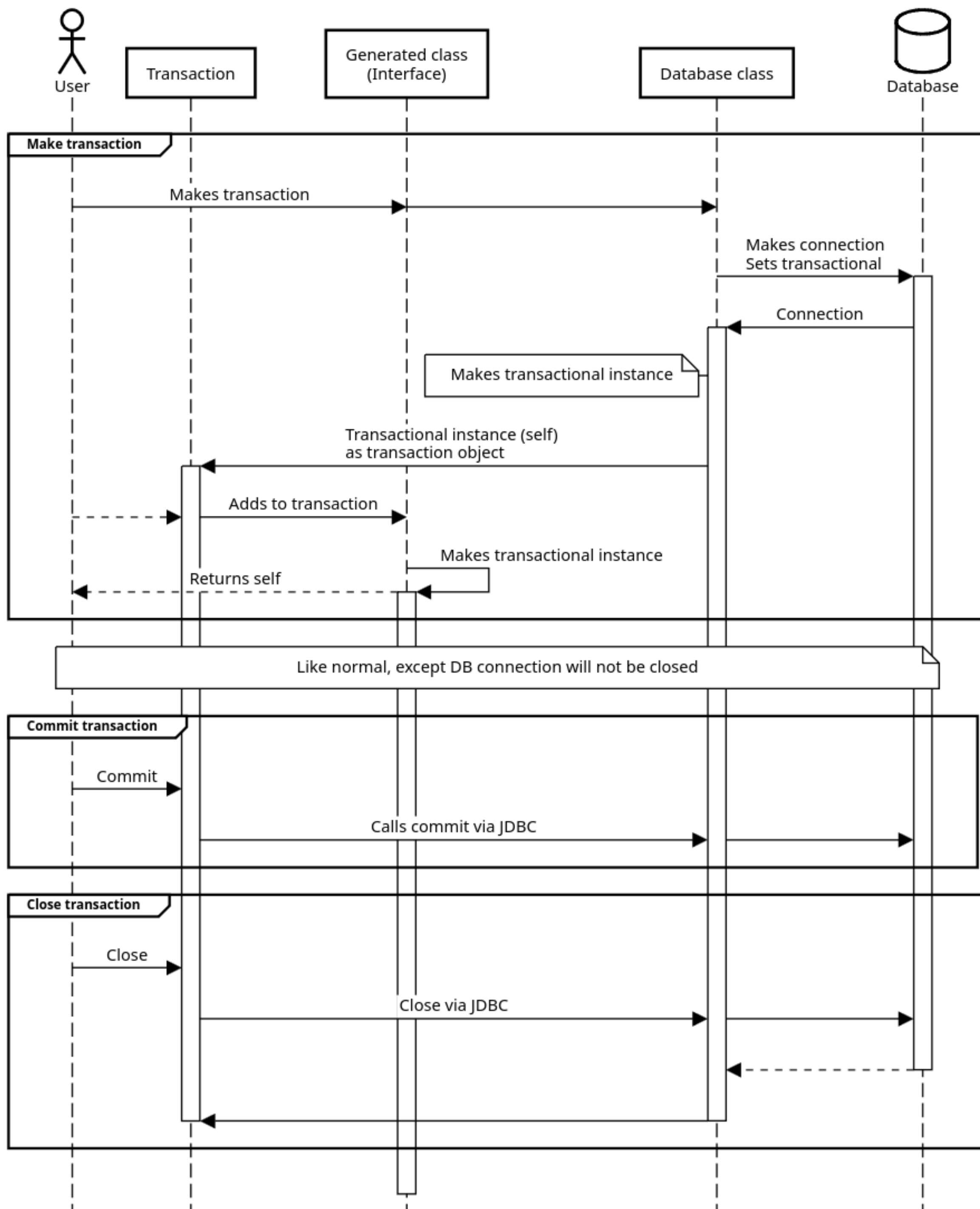

Figure 2. User database interaction in generated repository using transactions.

5. Results

5.1 Difficulties

During the development of the application, one of the main shortcomings identified was the inability to directly modify user-generated code. Annotation processing only allowed for the generation of new code, which made it impossible to augment existing implementations directly. A more elegant approach would have been to provide a more refined mechanism for creating interfaces, rather than relying on an external factory method.

Another major issue was the handling of null values. While setting values in PreparedStatements was relatively straightforward—since all values were inserted as objects, allowing JDBC to infer their types automatically—reading from a ResultSet posed greater challenges. Specifically, primitive-type read methods (e.g., `readInt`) were used instead of object-type methods to avoid unintended type mismatches. However, such methods return default values when encountering `null` values, which can be acceptable for primitives (e.g., `int`) but incorrect when object wrappers (e.g., Integer) are expected. As a result, it was necessary to call the `wasNull` method after each primitive read to determine whether the original value was actually `null`.

In addition to null handling, working with generics proved more complex than anticipated. When transforming data from TypeMirror (an abstract syntax tree representation of types) to TypeElement (which provides more detailed information about specific types) and back again, some information essential for understanding the nature of generic types was lost. This complexity was compounded by the need to interpret generic types more thoroughly. For instance, while a TypeElement might represent a type with type parameters such as `T` and `V`, the corresponding TypeMirror could provide concrete types such as Integer and String. This discrepancy required more sophisticated logic to correctly map and generate the appropriate code based on generic type parameters.

5.2 Future development

Future development could focus on decoupling the code generation logic from the runtime components. This would enable the database engine types to be placed into separate modules, thereby reducing the overall size of the Java archive. Currently, code minimization tools do not function correctly due to the use of dynamic programming techniques. For example, the application accesses the generated code via factory methods that rely on the Reflection API,

meaning that repository classes are not explicitly imported at compile time and thus cannot be easily optimized or pruned.

Another limitation lies in the current inability to reliably deserialize objects that contain nested complex objects as fields. The mapper presently expects an exact number of parameters, making such cases unsupported. A potential improvement would be to allow object creation via a no-argument constructor combined with setter methods, increasing flexibility and compatibility with more diverse object structures.

Additionally, compile-time SQL validation could be introduced to verify query correctness. This could include SQL-to-Java type compatibility checks to prevent situations where, for instance, a developer inadvertently attempts to map a SQL VARCHAR value to a Java Integer field. Such validations would reduce runtime errors and improve overall type safety.

5.3 Comparing to other similar applications

It cannot be claimed that this application represents something entirely new, as there are already numerous existing applications with similar, or even more advanced, functionality. Examples of such applications include Hibernate, EclipseLink, Spring Data JPA, and Spring Data JDBC. While the aforementioned solutions are generally more powerful, it is also possible to create a database-interacting application using the Java Database Connectivity (JDBC) API. Therefore, the developed application can be considered functionally similar to these libraries.

In comparison to Hibernate, which is one of the most prominent and widely used frameworks, it is evident from Hibernate's documentation [12] that it offers significantly more functionality than the developed application. For instance, Hibernate supports the mapping of complex objects and facilitates convenient operations such as saving, modifying, and deleting entities. These capabilities are enabled by the requirement that objects be annotated in advance. This approach can be beneficial for larger and more complex systems, as they can leverage Hibernate's ability to automatically create and update database tables. However, for smaller applications, this feature may become burdensome or unnecessary.

Despite its extensive functionality, Hibernate also introduces a greater theoretical performance overhead. According to an article describing Hibernate's internal mechanisms [13], the framework makes use of proxy objects for lazy loading and various other purposes. One such use is maintaining database sessions and tracking changes. Although proxies are supposed to be the best performant solution for such things, they cannot always be disabled. In scenarios where

it is important for the data object to be fully loaded and in an unaltered state (since the proxy's class may not match the original entity's class exactly), the developed application may be a more suitable choice. Once the object is loaded, it is entirely under the developer's control, with no remaining references held by the framework. Moreover, proxy objects typically introduce more performance overhead compared to their non-proxy equivalents.

Similarly to Hibernate, Spring Data JDBC also enables reading objects from the database, although according to the Spring Data JDBC documentation [14] and Hibernate's documentation [12], it supports fewer database systems. While Spring Data JDBC requires fewer annotations, it still offers more advanced object mapping capabilities than the developed application. However, it too relies on proxy classes for its repository implementation during runtime.

Both of the aforementioned frameworks perform a certain degree of runtime analysis, as they have processed some info during compilation but not all, whereas the developed application avoids this by generating all its code at compile time. This approach ensures that mappers are identified and created before the application starts, allowing for a faster and more reliable startup process. Hibernate and Spring Data JDBC, on the other hand, must dynamically create classes during application startup due to their reliance on proxy objects. Hibernate, in particular, also creates proxy instances during data retrieval for lazy loading purposes, which can further impact performance.

A somewhat lesser-known tool in this context is Apache Torque, an Object-Relational Mapping (ORM) framework that generates its entire needed classes during the compilation phase. This functionality can be configured via Maven. According to the Apache Torque wiki [15], the framework requires a detailed XML file that fully describes the data model in order to function. In addition to generating database-related classes, the code generator also produces the corresponding data classes. As a result, it is necessary either to use the generated code directly or to copy it manually to avoid issues where code analysis tools report missing classes simply because the generator has not yet been invoked. This generator shares several similarities with developed application but differs in that it assumes the availability of a comprehensive data model and mandates the use of the classes it produces. While it is possible to customize the generated code through template modifications, according to the documentation, this process is not particularly convenient.

Like the previous solution, the developed application generates all necessary logic into the class at compile time, which enables type checking by the compiler and allows the Just-In-Time (JIT)

compiler to optimize the code during runtime. Since all analysis is performed at compile time, the runtime is reduced to merely executing the query and mapping the results.

In addition to comparing with the aforementioned frameworks, the developed application can let developer access directly Java Database Connectivity (JDBC) API. As the application is implemented on top of JDBC, it somewhat inherits its functionality by exposing enough of JDBC's interface to allow developers to utilize raw JDBC operations if needed. For instance, the application provides access to the ResultSet, enabling the execution of queries without automatic mapping. Transaction management is also implemented using JDBC.

6. Conclusion

The objective of this bachelor's thesis was to develop a Java application capable of generating the classes necessary for interacting with a database. This objective was achieved through the use of annotation processing and custom annotations. One of the goals of the generated code was to minimize the use of the Reflection API, which was successfully accomplished; reflection was used only to access the generated classes at runtime.

However, the application was unable to map more complex structures due to limited SQL introspection capabilities. Enhancing this functionality through annotations would have compromised the simplicity of using the application. Nevertheless, this is a shortcoming that could likely be addressed in future work.

Despite its limitations, the significance of the application lies in its ability to generate all necessary code at compile time, thereby avoiding any runtime operations beyond query execution and result mapping. Many other frameworks may write metadata at compile time, but they typically do not fully generate classes and instead rely heavily on dynamic programming techniques.

References

[1] Lakatos D. What is Java Persistence API (JPA)? 2024. https://medium.com/@lktsdvd/what-is-java-persistence-api-jpa-2763d0c1ee73 (12/08/2024).

[2] Ovidiu M. T. and Michael K. Is Java Reflection Bad Practice? 2024. https://www.baeldung.com/java-reflection-benefits-drawbacks (12/08/2024).

[3] Oracle. Module java.sql. https://docs.oracle.com/en/java/javase/21/docs/api/java.sql/module-summary.html (05/12/2024).

[4] Oracle. Java SE support roadmap. 2024. https://www.oracle.com/my/java/technologies/java-se-support-roadmap.html (12/08/2024).

[5] Horstmann S. C. Pattern Matching for switch (JEP 441). 2022. https://javaalmanac.io/features/typepatterns/ (12/08/2024).

[6] The Apache Software Foundation. Feature Summary. 2024. https://maven.apache.org/maven-features.html (12/08/2024).

[7] Gradle Inc. Gradle and Maven Comparison. 2024. https://gradle.org/maven-and-gradle/ (12/08/2024).

[8] Oracle. Java Platform, Standard Edition & Java Development Kit Version 21 API Specification. 2024. https://docs.oracle.com/en/java/javase/21/docs/api/index.html (12/08/2024).

[9] Google. AutoService. 2024. https://github.com/google/auto/blob/main/service/README.md (12/08/2024).

[10] Palantir Technologies. JavaPoet. 2024. https://github.com/palantir/javapoet (12/08/2024).

[11] JSQLParser. Java SQL Parser Library. 2022. https://jsqlparser.github.io/JSqlParser/ (05/12/2024).

[12] King G. An Introduction to Hibernate 6. 2025. https://docs.jboss.org/hibernate/orm/6.6/introduction/html_single/Hibernate_Introduction.html (05/01/2025).

[13] Tuấn A. T. How Hibernate ORM Works Under the Hood. 2025. https://medium.com/tuanhdotnet/how-hibernate-orm-works-under-the-hood-af26547136ba (05/01/2025).

[14] Broadcom Inc. Spring Data JDBC. 2025. https://docs.spring.io/spring-data/relational/reference/jdbc.html (05/04/2025).

[15] The Apache Software Foundation. Apache Torque 6.0 homepage. 2024. https://db.apache.org/torque/torque-6.0/index.html (05/10/2025).

Appendices

1. Repository interface code

Full code for repository example.

```
package me.gregorsomething.example;

import me.gregorsomething.database.Transactional;
import me.gregorsomething.database.annotations.Query;
import me.gregorsomething.database.annotations.Repository;
import me.gregorsomething.database.annotations.Statement;

import java.sql.SQLException;
import java.util.List;
import java.util.Optional;

@Repository(value = """
        CREATE TABLE IF NOT EXISTS users (
            id SERIAL PRIMARY KEY,
            name VARCHAR(255),
            email VARCHAR(255)
        );
        """, additionalTypes = {EmailTypeReader.class})
public interface UserRepository extends Transactional<UserRepository> {

    @Query(value = "SELECT id, name, email FROM users WHERE id = [( id )];",
            onNoResultThrow = true)
    User byId(int id);

    @Query("SELECT id, name, email FROM users WHERE id = [( id )];")
    Optional<User> byIdOptional(int id);

    @Query("SELECT id, name, email FROM users;")
    List<User> getAll(int id);

    @Query(value = "SELECT count(id) FROM users;",
            defaultValue = "-1")
    int getUserCount();

    @Statement(value = "INSERT INTO users (name, email) " +
```

```
            "VALUES ([( name )], [( email.toString() )]);")
    void addUser(String name, Email email) throws SQLException;

    @Statement(value = "UPDATE users SET name = [( u.name )], email =
        [( u.email().toString() )] WHERE id = [( u.id )];")
    void updateUser(User u);
}
```

2. Generated repository interface

Generator output code for the repository defined in appendix 1.

```
package me.gregorsomething.example;

import java.lang.Override;
import java.lang.RuntimeException;
import java.lang.String;
import java.sql.ResultSet;
import java.sql.SQLException;
import java.util.ArrayList;
import java.util.List;
import java.util.NoSuchElementException;
import java.util.Optional;
import lombok.SneakyThrows;
import me.gregorsomething.database.Database;
import me.gregorsomething.database.Transaction;
import me.gregorsomething.database.Transactional;

public class UserRepositoryImp implements UserRepository,
        Transactional<UserRepository> {
  private final Database database;

  @SneakyThrows
  public UserRepositoryImp(Database database) {
    this.database = database;
    this.database.execute("CREATE TABLE IF NOT EXISTS users (\n"
            + "    id SERIAL PRIMARY KEY,\n"
            + "    name VARCHAR(255),\n"
            + "    email VARCHAR(255)\n"
            + ");\n");
```

```
}

private User byIdMapper(ResultSet rs) throws SQLException {
  int id = rs.getInt(1);
  String name = rs.getString(2);
  Email email = EmailTypeReader.from(rs, 3);
  return new User(id, name, email);
}

private User byIdOptionalMapper(ResultSet rs)
      throws SQLException {
  int id = rs.getInt(1);
  String name = rs.getString(2);
  Email email = EmailTypeReader.from(rs, 3);
  return new User(id, name, email);
}

private User getAllMapper(ResultSet rs)
      throws SQLException {
  int id = rs.getInt(1);
  String name = rs.getString(2);
  Email email = EmailTypeReader.from(rs, 3);
  return new User(id, name, email);
}

@Override
public void addUser(String name, Email email)
      throws SQLException {
  this.database.execute("INSERT INTO users (name, email)
          VALUES (?, ?);", name, email.toString());
}

@Override
public void updateUser(User u) {
  try {
    this.database.execute("UPDATE users SET name = ?, email = ? WHERE
          id = ?;", u.name(), u.email().toString(), u.id());
  } catch (SQLException e) {
    throw new RuntimeException(e);
  }
```

```
}

@Override
public User byId(int id) {
  try (ResultSet rs = this.database.query("SELECT id, name,
          email FROM users WHERE id = ?;", id)) {
    if (!rs.isBeforeFirst()) {
      throw new NoSuchElementException();
    }
    rs.next();
    return this.byIdMapper(rs);
  } catch (SQLException e) {
    throw new RuntimeException(e);
  }
}

@Override
public Optional<User> byIdOptional(int id) {
  try (ResultSet rs = this.database.query("SELECT id, name,
          email FROM users WHERE id = ?;", id)) {
    if (!rs.isBeforeFirst()) {
      return Optional.empty();
    }
    rs.next();
    return Optional.ofNullable(this.byIdOptionalMapper(rs));
  } catch (SQLException e) {
    throw new RuntimeException(e);
  }
}

@Override
public List<User> getAll(int id) {
  try (ResultSet rs = this.database.query("SELECT id, name,
          email FROM users;", id)) {
    if (!rs.isBeforeFirst()) {
      return List.of();
    }
    List<User> list = new ArrayList<>();;
    while (rs.next()) {
      list.add(this.getAllMapper(rs));
```

```
      }
      return list;
    } catch (SQLException e) {
      throw new RuntimeException(e);
    }
  }

  @Override
  public int getUserCount() {
    try (ResultSet rs = this.database
            .query("SELECT count(id) FROM users;")) {
      if (!rs.isBeforeFirst()) {
        return -1;
      }
      rs.next();
      var tmp = rs.getInt(1);
      if (rs.wasNull()) {
        return -1;
      }
      return tmp;
    } catch (SQLException e) {
      throw new RuntimeException(e);
    }
  }

  @Override
  @SneakyThrows
  public Transaction getNewTransaction() {
    return new Transaction(this.database.getConnection());
  }

  @Override
  public UserRepository asTransactional(Transaction transaction) {
    return new UserRepositoryImp(
        transaction.getTransactionalDatabase()
    );
  }
}
```

License

I, Gregor Suurvarik,

1. grant the University of Tartu a free permit (non-exclusive licence) to reproduce, for the purpose of preservation, including for adding to the digital archives of the University of Tartu until the expiry of the term of copyright, my thesis Generating Code for Database Classes in Java at Compile Time, supervised by Stefan Hermann Kuhn;
2. grant the University of Tartu a permit to make the thesis specified in point 1 available to the public via the web environment of the University of Tartu, including via the digital archives, under the Creative Commons licence CC BY NC ND 4.0, which allows, by giving appropriate credit to the author, to reproduce, distribute the work and communicate it to the public, and prohibits the creation of derivative works and any commercial use of the work until the expiry of the term of copyright;
3. am aware of the fact that the author retains the rights specified in points 1 and 2;
4. confirm that granting the non-exclusive licence does not infringe other persons' intellectual property rights or rights arising from the personal data protection legislation.

Gregor Suurvarik

15/05/2025