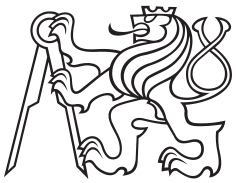
Bachelor's Thesis



Czech Technical University in Prague

F3

Faculty of Electrical Engineering
Department of Computer Science

Domain Object Change Tracking Module

Miroslav Holeček

Supervisor: Ing. Martin Ledvinka, Ph.D.
May 2022

BACHELOR‘S THESIS ASSIGNMENT

I. Personal and study details

Student's name: **Holeček Miroslav** Personal ID number: **492242**

Faculty / Institute: **Faculty of Electrical Engineering**

Department / Institute: **Department of Computer Science**

Study program: **Software Engineering and Technology**

II. Bachelor’s thesis details

Bachelor’s thesis title in English:

Domain object change tracking module

Bachelor’s thesis title in Czech:

Modul sledování změn doménových objektů

Guidelines:

1. Become familiar with the topics of change tracking in domain objects and semantic web technologies.
2. Compare different ways of tracking changes to domain objects (e.g, storing complete snapshots, change vectors). Discuss their suitability in different use cases.
3. Design a domain object change tracking module for information systems based on Semantic Web technologies.
4. Implement the designed solution. The implementation should support change tracking on objects compatible with the JOPA library.
5. Evaluate the solution by integrating it with the TermIt terminology editor.

Bibliography / sources:

[1] D. Allemang, J. Hendler, Semantic Web for the Working Ontologist: Effective Modeling in RDFS and OWL, Morgan Kaufmann, 2011
[2] Hibernate Envers, https://hibernate.org/orm/envers/, 2021
[3] E. Gamma, R. Helm, R. Johnson, J. Vlissides, Design Patterns: Elements of Reusable Object-Oriented Software, Addison-Wesley, 1994

Name and workplace of bachelor’s thesis supervisor:

Ing. Martin Ledvinka, Ph.D. Knowledge-based Software Systems FEE

Name and workplace of second bachelor’s thesis supervisor or consultant:

Date of bachelor’s thesis assignment: **02.02.2022** Deadline for bachelor thesis submission: **20.05.2022**

Assignment valid until: **30.09.2023**

Ing. Martin Ledvinka, Ph.D. Supervisor’s signature	Head of department’s signature	prof. Mgr. Petr Páta, Ph.D. Dean’s signature

III. Assignment receipt

The student acknowledges that the bachelor’s thesis is an individual work. The student must produce his thesis without the assistance of others, with the exception of provided consultations. Within the bachelor’s thesis, the author must state the names of consultants and include a list of references.

.	
Date of assignment receipt	Student’s signature

Acknowledgements

Most of all, I would like to thank my supervisor Martin Ledvinka for exemplary guidance and leadership, my family and friends for extensive support, and my colleagues from the faculty not only for all the joyful times discussing our theses.

Declaration

I hereby declare that the presented thesis is my own work and that I have cited all sources of information in accordance with the Guideline for adhering to ethical principles when elaborating an academic final thesis.

I acknowledge that my thesis is subject to the rights and obligations stipulated by the Act No. 121/2000 Coll., the Copyright Act, as amended, in particular that the Czech Technical University in Prague has the right to conclude a license agreement on the utilization of this thesis as a school work under the provisions of Article 60 (1) of the Act.

In Prague in May 2022.

Abstract

This bachelor's thesis describes the analysis, design and beginning of the implementation of an application module allowing domain object change tracking in the context of the semantic web and linked data. One of the key requirements is compatibility with the Spring framework and JOPA, a database library developed by the KBSS group at Czech Technical University's Faculty of Electrical Engineering. In spite of that, the module has to be reusable in other environments, which constrains the permissible degree of vendor lock-in. An approach utilizing change vectors was chosen due to their low storage demands. Relational databases were selected as the backing storage solution based on superior performance and their alternative, NoSQL stores, lacking benefits exploitable by the module. An imperative API was designed and implemented in a prototype written in the Java programming language, focusing on extensibility. The module's usability was evaluated by integrating it into an existing application, TermIt.

Keywords: change tracking, database auditing, data versioning, module, tool, library, change vector, change record

Supervisor: Ing. Martin Ledvinka, Ph.D.

Abstrakt

Bakalářská práce popisuje analýzu, návrh a implementaci aplikačního modulu sloužícího pro sledování změn doménových objektů v kontextu sémantického webu a propojených dat. Klíčovým požadavkem na modul je jeho kompatibilita s frameworkem Spring a databázovou knihovnou JOPA, který byl vyvinut výzkumnou skupinou KBSS na Fakultě elektrotechnické ČVUT. Zároveň však je třeba vyhnout se přílišné platformní závislosti, aby modul byl lehce přepoužitelný i v jiných prostředích. Pro ukládání změn byl využit koncept změnových vektorů, zejména díky prostorové nenáročnosti. Jako datové úložiště byly pro svou výkonnost a univerzalitu zvoleny relační databáze. Navržené API je imperativní a implementováno v jazyce Java prototypem s důrazem na rozšiřitelnost. Použitelnost modulu je zvalidována funkční integrací do existující aplikace TermIt.

Klíčová slova: sledování změn, nástroj, modul, knihovna, verzování dat, auditování, změnový vektor

Překlad názvu: Modul sledování změn doménových objektů

Contents

Figures

Tables

Chapter 1

Introduction

Modern information systems often require access to both the current and historical versions of the data they store. One of the primary reasons for this is the need for a continuous auditing mechanism, i.e. a way to monitor changes of a given record step by step. Some applications opt to implement this functionality ad hoc, but for easier reusability, such functionality may and shall be implemented in the form of an independent application module or library.

The goal of this bachelor's thesis is to design the inner workings of such a reusable change-tracking application module and provide its implementation to be used in conjunction with the JOPA[1] database library developed by the Knowledge-Based Software Systems research group at the Czech Technical University in Prague [7] as part of the TermIt[2] project by the same authors.

Several approaches exist that can be used for storing such changes, the most prominent of which are change vectors and whole object snapshotting. Storage and access complexity are discussed for both of these. So is the selection of a database backend of the module, based on performance and general fitness for the purpose. Lastly, an imperative API is proposed with regard to reusability and ease of integration, and implemented in a prototype of the module, which is then integrated with the TermIt application.

[1] Java OWL Persistence API; available from `https://github.com/kbss-cvut/jopa`.

[2] A SKOS-compliant terminology manager based on Semantic Web technologies; available from `https://github.com/kbss-cvut/termit`.

Chapter 2

Change storage format

As previously mentioned, two main approaches come to mind when designing the storage format of a change tracker: change vectors and full object snapshots. Let us discuss both of these variants in greater detail and explain how to perform the two main tasks at hand:

1. viewing a change log (differences between revisions);
2. viewing complete revisions.

We will use the following model, depicted in Figure 2.1, for all examples in this chapter.

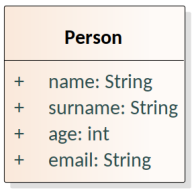


Figure 2.1: Example class model.

2.1 Change vectors

A change vector (also sometimes—more generally—known as "change record") is a minimal representation of a difference between multiple objects (two in our case). In general, it might look like Figure 2.2[1]:

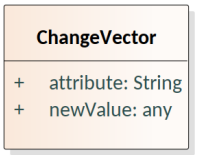


Figure 2.2: Minimalist change vector.

[1] Of course, in real world applications, in addition to identifying the object they correspond to, change vectors would serve a much better purpose if they included other attributes like a timestamp of the change and an identifier of the user responsible for it. Here, we are just illustrating the basic concept.

An important fact that we have to note here is that **a single change vector will only be able to record a change a single attribute**. If an object differs in multiple attributes between two revisions, a change vector will have to be created for every single changed attribute. This is by design: if we wanted a vector to be able to hold multiple attribute changes, we would considerably increase the complexity of designing a proper database schema, if applicable. For example, if a relational database were used as the vector store, a column would have to be introduced for every single attribute being change-tracked. This column would then have to be nullable[2] to accommodate for all vectors representing revisions not differing in this attribute. However, that would cause potential inconsistencies with attributes that themselves are of nullable types (i.e. non-mandatory attributes). This would also essentially reduce (or eliminate altogether) the space savings when comparing to snapshots.

As we can assume from the aforementioned Figure 2.2, the primary advantage of using change vectors is their simplicity, and, in turn, their low storage space requirements. They do have a significant drawback, though: the complexity of retrieving an object at a specific revision increases with every single further change to it.

To illustrate, let us take a look at how an object could change in time. First of all, we will define the starting object: Jacob Appleseed, age 23, instance of the previously introduced *Person* class—illustrated in Figure 2.3—and the first change to it: Jacob decides to change his official name to *Jake*. This change is recorded using the change vector shown in Figure 2.4.

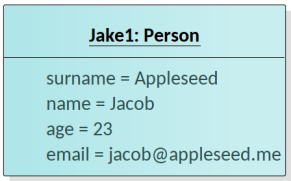


Figure 2.3: Base version of our Person instance.

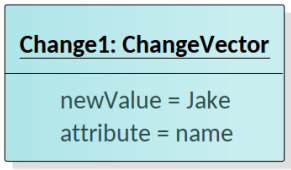


Figure 2.4: First change vector, updating the *name* attribute.

To reconstruct the object at the latest revision at this point (i.e. extract the object *after* the first change), we would have to apply the first change vector. Right now, this looks fine – only one setter method has to be executed so far. However, when we add further changes, like updating Jake's age (Figure 2.5) or his surname (Figure 2.6), the number of setter methods required to run

[2] Allowing (and in some cases even expecting) null values.

(and, prior to that, vectors retrieved from the database) before we receive the latest revision (Figure 2.7) grows linearly.

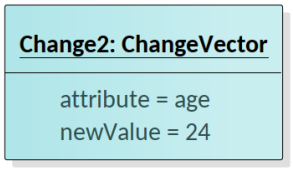


Figure 2.5: Second change vector, updating the *age* attribute.

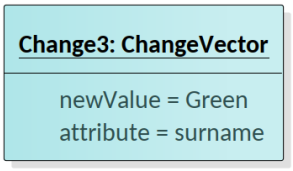


Figure 2.6: Third change vector, updating the *surname* attribute.

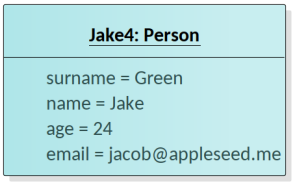


Figure 2.7: The most up-to-date version of our Jake object.

Sure, it is possible to optimize this approach of applying changes by only querying the change vector store for distinct *attribute* values, but in case of complex objects with a large number of attributes, this approach still leaves the burden of combining the change vectors on the *read* operation. That is not ideal because even though historical data is important, it is usually not as important as the most recent version(s) of the data, and thus, performance should be optimized in favor of the read operation *on the latest revisions*.

One option to do so in quite an elegant way and move the burden to the *update* operation would be to use **negative change vectors**: instead of storing the original version of an object and incrementally updating it by applying each change vector, the most recent version of the object would be stored and change vectors would store the *previous* version of the changed data (attribute). Such a change vector class might then look like depicted in Figure 2.8. Our previous changes would then be recorded as shown in Figure 2.9.

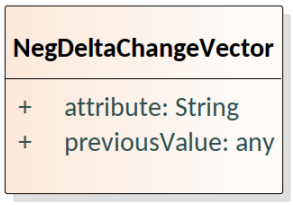


Figure 2.8: Minimalist negative-Δ vector class schema.

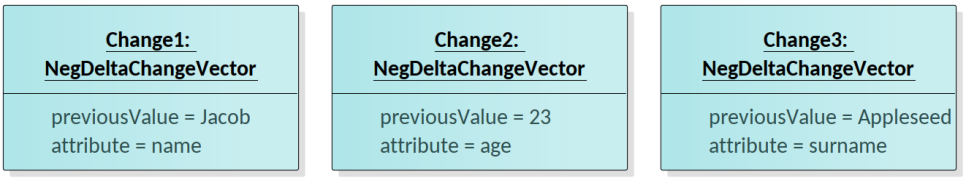


Figure 2.9: Negative-Δ vectors for all previously described changes.

The vector-saving operation can then be implemented as follows (also in code in Listing 1):

1. Iterate over all fields and discover differences.

2. Extract the original values of the fields and create change vectors.

3. Save the change vectors to the database.

```java
import java.util.List;
import java.util.ArrayList;
import java.util.stream.Collectors;
import java.util.reflect.Field;

public class ChangeTracker {
    private final PersonDao personDao;
    private final VectorDao vectorDao;

    public void compare(Object orig, Object newer) throws Exception {
    List<Field> changed = new ArrayList<>();
    for (var f : orig.getClass().getFields()) {
      if (!f.get(orig).equals(f.get(newer))) {
        changed.add(f);
      }
    }

    List<NegDeltaVector> vectors;
    vectors = changed
                    .stream()
                    .map(field -> new NegDeltaVector(
                        field.getName(),
                        field.get(orig)
                    ))
                    .collect(Collectors.toList());

    personDao.save(newer);
    vectorDao.saveAll(vectors);
  }
}
```

Listing 1: Simple implementation using Java Reflection API.

2.1.1 Data type ambiguity

A new burden may be introduced by employing change vectors of any kind as the format for storing changes: attributes of a given entity may very well be of different types. Even in our simple *Person* class, there are string and integer attributes. This means that our *ChangeVector* class's *previousValue* (or *newValue*, if not considering negative change vectors) can differ by its type even between vectors concerning the same entity. While this may not cause any trouble with some NoSQL stores (especially the ones not enforcing a schema), it does pose an issue when considering other data storage solutions, such as relational databases. We will be returning to this issue in the relevant following chapter.

2.1.2 Viewing a change log

A change log is very straightforward to manipulate when using change vectors. It is simply a matter of loading the change vectors that concern the desired

object and sorting them appropriately, for example based on the timestamp of the change, which is likely to be included within the change vector itself.

2.1.3 Viewing complete revisions

Reconstructing objects at a given revision, however, proves to be a slightly more difficult task when working with change vectors, especially negative ones. If we worked with basic forward change vectors, it would simply be a question of querying the change vector store for all change vectors concerning the desired object, preferably with a query to only select the most recent (up to the historical timestamp we want to view the object at) vector for each distinct attribute, and then applying those vectors to a base revision of the object. It would indeed require the base revision to be stored as well, which does not seem very logical or useful, as previously discussed in this chapter; now, let us take a look at what the situation would be with negative change vectors.

With them, we have to reverse our thinking slightly, but the core logic applied is still the same: query the vector store for all vectors created *after* (or at) the desired timestamp. We can again benefit from only selecting change vectors for distinct attributes but we have to be careful to always select the *least* recent ones. Then we can apply them in a similar as in the previous paragraph, and we can use the current version of the object as the base revision, which allows us not to be concerned with its storage.

2.2 Snapshots

Snapshots, on the other hand, are full copies of the versioned objects, including possibly repeating values in all unchanged fields. So, for the Figure 2.1 model, and for the same changes as in the previous section, we would store the snapshots illustrated in Figure 2.10 in the database.

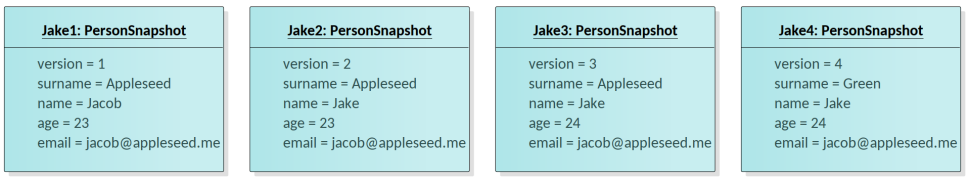


Figure 2.10: The snapshots saved for all previously described changes.

We can see that the storage space complexity is significantly higher. One can argue whether this still is a cause for concern in 2022 but I believe it is the approach using more space that requires a justification, not the one more economic with it. Furthermore, this approach can essentially be implemented on the database-interfacing layer of any application at will and does not necessarily warrant extraction into a separate module with the sole focus on change tracking.

This does not stand to say that snapshots do not have suitable real world applications; quite on the contrary: an application with emphasis on viewing full historical versions of objects or with the requirement of quickly switching between multiple revisions of an object can benefit from using them very much. A good example of that is WordPress[3]: it saves a new *revision* of a post whenever it is updated, allowing the content creator to revert to an older revision rather effortlessly. The revisions are stored in the same database table as the current versions of the posts themselves[4].

2.2.1 Viewing a change log

Because snapshots are whole revisions of objects, a change log can only be created by comparing two objects using the same method that is used when creating change vectors: iterating through their attributes and comparing the old value of each one to the new value. Snapshots do hold an advantage in this regard: multiple attributes can be changed in a single revision; when using vectors, only one can.

2.2.2 Viewing complete revisions

In contrast to change vectors, a snapshot-based versioning system is rather straightforward when trying to access entire older revisions. It does not require any kind of a base revision of the object to be stored, as snapshots are in fact instances of the object's class themselves, so they can directly be used in their place[5]

2.3 Verdict

Based on arguments presented in sections 2.1 and 2.2, change vectors (more specifically, negative ones) were chosen to be used for the purposes of the module as their lower storage requirements outweigh the simplicity of operating with snapshots, partly also because the primary use case of the module is to be integrated with TermIt, a terminology management tool[6], which seems to benefit from viewing changes individually (first, in a log-like view, and then being able to reconstruct entire revisions on demand) more than from viewing the complete revisions immediately and then calculating their differences.

We have only discovered two advantages of snapshots: (i) being able to contain several changed attributes in a single revision (mentioned in Section 2.2.1), which can be easily replicated with change vectors by grouping them based on matching (or differing by very little) timestamps, and (ii) not

[3] An open source web content management system, often used to create blog-style websites.

[4] `https://github.com/WordPress/WordPress/blob/5.8/wp-includes/post.php#L105`

[5] Barring the extra version-marking fields – *version* in our example in Figure 2.10.

[6] Also developed at KBSS, available from `https://github.com/kbss-cvut/termit`.

having to deal with type ambiguity as with vectors on the database level, which is further discussed in Section 3.2.1.

Chapter 3

Data storage solution

Now that we have decided what storage format we are going to use, we can move forward and take a look at our options of actually storing the data.

As mentioned in the introduction, the module is envisioned to be used in a context of linked data. There already has been extensive research in the area of storing linked data using several database approaches and technologies. Most such experiments arrive at the conclusion that relational databases are superior in performance compared to both triplestores and other, more standard NoSQL solutions, such as the graph database Neo4j. Of course, these experiments are based on querying *raw*, unversioned RDF data [8][9]. In the case of this module, which operates with relatively simple-to-index change vectors, the performance difference should be even more pronounced.

However, raw query performance is usually not the biggest selling point for NoSQL databases of any kind, and neither is it as important a consideration factor in the design of the versioning module. Therefore, let us take a closer look at some of the non-relational database solutions, their most significant advantages and review their fitness for our use case.

3.1 NoSQL databases

Each category of these database solutions has its advantages and ideal use cases where they excel, which we will discuss based on the sources [10][11]. In general, one of the key advantages of NoSQL is the fact that most of such databases do not have a fixed/rigid schema and thus enable developers to prototype against them rapidly.

Key-value stores, illustrated in Figure 3.1, are the simplest NoSQL databases. They store a *value*, an arbitrary object without complex introspection by the database engine, under a *key*—an identifier. They best suit applications where entire objects can be manipulated based on just their identifiers; i.e. where no further access to the stored object is required. Probably the best known example of this category is Redis[1], an in-memory key-value store often used for caching and session data storage, especially in conjunction with Node.js applications[2].

[1] Homepage: `https://redis.io/`.

[2] Based on data from `https://www.npmjs.com/package/redis`.

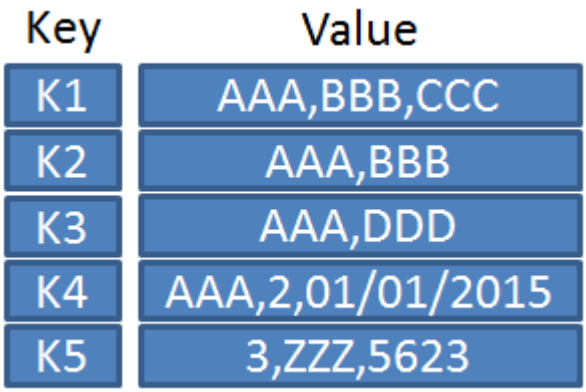


Figure 3.1: Illustration of the key-value concept. Source: [1]

Document databases, such as MongoDB or Elasticsearch, generally store self-describing documents in formats like JSON or XML. An example document is provided in Listing 2. Each document has an unique identifier (line 2 of the example), equivalent to a primary key in relational databases. These documents are then grouped into collections, functionally equivalent to tables in relational databases, which can then be queried based on complex criteria that may, like in SQL, be based on all attributes of the stored objects. They are thus suitable for storing documents with semi-lax[3] structure and querying them in a wide variety of ways. MongoDB also offers a notable advantage of native geographical sharding and data replication capabilities.

```
1  {
2      "_id" : ObjectId("5d1a729568758c3d46fd0fda"),
3      "from" : {
4          "lon" : 14.3915194,
5          "lat" : 50.1003628,
6          "address" : "Evropská, Prague, Czech Republic"
7      },
8      "date" : ISODate("2019-06-21T00:00:00.000Z"),
9      "track" : {
10         "type" : "Feature",
11         "geometry" : {
12             "coordinates" : [[ 14.391523, 50.100334 ]],
13         },
14         "properties" : {
15             "description": null
16         }
17     }
18 }
```

Listing 2: Example MongoDB document describing a part of a public transport route.

[3]Not entirely lax, as the queries still require the document structure to be stable to some degree, but also not exactly strict, as many attributes of the documents might be missing in some documents or contain further de-normalized data such as subdocuments or arrays.

Wide-column stores operate with the concept of *column families* as a replacement for tables (or collections). These families are groups of similar rows, which are in turn groups of columns (often ones that are frequently queried together) and have a unique key that identifies them. Not every row has to contain all columns the other documents do, plus columns may contain non-scalar values such as sets. That is what makes this category fit for applications with non-normalized, but at the same time flat, data, such as application and event logging and blog content management. The best known wide-column store may perhaps be Apache Cassandra. An illustration of such a store is featured in Figure 3.2.

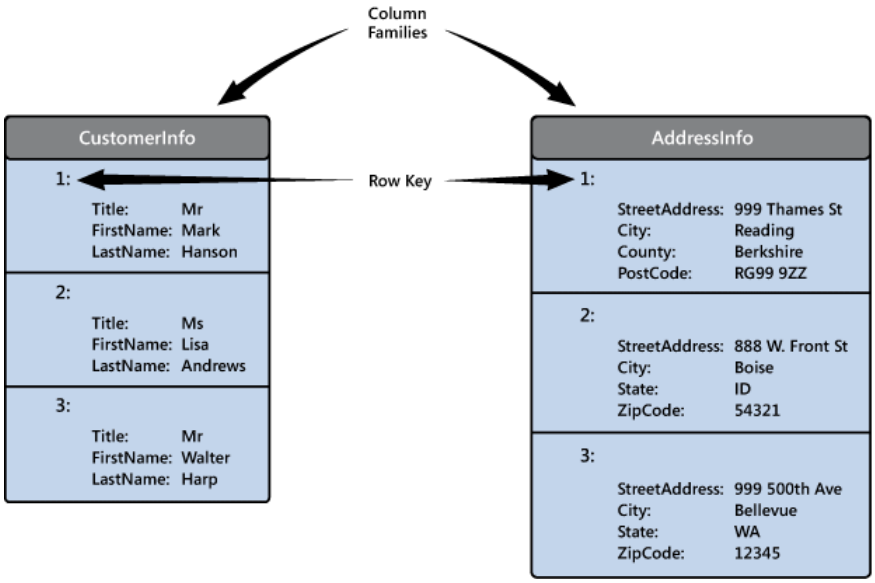


Figure 3.2: A wide-column store example. Source: [2]

Graph databases draw a lot of attention in the context of social networking, where they accurately represent the relationships (arcs or edges) between individual nodes (vertices). They may operate with both directed and undirected graphs, where both nodes and relationships may feature any number of particular attributes. They are applicable to where efficiency of graph traversal is of concern, as well as applications that might benefit from similarity-based queries. Neo4j is the most widely used graph database at the time of writing[4], and an illustration of a graph in it is displayed in Figure 3.3.

Triplestores (also known as RDF stores), represented by Apache Jena or Eclipse RDF4J (formerly Sesame), are perhaps the most interesting mention in this section. As the name suggests, they are based on RDF triples which are statements consisting of a subject, predicate and object. These statements may also be viewed as graphs: an edge (predicate) between two vertices (subject and object). They are queried using SPARQL, the de facto standard query language for RDF, which itself is the standard for storing linked data.

[4]Based on Maven Central repository artifact usage statistics, retrieved in May 2022: `https://mvnrepository.com/open-source/graph-databases`.

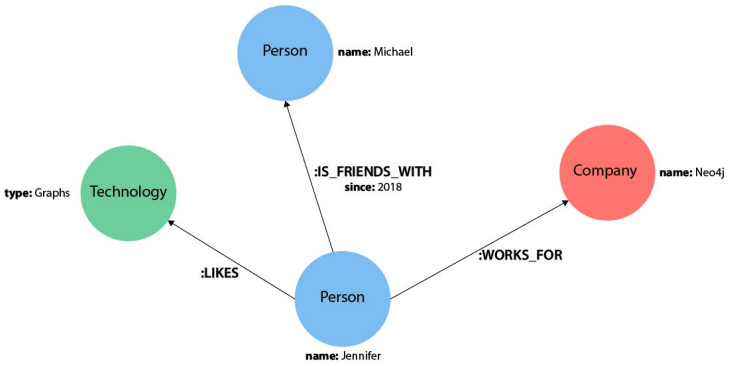


Figure 3.3: Illustration of a Neo4j graph. Source: [3].

Figure 3.4 contains a visualization of a triplestore.

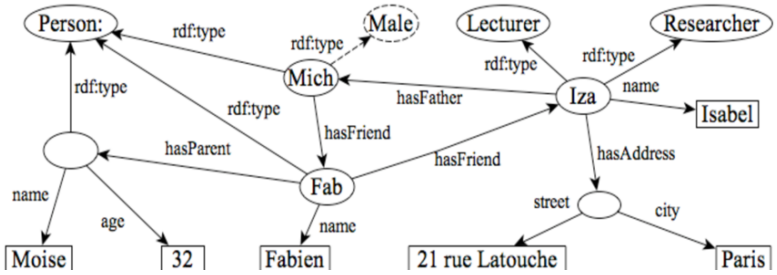


Figure 3.4: Example of a triplestore (in space). Source: [4].

3.2 Relational databases

After the short introduction to NoSQL databases, we can come to the conclusion that in our case, none of the presented solutions offer any significant benefits (such as improved usability), except preventing the type ambiguity issue (discussed in Section 2.1.1) by not enforcing a rigid schema like relational databases do. Based on that and the generally inferior performance of NoSQL stores [9], relational databases were chosen as the more fitting approach instead.

3.2.1 Tackling type ambiguity

Let us elaborate on the problem of dealing with an attribute that can have different types of values as is the case with change vectors. A good starting point is outlined in a Stack Overflow answer to a question dealing with a similar issue, where two solutions are proposed [12]:

The first is to define a *sparse table*: create columns for each of the possible types and only use the applicable column for each row (vector), inserting `NULL` values elsewhere. This approach allows two options when defining the table:

either create columns for all types, regardless of whether they are actually used in any entity class in the application, or only create columns for the used types. The second option is prone to breaking when the entity classes are updated, so the first one is much safer, but also much more wasteful in terms of storage space.

To mitigate that, a sparse table can be avoided altogether by creating multiple tables for the vectors, each having a value column of a distinct type. Null values would not be required, but the complexity of querying such a database would increase significantly – joins over a vast amount of tables (one for each type) would be required when looking up changes to even a single entity class. Combined with the fact that the enumeration of types would have to be exhaustive (in order not to limit the attribute types an application can use), this solution is still far from an ideal.

The second solution proposed is to introduce a new column to describe the attribute's type. The value of the attribute is then saved in a generic format, such as JSON[5], and stored in either a `JSON` or a `TEXT` column. It then has to be converted to the original type when being read from the database by the versioning module. This even allows exotic data types to be used in the application using the module, provided there is an implementation to convert them to and from JSON. Therefore, this solution seems appropriate in our case. The change vector class will then be defined as depicted in Figure 3.5.

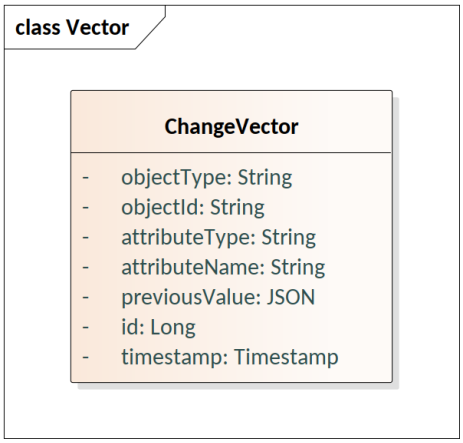


Figure 3.5: Change vector class definition.

To implement this solution in a future-proof way, the module has to be able to process data of the most common types (such as strings and basic numeric types), as well as provide a means for developers to extend the module's serialization and deserialization capabilities by implementing their own logic for processing custom data types.

[5]JavaScript Object Notation, a modern common data interchange format specified in `https://datatracker.ietf.org/doc/html/rfc8259`.

Chapter 4

API and internal design

Since the module is primarily being developed for usage in conjunction with JOPA[1] and TermIt[2] (both developed at KBSS, the latter based on Spring Boot), it has to be implemented in a JVM language. According to the TIOBE index[3], the most popular of these (as of January 2022) are Java, Groovy and Kotlin, placing 3rd, 17th and 29th, respectively. Out of those three, I have the broadest experience with Java, and while I believe that both Groovy and Kotlin have their applications, I chose to develop the module in Java due to personal preference and the smoothest interoperability—based on sharing the same toolchain without introducing a Kotlin compiler, for example—with both of the aforementioned projects.

Nevertheless, it has been an important design decision to make the module as reusable as possible. For this reason, runtime dependencies are kept to a minimum and SOLID[4] principles are applied.

As a means to further reduce platform dependencies of the module, the entire API is designed to be imperative. While this may go against the convention of current Spring Boot-based applications, where developers might expect to simply annotate their entity classes and have the module automatically monitor instances for changes, it greatly reduces the potential platform lock-in.

In order to allow implicit change tracking in their applications, developers can leverage technologies such as Jakarta Interceptors[5] or aspect-oriented programming, for example using the AspectJ extension for Java[6]. TermIt itself utilizes AspectJ, even specifically for this purpose[7].

Along with the imperative API goes an annotation marking a class as an entity whose changes can be tracked: the `Audited` class-level annotation. The

[1] `https://github.com/kbss-cvut/jopa`

[2] `https://github.com/kbss-cvut/termit`

[3] `https://www.tiobe.com/tiobe-index/`

[4] Five design principles intended to make software designs more understandable, flexible and maintainable: the Single-responsibility, Open-closed, Liskov substitution, Interface segregation and Dependency inversion principles. [13]

[5] Formerly named Java EE Interceptors, described in the Java EE 6 Tutorial – *Overview of Interceptors*: `https://docs.oracle.com/javaee/6/tutorial/doc/gkigq.html`.

[6] AspectJ project homepage: `https://www.eclipse.org/aspectj/`.

[7] As seen in: `https://github.com/kbss-cvut/termit/blob/v2.11.2/src/main/java/cz/cvut/kbss/termit/aspect/ChangeTrackingAspect.java`

change tracker will not look for changes between instances of a class which does not have this annotation. Entity classes can also contain fields whose changes should not be recorded. Examples of such fields include generated values, such as arbitrary and synthetic identifiers, and other transient fields, such as inferred attributes of objects, like a person's current age computed from their birth date, which itself is stored in the database (and change-tracked). For this purpose, another annotation is introduced: `IgnoreChanges`. It is to be used on the field level. Any changes in fields annotated with `IgnoreChanges` will not cause change vectors to be created.

What is also an imperative concept is the fact that the client application will be responsible for providing a database connection. This allows, for example, applications already using JPA to leverage the same persistence configuration for change-tracking purposes by simply passing an `EntityManager` instance to the `JpaStorageStrategy` constructor, instead of either having to configure it again for the change tracker or relying on a third-party solution, such as a dependency injection framework, which, as already mentioned, would only cause the module to become unnecessarily vendor- or platform-locked.

To allow simple integration with enterprise Java applications, the module should expose a primary point of interaction—the `ChangeTracker` class. It serves as a facade[8] for the client application to interface with the change-tracking functionality without having to deal with the internals of the module.

When creating a `ChangeTracker` instance, two dependencies are required in its constructor. Both are strategies, which we will take a closer look at in the following section.

4.1 Strategies

While the design choices discussed in the previous chapters are binding for the initial prototype implementation, we wanted to create an easily extensible and reusable module. What this meant was that even though a SQL database had been decided as the backing storage solution and JOPA entity classes as the primary subjects of tracking, a developer using the module had to have the option to use both a different entity persistence platform (for business entities) and a vector storage solution of their own liking.

Thankfully, this issue is tackled relatively frequently in software development and there is a conventional way of approaching it – the *Strategy* design pattern [14], described in detail in Figure 4.1. In the module, it is used for the two aforementioned cases: vector storage (`StorageStrategy`) and entity class management (`EntityStrategy`). This way, a developer may create their own implementations of both strategies and adapt the module to their environment, such as versioning JPA entities and storing the changes in a triplestore. Needless to say, the module does contain a JOPA entity strategy and a JPA storage strategy reference implementation.

[8] Facade is a structural software design pattern that provides a simplified interface to a library, a framework, or any other complex set of classes. [14][15]

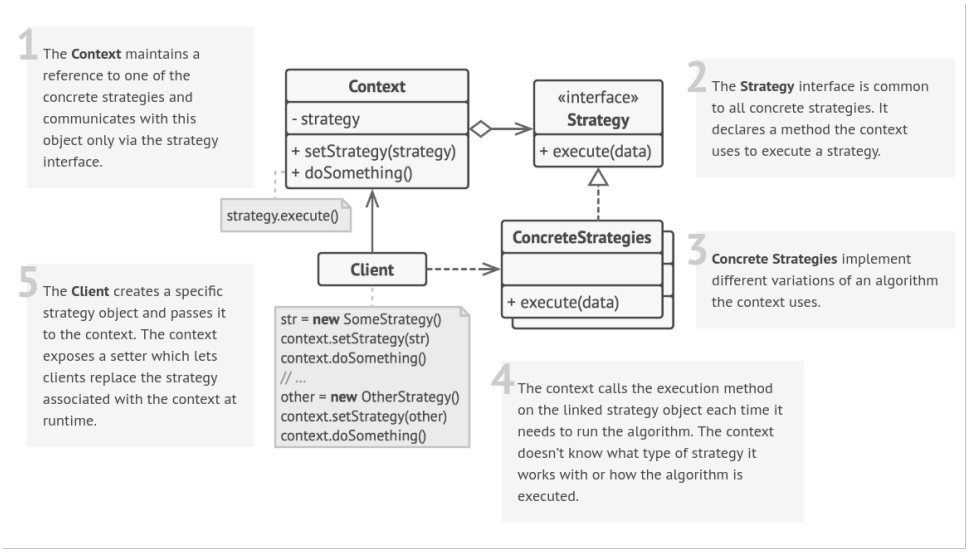


Figure 4.1: A visualization and description of the Strategy design pattern. Source: [5].

Furthermore, a substantial part of both the entity- and storage-related code may be reused in implementations targeting either a different entity framework (such as JPA) or a different solution for storing change vectors (like a triplestore), the `BaseEntityStrategy` and `JsonBasedStorageStrategy` classes are introduced as abstract implementations of the strategy interfaces.

4.2 Vector API

In order to properly isolate the client application from handling change-tracking internals, such as change vector format, exposing some JSON-specific fields, such as `attributeType`[9], is not very desirable.

An important consideration that has not yet been taken into account is the authorship of changes. It can be assumed that a vast majority of the module's applications will benefit from (and possibly even require) such functionality (an example being any change-tracking related to audits). Based on the requirements of TermIt, as the prototypical client application, a single field containing the identifier of the user responsible for the change should be enough. Such a field, provisionally named `authorId`, has to be added to the vector model.

During the alteration of the existing vector class, we considered whether there is any other metadata that may be useful to include as part of change vectors. In fact, we considered the option of introducing a generic key-value map of user-defined data as part of the vector, so that applications could store their specific metadata about entities without us having to alter the vector model ever again. However, this would require designing a "proper"

[9]Shown in Figure 3.5.

way of persisting such maps and, combined with the fact that we could not come up with any other metadata, we agreed that the `authorId` field was the sounder approach. To include this as part of the vector generation, there was only one way of providing an author identifier (not taking into consideration vendor lock-in-heavy options like obtaining the user from a Spring Security Context): drilling it down all the way from the first facade method.

Because the module obviously still has to process JSON data internally, a `JsonChangeVector` class is introduced, but it is not part of the public API and should not be directly used by the client application.

4.3 Exceptions

The last part of the module's API are exceptions. All exceptions actively generated (thrown) by the change-tracking module inherit from the common `ChangeTrackingException` class[10], which itself extends `RuntimeException`. This means that all module-created exceptions in are unchecked. This is in alignment with the general trend of preferring them over checked ones, except in justified special cases, such as recoverable conditions, which are not the case here [16].

4.4 API class diagram

Together, the classes discussed in this chapter make up the public API of the module. A class diagram depicting the application's public classes and interfaces is in Figure 4.2.

4.5 Example interaction

An example interaction with a client application is presented in the sequence diagram in Figure 4.3.

[10] With the exception of `NullPointerException`, which may be triggered by several null checks.

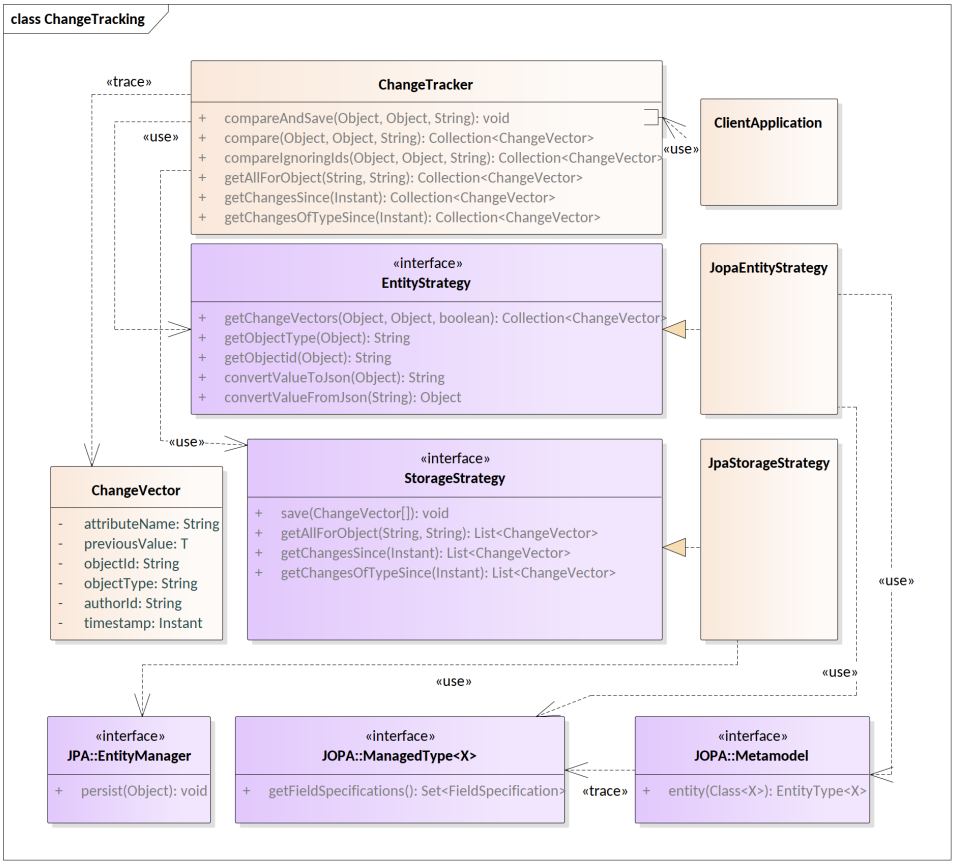


Figure 4.2: Application class diagram.

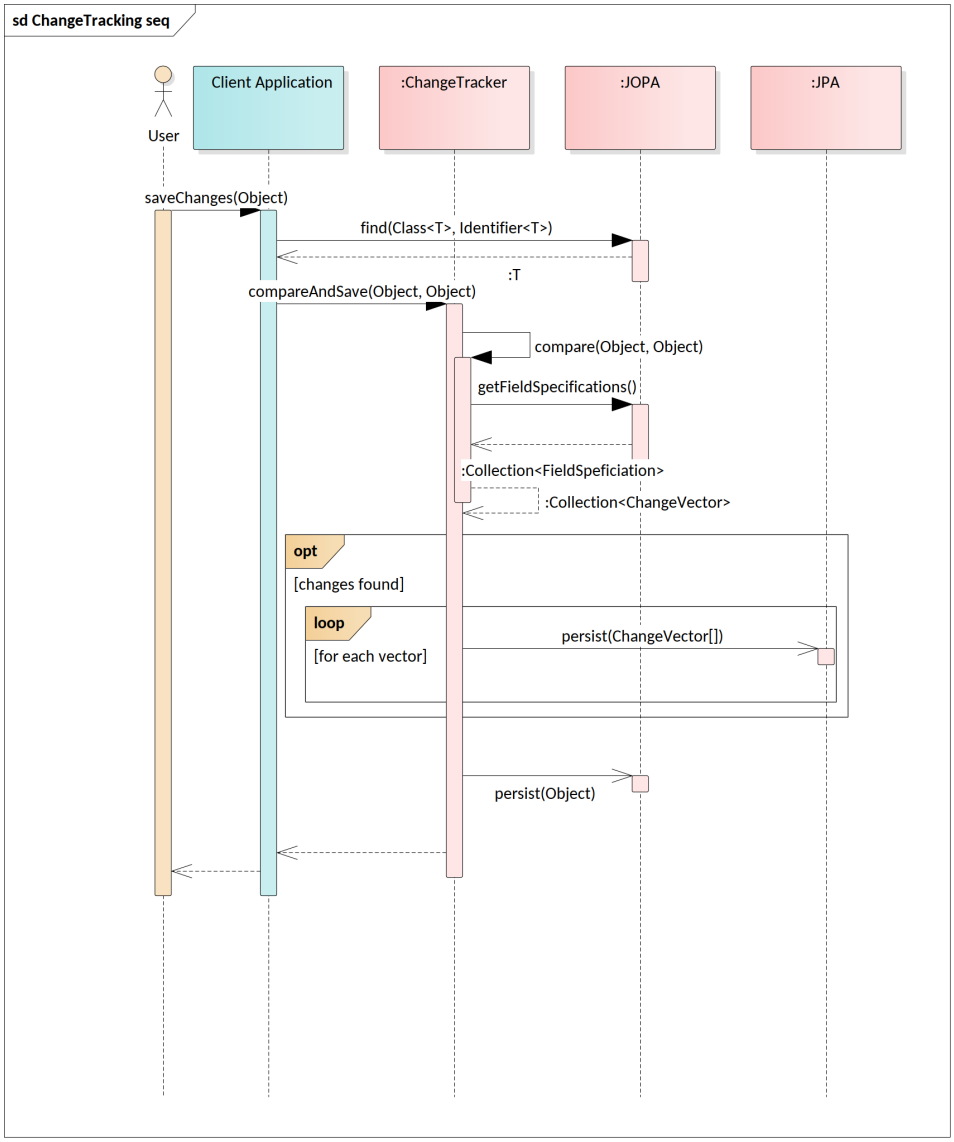


Figure 4.3: Example interaction with a client application.

Chapter 5

Implementation

During development, some unexpected issues were encountered and handled. This chapter focuses on these issues and explains the methods and rationale used for their resolution.

5.1 Type ambiguity handling

Unconstrained generic types cannot be used for entity attributes with JPA because the underlying implementation would not be able to accurately map them to a corresponding column type, not even if the column type is explicitly defined to be JSON: while this approach may be idiomatic for primitives such as strings or numeric types, there is no predefined implicit behavior for more complex types, such as collections or even other entities.

Consequently, the Jackson[1] library was introduced as a module dependency in order to imperatively handle the JSON conversion. Its implicit behavior allows mapping JSON values not just to platform primitives, but also collections and complex objects, such as maps or class instances. Custom deserializers may also be registered[2].

5.2 Handling collections

One of the considerations undertaken while implementing the module was how to handle collections. Due to the limitations of JSON, all collections (including subclasses of the `Collection` interface as well as arrays) have to be serialized into JSON arrays. This, at first glance, does not seem like much of an issue: as long as we have another column to store the type in, we should be able to deserialize those JSON arrays directly into the original type. While this works out of the box for arrays, i.e. storing the array type in the type column, such as `java.lang.String[]` for strings, it is not as straightforward with Java collections. To specify the element type of a collection, type parameters are used when writing code, for example `List<String>`. But

[1] Available from `https://github.com/FasterXML/jackson`.

[2] The change tracker API allows Jackson's behavior to be configured by optionally passing a custom `ObjectMapper` instance to the constructor of `JpaStorageStrategy`.

because the module internally uses the `getName()` method of the `Class` class to extract type names, and this method does not return type parameters[3], it would require a custom type name extraction implementation in order to properly annotate the saved data. I did not want to go down this route because it would mean designing a universal format and then manipulating strings, such as composing them to resemble the generic type invocations in code, which would require a recursive algorithm similar to this one:

1. If the output has not yet been initialized, initialize it as an empty string.
2. Get the "root" class name (considering the `List<String>` example from before, saved in the `myList` variable, this would mean calling `myList.getClass().getName()`, which will return the name of the `List` implementation that was really used[4]—that may for example be `"java.util.ArrayList"`) and append it to the output string.
3. Check if there are any type parameters using the Reflection API[5]. If so, append the type parameter opening character (<). If not, return.
4. Retrieve the type parameter class and recursively start this algorithm again on the retrieved class. This step is notoriously complex, if not outright impossible without modifying the calling code[6].
5. Append the type parameter closing character (>) and return the string.

As we can see, this method of manually composing the strings is rather complex and prone to breakage. Instead, I opted to use a more primitive approach: *a*) serialize all collections into JSON arrays, using their respective source classes without extracting type arguments, *b*) deserialize arrays into collections based on the target collection type, and *c*) not handle "edge cases" (such as nested collections).

One of the drawbacks of this decision is that complex types in collections will likely fail to deserialize properly. I fully expected this to prove insufficient while integrating with TermIt. Surprisingly enough, though, no such issues have arisen in the integration process; therefore, I consider this solution sufficient for the purposes of the initial module release. I do, however, believe that a more complete and resilient solution should be implemented as part of future work.

[3] And neither do its sibling methods `getSimpleName()` and `getCanonicalName()`.

[4] In case of anonymous classes defined inline, such as `new List<String>() { /* methods' implementation */}`, this method call returns `"$0"`. If those were to be supported, this step would require a new method, traversing the class hierarchy until it finds a collection class/interface this anonymous class inherits from.

[5] `var t = ((ParameterizedType)myList.getClass().getGenericInterfaces()[0]);` And then `var hasAnyTypeParameters = t.getActualTypeArguments().length > 0;`

[6] Due to compile-time type erasure, as noted in a lengthy discussion on Stack Overflow: `https://stackoverflow.com/q/1901164`.

5.3 JSON columns

Another interesting revelation was that while MariaDB, PostgreSQL and H2[7] all support a `JSON` column type, they handle insertions (and `SELECT`s) differently. My initial understanding was that columns defined as such were to be used as standard `VARCHAR` or `TEXT` ones, with the added capability of executing JSON-based queries on the stored values and validating the input strings. This, however, is only the case for MariaDB, which, unlike other databases (including, perhaps surprisingly, MySQL, with which it shares a significant part of its history), introduces the `JSON` column type as an alias for `LONGTEXT` (and automatically adds a `CHECK` constraint calling its `JSON_VALID` function to validate the document before saving it into the column). If one creates a `JSON` column in PostgreSQL, strings containing JSON cannot be directly and implicitly inserted with JPA from a `String` field annotated with a JSON column type definition[8]: a `PSQLException` is thrown when attempting the insertion.

H2 does allow insertions with the same configuration as mentioned (at least simple ones – complex `INSERT`s were not properly tested), but when reading from the database, there may be unexpected behavior. For example, imagine saving a string. Converting a raw string into JSON means enclosing it in double quotes (and possibly escaping some entities, such as embedded double quotes), e.g. `"like this"`. After persisting and reading this value back from the database, I expected to receive the original, double-quote-wrapped, string again. What I got instead was `"\"like this\""` – a string wrapped within another string. A quick search on the Internet[9] explained the issue. The solution was to replace the `JSON columnDefinition` (the type definition) with a `TEXT` one. This resulted in all the databases mentioned properly storing and retrieving (unchanged) strings, which was the goal here.

5.4 Supporting entities with differing identifiers

Perhaps surprisingly, during the module's conception, we have not thought of the scenario where one the user may want to compare two entity instances differing in their identifier. While this does not make sense in the original use case of tracking changes across an entity's lifetime, it may sometimes be useful to just "diff" (find differences between) two different entities. An example of where this may be desired is comparing two user accounts when deduplicating database records.

[7] An in-memory database regarded as a default when testing JPA-related functionality in Java applications. Homepage: `https://www.h2database.com/html/main.html`.

[8] `@Column(columnDefinition = "JSON")`

[9] Namely, a Stack Overflow answer (`https://stackoverflow.com/a/59679109`) based on this GitHub issue: `https://github.com/h2database/h2database/issues/2389`.

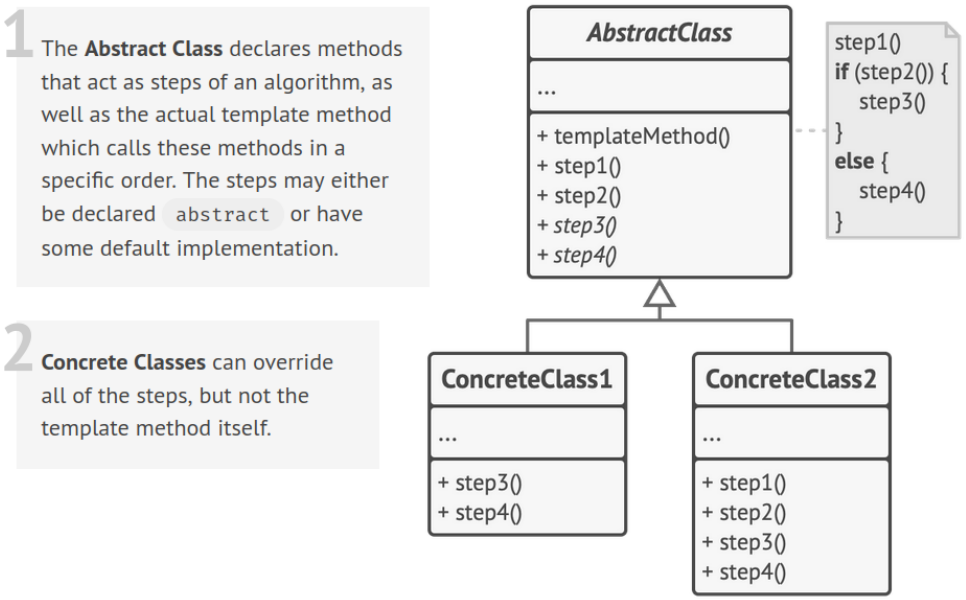


Figure 5.1: A visualization and description of the Template Method design pattern. Source: [6].

5.5 Separation of strategy code

As mentioned in Section 4.1, in order to allow easy extension of the module, I have extracted a significant part of the code that I envision may be reused for other `EntityStrategy` and `StorageStrategy` to `BaseEntityStrategy` and `JsonBasedStorageStrategy`, respectively. Both are abstract classes. `BaseEntityStrategy` mostly (with one exception mentioned in the next paragraph) just declares methods that are likely to be useful when implementing a new EntityStrategy, while `JsonBasedStorageStrategy` also includes definitions of methods handling the conversion to and from JSON.

Both of these abstract strategies make sparing use of the *Template Method* design pattern [14], where a method implementation in the abstract class calls an abstract method (which then is implemented in the subclasses). The design pattern is better explained in the Figure 5.1 illustration. In the case of `JsonBasedStorageStrategy`, the `save(ChangeVector<?>...)` method calls the abstract `saveVector(JsonChangeVector)` method, which is implemented the JPA subclass. In `BaseEntityStrategy`, `getObjectType(Object)` calls the abstract `getTypeName(Class<?>)`.

5.6 Design disagreements and discussions

During development, I came up with several changes to both the API and the inner design of the module that I thought might improve its usability. Some of them, such as allowing the comparison of entities differing in IDs

(discussed in Section 5.4), were accepted by my supervisor, while some were rejected or heavily modified. Similarly, sometimes my supervisor came to me with a feature request that we then discussed and came to the conclusion that implementing it is not viable. This section strives to point out the most interesting discussions we have had in order to include provide a better view of what considerations have taken place as part of the development process.

- `ChangeTracker.compareAndSave(T, T, String)` returning a boolean, representing whether any changes have been found. This was rejected because it could potentially create a false impression that the client application does not have to store a new revision of the object because the change tracker could not find any changes, even though there could have been some, perhaps because the changed fields have been annotated with `@IgnoreChanges` but persisting them in the primary database does make business sense.
- Not treating `null` the same as an empty collection (`Set`, for example): strictly mathematically speaking, a singleton (also known as a 1-tuple or a unit-set – a set containing a single element) is not equal to the element itself [17]. However, in practice (at least in the case of TermIt), there is no difference in the meaning, thus the values are to be treated as the same (which is the module's current behavior).

5.7 Source code and license

The module is released under the MIT open-source license with the source code publicly available at `https://github.com/kbss-cvut/change-tracker`.

Chapter 6

Evaluation

In order to verify that the created change-tracking module complies with the expected behavior and is usable in a real-world application, it was necessary to both extensively test it and integrate it with an existing application.

6.1 Unit and integration tests

The project includes automated tests based on the de facto standard Java test framework – JUnit 5[1] (the latest major release at the time of writing). Extended mocking funcionality is provided by the Mockito library. These components are both open source and provided under the Eclipse Public License and MIT License, respectively.

Unit tests follow the *UnitOfWork_StateUnderTest_ExpectedBehavior* naming principle[2] and cover most non-trivial functionality of the module. Following a similar concept used in JOPA, I have separated integration tests from the module's implementation into an isolated project which includes a Maven dependency on Spring Boot, Spring Data JPA, a JPA-compatible database driver (H2) and KBSS's JOPA implementation. The demo application itself does not contain any functionality, but the intended purpose of the separate project is to provide a clean environment for running JUnit integration tests covering a JOPA entity's lifecycle and saving the changes into a real (albeit in-memory) database. Interim assertions are placed after every logical step.

6.2 Integration with TermIt

To validate the module's fitness for real use, it was to be integrated with TermIt, a terminology manager also developed at KBSS[3]. Prior to the conception of this project, TermIt had already had a change-tracking module, but a reusable solution was desired so that change-tracking could be effortlessly implemented in other applications and projects of the research group,

[1] `https://github.com/junit-team/junit5`

[2] Which is the long-term industry standard (note the year): `https://osherove.com/blog/2005/4/3/naming-standards-for-unit-tests.html`.

[3] Available from `https://github.com/kbss-cvut/termit`.

while also centralizing its development to ease bug-fixing and new feature implementation.

The previous change-tracking implementation included a significant amount of automated tests. Owing to that, the integration was easy to be thoroughly tested by retrofitting a portion of the original solution's tests. While a number of those are redundant with the tests included in the change-tracking project itself, for the most part, they were retained for the sake of completeness. Some tests had to be removed as they focused on unit-testing parts of the original module that do not have equivalents in the newly developed library. As it stands now, many of the retrofitted tests are not correctly named – their class and method names were preserved even though they may have been named based on classes and methods either that no longer exist or were renamed to reflect the current structure of the change-tracking code. In the future, we expect to remove or refactor these tests, but for the sake of evaluation, they are left in this form.

While replacing the original change-tracking service in TermIt, all of its original functionality has been preserved, with the sole exception of what was referred to as `PersistChangeRecord` – a record of a database entity (instance) being created. While discussing this with my supervisor, we arrived at the conclusion that it would also be viable to create a record when deleting an entity from the database. Both of these features are not possible to be implemented concisely while retaining the current change vector schema: each vector, as currently designed, has a (mandatory) field that identifies the entity attribute which has changed. In the case of the entire entity changing (which is, semantically speaking, what happens when creating or deleting), such an identification is not possible. It sure is possible to make this field nullable and just not provide a value whenever it is not applicable, but that does break the cleanness of the code and schema design. Therefore, after considering the importance of creation/deletion change vectors, we have decided against implementing this functionality in the initial version of the change-tracking module, and we consider it a part of potential future work.

The source code of TermIt with the change-tracking module integrated is publicly available online as a fork of the KBSS TermIt repository on GitHub, at `https://github.com/HolecekM/termit`. The modified code including the integration is all placed in the `feat/changetracking_dev` branch.

Chapter 7

Conclusions and future work

We have established that negative-delta vectors are the most fitting format of storing changes to domain objects for our use case, based on low storage requirements and relative simplicity of operation with them. Furthermore, based on cited performance benchmarks and the lack of benefits from using a NoSQL store, we have decided to use relational databases as the data storage solution for the module. A basic yet extensible imperative API has been designed with SOLID principles in mind and a prototype supporting basic data types has been created along with a set of test scenarios demonstrating the functioning of the module.

The module has successfully been designed, developed and integrated with the TermIt terminology manager and has replaced its original change-tracking solution. As such, all assignment goals have been fulfilled and this bachelor's thesis can be labeled as complete.

7.1 Future work

While the thesis itself can be considered complete, the development of the module is far from finished: several "nice-to-have" (and generally lower-priority) features have been discussed and placed on the roadmap, such as change vectors, or more generally, records, of object creation and deletion, first mentioned in Section 6.2. These are considered viable additions to the current capabilities of the change tracker. However, implementing them will likely require a significant overhaul of the change vector model.

As part of this overhaul, the whole concept of negative change vectors is to be revisited: a major motivation for introducing it was reducing the complexity of retrieving the latest object revision from the database by decreasing the required amount of retrievals and setter method calls. This does not take into account the fact that the tracked entities themselves and change vectors are, at least in the case of the prototype's integration with TermIt, not stored in the same database, and in order to retrieve the latest revision of an entity, it can be directly loaded from its respective database (a triplestore) and no vectors have to be processed at all.

Section 5.2 mentions that the current handling of Java collections may prove insufficient when the module is used with an application requiring complex

objects in collections to be compared (for the purpose finding differences between them). We already have a solution in mind: instead of saving the original collection type in the `attributeType` column, we may instead save the array type of the element (i.e. `"String[]"` for both string lists and sets)[1]. When deserializing such a value, `List.class` would be passed to Jackson, no matter the original collection. The rationale behind always using a `List` instead of accurately reflecting the original type is that the JSON array does preserve the order of elements and care should be taken not to lose it in the conversion process. In the case of sets, a new `Set` can always be created from a `List` instance without losing any information in the process. Doing the opposite—first deserializing into a `Set` and then potentially converting to a `List`, if required—introduces the risk of both losing the original element order and losing some elements themselves, as a `List` can contain several elements "equal to each other" (whose `equals(Object)` methods return `true` when called on each other), while a `Set` cannot[2]. This approach breaks native array type compatibility, so it still must be refined in order to accommodate for all possible use-cases.

One of the next steps for this project is also the implementation a triplestore-based `StorageStrategy`. This is mainly to allow TermIt to store both change vectors and the change-tracked entities in the same database.

Another potentially useful capability that the module currently lacks is object reconstruction. This is discussed in greater detail in Chapter 2, but because the change tracker is not responsible for managing the entity itself, it will likely require a significant amount of code to be written on the side of the client application. The change tracker already exposes the `getAllForObject(String, String)` method, which allows the user to retrieve all change vectors pertaining to an object identified by its type and identifier, which can serve as the basis for the object reconstruction. However, we want to examine opportunities to extract as much of this logic as possible into the change-tracking module, as the code on the side of the client application is likely to be very similar, no matter the client application.

[1] This is a simplification for the sake of readability. In reality, `String[].class.getName()` (which is the approach used to obtain class names) would return `"[Ljava.lang.String;"`.

[2] As per the `Set` interface documentation: `https://docs.oracle.com/javase/8/docs/api/java/util/Set.html`.

Glossary

Acronym	Meaning
API	Application Programming Interface
JOPA	Java OWL Persistence API
JPA	Java Persistence API
JSON	JavaScript Object Notation
JVM	Java Virtual Machine
KBSS	Knowledge Based Software Systems Group
RDB	Relational Database
RDBMS	Relational Database Management System
RDF	Resource Description Framework
SQL	Structured Query Language

Table 7.1: Glossary.

Bibliography

[1] Clescop. (2016) Key value concept illustration. San Francisco, California. [Online]. Available: https://commons.wikimedia.org/wiki/File:KeyValue.PNG

[2] S. Uzunbayir, "A Comparison Between Relational Database Models and NoSQL Trends On Big Data Design Challenges Using A Social Shopping Application," Ph.D. dissertation, 06 2015. [Online]. Available: https://www.researchgate.net/publication/324261323_A_Comparison_Between_Relational_Database_Models_and_NoSQL_Trends_On_Big_Data_Design_Challenges_Using_A_Social_Shopping_Application

[3] (2022) Getting Started with Cypher. San Mateo, CA, United States. [Online]. Available: https://neo4j.com/developer/cypher/intro-cypher/

[4] A. Basse, F. Gandon, I. Mirbel, and M. Lo, "Incremental characterization of RDF Triple Stores," 04 2012. [Online]. Available: https://www.researchgate.net/publication/265359709_Incremental_characterization_of_RDF_Triple_Stores

[5] A. Shvets and D. Zhart. (c2014-2022) Strategy. Kamianets-Podilskyi, Ukraine. [Online]. Available: https://refactoring.guru/design-patterns/strategy

[6] ——. (c2014-2022) Template Method. Kamianets-Podilskyi, Ukraine. [Online]. Available: https://refactoring.guru/design-patterns/template-method

[7] M. Ledvinka and P. Křemen, "JOPA: Accessing Ontologies in an Object-oriented Way," *Proceedings of the 17th International Conference on Enterprise Information Systems*, pp. 212–221, 2015-4-27. DOI 10.5220/0005400302120221. [Online]. Available: http://www.scitepress.org/DigitalLibrary/Link.aspx?doi=10.5220/0005400302120221

[8] D. Hernández, A. Hogan, C. Riveros, C. Rojas, and E. Zerega, "Querying Wikidata: Comparing SPARQL, Relational and Graph Databases," *The Semantic Web – ISWC 2016*, pp. 88–103,

2016. DOI 10.1007/978-3-319-46547-0_10. [Online]. Available: https://link.springer.com/10.1007/978-3-319-46547-0_10

[9] F. Ravat, J. Song, O. Teste, and C. Trojahn, "Efficient querying of multi-dimensional RDF data with aggregates: Comparing NoSQL, RDF and relational data stores," *International Journal of Information Management*, vol. 54, 2020. DOI 10.1016/j.ijinfomgt.2020.102089. [Online]. Available: https://linkinghub.elsevier.com/retrieve/pii/S0268401219306097

[10] A. Corbellini, C. Mateos, A. Zunino, D. Godoy, and S. Schiaffino, "Persisting big-data: The NoSQL landscape," *Information Systems*, vol. 63, pp. 1–23, 2017. DOI 10.1016/j.is.2016.07.009. [Online]. Available: https://linkinghub.elsevier.com/retrieve/pii/S0306437916303210

[11] M. Svoboda, "Introduction: Big Data, NoSQL Databases," *Database Systems 2.* [Online]. Available: https://www.ksi.mff.cuni.cz/~svoboda/courses/211-B4M36DS2/lectures/B4M36DS2-Lecture-01-Introduction.pdf

[12] G. Edmonds. (2013) Sane way to store different data types within same column in postgres. New York City, New York, U.S. [Online]. Available: https://stackoverflow.com/a/18233822

[13] R. C. Martin, *Design Principles and Design Patterns.* www.objectmentor.com, 2000. [Online]. Available: https://web.archive.org/web/20150906155800/http://www.objectmentor.com/resources/articles/Principles_and_Patterns.pdf

[14] E. Gamma, R. Helm, R. Johnson, and J. Vlissides, *Design Patterns.* Boston: Addison-Wesley, c1995. ISBN 978-0201633610

[15] A. Shvets and D. Zhart. (c2014-2022) Facade. Kamianets-Podilskyi, Ukraine. [Online]. Available: https://refactoring.guru/design-patterns/facade

[16] J. Bloch, *Effective Java*, Third ed. Boston: Addison-Wesley Professional, 2017, pp. 293–300. ISBN 978-0134686097

[17] Wikipedia contributors, "Singleton (mathematics)," 2022. [Online]. Available: https://en.wikipedia.org/w/index.php?title=Singleton_(mathematics)&oldid=1071184601