Data Model Evolution using Object-NoSQL Mappers: Folklore or State-of-the-Art?

Andreas Ringlstetter
OTH Regensburg
Regensburg, Germany
andreas.ringlstetter@st.oth-regensburg.de

Stefanie Scherzinger
OTH Regensburg
Regensburg, Germany
stefanie.scherzinger@oth-regensburg.de

Tegawendé F. Bissyandé
University of Luxembourg
tegawende.bissyande@uni.lu

ABSTRACT

In big data software engineering, the schema flexibility of NoSQL document stores is a major selling point: When the document store itself does not actively manage a schema, the data model is maintained within the application. Just like object-relational mappers for relational databases, object-NoSQL mappers are part of professional software development with NoSQL document stores. Some mappers go beyond merely loading and storing Java objects: Using dedicated *evolution annotations*, developers may conveniently add, remove, or rename attributes from stored objects, and also conduct more complex transformations. In this paper, we analyze the dissemination of this technology in Java open source projects. While we find evidence on GitHub that evolution annotations are indeed being used, developers do not employ them so much for evolving the data model, but to solve different tasks instead. Our observations trigger interesting questions for further research.

CCS Concepts

•**Software and its engineering → Agile software development; Software evolution;**

Keywords

Object-NoSQL mappers, data model evolution

1. INTRODUCTION

NoSQL document stores are becoming increasingly popular in big data software development, with MongoDB currently ranking among the top five database systems[1]. MongoDB as a schema-less document store does not require a fixed data model declared up front. Agile software engineers may thus evolve their data model over time.

Along with the proliferation of NoSQL document stores, dedicated object-NoSQL mappers have become available [7].

[1]http://db-engines.com/en/ranking as of February 2016.

BIGDSE'16, May 16 2016, Austin, TX, USA
© 2016 ACM. ISBN 978-1-4503-4152-3/16/05…$15.00
DOI: http://dx.doi.org/10.1145/2896825.2896827

```
@Entity
class Player {
  @Id
  Integer id;


  String lastName;
  String firstName;
  Char middleInitial;

}
```

a: Entity class *Player*.

```
@Entity
class Player {
  @Id
  Integer id;

  @AlsoLoad("lastName")   // Rename
  String surname;
  String firstName;
                           // Remove
  String nickname = "N/A"; // Add
}
```

b: Refactored entity class.

```
{
  "Kind": "Player",
  "id": 42,
  "lastName": "Cooper",
  "firstName": "Sheldon",
  "middleInitial": "L"
}
```

c: Initial player entity.

```
{
  "Kind": "Player",
  "id": 42,
  "surname": "Cooper",
  "firstName": "Sheldon",
  "nickname": "N/A"
}
```

d: Migrated player entity.

Figure 1: (a, b) Entity classes with object-NoSQL mapper annotations, and matching entities (c, d resp.).

They make storing and loading objects convenient, as illustrated by the following example. Figure 1a shows a Java class representing players in an online game. In the tradition of the Java Persistence API [2], a de-facto standard for object-relational mappers, annotation `@Entity` declares that player objects can be stored. Annotation `@Id` denotes the unique identifier. Based on the annotations, an object-NoSQL mapper marshals player objects into the NoSQL document store. Figure 1c shows an instance of a stored entity in JSON format. Throughout this paper, we refer to Java classes with annotation `@Entity` as *entity classes*. We distinguish the instances of an entity class, the *objects*, from the stored objects, the *entities*.

Figure 1b shows evolutionary changes to the entity class. Attribute *lastName* has been renamed to *surname*, attribute *middleInitial* has been removed, and *nickname* has been added. Adding and removing attributes can be carried out by all common object-NoSQL mappers. However, the mappers Morphia [4] (for MongoDB) and Objectify [5] (for Google Cloud Datastore) offer dedicated *evolution annotations* for more advanced changes. The evolution annotation `@AlsoLoad` in Figure 1b has the following effect when the entity from Figure 1c is loaded: Attribute *lastName* is renamed to *surname*. Saving the object back to storage yields the entity in Figure 1d. This on-demand migration of single entities is commonly referred to as *lazy* or *online* migration.

At the time of writing this paper, more object-NoSQL mappers are starting to follow the lead of Morphia and Objectify, c.f. the current roadmap for Hibernate OGM [6]. The excitement among developers over these opportunities for agile development, perceivable in blogs and mailing lists, motivates our research question: *How established is data model evolution driven by evolution annotations in practice?*

To answer this question, we have conducted an analysis of approx. 900 GitHub projects using Morphia or Objectify with a history of at least 20 commits:

- While we have found evidence of evolution annotations being used in approx. 100 projects, a close-up investigation reveals that in the majority of cases, evolution annotations are *abused* to solve different tasks.
- By screening the projects with evolution annotations, we gain insight on the developers' intentions in using these features. We are able to distinguish five use cases, where data model evolution is actually among the least frequent: In only 5.6% of projects with evolution annotations, these were actually employed for the purpose of lazy migration.
- We observe cases where evolution annotations are used for managing transient attributes, even though there are dedicated annotations for this purpose. At the same time, there are evolution annotations where we could not find any evidence of them being used. This may be because developers find it difficult to grasp the intended usage of some annotations, or because they do not need these annotations in the first place.
- We discuss and try to interpret our findings, and propose questions for further research.

Structure: This paper is structured as follows. Section 2 introduces evolution annotations in Morphia and Objectify. In Section 3, we present our methodology in analyzing open source projects. We present our results in Section 4, and discuss them in Section 5. Section 6 concludes.

2. EVOLUTION ANNOTATIONS

We distinguish two groups of evolution annotations. The first group controls the evolution of class attributes, while the second controls the invocation of class methods.

Migration Annotations: Annotation `@AlsoLoad` renames a single attribute and is available both in Morphia and Objectify. Morphia also provides the annotation `@NotSaved` so that an attribute in a Java object is not stored in the entity. Objectify has the corresponding attribute `@IgnoreSave` (and its deprecated equivalent `@NotSaved`), as well as an annotation with the inverse effect: `@IgnoreLoad` does not load an attribute from an entity into the Java object.

Life-Cycle Annotations: The syntax of Morphia life-cycle annotations resembles that of the Java persistence API (JPA) [2], originally devised for object-relational mappers. Traditionally, these all-purpose annotations have been devised for managing the object life-cycle. In Morphia, they are intended for managing data model evolution as well: Methods annotated with `@PrePersist` or `@PreSave` are invoked before saving an object. The difference is subtle: A method annotated `@PrePersist` is invoked before the Java object is stored. A `@PreSave` method is invoked one step further in the life cycle, when the object has been mapped to an entity and the raw entity is about to be stored. Accordingly, a `@PostPersist` method is invoked after saving an object as an entity. The behavior of annotations `@PreLoad` and `@PostLoad` for loading is likewise. In Objectify, methods annotated with `@OnLoad` or `@OnSave` are invoked after loading an entity as an object, or before saving an object.

3. METHODOLOGY

Our analysis is performed in four stages. We first identify Github projects that use either Morphia or Objectify. We make an effort to be exhaustive, and as of December 4th, 2015, count a total of 633 projects using Morphia and 1,621 projects using Objectify. In the second stage we filter out unsuitable projects. This yields 258 projects using Morphia and 648 projects using Objectify for further analysis. Next, we identify entity classes with evolution annotations and manually classify them according to the use cases sketched below. We now provide details on each phase.

Phase 1: We query Github for Java projects using Morphia or Objectify.[2] Our heuristic for finding these projects is to look for framework-specific import statements.

Phase 2: Roughly 60% of the projects found in phase 1 need to be excluded from further analysis: Some projects merely fork from the Morphia or Objectify source code, or embed these code bases without actually using them.

We then identify all entity classes within projects, but ignore those contained in test suites. By now, all remaining projects contain at least one self-written entity class. We ignore projects with fewer than 20 commits, as these are mainly toy projects or serve educational purposes. While evolution annotations do occur in these projects, investigating a sample of 200 projects reveals that they are not viable development projects with evolving data models.

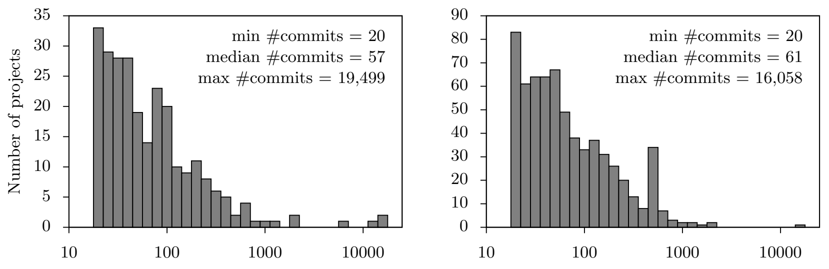
Phase 3: A simple text search identifies entity classes with evolution annotations. We manually inspect to which purpose the annotations are actually employed. Annotated methods that are empty or contain dead code are excluded from further analysis. We distinguish the following use cases:

1. Data model evolution
2. Transient attributes
3. Timestamps
4. Serialization
5. Dispatch of events

We briefly describe each use case. (1) In detecting data model evolution, we focus on annotation-based changes only (e.g., `@AlsoLoad` in the example from Figure 1). Thus, we disregard implicit migrations realized by adding or removing an attribute in an entity class.

(2) Transient attributes are common in database applications. They reside only in objects, but not in entities. For instance, the date of birth of a player in our example from Figure 1 would be a persistent attribute. Since the player's age can be derived from the date of birth, it need not be stored. By declaring attribute *age* as transient, it can be initialized when the player entity is loaded. Traditionally, `@Transient` marks transient attributes [2]. With `@Ignore`, Objectify provides its own annotation for this purpose.

[2]We boost query throughput by using global code search through the GitHub web interface (rather than the GitHub API which is limited in scope to a single project), partitioning the result set with auxiliary terms and the identity $R = (R \cap A) \cup (R \setminus A)$ until it meets the pagination limit.

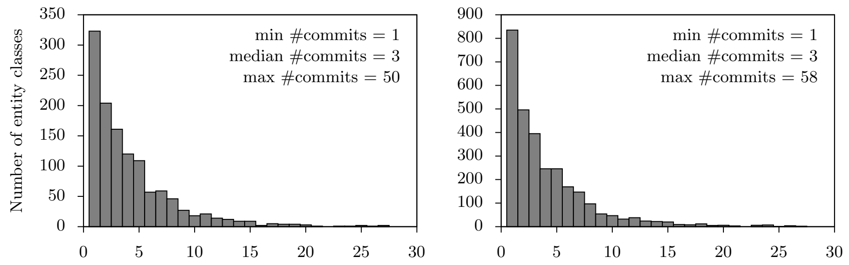


a: Morphia-based projects. b: Objectify-based projects.

Figure 2: Project activity: Commits per project (log scale).

Use Case	@PrePersist	@PreSave	@PostPersist	@PreLoad	@PostLoad	@AlsoLoad	@NotSaved	Total
Evolution						2		2
Transient							7	7
Timestamp	12							12
Serialization	25	1			22			34
Dispatch								0

Table 1: Morphia evolution annotations.



a: Morphia-based projects. b: Objectify-based projects.

Figure 3: Data model activity: Commits per entity class.

Use Case	@OnLoad	@OnSave	@PrePersist	@PostLoad	@AlsoLoad	@IgnoreLoad	@IgnoreSave	Total
Evolution					4			4
Transient							9	9
Timestamp	2	14	6	1				22
Serialization	17	11	8	2				32
Dispatch		2						2

Table 2: Objectify evolution annotations.

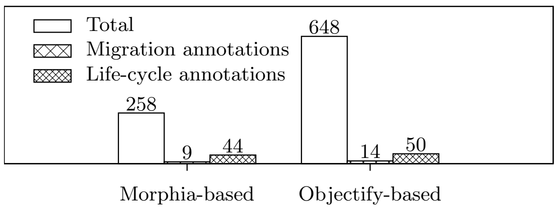


Figure 4: Usage of evolution annotations.

However, we encounter cases where transient attributes are realized using evolution annotations.

(3) Some developers maintain timestamp and version attributes to record the last change to an entity.

(4) By serialization (and de-serialization), we summarize application logic related to storing and loading, e.g., handling attributes which default to *null*. This includes the initialization of data store index structures after de-serialization, and initialization of transient attributes.

(5) Further, we find cases where developers dispatch events to other components, e.g., for logging purposes.

4. RESULTS

By counting commits, we are able to confirm activity in projects and their data models. We then investigate which role evolution annotations play in evolving the data model.

Project Activity: The histogram in Figure 2 shows the number of commits to Morphia- and Objectify-based projects. As expected [3], the majority of projects is short-lived, yet some projects count several hundred commits.[3]

Data Model Size: Morphia and Objectify-based projects contain three entity classes on average. One Morphia-based project[4] has 44 and one Objectify-based project[5] has even 52 entity classes. There are projects with even larger data models, but these numbers are distorted by duplicate code.

Data Model Activity: Figure 3 is fundamental to our analysis. It shows how often an entity class changes, counting the commits per entity class. A typical entity class changes at least three times. Manual investigation reveals that the majority of commits indeed changes the data model (rather than class methods that do not affect the structure of an entity, e.g., getters or setters). There is even one entity class for each framework with at least 50 changes[6,7].

Usage of Evolution Annotations: Having confirmed that data models evolve, we investigate whether evolution annotations are being used. Figure 4 states the number of projects in which evolution annotations occur. Note that the same project may use both migration and life-cycle annotations. The share of Morphia-based projects with evolution annotations is higher than for Objectify-based projects.

Use Cases for Evolution Annotations: In Tables 1 and 2, we categorize occurrences of evolution annotations into the use cases from Section 3. The first column lists the use case. The following columns read as follows. For each evolution annotation, the column states how many projects employ this annotation for the given use case (at least once). Column "total" sums up the findings per use case.[8]

For instance, among the 258 Morphia-based projects, only two use the `@AlsoLoad` annotation. This is surprising insofar as `@AlsoLoad` has straightforward semantics. In contrast, the fact that `@PreLoad` is not used by any project may be explained by its subtlety (c.f. Section 2). As can be expected, developers have more need for manipulating the object (c.f. using `@PostLoad`) rather than the raw entity (c.f. `@PreLoad`). We next discuss each use case in turn.

Use Case "Data Model Evolution": Data model evolution

[3]The peaks in Figures 1a and 1b at approx. 100 and 700 commits are due to forks late in the development cycle.
[4]https://github.com/MKLab-ITI/simmo
[5]https://github.com/ishacrm/ishacrmserver

[6]https://github.com/karma-exchange-org/karma-exchange
[7]https://github.com/TuttiFruttiFT/tuttiFrutti
[8]A single project can use several annotations for the same use case, and, at the same time, can match several use cases. Hence, the numbers do not directly align with Figure 4.

is among the least frequent use cases. We refer to Section 5 for a discussion, and proceed with the other use cases.

Use Case "Transient Attributes": Morphia and Objectify provide annotations for declaring transient attributes (c.f. Section 3). However, we find cases where this functionality is replicated using `@NotSaved` or `@IgnoreSave`. Upon inspection, we can confirm that the annotations are not used for data model evolution, since the attribute values concerned are overwritten during load.

Use Case "Timestamps": A common use case is to tag entity classes with creation and update timestamps. Neither Objectify nor Morphia provide dedicated annotations. In implementing this use case, developers mimic date type columns in relational databases.

Use Case "Serialization": The most common use case is to write custom serialization and de-serialization logic. Mostly, this means serializing embedded objects into strings, so that they may be indexed by the NoSQL data store for query evaluation. Several Morphia-based projects serialize a transient `BigInteger` to a `String` in a method annotated with `@PrePersist`. The value is then de-serialized in a method annotated with `@PostLoad`. This pattern is unexpected, since developers might instead utilize the type conversion system provided by Morphia.

Use Case "Dispatch of Events": Merely two Objectify-based projects use `@OnSave` to notify components outside the model layer of save events.

Unclassified Findings: Table 1 lists no findings for the Morphia annotations `@PostPersist` and `@PreLoad`, although a small number of entity classes actually uses them: Inspecting the code, it becomes evident that the authors actually intended to use `@PrePersist` and `@PostLoad`. Confusing the annotations effectively renders these methods dead code.

As seen in Table 2, there are no findings for the Objectify annotation `@IgnoreLoad`.

5. DISCUSSION

Our analysis shows that the data model undergoes several evolutionary revisions in most projects that use the object-NoSQL mappers Morphia or Objectify (c.f. Figure 3).[9] Yet to our genuine surprise, only a negligible fraction of projects makes use of evolution annotations. To put this into perspective, we point out that with our analysis focusing on annotations, we miss the most basic evolutions realized by merely adding or removing attributes in entity classes. Sampling entity classes and their revisions, we find this form of evolution in 50% of all projects. Thus, a large share of developers is – intentionally or not – already using this basic form of lazy migration specific to object-NoSQL mappers.

Yet the lack of evidence for more complex evolution operations, such as renaming attributes or transformations of attribute values, stands in stark contrast to the fact that data models do evolve. At the beginning of our analysis, we were aware that life-cycle annotations can be used for various tasks. However, we did expect to find at least some evidence that they are utilized for data model evolution, as frequently discussed in blogs and mailing lists.

One interpretation of this negative result is that data model evolution remains a challenge, even when developers use NoSQL data stores. To spin this thought further, it would mean that the promises of NoSQL technology w.r.t. schema flexibility have not arrived in practice yet. Here, a word of caution is required, since we have only analyzed projects hosted on GitHub and no commercial code bases.

[9]We even find evidence that the data model evolves *after* the application has been released into production, based on version control release tags and publicly accessible instances.

It remains unclear whether developers are even aware that more complex data transformations *are even possible* with object-NoSQL mappers. Judging from the coding patterns we were able to observe, a significant number of developers appears to have worked with relational databases before, where different limitations apply. Thus, it may very well be that developers stick to the annotations that are familiar from object-relational mapping.

One conclusion that we may draw with more confidence is that our results suggest the need for a new feature in object-NoSQL mappers: The popular use case of managing entity timestamps reveals a shortcoming of Morphia and Objectify. Given that in almost 5% of the projects, developers went so far as to implement their own timestamp mechanism, indicates a need for a dedicated annotation.

6. SUMMARY

Our investigation reveals that there is little to no evidence in GitHub projects today that data model evolution driven by evolution annotations has become state-of-the-art. As discussed in the previous section, various explanations deem reasonable. At the same time, caution is justified in interpreting these facts: It may very well be the case that developers do not check evolution-related code into GitHub, assuming that whoever pulls the project will do a setup from scratch. This justifies a follow-up study where we reach out to developers and query their motives.

Regarding the bigger picture, it is also worthwhile to investigate the correlation between development and deployment when it comes to data model evolution. With agile development, best practices for continuous delivery taking into account the data backend have emerged [1]. It is an open question whether these practices are reflected in open source code repositories.

7. ACKNOWLEDGMENTS

We thank Kim Dongsun and Daoyuan Li from the University of Luxembourg for their advice and assistance.

8. REFERENCES

[1] J. Humble and D. Farley. *Continuous Delivery: Reliable Software Releases Through Build, Test, and Deployment Automation.* Addison-Wesley Professional, 2010.

[2] Java Persistence 2.0 Expert Group. *JSR 317: Java Persistence 2.0*, 2009.

[3] E. Kalliamvakou, G. Gousios, K. Blincoe, L. Singer, et al. The Promises and Perils of Mining GitHub. In *Proc. MSR'14*, 2014.

[4] Morphia. *Morphia*, Feb. 2016. https://github.com/mongodb/morphia.

[5] Objectify. *Objectify*, Feb. 2016. https://github.com/objectify/objectify.

[6] Red Hat. Hibernate OGM Roadmap for version 5.0, Feb. 2016. http://hibernate.org/ogm/roadmap/.

[7] U. Störl, T. Hauf, M. Klettke, and S. Scherzinger. Schemaless NoSQL Data Stores – Object-NoSQL Mappers to the Rescue? In *BTW'15*, 2015.