

# Ein Eclipse-Plugin zur kontrollierten Schema-Evolution im NoSQL Datenbanksystem MongoDB

Dennis Schmidt<sup>1</sup>

**Abstract:** Schemafreie NoSQL-Datenbanken wie MongoDB bieten in der Softwareentwicklung große Flexibilität. Dies ist vor allem in der agilen Softwareentwicklung sehr nützlich, bei der die Software früh veröffentlicht und in kurzen Abständen aktualisiert wird. Des Weiteren werden in der professionellen Anwendungsentwicklung gerne Objektmapper als Bindestück zwischen einer objektorientierten Sprache und dem Datenbankmodell verwendet. Durch einen Objektmapper wird über die Klassendeklaration der zu speichernden Objekte ein Schema in die eigentlich schemafreie NoSQL-Datenbank impliziert. Durch häufige Aktualisierungen der Software ergeben sich große Probleme beim Verändern des Schemas bzw. der Klassendeklarationen (Schema-Evolution): Während sich die Klassendeklaration verändert, behalten bereits gespeicherte Objekte ihr Schema bei und werden anschließend über die neue Klassendeklaration geladen. Dadurch kann es zu Datenverlust oder gar Laufzeitausnahmen kommen.

In diesem Beitrag wird ein Eclipse-Plugin vorgestellt, das den Entwickler bei der kontrollierten Schema-Evolution mit dem Objektmapper Morphia für MongoDB unterstützt. Veränderungen an Klassendeklarationen werden mit früheren Veröffentlichungen einer Software verglichen. Dabei werden mögliche Probleme erkannt und passende Lösungsvorschläge angeboten.

**Keywords:** Schemafreie NoSQL-Datenbanken, MongoDB, Objektmapper, Schema-Evolution

## 1 Einleitung

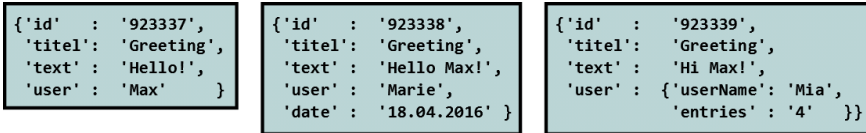
NoSQL-Datenbanken wie MongoDB verzichten im Gegensatz zu relationalen Datenbanken auf ein festes Schema für die gespeicherten Daten. Durch diese Flexibilität werden sie vor allem in der agilen Softwareentwicklung gerne eingesetzt. In einem solchen Entwicklungsprozess wird die Software in einem frühen Stadium veröffentlicht und dann in kurzen Abständen aktualisiert [Hü07]. Durch die häufigen Aktualisierungen ist es schwer zu Beginn ein Schema zu definieren, das in den folgenden Versionen gleich bleiben kann. MongoDB gehört zu den dokumentenorientierten Datenbanktypen, wobei *JSON* ähnliche Dokumente verwaltet werden. Ein Eintrag in MongoDB heißt Dokument und ist eine Datenstruktur aus Feld-Wert Paaren [RW12]. In MongoDB *Collections* werden üblicherweise Dokumente gruppiert, die denselben oder einen ähnlichen Zweck erfüllen. Dabei schreiben die *Collections* kein festes Schema für die gespeicherten Dokumente vor.

Die Abbildung 1 gibt im *JSON*-Format drei unterschiedlich strukturierte Dokumente in derselben *Collection* an. Durch die Schemafreiheit von MongoDB ist es kein Problem, unterschiedlich strukturierte Dokumente in derselben *Collection* zu speichern und zu laden. In der professionellen Anwendungsentwicklung werden gerne Objektmapper verwendet,

---

<sup>1</sup> Ostbayerische Technische Hochschule Regensburg, Fakultät für Informatik und Mathematik, dennis.schmidt@st.oth-regensburg.de

um ein Objekt einer Programmiersprache bequem in der Datenbank zu speichern und zu laden. Der Objektmapper übernimmt dabei das Erstellen eines passenden Datenbankobjekts und das Abbilden der Attribute auf dieses. Morphia ist ein Objektmapper für MongoDB, der ein gewöhnliches Java-Objekt auf ein Dokument für MongoDB abbildet [Mo15].



```

{ 'id' : '923337',
  'titel': 'Greeting',
  'text' : 'Hello!',
  'user' : 'Max' }

{ 'id' : '923338',
  'titel': 'Greeting',
  'text' : 'Hello Max!',
  'user' : 'Marie',
  'date' : '18.04.2016' }

{ 'id' : '923339',
  'titel': 'Greeting',
  'text' : 'Hi Max!',
  'user' : { 'userName': 'Mia',
             'entries' : '4' } }

```

Abb. 1: Drei unterschiedlich strukturierte Dokumente in derselben *Collection*

Im Folgenden wird eine Gästebuch-Software betrachtet, die in der agilen Softwareentwicklung mit MongoDB als NoSQL-Datenbank und Morphia als Objektmapper realisiert wird (Abbildung 2). Im oberen Bereich der Abbildung wird die Deklaration der Klasse „*Guestbook*“ in der Entwicklungsumgebung dargestellt. Mit *@Entity* wird die Klasse zum Speichern markiert und mit *@Id* wird das eindeutige Id-Feld des Dokuments markiert. Beim Speichern eines Objekts dieser Klasse erstellt der Objektmapper ein passendes Dokument, indem es die Attribute auf Felder des Dokuments abbildet. Der Klassename wird dabei zum Namen der *Collection* und die Namen der Attribute werden zu den Namen der Felder innerhalb des Dokuments. Die gespeicherten Dokumente werden im JSON-Format im unteren Bereich der Abbildung in der Produktionsumgebung dargestellt. Durch die agile Softwareentwicklung wird die erste Version früh veröffentlicht. Der erste Nutzer „*Max*“ verfasst einen Eintrag mit der Gästebuch-Software. Dabei wird zuerst ein Objekt der Klasse „*Guestbook*“ erzeugt und dann mit dem Morphia-Aufruf *save* abgespeichert. Dies wird mit einem Pfeil von der ersten Version der Klasse „*Guestbook*“ nach dem erzeugten Dokument dargestellt. Wird dieses Dokument mit Morphia geladen, wird ein Objekt der Klasse „*Guestbook*“ erstellt und die Felder werden auf Attribute abgebildet. Nun wird eine zweite Version der Software entworfen und anschließend veröffentlicht. Dabei hat der Entwickler die Klassendefinition verändert: Das Attribut „*text*“ wurde auf „*content*“ umbenannt. Werden nun Objekte dieser Klasse abgespeichert, haben diese das Feld „*content*“. Neu abgespeicherte Dokumente haben jetzt die Struktur der zweiten Klassendeklaration und werden über diese erfolgreich geladen. Durch die Schemaflexibilität von MongoDB können die neu erzeugten Dokumente in derselben *Collection* gespeichert werden, obwohl sich die Struktur von denen der älteren Dokumente unterscheidet. Das Problem ist nun, dass auch ältere Dokumente, die bereits in der Datenbank gespeichert wurden, mit der neuen Klassendeklaration geladen werden. Dies wird mit einem Pfeil zwischen dem Dokument mit der alten Struktur und der zweiten Version der Klassendeklaration dargestellt. Zum Laden des Dokuments wurde zum Beispiel der Morphia-Aufruf *get* verwendet. Dabei erstellt der Objektmapper ein Java-Objekt, wobei das Feld „*text*“ nicht mehr zugeordnet werden kann. Das Attribut „*content*“ hat kein passendes Feld in dem geladenen Dokument und wird deshalb ohne Wert initialisiert. Der Wert von „*text*“ ist also nicht mehr abrufbar. Wird dieses geladene Dokument gespeichert, wird das alte Dokument überschrieben und der Wert von „*text*“ ist verloren. Dies wird mit einem Pfeil zwischen der zweiten Version der Klassendeklaration und dem Dokument rechts unten dargestellt. Das Dokument wird mit der neuen Klassendefinition abgespeichert, das heißt, das spei-

cherte Dokument besitzt nun statt „text“ das Feld „content“ ohne Wert.

Bei einem Objektmapper ist also die Klassendefinition der zu speichernden Objekte wichtig, da diese das Schema der Datenbankobjekte bestimmen. Nach dem Speichern werden die Datenbankobjekte über diese Klassendeklaration geladen. Obwohl viele NoSQL-Datenbanken kein festes Schema besitzen, implizieren Objektmapper also ein Schema über die Klassendeklaration der zu speichernden Objekte. Beim Verändern der Klassendeklarationen, auch genannt Schema-Evolution, kann es zu Probleme beim Laden älterer Datenbank-Objekte kommen.

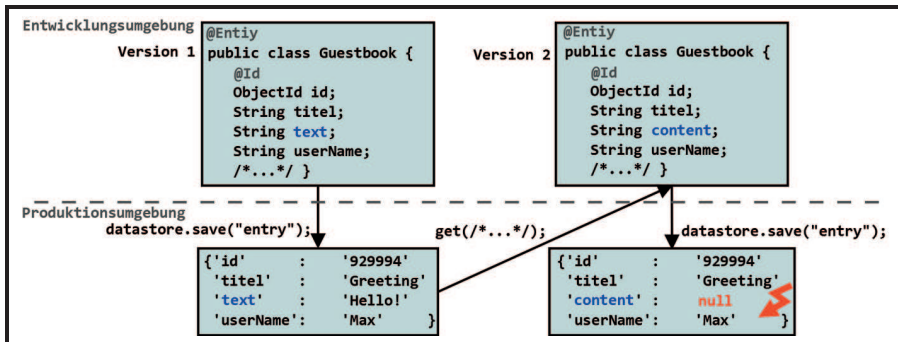


Abb. 2: Beispiel einer unkontrollierten Schema-Evolution: Umbenennung führt zu Datenverlust

### Contributions:

1. In diesem Beitrag wird das erste Eclipse-Plugin vorgestellt, das Probleme bei der Schema-Evolution in MongoDB frühzeitig erkennt und über das *Quickfix*-Fenster passende Lösungsvorschläge anbietet.
2. Das Plugin wurde ursprünglich für den Objektmapper Objectify für den Google Datastore implementiert und wurde nun in seinem Funktionsumfang auf den Objektmapper Morphia und MongoDB, einer der bekanntesten NoSQL-Datenbanken, erweitert [CCS15, DB16].

**Aufbau:** In Kapitel 2 wird gezeigt, wie das Eclipse-Plugin das in der Einleitung angegebene Problem erkennt und Lösungsvorschläge anbietet. In Kapitel 3 wird das Konzept des Eclipse-Plugins beschrieben.

## 2 Die Plugin gestützte Schema-Evolution mit Morphia Annotationen

Im Folgenden wird gezeigt, wie das Eclipse-Plugin den Entwickler bei der kontrollierten Schema-Evolution unterstützt. Dabei wird auf das in Kapitel 1 gezeigte Problem eingegangen. Viele Objektmapper wie Morphia oder Objectify bieten spezielle Annotationen an, mit denen angegeben werden kann, wie auch ältere Objekte über eine neue Klassendeklaration verlustfrei geladen werden können. Diese werden in der jeweiligen Klassendeklaration der Datenbankobjekte platziert. Somit wird die Schema-Evolution den Softwareentwicklern überlassen. Diese haben möglicherweise wenig Erfahrung im Bereich Datenbanken, wodurch sich viele Fehlerquellen ergeben können. In Kapitel 1 wird beschrieben,

wie ein Entwickler in einer Gästebuch-Software ein Attribut ohne weiteres umbenennet und deshalb Datenverlust auftritt. Wird der Typ eines Attributs verändert, kann sogar eine Laufzeitausnahme auftreten, z.B. wenn der Typ von *String* auf *Integer* verändert wird.

Mit dem Plugin kann der Entwickler die erste Version der Gästebuch-Software als Veröffentlichung kennzeichnen. Wenn er anschließend das Attribut „*text*“ auf „*content*“ umbenennet, erkennt das Plugin, dass Datenverlust beim Laden älterer Datenbank-Objekte über diese Klassendeklaration auftreten kann und erstellt eine Markierung. In Bild 3 werden die Lösungsvorschläge angegeben. Das Plugin erkennt, welcher Objektmapper benutzt wurde und bietet deshalb Lösungsvorschläge mit Morphia Annotationen an, welche bei Auswahl automatisch so platziert werden, dass das jeweilige Problem gelöst wird. Wählt der Entwickler die Lösung mit der Annotation `@AlsoLoad("text")`, wird das Attribut „*content*“ damit markiert (Abbildung 4). Dokumente mit dem Feld „*content*“ werden

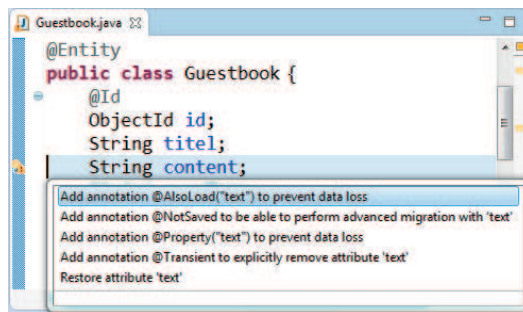
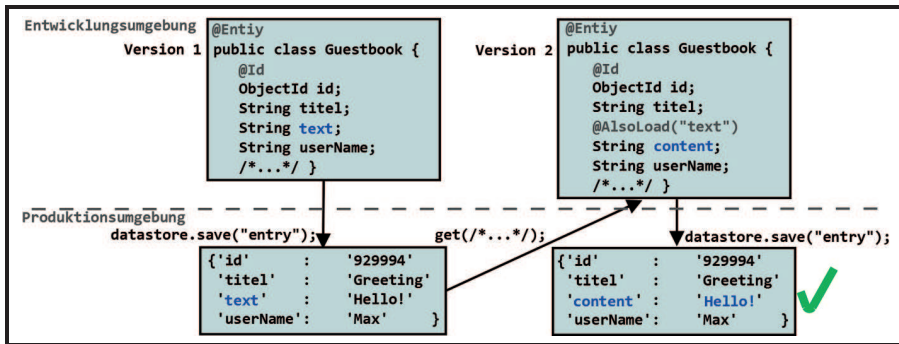


Abb. 3: Das *Quickfix*-Fenster bietet nach dem Umbenennen eines Attributs Lösungsvorschläge an

wie bisher geladen und gespeichert. Aber nun werden auch ältere Dokumente, die das Feld „*text*“ statt dem Feld „*content*“ besitzen verlustfrei geladen. Wird ein Dokument mit dem Feld „*text*“ geladen, wird der Wert von „*text*“ nach „*content*“ kopiert. Beim anschließenden Speichern dieses Dokuments wird „*text*“ verworfen, da es in der Klassendeklaration nicht mehr vorhanden ist. Dafür wird das Attribut „*content*“ mit dem Wert von „*text*“ gespeichert. Der Entwickler hat außerdem die Möglichkeit, das Attribut „*content*“ mit `@Property("text")` zu markieren. Damit wird das Attribut „*content*“ unter dem Bezeichner „*text*“ gespeichert und geladen. Das heißt, das Datenbank-Objekt behält das Feld „*text*“ bei, während in der Entwicklungsumgebung das Attribut „*content*“ oder ein anderer beliebiger Bezeichner benutzt werden kann. Außerdem gibt es die Möglichkeit, das Umbenennen rückgängig zu machen oder den alten Bezeichner als eigenes Attribut mit der Annotation `@Transient` bzw. `@NotSaved` explizit zu entfernen. Bei `@Transient` wird das Attribut nicht mehr gespeichert oder geladen. Bei `@NotSaved` wird das Attribut weiterhin geladen, was für eine komplexe Migration nützlich ist. Dafür kann eine Methode mit der Annotation `@PostLoad` definiert werden, wodurch diese nach dem Laden eines Objekts dieser Klasse ausgeführt wird. In dieser kann beispielsweise überprüft werden, ob ein Wert bei „*text*“ geladen wurde, um diesen nach einer Typkonvertierung nach „*content*“ zu kopieren. Wird „*content*“ auf „*post*“ umbenannt, kann überprüft werden ob ein Wert bei „*text*“ oder „*content*“ geladen wurde, um diesen nach „*post*“ zu kopieren, da mehrere `@AlsoLoad` Annotationen bei einem Attribut nicht kombiniert werden können.

Abb. 4: Beispiel einer kontrollierten Schema-Evolution mit der Annotation `@AlsoLoad`

### 3 Das Konzept des Plugins zur kontrollierten Schema-Evolution

Damit das Plugin mögliche Probleme erkennen kann, speichert es alle Attribute von Klassen, die in die Datenbank abgebildet werden, zusammen mit der Information über eventuell vorhandene Annotationen. Das Plugin erkennt anhand der vorhandenen Bibliotheken, welcher Objektmapper verwendet wurde. Sind Morphia-Bibliotheken vorhanden, benutzt das Plugin Morphia-spezifische Typprüfungsregeln, die definieren, welche Veränderungen an Klassendeklarationen erlaubt sind [Sc16]. Dabei wird die aktuelle Version der Deklaration eines Attributs paarweise mit allen älteren Deklarationen verglichen. Es gibt eine schreibende Klassendeklaration  $C_W$  und eine lesende Klassendeklaration  $C_R$ , die ohne Datenverlust oder Laufzeitausnahme lesen kann. Beispielsweise gibt es die Regel 1, die definiert, dass der Typ eines Attributs, welches mit `@Transient` markiert ist, verändert werden darf, da dieses Attribut nicht mehr geladen oder gespeichert wird.

$$\frac{C_W \vdash \text{type}_1 \text{ att};}{C_R \circ C_W \vdash @\text{Transient type}_2 \text{ att}; \text{ OK}} \quad (1)$$

Weitere Regeln geben an, dass jedes Attribut, das jemals in die Datenbank abgebildet wurde entweder als eigenes Attribut, explizit mit `@NotSaved` bzw. `@Transient` entfernt oder innerhalb einer `@Property` oder `@AlsoLoad` Annotation eines Attributs vorhanden sein muss. Das Plugin unterscheidet zwischen drei Problemkategorien: Das Umbenennen, das Entfernen und das Ändern in einen inkompatiblen Typ. In Bild 3 werden die Lösungen für das Umbenennen gezeigt. Wurde beim Umbenennen nicht das `Refactoring`-Werkzeug von Eclipse verwendet, erkennt das Plugin diese Änderung nur als Löschen und neu Hinzufügen eines Attributs. Genauso verhält es sich, wenn bei einem Attribut, welches umbenannt wird, bereits eine `@AlsoLoad` vorhanden ist, da mehrere `@AlsoLoad` Annotationen nicht miteinander verknüpft werden können. Wurde ein Attribut gelöscht, kann der Entwickler dieses wiederherstellen bzw. mit `@Transient` oder `@NotSaved` explizit entfernen. Wurde der Typ in einen inkompatiblen Typ geändert, hat der Entwickler die Möglichkeit die Änderung rückgängig zu machen oder dieses mit `@Transient` auszuschließen (Regel 1). Außerdem kann er das ursprüngliche Attribut mit `@NotSaved` markieren und ein Methoden-Gerüst zur Typkonvertierung einfügen. Dadurch, dass das Plugin die aktuelle

Version mit allen älteren Versionen vergleicht, erkennt es auch, ob ein Attribut in einer früheren Version gespeichert, anschließend über mehrere Versionen mit *@Transient* entfernt und danach mit einem inkompatiblen Typ wieder hinzugefügt wurde.

### Interaktive Demo

1. Es wird eine Gästebuch-Software mit MongoDB als NoSQL-Datenbank vorgestellt. Diese besitzt die in Abbildung 2 links oben angegebene Klassendeklaration.
2. Nun kennzeichnen wir mit dem Plugin die Software als Veröffentlichungsversion. Anschließend speichern wir ein Objekt der Klasse mit dem Objektmapper Morphia in der Datenbank und laden dieses erfolgreich.
3. Danach nennen wir ein Attribut der Klasse um. Dabei erzeugt das Plugin eine Warnung. Nun laden wir das vorhin gespeicherte Objekt, ohne die Warnung zu beachten. Dabei können wir beobachten, dass das veränderte Attribut aus der Datenbank nicht mehr geladen wird und ein Datenverlust auftritt.
4. Das Plugin bietet mehrere Lösungsvorschläge an. Wir wählen den Lösungsvorschlag mit der Annotation *@AlsoLoad*. Danach laden wir das Objekt ein weiteres Mal. Dabei können wir beobachten, dass das Objekt jetzt erfolgreich geladen wird.

### Zusammenfassung

Viele Anwendungen nutzen die Flexibilität, die NoSQL-Datenbanken durch ihre Schemafreiheit bieten. Doch werden Anwendungen zusammen mit Objektloggern entwickelt, kann es zu Probleme bei der Schema-Evolution kommen. Um Datenverlust und Laufzeitausnahmen beim Laden älterer Objekte zu verhindern, müssen Annotationen richtig angewendet werden. In diesem Beitrag wird ein Eclipse-Plugin vorgestellt, das Probleme frühzeitig erkennt und passende Lösungsvorschläge anbietet.

**Danksagung:** Ein besonderer Dank gilt Prof. Dr. Stefanie Scherzinger, welche die zugehörige Bachelorarbeit betreute [Sc16].

### Literaturverzeichnis

- [CCS15] Cerqueus, T.; Cunha de Almeida, E.; Scherzinger, S.: Safely Managing Data Variety in Big Data Software Development. In: Proc. BIGDSE'15. 2015.
- [DB16] DB-Engines Ranking, <http://db-engines.com/de/ranking>, Stand: 02.05.2016.
- [Hü07] Hüttermann, M.: Agile Java-Entwicklung in der Praxis. O'Reilly, 2007.
- [Mo15] Morphia, <http://mongodb.github.io/morphia/>, Stand: 02.05.2016.
- [RW12] Redmond, E.; Wilson, R.: Seven Databases in Seven Weeks: A Guide to Modern Databases and the NoSQL Movement. Pragmatic Bookshelf, 2012.
- [Sc16] Schmidt, D.: Ein Eclipse-Plugin zur kontrollierten Schema-Evolution im NoSQL Datenbanksystem MongoDB. Bachelorarbeit an der Ostbayerischen Technischen Hochschule Regensburg, 2016.