Enhancing an Object Database with Constraint Functionality to Improve Data Quality

Bachelor Thesis

Author(s):
Oser, Benjamin

Publication date:
2013

Permanent link:
https://doi.org/https://doi.org/10.3929/ethz-a-010286836

Enhancing an Object Database with Constraint Functionality to Improve Data Quality

Bachelor Thesis

Benjamin Oser
<oserb@ethz.ch>

Prof. Dr. Moira C. Norrie
David Weber

Global Information Systems Group
Institute of Information Systems
Department of Computer Science

17th December 2013

ETH
Eidgenössische Technische Hochschule Zürich
Swiss Federal Institute of Technology Zurich

globis

Abstract

In this bachelor thesis an approach for extending any Java Data Objects (JDO) aware object database with constraint functionality is developed. To improve data quality it is not only possible to assign constraints a severity level but also some profiles. This allows fine grained control over constraint evaluation. While the severity level serves as data quality indicator and facilitates automatic conflict resolution, profiles can be used to tweak performance. Furthermore, constraints can be retained, removed and added at runtime. The framework is based on the JDO-API to be completely independent of the underlying object database. Besides basic constraint functionality as (composite) `Primary-` and `ForeignKey` as well as `Unique`, it also contributes the Object Constraint Language (OCL) as an expression language for `Checks` of the Object Validation Framework for Java (OVal). Moreover, we take account of the additional constraint types raised by the object database. A demonstrator web application serves as proof-of-concept that the framework actually has the potential for being used.

Contents

1 Introduction

1.1 Motivation

Constraints are an important part of relational databases for ensuring basic data quality and integrity. In contrast to them, object databases provide only very limited support for constraints yet. As a consequence, checks regarding data quality are distributed across the different layers of the application. This not only complicates maintenance but also impairs performance. For this reasons a central constraint definition, management and validation framework is demanded, preferably just depending on the JDO Application Programming Interface (API) to stay independent of the actual object database.

1.2 Structure

This thesis begins in chapter 2 with acquiring some background information, especially gaining an overview about the different constraint types and categories from section 2.1.1 to 2.1.3. Also some existing concepts of constraint functionality as in section 2.2 and the possible interaction methods of JDO in section 2.3 will be compared. In section 2.4 we reason whether to use annotations or a configuration file for constraint configuration. We proceed by evaluating different approaches for the framework in chapter 3 before motivating our decision and describing our approach including the implementation in detail in chapters 4 and 6. In between, in chapter 5, we provide an use case for an easy start into our framework. Moreover, we show in chapter 7 how to extend the framework with new constraint types in different ways. Furthermore, we describe our proof-of-concept in chapter 8, before we conclude with our contribution and future work to be done in the final chapter 9.

2 Background

2.1 Overview of Constraints

This section is about gaining an overview about the different constraint types and categories. Due to the fact that the object-oriented data model is a generalisation of the relational data model the constraints and their type categories are analogous. The terms 'column', 'record' and 'relation' in the relational data model correspond to 'attribute', 'object' and 'class' in the object-oriented model. Nevertheless, the additional concepts rise some new constraints. [18]

2.1.1 Constraint Types

First we introduce five constraint categories based on the qualitative complexity of the constraint evaluation as proposed by Kim et al. [18]. The following classification is independent of the existence of special data structures. However, it is possible to exploit the presence of an index to lower constraints of type 4 or 5 to type 2 or 3.

1. **Type-1**: Constraints which can be checked in a single attribute within a single object.
2. **Type-2**: Constraints which can be checked in multiple attributes within a single object.
3. **Type-3**: Constraints which require to access a set of instances of a class.
4. **Type-4**: Constraints which require to access all instances of a class.
5. **Type-5**: Constraints which require to access instances of multiple classes.

Furthermore, we classify constraints regarding the data quality as introduced by our research group in an earlier paper [17]. We differentiate between mandatory *hard* constraints to maintain data integrity and optional *soft* constraints, which can be considered as indicators for data quality by categorising them into different data quality levels.

2.1.2 Constraint Keywords

Kim et al. mentioned the following constraint keywords as they were available since SQL-89 (ISO 9075:1989), which introduced referential integrity. The number in brackets behind the keyword indicates the corresponding constraint type.

- **Not Null**(1)
  This constraint prevents *null* values in attributes. Constraints only fail if they evaluate to *false*. However, *null* values will in general not evaluate to *false* but rather to *unknown*, which makes this kind of constraint crucial. It is sufficient to check only one attribute for that kind of constraint, hence it is of the first constraint category.

- **Primary Key/Unique**(4)
  The *primary key* constraint ensures *not null* and *unique*, while *unique* only ensures that a value or a combination of values is unique within all instances of a class. Therefore, it is of constraint category four.

- **Foreign Key/References**(4)
  This constraint ensures referential integrity. Referenced values must exist. For that reason all of the possibly referenced objects need to be checked, this makes it also to a constraint of category four.

- **Check**(2)
  Checks can be evaluated within a single object. Therefore, it must not include any subqueries or aggregation as they may involve other objects. However, it is not limited to a single attribute. Therefore, it is of the second constraint category. *Checks* are defined in an arbitrary logical expression language and are evaluated at inserts or updates. It is favourable to define multiple checks per attribute to be able to distinguish the reason for a constraint violation.

- **With Check Option**(2)
  Similar to *check* this constraint type is also checked for each record. Thus, it is of the same constraint category. In contrast to the former *check* constraint *with check option* builds upon views. On updates or inserts the conditions defining the view get checked.

2.1.3 Additional Concepts

So far, the introduced constraint types and keywords in the object-oriented data model have their direct counterparts in the relational data model. However, the object-oriented data model includes some additional concepts. [18]

- **Methods**
  With methods it is possible to implement triggers because they are sufficient to specify constraints and their corresponding action. Due to the complexity and the universal computing power they should be used with care to not impair system performance or destroy database integrity by side effects.

- **Nested Attributes**
  An attribute can have as its value an instance of an arbitrary class, whose attributes can also be used for constraint specification. This so called traversal is logically equivalent to join in the relational data model. Although this constraint type belongs to category five the performance overhead gets limited by the existence of the object identifiers. It might be worth considering whether to place the constraint in the starting or in the terminal class.

- **Set-Valued Attributes**
  Since it is possible to have a set of values for an attribute, aggregation becomes applicable here as well. An aggregation constraint can make use of count, maximum, minimum, sum and average. As a set is involved it is of constraint category three.

- **Complex Objects**
  In general a complex object recursively references any number of other objects. This can be modelled by combining the concepts *nested attributes* and *set-valued attributes* as introduced above. In contrast to the relational data model, such a constraint can be specified for a single object. Hence, it is of the first constraint category.

- **Class Hierarchy**
  The class hierarchy due to inheritance also influences consideration of constraints. Accessing all direct instances is only of constraint category three. Taking also all indirect instances into account upgrades the category to five. As the constraint specification is a property of a class it should be inheritable. Such inherited constraints need to be modifiable to specialise the domain. In general it is very difficult to automatically check conflicts between constraints, whether they can co-exist or not. Proper naming indicates whether to inherit or override. *Unique*, *PrimaryKey* and *ForeignKey* also gets inherited unless they are redefined. If such constraints consist of a single attribute it should be possible to directly mark them in the attribute definition without the need of a separate statement. Be aware that although the unique object identifier guarantees uniqueness of the object, it does not imply uniqueness of its attributes as it does not carry any semantics such as referential integrity. In other words, two objects can have the values of all their attributes in common in spite of their different unique object identifiers.

2.1.4 OVal Constraints

The Object Validation Framework for Java (OVal) provides already a rich set of constraints out of the box, such as `length`, `max` and `min`. The complete set of constraints as defined by the schema for the Extensible Markup Language (XML) configuration[1] is listed in the appendix A.2.

[1] `http://oval.sourceforge.net/oval-configuration.xsd/`

2.2 Analysis of Existing Concepts

There exist several mechanisms for enforcing integrity constraints. The most general form is the event-condition-action paradigm, which is also basis for triggers. Even more fundamental but less flexible are assertions.

2.2.1 Event-Condition-Action

The Event-Condition-Action (ECA)-paradigm consists of three parts. It is possible to set the coupling mode in between the triggering event and the condition evaluation as well as between the condition evaluation and the execution of the action. ECA rules are proposed to generalise mechanisms such as assertions and triggers. [4]

Events

For the practical part of this thesis it is important to understand the various events. Kim et al. [18] differentiate between conditioned external interrupts and database access. *External interrupts* do not involve any object of the database. This kind of events can be issued by timer or real-time interrupts. *Timer interrupts* often occur periodically e.g. for updating database statistics, storage compaction, check-pointing or deadlock detection. *Real-time interrupts* which require quick response as for user interaction or sensor data also belong to the category of external interrupts. *Database access* is categorised into four types considering the scope of access (single object, single class, multiple classes or the entire database). Ischikawa et al. [9] call their basic event and triggering mechanisms demons. All transaction- (start, commit, abort) and CRUD-methods (create, read, update, delete) belong to the category of *system-defined* methods. *User-defined* methods and their events are in general far more costly to catch. For example, it becomes possible to define an event for the first respectively the n-th access. Furthermore, composite events consist of several primitive events.

Condition

In the condition the constraints as introduced in section 2.1 get specified.

Action

If a condition evaluates to true it would be possible to simply *reject* that operation and let the transaction manager handle the recovery. In some cases there might be enough information to proceed by issuing an appropriate query to *update* the database into a consistent state regarding the constraints. Even more, it is possible to carry out a *general operation* written in the host programming language.

Coupling Modes

Evaluating the condition as soon as the associated event occurs within the same transaction is called *immediate*. This evaluation of conditions can be *deferred* right before the end of but still within the same transaction. The order in which the conditions are triggered will be maintained (first-come, first-serve). If the condition will be evaluated in a transaction other than the triggering transaction it is called *separate*.

2.2.2 Triggers

A trigger is a general mechanism for automatically preserving integrity conditions and/or taking arbitrary actions in an active database system. [3] Whereas the term active describes a system that supports automatic updates of views and derived data as base data are updated. [4], [12] Triggers consist of three components: a set of events, conditions and a set of actions. Their general form is:

ON event IF condition THEN DO action

Where conditions are constraints as already discussed in section 2.1 while events and actions are part of the ECA-paradigm from subsection 2.2.1. It can be specified whether the trigger is activated for an object or the entire class. Furthermore, it is possible to specify whether the trigger is executed before or after the firing event.

2.2.3 Assertions

Assertions are an important part of the design by contract design pattern [11], which consists of invariants, pre- and postconditions. While invariants have to hold during the entire execution of a code section, preconditions only have to hold just prior and postcondition just after that code section. In contrast to the programming language Eiffel [10] which provides design by contract natively, in Java it has to be implemented manually with assertions. Either way it is possible to disable assertions to improve performance of execution. Their disadvantage is to be static so they cannot be added or modified at runtime.

2.3 JDO Concepts

2.3.1 Life Cycle Listeners

Life cycle listeners are an implementation of the publisher-subscriber respectively observer design pattern. [7] It it possible to register an event handler for each kind of event. On occurrence of such an event the publisher will notify the corresponding listener, which need to be registered at the persistence manager or its factory. For an example control flow see figure 2.1. In the JDO-API there are several listeners defined:

- AttachLifecycleListener - all attach events
- ClearLifecycleListener - all clear events
- CreateLifecycleListener - all object create events
- DeleteLifecycleListener - all object delete events
- DetachLifecycleListener - all detach events
- DirtyLifecycleListener - all dirty events
- LoadLifecycleListener - all load events
- StoreLifecycleListener - all store events

Since JDO 2.0 it is possible to detach[2] an object from the persistence graph, allowing it to be worked on in the users application and to be attached to the persistence graph again later

[2] `http://www.datanucleus.org/products/datanucleus/jdo/attach_detach.html`

on, which avoids manual copying of fields. *Attach* events are thrown during the execution of `makePersistent` on the detached instance before the copy is made, respectively on the persistent instance after the copy is made. Inversely, *detach* events are thrown during the execution of `detachCopy` on the persistent instance before the copy is made respectively on the detached instance after the copy is made.
Clear events are thrown before the implementation clears the values in the instance to their Java default values. This happens during an application call to evict, and in `afterCompletion` for commit with `RetainValues` set to false and rollback with `RestoreValues` set to false. [15]

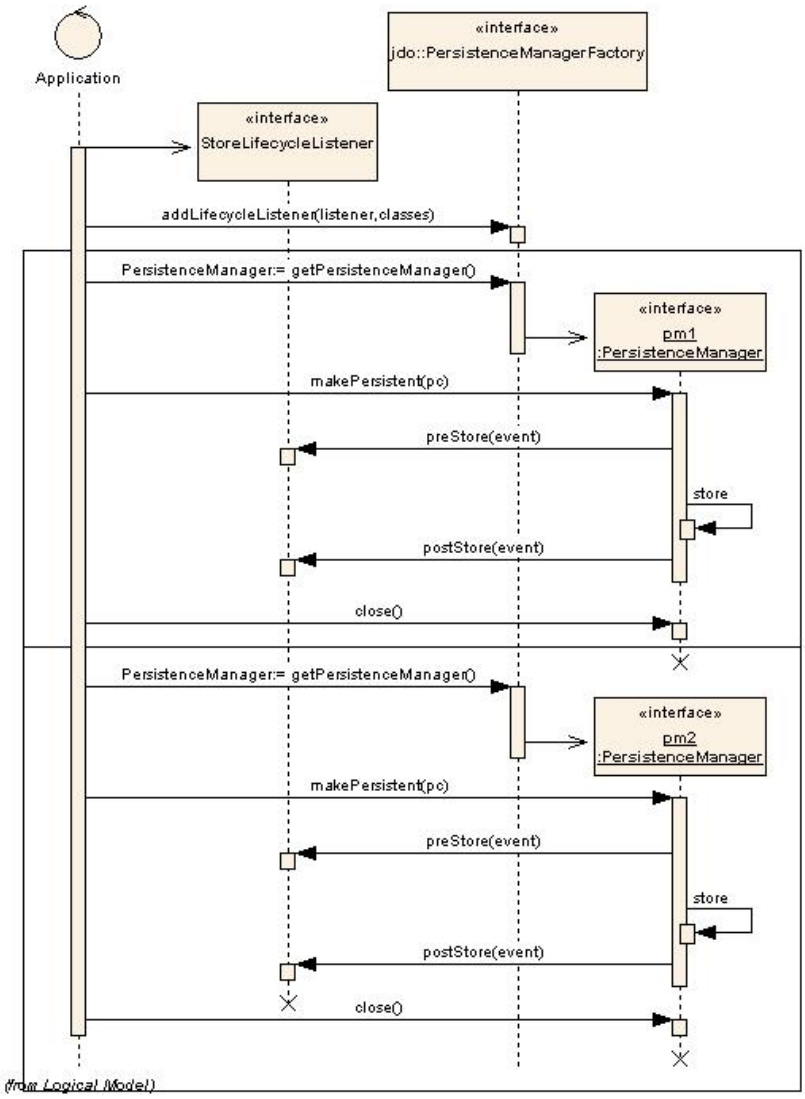

Figure 2.1: JDO Life Cycle Listener[3]

[3] `http://www.datanucleus.org/products/accessplatform/jdo/lifecycle_callbacks.html`

2.3.2 Instance Callbacks

Another very intuitive way are instance callbacks. For this purpose each persistent class needs to implement the instance callback interface, which provides the four methods `jdoPostLoad`, `jdoPreClear`, `jdoPreDelete` and `jdoPreStore`. A list of managed instances could then be maintained by adding objects on load and removing them on clear. In general it gets possible to inform other instances about state changes.
As interfaces are abstract the implementation needs to be done in the class definition of the object to store. This might be cumbersome as there exists already a managed objects list.

2.3.3 Managed Objects

Objects underlay also a life cycle. Right after creation they are in state *transient*. Making them persistent changes their state to *persistent new*. Marking an object for deletion changes its state to *persistent (new) deleted*. On commit, all *persistent* objects change to state *hollow*. Loading a *persistent* object makes it *persistent clean*, on modification it gets *persistent dirty*. Finally an object is in *detached* state when disconnected from the manager, for example, when the manager gets closed. So all objects in the current working set are maintained by the managed object list. By iterating through this list it is possible to check if an object is dirty and if so perform the appropriate constraint check. For detailed state transition see figure 2.2.

2.3.4 Synchronisation

With transaction synchronisation it is possible to set a single synchronisation instance to get informed about transaction state transitions. There are two methods which will be called before respectively after the state transition induced by either a commit or rollback.
In order to stay in a consistent state the constraint check need to be performed in the `beforeCompletion` method of the synchronisation instance, to prevent a constraint violating object to get stored.

2.4 File versus Annotation based Constraint Configuration

Although Plain Old Java Objects (POJOs) are great for runtime constraint configuration, it is tedious to establish them and not well documented how to do so. Apart from POJOs, there is a debate whether to annotate classes directly or to use a separate configuration file. Annotations are available since J2SE 5.0 released in 2004. [13] They are meta-data features allowing to associate additional information to Java elements without direct effect to the code or the way in which the source code is compiled. [14] Rocha et al. showed the so-called *annotation-hell* phenomena: extensive use of annotations could massively contribute to code pollution leading to poor code readability and understandability. On the other hand, annotations could serve as code level comments and therefore improve readability. Although changes to annotations always require code recompilation, refactoring is easier and often also assisted by the Integrated Development Environment (IDE). Sometimes it is, however, not possible to annotate the class itself as the source code is not available, then it is obvious to use an external configuration file. A configuration file is favoured when clear code separation is demanded. In particular, it is possible to do changes at runtime and then invoke a reconfiguration. In the end, the decision depends on many factors regarding the concrete application.

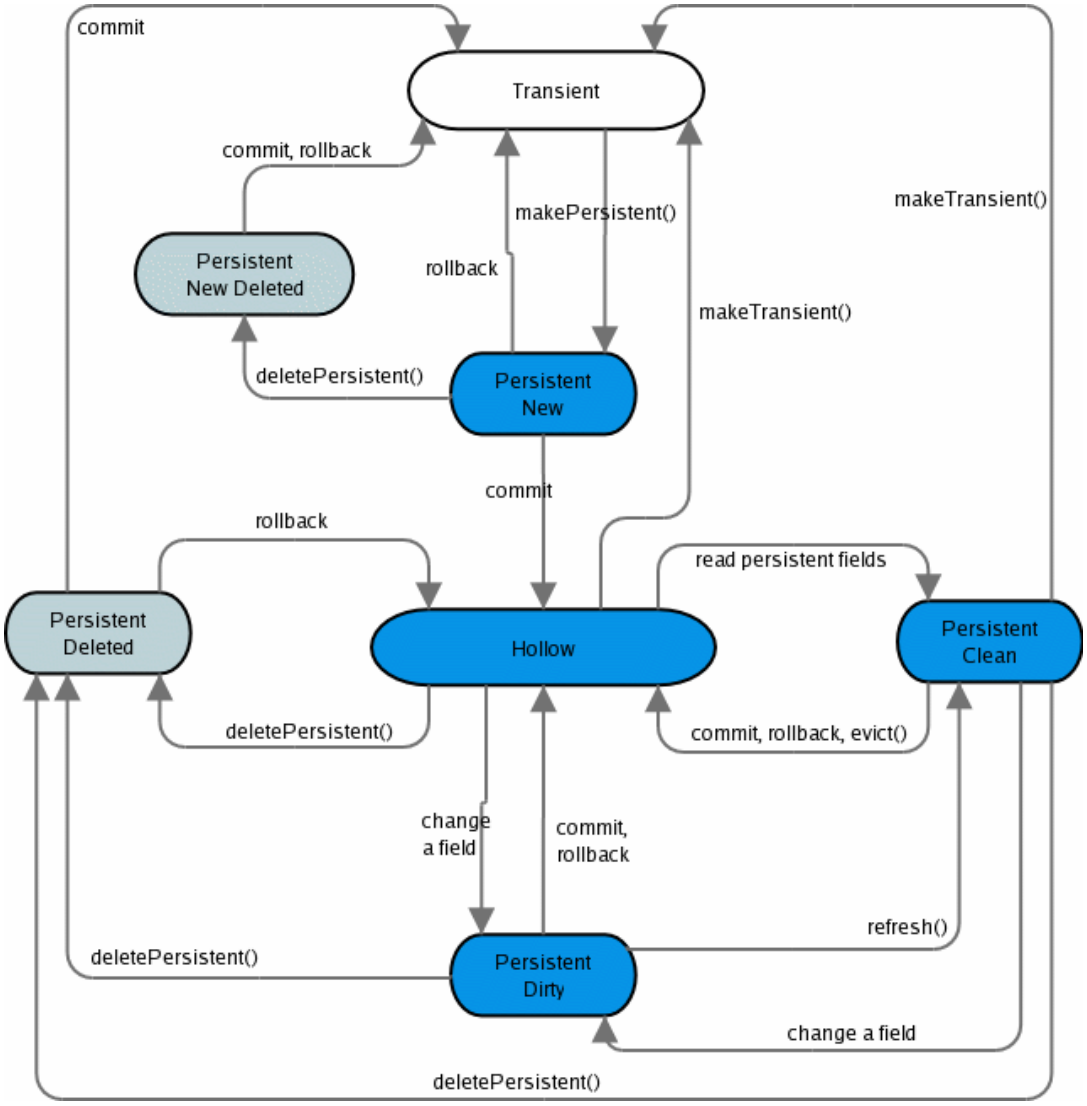

Figure 2.2: JDO State Transition[4]

[4] http://db.apache.org/jdo/state_transition.html

2.5 Dependencies

In this section we introduce the dependencies of our framework. In particular, we present the object database, the validation framework as well as the OCL parser and interpreter. The licenses listed for the dependencies are that of the projects themselves, and not of their dependencies.

2.5.1 ZooDB

ZooDB[5] is an object oriented database based on the JDO 3.0 standard. It is written by Tilmann Zaeschke of our GlobIS[6] research group and is still under development. Some of its current limitations directly affect our project. For example, *ZooDB* is limited to a single session and is not thread-safe. Furthermore, there is manual enhancement of classes required and some query features, such as aggregation, are only partially supported. It is licensed under GPLv3 (GNU Public License).

2.5.2 OVal

OVal[7] is a pragmatic and extensible general purpose validation framework. Constraints can be declared with annotations, POJOs or XML configuration files. Furthermore, custom constraints can be expressed as Java classes or by using scripting languages. It is licensed under EPLv1 (Eclipse Public License).

2.5.3 DresdenOCL

DresdenOCL[8] is a project of the Software Technology Group at Technische Universität Dresden, which provides a set of tools to parse and evaluate OCL constraints on various models like UML, EMF and Java. It is licensed under LGPLv3 (GNU Lesser General Public License).

[5] `https://github.com/tzaeschke/zoodb`
[6] `http://www.globis.ethz.ch/`
[7] `http://oval.sourceforge.net/`
[8] `http://www.dresden-ocl.org/`

3 Evaluated Approaches

3.1 Eclipse OCL

There is the aim for this thesis to use a general constraint definition language. Such a language is the Object Constraint Language (OCL) which was first developed at IBM as a complement to the Unified Modeling Language (UML). As UML is a graphical notation its expressiveness is limited to a subset of all relevant information. Whereas OCL as a textual language allows to precisely specify all detailed aspects. As a consequence OCL became integrated as a standard in UML only two years later in 1997. [2]
The IDE Eclipse, which is in widespread use and whose code base also originates from an IBM project, already provides support for *OMG OCL 2.3.1* as a plugin. So it was obvious to evaluate this first. While the language *Complete OCL* enhances an existing meta-model with invariant definitions in a separate file, the language *Essential OCL* is of limited use by itself as the model cannot be provided and hence the grammar needs to be extended to provide the missing context. The *OCLinEcore* language introduces OCL annotations for the Ecore model of the Eclipse Modeling Framework (EMF), so there is strong support by the editor for semantic checking etc. To be clear, Ecore is the core meta-model of the EMF and is defined in terms of itself.
There is also a generator provided by the EMF to translate the model in its corresponding model code. Furthermore, it is possible to use annotated interfaces and classes instead of a model for code generation.
Parsing an OCL string to a constraint and evaluating it against an object is then fairly easy.

3.2 OVal & DresdenOCL

There exist various validation frameworks for Java. Based on a comparison[1] of open source validation frameworks which was recommended on Stackoverflow[2] we decided to use the Object Validation Framework for Java (OVal) for our purpose as it provides the highest flexibility and extensibility. With OVal it is possible to declare constraints with annotations, Plain Old Java Objects (POJOs) and XML configuration files out of the box. This works for any kind of Java objects and is not restricted to Java Beans. [8] Nonetheless, it is possible to configure OVal to interpret Bean Validation (JSR303) and Enterprise JavaBean (EJB3)-Java Persistence API (JPA) annotations. [1], [5], [6]
Although OVal already supports many expression languages as beanshell, groovy, javascript, ruby and some more, there is no built-in support for OCL. Fortunately, it is possible to implement the support of new expression languages and register them for assertions as well as pre- and post-conditions. Moreover, it is possible to define custom annotations. While OVal is the glue and takes care of the validation process, *DresdenOCL* serves as its interpreter for OCL expressions, since it can be used as a standalone library.

3.3 OCL to Java by Hand

Building an OCL interpreter by hand requires first to write an OCL parser, which is very complex since OCL tightly relies on a model. *EssentialOCL* is extended to provide the missing context while supporting only the core functionality. Since the grammar is ambiguous it needs additional rules to make it more regular. Given such a parser, OCL expressions are translated into corresponding constraint objects, which provide some *validate* methods to be called by the validator itself. The validation framework has to maintain a mapping between objects and checks to perform.

3.4 Decision

With OVal there exists already a validation framework that takes care about the validation process and which is flexible enough for our needs. Its internal data structure retrieves all checks per class and validates them against the object. So primarily the OCL interpreter is to debate.
Building an own interpreter even for a subset of the OCL language is out of the scope of this bachelor thesis. Nonetheless, there would be some handy benefits. For example, all objects necessary to evaluate the expression could directly be requested from the database to set up the model instance of the interpreter.
Although *Eclipse OCL* is delivered with many tools for modelling and validation, its programmatic capabilities are limited. Constraint annotations are on the level of the model and not on the implementation code. The validation works only for one constraint per object and vice versa. However, this is the minor limitation. The major limitation is the dependency on an Ecore or UML meta model. Although there is a way to generate the model code out of existing code, it is very cumbersome as all methods have to be annotated first. Also the

[1] `http://java-source.net/open-source/validation`
[2] `http://stackoverflow.com/questions/397852/java-validation-frameworks`

generated output has to be revised by the developer. Furthermore, changes to the code need to be marked so they do not get overwritten on the next model generation run.
For all this reason we decided to use *DresdenOCL* as standalone interpreter. Its simplified model in Java is just much less a burden to the application programmer. There is no need for an UML or EMF meta-model to parse and evaluate OCL expressions.

4 Approach

In this chapter we describe our approach for defining, managing and validating constraints in object databases. Our framework supports basic constraint functionality as `Primary-` and `ForeignKey` checks as well as `Unique` checks. Furthermore, it is possible to define constraints in an uniform way in the OCL expression language, either as annotation, in a configuration file or as Plain Old Java Object (POJO)s. For that purpose we enhanced OVal with capabilities of *DresdenOCL* by implementing the support for OCL as expression language using the parser and interpreter provided by *DresdenOCL*.

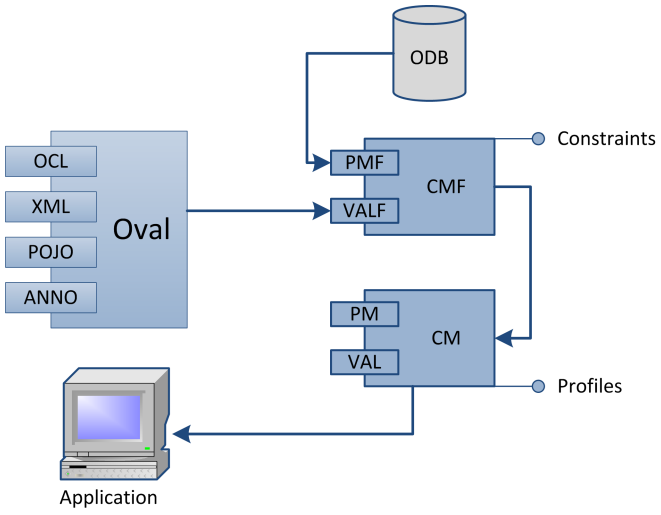

Figure 4.1: Simplified Architecture

The entry point into our framework is the Constraint Manager Factory (CMF) as shown on the upper right part of figure 4.1. It consists of a Persistence Manager Factory (PMF) and a Validator Factory (VALF), which acts as a reference validator for managing the constraints. Therefore, the CMF provides an interface to retain, add and remove constraint checks. As

motivated before in section 3.4 and indicated on the upper left part of figure 4.1, we selected OVal as validation framework. It supports three constraint configuration methods out of the box, such as central constraint management via XML configuration files, runtime configuration via POJOs as well as annotations on classes, attributes and methods. Furthermore, we enhanced the framework with an additional configurer for OCL configuration files and defined OCL as an additional expression language.
Keep in mind that the Persistence Manager Factory (PMF) as well as the Persistence Manager (PM) are integral parts of the JDO (JSR243) specification. [15] "The `PersistenceManager` is the primary interface for JDO-aware application components. It is the factory for `Query` and `Transaction` instances, and contains methods to manage the life cycle of `PersistenceCapable` instances."[1] Our conceptual idea is to replace the PMF and the PM in the application code with their counterparts CMF respectively CM. The Constraint Manager (CM) is the central part in our framework, as shown on the lower right part of figure 4.1. It consists of a PM, which is responsible for the transactional part, and a validator, which takes care about the validation process. The CM brings them into accordance, based on the coupling mode introduced in section 2.2.1 about the ECA paradigm; it determines when to invoke the validation process and on which objects. For each object to validate, the validator will request a set of applicable constraints from all its configurers. An interface at the CM allows to individually enable and disable constraint evaluation by profiles.

4.1 Constraints

We considered the evaluation of `Primary-` and `ForeignKey` checks as well as `Unique` checks. All of them have to check the database for corresponding entries. In case of the `ForeignKey` check, a match on the specified primary key is expected, while for `PrimaryKey` and `Unique` checks there must not exist an entry with the same key already. As we support composite keys, we therefore set up a query involving all attributes participating in the key.

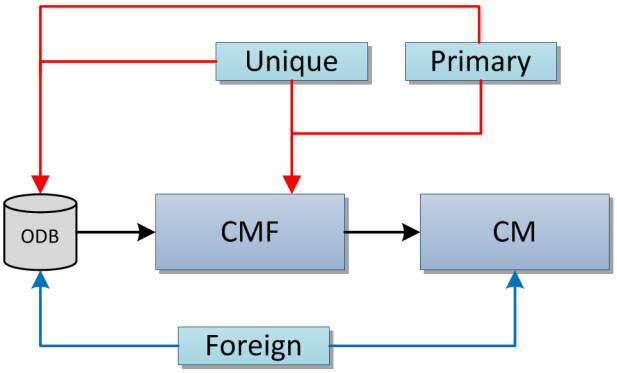

Figure 4.2: Simplified Checks

However, this is only a necessary and not a sufficient condition. Furthermore, we have to consider parallel transactions which is slightly more sophisticated. For `PrimaryKey` or `Unique` checks, only the first transaction applying for that key can proceed, while the later

[1] `http://db.apache.org/jdo/api30/apidocs/javax/jdo/PersistenceManager`

ones have to back off. Therefore, we have to check all managed object sets of concurrent transactions. For this purpose, the CMF maintains a list of released CMs. This is a conservative approach, as the earlier transaction still might abort. For the same reason, it is critical to `ForeignKey` checks, that the corresponding primary key is persisted at the database already or at least will be persisted within the same transaction.
Regarding the constraint types of section 2.1.2 we already covered `Primary-` and `ForeignKey` checks as well as `Unique` checks. Next we will consider `Check` constraint checks for OCL as expression language. As it is favourable to define multiple checks per object to be able to distinguish the reason for a constraint violation, such as profiles or severity, we defined a container element called `OclConstraints`. Listing 4.1 shows such a composite annotation containing two `OclConstraint`. Note that this container is not limited to annotations but can also be instantiated as a POJO.
To be complete, we also have to consider `NotNull` checks. However, as they are already an integral part of OVal, they do not require any further consideration.
Although in early papers views for object databases are mentioned, we did not encounter them during our research. In relational databases views provide a powerful abstraction mechanism. [16] According to the developer of *ZooDB*, the object oriented approach made them dispensable. Therefore, we did not follow `CheckWithOption` constraint checks.

Listing 4.1: OclConstraints

```
1 @OclConstraints({
2     @OclConstraint(expr="...", severity=..., profiles="..."),
3     @OclConstraint(expr="...", severity=..., profiles="...")
4 })
```

As introduced in section 2.1.3, object databases raised some new constraint concepts. OVal provides the possibility to specify a method or an inline class to check a property with. Just keep in mind that due to their universal computing power, they should be used with care, as side effects may destroy database integrity. Furthermore, OVal takes care of nested properties by specifying the target, which is the path to the object where the constraint should be applied. For that purpose, the object activation level as illustrated in figure 4.3 has to be determined carefully. The activation level, which is also known as activation depth, prevents the database to retrieve the complete object graph on traversing references. However, consider to place the constraint at the target itself.
To deal with sets, OCL as expression language for OVal is perfectly applicable. Complex objects do not require any further considerations, as they can be modelled by combining the concepts above. Last but not least OVal does full validation of the class inheritance tree, hence class hierarchy is also covered.

4.2 Constraint Evaluation Mode

The Constraint Manager (CM) requires considerations about the constraint evaluation mode introduced in section 2.2.1. *Deferred* constraint evaluation will be invoked automatically when the transaction commits. Furthermore, we provide *immediate* constraint evaluation, but we do not provide *separate* constraint evaluation so far. This behaviour is controlled by the coupling mode and can be changed at any time. Either way, when constraint evaluation gets

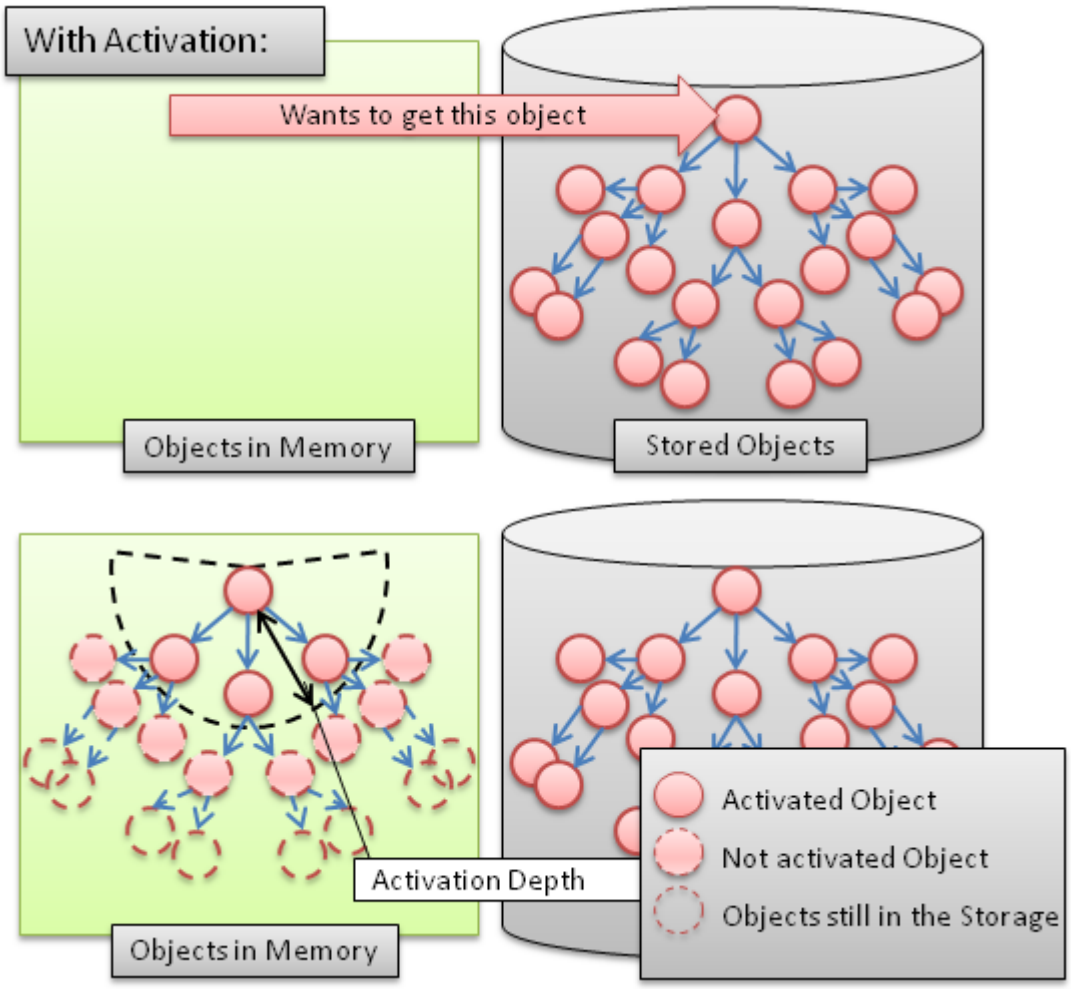

Figure 4.3: Object Activation Level[2]

invoked, the CM has to determine the objects to concern about. While *immediate* evaluation uses events to determine the single object to validate, *deferred* evaluation has to check a set of objects. Fortunately, JDO already demands the Persistence Manager (PM) to maintain a managed object list. This set can be (dramatically) reduced by demanding certain states[3] the objects have to be in. According to figure 2.2 the blue states are of interest:

- **Hollow**
  Any persistent object that represents data in the datastore, but where its values are not in the instance.
- **Persistent New**
  Any object that is newly persisted in the current transaction. A JDO identity has been assigned to these objects.

[2] `http://community.versant.com/Documentation/Reference/db4o-8.0/java/reference/Content/basics/activation.htm`

[3] `http://db.apache.org/jdo/state_transition.html`

- **Persistent Dirty**
  Any persistent object that has been changed in the current transaction.
- **Persistent Clean**
  Any persistent object that represents data in the datastore, and where its values have not been changed in the current transaction.

Obviously we have to check *persistent dirty* and *persistent new* objects. Objects in the state *persistent hollow* and *persistent clean* can be ignored as they are either already stored in the database or are unmodified and hence are supposed to be valid. If at least one constraint does not hold an exception will be thrown containing a list of violations and their causes.
Moreover, it is possible to assign to each constraint a severity level and some applicable profiles as shown in listing 4.1. Regarding the performance, profiles can be used to disable and enable constraint evaluation for some profiles individually, while the severity level can indicate the impact of constraint violations on data quality or automate their resolution. Furthermore, constraint checks can be retrieved, added or removed at runtime. While each CM inherits the checks of the reference validator at the point of creation, there is no upwards propagation of constraint checks.

5
Use of Framework

Often learning by doing facilitates understanding. So let us demonstrate how to set up the Constraint Manager Factory (CMF). First of all we need to obtain a Persistence Manager Factory (PMF) from the database. For this purpose we make use of the *JDOHelper* defined by the JDO-API. As arguments it takes properties obtained from the concrete database implementation, as shown in listing 5.1.

Listing 5.1: Persistence Manager Factory

```
1 // initialize persistence manager factory
2 ZooHelper.createDb(dbName);
3 Properties props = new ZooJdoProperties(dbName);
4 PersistenceManagerFactory pmf = JDOHelper.
    getPersistenceManagerFactory(props);
```

The next step is to initialise the Validator Factory (VALF) with a simplified OCL meta-model for *DresdenOCL* and a set of actual configurations. Each configuration has an individual profile and severity level assigned, as depicted in listing 5.2. In the example we set the profile to *oclConfigurer* and its severity to zero.

Listing 5.2: Validator Factory

```
1 // initialise validator factory
2 File modelProviderClass = new File(resources,"model/ch/ethz/oserb/
    ModelProviderClass.class");
3 File oclConfig = new File(resources, "constraints/constraints.ocl")
    ;
4 ArrayList<OCLConfig> oclConfigs = new ArrayList<OCLConfig>();
5 oclConfigs.add(new OCLConfig(oclConfig, "oclConfigurer", 0));
6 ValidatorFactory vf = new ValidatorFactory(modelProviderClass,
    oclConfigs);
```

Given the PMF and VALF we can finally initialise the CMF itself. From then on we can request a Constraint Manager (CM) from its factory. Although it is possible to change the coupling mode later on, we have to state the initial mode. For the example in listing 5.3 we selected *deferred* constraint evaluation mode.

Listing 5.3: Constraint Manager Factory

```
1 // initialise constraint manager factory
2 ConstraintManagerFactory.initialize(pmf,vf);
3 ConstraintManager cm = ConstraintManagerFactory.
    getConstraintManager(CouplingMode.DEFERRED);
```

If a database requires the schemes to be defined explicitly as *ZooDB* does, this should be done in a separate transaction preferably as soon as possible, as we did in listing 5.4.

Listing 5.4: Schema Definition

```
1 // schema definition
2 cm.begin();
3 ZooSchema.defineClass(cm.getPersistenceManager(), ExamplePerson.
    class);
4 cm.commit();
```

In listing 5.5 an example transaction is shown. First we disable all profiles and re-enable only the profile ”hard”. Then we make an *ExamplePerson* persistent so it gets managed by the persistence manager. On *immediate* evaluation mode every interaction causing a state transition of an object can throw a `ConstraintViolatedException`, which contains a list of `ConstraintViolations`. On *deferred* evaluation mode only commit will throw the exception. In case of an `OclConstraint` the violation provides additionally a list of causes which made the OCL expression to fail. Therefore, it is recommended to put the whole transaction inside a try-catch block and then iterate over the exception.

Listing 5.5: Example Transaction

```
1  try{
2      // begin transaction: write
3      cm.begin();
4      cm.disableAllProfiles();
5      cm.enableProfile("hard");
6      cm.makePersistent(new ExamplePerson("Fred",12,1));
7      cm.commit();
8  }catch (ConstraintsViolatedException e) {
9      // get constraint violations
10     for(ConstraintViolation constraintViolation : e.
           getConstraintViolations()){
11         LOG.error(constraintViolation.getMessage());
12     }
13     cm.forceCommit();
14 }finally{
15     assert(!cm.currentTransaction().isActive());
16 }
```

6 Implementation

6.1 Constraint Manager

As the Constraint Manager (CM) surrogates the Persistence Manager (PM) it simply delegates most of its methods. However, the commit method gets enhanced to invoke constraint evaluation. As motivated before *deferred* evaluation, such as in listing 6.1, will check all *persistent dirty* and *persistent new* objects in the managed objects set. For *immediate* evaluation, as shown in listing 6.2, it is sufficient to add a `ListenerCreate` and a `ListenerDirty` to the `InstanceLifecycleListeners`. The event-source already contains the reference to the object to validate. Validation collects all violations and will eventually throw an `ConstraintsViolatedException` or, in case of *deferred* evaluation, commit the transaction. If an exception is thrown, it is the obligation of the programmer to decide whether to force a commit or roll-back the transaction!

Listing 6.1: Deferred Evaluation

```
1  // deferred constraint evaluation
2  if(couplingMode==CouplingMode.DEFERRED){
3      List<ConstraintViolation> constraintViolations = new LinkedList
           <ConstraintViolation>();
4      for(Object obj:pm.getManagedObjects(EnumSet.of(ObjectState.
           PERSISTENT_DIRTY, ObjectState.PERSISTENT_NEW))){
5          constraintViolations.addAll((validate(obj)));
6      }
7      if(constraintViolations.size()>0)   throw new
           ConstraintsViolatedException(constraintViolations);
8  }
9  // if no constraint violated -> commit
10 pm.currentTransaction().commit();
```

Listing 6.2: Immediate Evaluation

```
1 // immediate evaluation
2 if(couplingMode==CouplingMode.IMMEDIATE)validateImmediate(event.
    getSource());
```

6.2 OCL Expression Language

Parsing an expression to a list of constraints requires a model. Interpreting this list of constraints then requires a corresponding model instance. For this purpose we make use of the simplified model provided by *DresdenOCL*. It is sufficient to put all class files into a special folder hierarchy in the resources folder corresponding to the package name. The necessary model can be simplified with a `ModelProviderClass` as shown in listing 6.3 and figure 6.1. Moreover, it is still possible to use an UML, XML or EMF model.

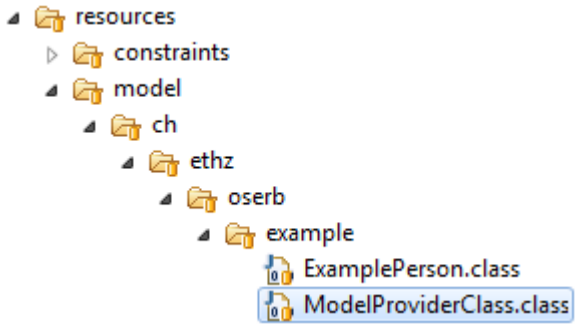

Figure 6.1: Simple Model Folder Hierarchy.

Listing 6.3: ModelProviderClass

```
1 package ch.ethz.oserb.example;
2 public class ModelProviderClass {
3     protected ExamplePerson examplePerson;
4 }
```

Listing 6.4: constraints.ocl

```
1 package ch::ethz::oserb::example
2     context Student
3     inv: self.age>21
4     inv: self.name.size()>6
5 endpackage
```

For each validation we set up a fresh model instance to put the object to validate in, as it is done in listing 6.5 on line 5. On line 8 we parse the given constraint expression versus the model and interpret the parsed constraints in the set up model instance on line 11. For that purpose we rely on the standalone parser and interpreter of *DresdenOCL*. Finally we prepare the list of constraint violations.

Listing 6.5: OCL Expression Language

```
// create empty model instance
modelInstance = new JavaModelInstance(model);

// add object to model
 modelInstance.addModelInstanceElement(values.get("_this"));

// parse OCL constraints from annotation
 List<Constraint> constraintList = Ocl22Parser.INSTANCE.
    parseOclString(expr, model);

// interpret OCL constraints
 List<IInterpretationResult> interpretationResults =
    StandaloneFacade.INSTANCE.interpretEverything(modelInstance,
    constraintList);

//return interpretationResults;
for (IInterpretationResult result : interpretationResults) {
    if(!((JavaOclBoolean)result.getResult()).isTrue()){
        constraintViolations.add(...);
    }
}
```

6.3 OCL Constraint

An `OclConstraint` is basically an assertion with the expression language predefined. Only the expression to evaluate is mandatory, all other parameters are optional, such as `profiles` and `severity`. As we want to allow several expressions in combination with different profiles respectively severities, but Java does not allow multiple annotations of the same type, we had to introduce a wrapper called `OclConstraints` with a list of `OclConstraint` as the only argument. Listing 6.6 shows the corresponding annotation interface. For each defined annotation we have to declare a class to check the annotation with. In particular this check-class provides the method *isSatisfied* to evaluate the check. Since we already implemented an interpreter, described in section 6.2, we make us of it analogous as general assertions also do. In contrast to them, we do not call the boolean method *isSatisfied* but *evaluate* directly so we do not discard the causes.

Listing 6.6: oclConstraints

```
@Retention(RetentionPolicy.RUNTIME)
@Target({ElementType.TYPE})
@Constraint(checkWith = OclConstraintsCheck.class)
public @interface OclConstraints
{
    OclConstraint[] value();
}
```

6.4 Primary- & ForeignKey, Unqiue

Regarding the constraint keywords introduced in subsection 2.1.2 we already cover `CHECK` with our OCL expression language. `PrimaryKey`- and `Unique`-checks are closely related, the first mentioned additionally ensures that none of the composite keys is null, as it is shown in listing 6.7.

Listing 6.7: notNull

```
1 // not null
2 if(keySet.containsValue(null)) return false;
```

For both constraints we have to check the database for corresponding entries. If such an entry already exists and it is not the object to validate itself, *unique* is not satisfied. Furthermore, there could be a concurrent transaction competing for the same *unique* composite key. Therefore, we have to check all managed object sets of such transactions whether the same composite key is already taken. For that purpose, the CMF maintains a list of released CMs. This approach is conservative, within the meaning of such a concurrent transaction still might abort. Even more, *ZooDB* is not prepared for concurrent transactions at all.
On the other hand, we conservatively determined that `ForeignKey`-checks only have to check the database and not the managed object sets, as the corresponding *PrimaryKey* could appear in a concurrent transaction that aborts. Thus, we demand that it needs to be persistent already.

6.5 OCL Configurer

Our OCL configurer works similar to OVals reference XML configurer. In principle it reads a document and feeds the validators POJO configurer with constraints. The validator then requests a `ClassConfiguration` per class. We do not want to parse the documents in its entirety yet, as the expressions get parsed and interpreted during validation. So we used a standard `StreamTokenizer` and a simple Finite State Machine (FSM) to identify and extract the important parts. In each **.ocl* document several contexts can be defined. Per context we then build a new `OclConstraint`.

6.5.1 StreamTokenizer

Given an input stream the `StreamTokenizer` parses it into tokens, such as `EOF`, `EOL`, `WORD` and `NUMBER`, which then can be processed one at a time. Since *.ocl files require a strictly defined structure a simple FSM is sufficient. We start in an initial state and expect the `package` to appear first. The package name implicitly states the path to the model provider class. During this first state transition, we remember the environment-path. Next we expect a `context` and perform another state transition. A context consists of its (class-)name, which must exist in the model provider class, and several invariants. The fully qualified path consists of the path to the environment and the class-name. If we encounter `EOF`, another context or the keyword `endpackage` we create a new constraint out of the StringBuilders content. In listing 6.8 we show the outline of the important parts of our simple parser.

Listing 6.8: StreamTokenizer & FSM

```
1         do {
2            int token = st.nextToken();
3            switch (token) {
4               ...
5               case StreamTokenizer.TT_WORD:
6                   // switch between keywords
7                   if(st.sval.equals("context")){
8                       if(state==STATE.PACKAGE){
9                           // fully qualified path
10                      }else if(state==STATE.CONTEXT){
11                          // create ocl constraint check
12                      }
13
14                      // start new context
15                      sb = new StringBuilder();
16                      sb.append(st.sval);
17
18                      // get context name
19                      st.nextToken();
20                      context = st.sval;
21                      sb.append("␣");
22                      sb.append(context);
23
24                      // state transition
25                      state = STATE.CONTEXT;
26                  }else if(st.sval.equals("package")){
27                      st.nextToken();
28                      sb.append(st.sval);
29                      // state transition
30                      state = STATE.PACKAGE;
31                  }else if(st.sval.equals("endpackage")){
32                     // create ocl constraint check
33                  }  else{
34                      // expression body
35                      if(state!=STATE.PACKAGE) sb.append("␣");
36                      sb.append(st.sval);
37                  }
38                  break;
39              ...
40           }
41        } while (!eof);
```

6.5.2 POJO Configurer

Another fairly large but notable part of the code is how to hand over the extracted constraints to the validator, as depicted in listing 6.9. OVal is designed for lazy evaluation. Checks for a class are obtained on demand and then cached at the validator itself. Thus, we either implement a method that extracts a specified context out of the configuration file or we parse the whole file at once and buffer it in a POJO-configurer as OVals reference XML-configurer does. We decided to stick to the reference implementation. Therefore, our OCL-configurer also maintains its own POJO-configurer, responsible for the methods `getClassConfiguration` and `getConstraintSetConfiguration`.
The first step in order to establish a new constraint as a POJO is to obtain the reference to the `ClassConfiguration` set. If there is none yet present, we have to create it first and register it at its direct ancestor in the hierarchy. While moving down the hierarchy, this step needs to be repeated for all elements if necessary. The second step is to obtain the corresponding `ClassConfiguration` and its object-level checks. There we lookup `OclConstraintsCheck` and add the `OclConstraintCheck`, which consists of the constraint expression, a set of profiles and its severity.

6.6 OVal Validator

Each Constraint Manager (CM) has its own independent validator, which requires separation of the `ClassConfiguration` set. Therefore, a kind of deep copy is involved to avoid sharing a reference to a common data structure, as depicted in listing 6.10. At least it requires independent lists of checks, so each CM can add or remove constraint checks without interfering with other CMs. For that purpose, we introduced a new constructor, which sets up new `ClassChecks` for each class occurring in the mapping of the reference validator. Since we provide methods for managing *object*-, *method*- and *field*-level checks as well as *constraint sets*, we have to go down on their level. For each kind of check we set up a new set and add all corresponding checks, before appending the set to the `ClassChecks`. Finally, we map the class to its corresponding `ClassChecks`. Although the checks are the same, we maintain for each CM a separate data structure. Thus, there is no upward propagation of constraint checks, even though each CM inherits the constraint checks of the reference validator on construction.

Listing 6.9: POJO Configurer

```
1  // get reference to class configurations
2  Set<ClassConfiguration> classConfigurations = pojoConfigurer.
       getClassConfigurations();
3  if(classConfigurations == null){
4      // setup new class configuration
5      classConfigurations = new HashSet<ClassConfiguration>();
6      pojoConfigurer.setClassConfigurations(classConfigurations);
7  }
8
9  // get reference to class configuration
10 ClassConfiguration classConfiguration = pojoConfigurer.
       getClassConfiguration(Class.forName(context));
11 if(classConfiguration == null){
12     // setup class configuration
13     classConfiguration = new ClassConfiguration();
14     classConfigurations.add(classConfiguration);
15     classConfiguration.type = Class.forName(context);
16 }
17
18 // get reference to object configuration
19 ObjectConfiguration objectConfiguration = classConfiguration.
       objectConfiguration;
20 if(objectConfiguration == null){
21     // setup object configuration
22     objectConfiguration = new ObjectConfiguration();
23     classConfiguration.objectConfiguration = objectConfiguration;
24     objectConfiguration.checks = new ArrayList<Check>();
25 }
26
27 // get reference to OclConstraintsCheck
28 List<Check> checks = objectConfiguration.checks;
29 OclConstraintsCheck oclConstraintsCheck=null;
30 for(Check check:checks){
31     if(check instanceof OclConstraintsCheck) oclConstraintsCheck =
           (OclConstraintsCheck) check;
32 }
33 if(oclConstraintsCheck==null){
34     // setup oclConstraintsCheck
35     oclConstraintsCheck = new OclConstraintsCheck();
36     objectConfiguration.checks.add(oclConstraintsCheck);
37 }
38
39 // define oclConstraint
40 OclConstraintCheck oclCheck = new OclConstraintCheck();
41 oclCheck.setExpr(expr);
42 oclCheck.setProfiles(profiles);
43 oclCheck.setSeverity(severity);
44
45 // append oclConstraint
46 oclConstraintsCheck.checks.add(oclCheck);
```

Listing 6.10: Deep Copy

```
1  // deep copy
2  for(Entry<Class<?>, ClassChecks> entry : classChecks.entrySet()){
3      ClassChecks cc = new ClassChecks(entry.getKey(),
           parameterNameResolver);
4      // object checks
5      Set<Check> oc = new LinkedHashSet<Check>(2);
6      oc.addAll(entry.getValue().checksForObject);
7      cc.addObjectChecks(oc);
8      // field checks
9      for(Entry<Field, Set<Check>> field:entry.getValue().
           checksForFields.entrySet()){
10         Set<Check> fc = new LinkedHashSet<Check>(2);
11         fc.addAll(field.getValue());
12         cc.addFieldChecks(field.getKey(), fc);
13     }
14     // method checks
15     for(Entry<Method, Set<Check>> method:entry.getValue().
           checksForMethodReturnValues.entrySet()){
16         Set<Check> mc = new LinkedHashSet<Check>(2);
17         mc.addAll(method.getValue());
18         cc.addMethodReturnValueChecks(method.getKey(), TRUE, mc);
19     }
20     checksByClass.put(entry.getKey(),cc);
21 }
22 // constraint set
23 for(Entry<String, ConstraintSet> entry:constraintSetsById.entrySet
       ()){
24     ConstraintSet cs = new ConstraintSet(entry.getValue().getId());
25     cs.setChecks(entry.getValue().getChecks());
26     constraintSetsById.put(entry.getKey(),cs);
27 }
```

7 Extensibility

7.1 Expression Language

Extending OVal with an additional expression language is straightforward. The desired expression language needs to implement the `ExpressionLanguage` interface, as depicted in listing 7.1. Only two methods have to be implemented whereas `evaluateAsBoolean` can refer to `evaluate`. Finally, the expression language implementation needs to be registered at the validator:

Validator.addExpressionLanguage(languageId, expressionLanguage)

Listing 7.1: Additional Expression Language

```
1  public class ExpressionLanguageImpl implements ExpressionLanguage {
2      @Override
3      public List<String> evaluate(String expr, Map<String, ?> val)
4      {
5          ...
6          return constraintViolations;
7      }
8      @Override
9      public boolean evaluateAsBoolean(String expr, Map<String, ?>
           val){
10         final List<String> result = evaluate(expr, values);
11         return result.size()==0;
12     }
13 }
```

7.2 Configurer

Extending OVal with an additional configurer is similarly straightforward as adding an expression language, but not that well documented. Therefore, we considered OVals XML configurer and adapted it for our purpose. The new configurer needs to implement the `Configurer` interface, as depicted in listing 7.2. The two methods `getClassConfiguration` and `getConstraintSetConfiguration` have to be implemented. This is preferably done via a POJO-configurer, which already provides corresponding methods.

Listing 7.2: Additional Configurer

```
1  public class OCLConfigurer implements Configurer {
2      private POJOConfigurer pojoConfigurer = new POJOConfigurer();
3      @Override
4      public ClassConfiguration getClassConfiguration(Class<?> clazz)
5      {
6          return pojoConfigurer.getClassConfiguration(clazz);
7      }
8      @Override
9      public ConstraintSetConfiguration getConstraintSetConfiguration
           (String constraintSetId){
10         return pojoConfigurer.getConstraintSetConfiguration(
               constraintSetId);
11     }
12 }
```

7.3 Annotation

Extending OVal with an additional annotation requires two steps. First, the annotation has to be defined, as already shown in listing 6.6. Then the corresponding `checkWith` class has to be implemented, as depicted in listing 7.3. The important part is the `isSatisfied` method, which will be called by the validator.

Listing 7.3: Additional Annotation

```
1  public class Check extends AbstractAnnotationCheck<Annotation> {
2      @Override
3      public void configure(final Check constraintAnnotation){
4          super.configure(constraintAnnotation);
5      }
6      @Override
7      public boolean isSatisfied(final Object validatedObject, final
           Object valueToValidate, final OValContext context, final
           Validator validator){
8          ...
9      }
10 }
```

8 Proof-of-Concept

In this section we demonstrate the constraint functionality of our framework, by covering the major constraint types. In particular, we developed a demonstrator web application to show that the framework actually has the potential for being used.

8.1 Test Cases

Test cases not only provide a good starting point for any programmer to comprehend the code, but also serve as a proof-of-concept. For that reason we usually establish a baseline which is expected to work properly. Then we violate intentionally some constraints and expect the framework to catch them all. Nevertheless, the main purpose of test cases is to detect flaws.

8.2 Demo Application

Although test cases already indicate the absence of flaws, it is nonetheless just a theoretical setting testing only limited parts at once. Thus, in practice there might occur situations not covered by a test case. Therefore, we considered a simplified university data model as a web application. According to figure 8.1, it consists only of students, professors, departments and lectures. Keep in mind that Java Beans require getter and setter methods. Students can attend several lectures, while lectures can be attended by several students. Each lecture is hold by exactly one professor, while a professor can hold several lecturers. Each professor belongs to exactly one department, but several professors can belong to the same department. For each entity we defined several constraints, so that every constraint type will be covered eventually. In particular, student entities have their age constrained by an OCL annotation, while their name length is constrained by an OCL configuration file. All entities have their *primary* key marked via annotation. Furthermore, professor and lecture entities make us of the *foreign* key annotation. While professor entities require a valid department, lectures require a professor. Lectures also ensure that the lecture name is *unique*.

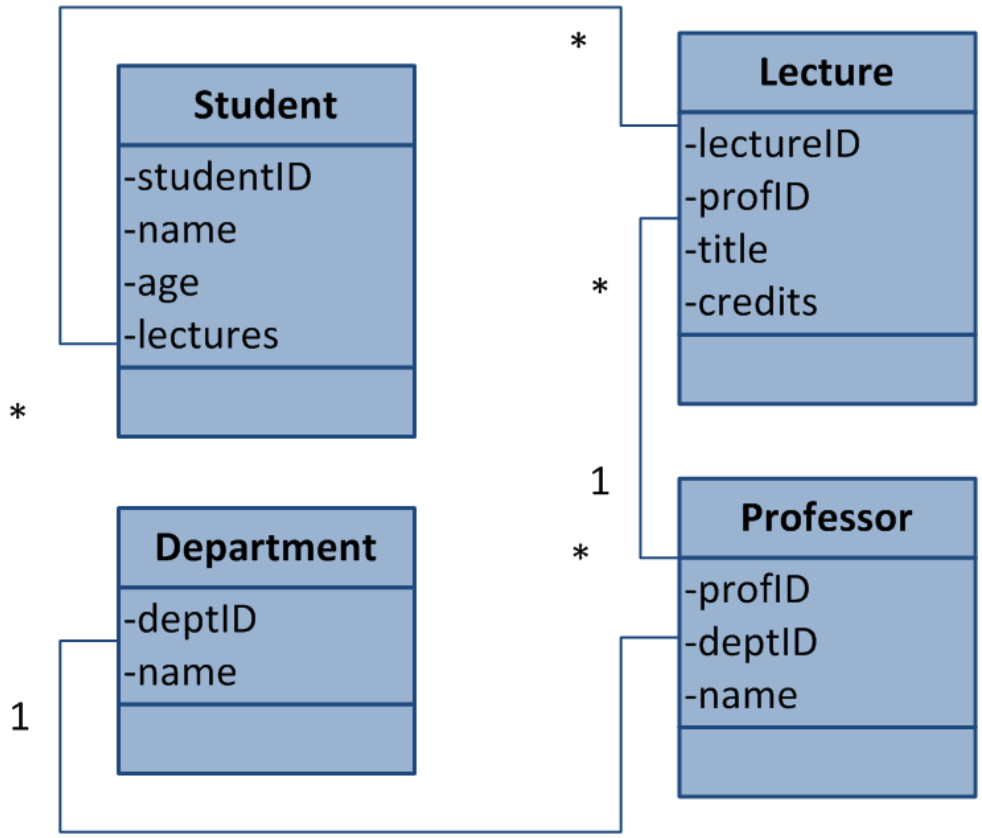

Figure 8.1: Demo Model

On the server side we used Apache Tomcat[1] version 7 in combination with Java Server Faces (JSF)[2] version 2.2. We separated the view from the actual data model by introducing a particular *managed bean* as intermediate layer according to the Model-View-Controller (MVC) design pattern. Each transactional bean get the reference to the constraint manager via dependency injection as a managed property. While views are obviously view-scoped, the constraint manager is application-scoped. Hence, its annotated post-construct and pre-destroy methods will be called only once. So we can ensure that the web application acquires only one Constraint Manager (CM) per session.

[1] `http://tomcat.apache.org/`

[2] `http://www.oracle.com/technetwork/java/javaee/javaserverfaces-139869.html`

Figure 8.2 shows the landing page of our demonstrator application. For each entity we provide an input form and an overview page where entities can be removed. Furthermore, it is possible to disable respectively enable profiles all at once or individually by name. Even more, the CM is prepared to retrieve, remove or add constraint checks at runtime. Figure 8.3 shows the example lecture form with the three major constraints violated. Figure 8.4 shows both kinds of OCL constraints violated.

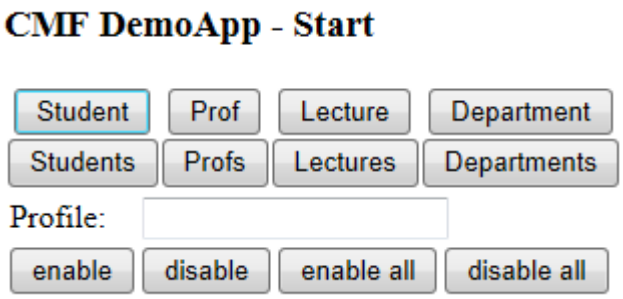

Figure 8.2: Demo Start

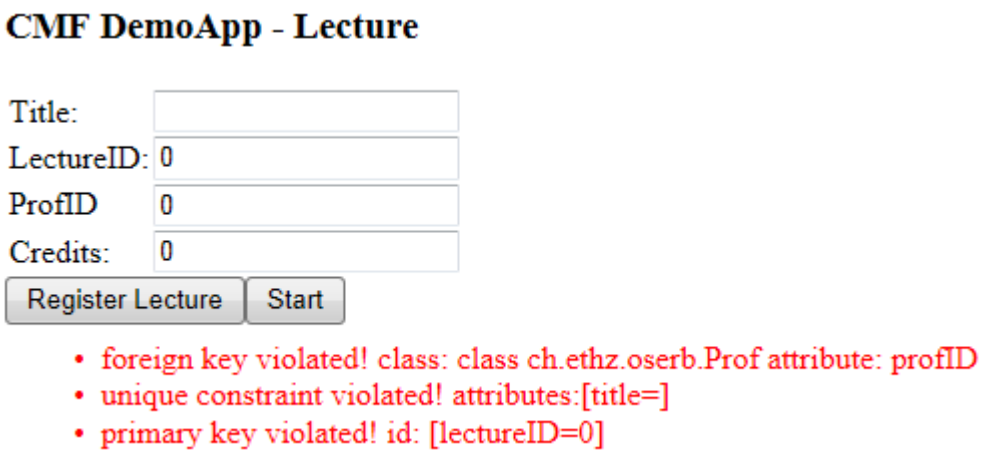

Figure 8.3: Demo Form

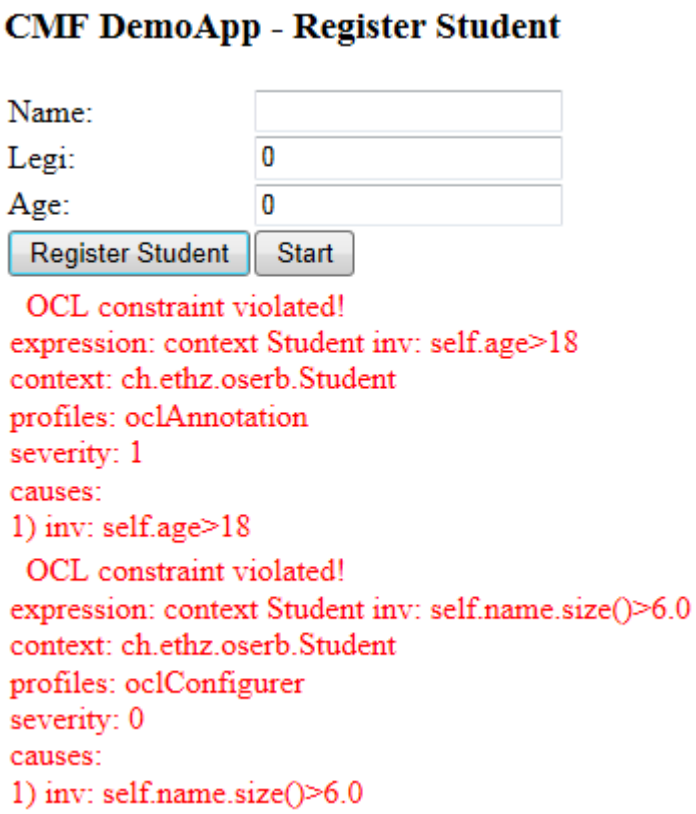

Figure 8.4: Demo OCL

9 Conclusion

9.1 Contribution

Besides the major contribution of providing constraint functionality on top of any JDO aware database, we enhanced OVal not only with OCL as an expression language but also as an configurer for configuration files. Furthermore, we introduced nested OCL annotations and outlined how to extend OVal in three different fashions. Beyond that, we support *ForeignKey* as well as composite *PrimaryKey* and *Unique* constraints, whereas especially the composite part outstands established object databases. For *ZooDB* in particular, we contributed a simple way of enhancing it with constraint functionality without any changes at its source code.

9.2 Future Work

9.2.1 Separate Evaluation

Separate constraint evaluation as proposed by the ECA paradigm introduced in section 2.2.1 requires a sophisticated roll-back mechanism. Especially cascading aborts have to be considered. It is likely that there is no way to achieve this in its entirety via the JDO-API. Hence, direct modification of the database become necessary. As we preferred to avoid any modification at the database, we suspended separate evaluation so far.

9.2.2 Referential Integrity

Although we provide partial referential integrity via *primary* and *foreign* keys, we did not implement cascading updates and deletions. On the one hand, changing the value of a *foreign* key still requires the referenced value to exists. On the other hand, changing the value of the referenced *primary* key does not automatically invoke re-validation of its *foreign* key. Furthermore, we do not prohibit to delete the corresponding primary key. So referential integrity is only guaranteed at the point of insertion.

9.2.3 Persist Constraint Checks

For the reuse of constraints, import and export functionality is desirable. This is straightforward for configuration files. However, it is much more complex for annotations. Furthermore, we would like to persist constraints modified at runtime. For that purpose, we would like to dump them to the database. This would require to persist the `ClassChecks` set. Unfortunately *ZooDB* requires each object to persist to implement the `ZooPCImpl` interface. To not limit our framework to *ZooDB*, we suspended this option.

A Appendix

A.1 Acronyms

API	Application Programming Interface 1
CM	Constraint Manager 18
CMF	Constraint Manager Factory 17
ECA	Event-Condition-Action 6
EJB3	Enterprise JavaBean 14
EMF	Eclipse Modeling Framework 13
FSM	Finite State Machine 28
IDE	Integrated Development Environment 9
JDO	Java Data Objects iii
JPA	Java Persistence API 14
JSF	Java Server Faces 36
MVC	Model-View-Controller 36
OCL	Object Constraint Language iii
OVal	Object Validation Framework for Java iii
PM	Persistence Manager 18
PMF	Persistence Manager Factory 17
POJO	Plain Old Java Object 17
UML	Unified Modeling Language 13
VALF	Validator Factory 17
XML	Extensible Markup Language 5

A.2 OVal Constraints

- **Assert**
  Check if evaluating the expression in the specified expression language returns true.
- **AssertConstraintSet**
  Check if the value satisfies the all constraints of specified constraint set.
- **AssertFalse**
  Check if the value is false. Note: This constraint is also satisfied when the value to validate is null, therefore you might also need to declare the NotNull constraint.
- **AssertFieldConstraints**
  Check if the value satisfies the constraints defined for the specified field.
- **AssertNull**
  Check if null.
- **AssertTrue**
  Check if the value is true. Note: This constraint is also satisfied when the value to validate is null, therefore you might also need to declare the NotNull constraint.
- **AssertURL**
  Check if the value is a valid URL. Note: This constraint is also satisfied when the value to validate is null, therefore you might also need to declare the NotNull constraint.
- **AssertValid**
  Check if the value passes a validation by Validator.validate().
- **CheckWith**
  Check the value by a method of the same class that takes the value as argument and returns true if valid and false if invalid.
- **DateRange**
  Check if the date is within the a date range. Note: This constraint is also satisfied when the value to validate is null, therefore you might also need to declare the NotNull constraint.
- **Digits**
  Check if the String representation has the given max/min number of integral and fractional digits. Note: This constraint is also satisfied when the value to validate is null, therefore you might also need to declare the NotNull constraint.
- **EqualToField**
  Check if value equals the value of the referenced field. Note: This constraint is also satisfied when the value to validate is null, therefore you might also need to declare the NotNull constraint.
- **Email**
  Check if the string representation is a valid e-mail address. Note: This constraint is also satisfied when the value to validate is null, therefore you might also need to declare the NotNull constraint.
- **Future**
  Check if the date is in the future. Note: This constraint is also satisfied when the value to validate is null, therefore you might also need to declare the NotNull constraint.

- **HasSubstring**
  Check if the string contains a certain substring. Note: This constraint is also satisfied when the value to validate is null, therefore you might also need to declare the NotNull constraint.
- **InstanceOf**
  Check if the value is an instance of the specified class or implements *all* specified interfaces. Note: This constraint is also satisfied when the value to validate is null, therefore you might also need to declare the NotNull constraint.
- **InstanceOfAny**
  Check if the value is an instance of the specified class or implements *one* of the specified interfaces. Note: This constraint is also satisfied when the value to validate is null, therefore you might also need to declare the NotNull constraint.
- **Length**
  Check if the string representation has certain length. Note: This constraint is also satisfied when the value to validate is null, therefore you might also need to declare the NotNull constraint.
- **MatchPattern**
  Check if the specified regular expression pattern is matched. Note: This constraint is also satisfied when the value to validate is null, therefore you might also need to declare the NotNull constraint.
- **Max**
  Check if the number is smaller than or equal to [max]. Note: This constraint is also satisfied when the value to validate is null, therefore you might also need to declare the NotNull constraint.
- **MaxLength**
  Check if the string representation does not exceed the given length. Note: This constraint is also satisfied when the value to validate is null, therefore you might also need to declare the NotNull constraint.
- **MaxSize**
  Check if an array or collection does not exceed the given size. Note: This constraint is also satisfied when the value to validate is null, therefore you might also need to declare the NotNull constraint.
- **MemberOf**
  Check if the string representation is contained in the given string array. Note: This constraint is also satisfied when the value to validate is null, therefore you might also need to declare the NotNull constraint.
- **Min**
  Check if the number is greater than or equal to [min] Note: This constraint is also satisfied when the value to validate is null, therefore you might also need to declare the NotNull constraint.
- **MinLength**
  Check if the string representation has at least the given length. Note: This constraint is also satisfied when the value to validate is null, therefore you might also need to declare the NotNull constraint.

- **MinSize**
  Check if the array or collection has at least the given number of elements. Note: This constraint is also satisfied when the value to validate is null, therefore you might also need to declare the NotNull constraint.
- **NoSelfReference**
  Check that the value is not a reference to the validated object itself. This is e.g. useful to avoid circular references.
- **NotBlank**
  Check if the string representation is not empty and does not only contain white spaces. Note: This constraint is also satisfied when the value to validate is null, therefore you might also need to declare the NotNull constraint.
- **NotEmpty**
  Check if the string representation is not empty (””). Note: This constraint is also satisfied when the value to validate is null, therefore you might also need to declare the NotNull constraint.
- **NotEqual**
  Check if the string representation does not equal a given string. Note: This constraint is also satisfied when the value to validate is null, therefore you might also need to declare the NotNull constraint.
- **NotEqualToField**
  Check if value does not equal the value of the referenced field. Note: This constraint is also satisfied when the value to validate is null, therefore you might also need to declare the NotNull constraint.
- **NotMatchPattern**
  Check if the specified regular expression pattern does not match. Note: This constraint is also satisfied when the value to validate is null, therefore you might also need to declare the NotNull constraint.
- **NotMemberOf**
  Check if the string representation is not contained in the given string array. Note: This constraint is also satisfied when the value to validate is null, therefore you might also need to declare the NotNull constraint.
- **NotNegative**
  Check if the number is greater or equal zero. Note: This constraint is also satisfied when the value to validate is null, therefore you might also need to declare the NotNull constraint.
- **NotNull**
  Check if not null.
- **Past**
  Check if the date is in the past. Note: This constraint is also satisfied when the value to validate is null, therefore you might also need to declare the NotNull constraint.
- **Range**
  Check if the number is in the given range. Note: This constraint is also satisfied when

the value to validate is null, therefore you might also need to declare the NotNull constraint.

- **Size**
  Check if the array or collection has the given size. Note: This constraint is also satisfied when the value to validate is null, therefore you might also need to declare the NotNull constraint.
- **ValidateWithMethod**
  Check the value by a method of the same class that takes the value as argument and returns true if valid and false if invalid

List of Figures

Acknowledgments

Foremost, I am deeply grateful to David Weber for offering me such an interesting topic of investigation. I appreciate his expertise as well as his inputs during our meetings.

Furthermore, I thank my fellow students in the lab for our questioning discussions. Special thanks goes to my esteemed friend Michael Troendle for reading my thesis and offering helpful advice based on his own recent experience in writing a thesis.

I also thank my girlfriend for her patience and motivating me for the last six months.

Last but not the least, I would like to thank my family, that they never stopped believing in me.

Bibliography

[1] E. Bernard. *JSR 303: Bean Validation*. Red Hat Inc, https://jcp.org/aboutJava/communityprocess/final/jsr303/, 2009.

[2] J. Cabot and M. Gogolla. Object constraint language (ocl): A definitive guide. In *Formal Methods for Model-Driven Engineering*, pages 58–90. Springer, 2012.

[3] C. Date. An introduction to database systems.

[4] U. Dayal, A. P. Buchmann, and D. R. McCarthy. Rules are objects too: A knowledge model for an active, object-oriented databasesystem. In *Lecture notes in computer science on Advances in object-oriented database systems*, pages 129–143, New York, NY, USA, 1988. Springer-Verlag New York, Inc.

[5] L. DeMichiel. *JSR 338: Java Persistence API 2.1*. Oracle, https://jcp.org/aboutJava/communityprocess/final/jsr338/, 2013.

[6] L. DeMichiel and M. Keith. *JSR 220: Enterprise JavaBeans 3.0*. Sun Microsystems Inc and Oracle, https://jcp.org/aboutJava/communityprocess/final/jsr220/, 2006.

[7] E. Gamma, R. Helm, R. Johnson, and J. Vlissides. *Design Patterns: Elements of Reusable Object-oriented Software*. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 1995.

[8] G. Hamilton. *Java Beans*. Sun Microsystems Inc, http://www.oracle.com/technetwork/java/javase/documentation/spec-136004.html, 1997.

[9] H. Ishikawa and K. Kubota. An active object-oriented database: A multi-paradigm approach to constraint management. In *VLDB*, pages 467–478, 1993.

[10] B. Meyer. *Eiffel: the language*. Prentice-Hall, Inc., 1992.

[11] B. Meyer. *Design by contract*. Prentice Hall, 2002.

[12] M. Morgenstern. Active databases as a paradigm for enhanced computing environments. In *Proceedings of the 9th International Conference on Very Large Data Bases*, VLDB '83, pages 34–42, San Francisco, CA, USA, 1983. Morgan Kaufmann Publishers Inc.

[13] M. Reinhold. *JSR 176: J2SE 5.0*. Sun Microsystems Inc, https://jcp.org/aboutJava/communityprocess/final/jsr176/index.html, 2004.

[14] H. Rocha and M. T. Valente. How annotations are used in java: An empirical study. In *SEKE*, pages 426–431, 2011.

[15] C. Russell. *JSR243: Java Data Objects 3.0*. Sun Microsystems Inc, `https://jcp.org/aboutJava/communityprocess/mrel/jsr243/index3.html`, 2010.

[16] M. H. Scholl and H.-J. Schek. Supporting views in object-oriented databases. *IEEE Data Eng. Bull.*, 14(2):43–47, 1991.

[17] D. Weber, S. Leone, and M. C. Norrie. Constraint-based data quality management framework for object databases. In *Proc. 21st European Conf. on Information Systems (ECIS 2013)*, June 2013.

[18] J. S. Won Kim, Yoon-Joon Lee. A framework for supporting triggers in object-oriented database systems. *International Journal of Intelligent and Cooperative Information Systems*, 1:127–143, 1992.