Investigating a Constraint-Based Approach to Data Quality in Information Systems

Investigating a Constraint-Based Approach to Data Quality in Information Systems

Master Thesis

Oliver Probst
<oprobst@student.ethz.ch>

Prof. Dr. Moira C. Norrie
David Weber

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

8th October 2013

ETH
Eidgenössische Technische Hochschule Zürich
Swiss Federal Institute of Technology Zurich

globis

ABSTRACT

Constraints are tightly coupled with data validation and data quality. The author investigates in this master thesis to what extent constraints can be used to build a data quality management framework that can be used by an application developer to control the data quality of an information system. The conceptual background regarding the definition of data quality and its multidimensional concept followed by a constraint type overview is presented in detail. Moreover, the results of a broad research for existing concepts and technical solutions regarding constraint definition and data validation with a strong focus on the Java™ programming language environment are explained. Based on these insights, we introduce a data quality management framework implementation based on the Java™ Specification Request (JSR) 349 (Bean Validation 1.1) using a single constraint model which avoids inconsistencies and redundancy within the constraint specification and validation process. This data quality management framework contains advanced constraints like an association constraint which restricts the cardinalities between entities in a dynamic way and the concept of temporal constrains which allows that a constraint must only hold at a certain point in time. Furthermore, a time-triggered validation component implementation which allows the scheduling of validation jobs is described. The concept of hard and soft constraints is explained in detail and supplemented with a implementation suggestion using Bean Validation. Moreover, we explain how constraints could be used to increase the data quality. A demonstrator application shows the utilisation of the data quality management framework.

ACKNOWLEDGMENTS

I am indebted to my supervisor David Weber who supported me during my master thesis and his colleague Dr. Karl Presser who gave us great insights into his point of view with respect to this master thesis topic. Thanks to global information systems group under the direction of Prof. Dr. Moira C. Norrie. Lastly, I would like to thank my colleagues, friends and family who have helped me in my work in any way.

CONTENTS

I INTRODUCTION 1

1 MOTIVATION 3

2 STRUCTURE 7

II CONCEPT BACKGROUND 9

3 DATA QUALITY 11

4 DATA QUALITY DIMENSIONS 15
4.1 DATA QUALITY DIMENSION: DISCOVERY 15
4.1.1 THEORETICAL APPROACH 15
4.1.2 EMPIRICAL AND INTUITIVE APPROACH 16
4.2 DATA QUALITY DIMENSION: DESCRIPTION 17
4.2.1 ACCURACY 17
4.2.2 COMPLETENESS 18
4.2.3 CONSISTENCY 19
4.2.4 TEMPORAL DATA QUALITY DIMENSIONS 20
4.2.4.1 TIMELINESS 20
4.2.4.2 CURRENCY 21
4.2.4.3 VOLATILITY 21
4.2.5 OTHER DATA QUALITY DIMENSIONS 21

5 CONSTRAINT TYPES 23
5.1 DEFINITION 23
5.2 TYPES 24
5.2.1 DATA RULES 24
5.2.2 ACTIVITY RULES 27

III RESEARCH BACKGROUND 28

6 CROSS-TIER VALIDATION 29
6.1 CONSTRAINT SUPPORT IN MDA TOOLS: A SURVEY 29
6.2 INTERCEPTOR BASED CONSTRAINT VIOLATION DETECTION 30

6.3 TOPES: REUSABLE ABSTRACTIONS FOR VALIDATING DATA . 31

7 PRESENTATION TIER VALIDATION 33
7.1 POWERFORMS 33

8 LOGIC TIER VALIDATION 35
8.1 CROSS-LAYER VALIDATION 35
8.1.1 INTEGRATION OF DATA VALIDATION AND USER INTERFACE CONCERNS IN A DSL FOR WEB APPLICATIONS 35
8.2 PRESENTATION LAYER VALIDATION 37
8.2.1 MODEL-DRIVEN WEB FORM VALIDATION WITH UML AND OCL 37
8.3 BUSINESS LAYER VALIDATION 38
8.3.1 OVERVIEW AND EVALUTION OF CONSTRAINT VALIDATION APPROACHES IN JAVA 39
8.3.2 LIMES: AN ASPECT-ORIENTED CONSTRAINT CHECKING LANGUAGE 39
8.3.3 VALIDATION APPROACHES USING OCL 41
8.4 DATA ACCESS LAYER VALIDATION 41
8.4.1 CONSTRAINT-BASED DATA QUALITY MANAGEMENT FRAMEWORK FOR OBJECT DATABASES 42

9 DATA TIER VALIDATION 43
9.1 OBJECT-ORIENTED DATABASES 43
9.2 RELATIONAL DATABASES 43

IV TECHNOLOGY BACKGROUND 45

10 CROSS-TIER VALIDATION 47
10.1 BEAN VALIDATION 47
10.1.1 JSR 349: BEAN VALIDATION 1.1 47
10.1.1.1 HIBERNATE VALIDATOR 49
10.1.2 JSR 303: BEAN VALIDATION 1.0 50
10.1.2.1 APACHE BVAL 51

11 PRESENTATION TIER VALIDATION 55
11.1 HTML 5 55

12 LOGIC TIER VALIDATION 59
12.1 CROSS-LAYER VALIDATION 59
12.2 PRESENTATION LAYER VALIDATION 59
12.2.1 JSR 314: JAVASERVER™ FACES 59
12.2.1.1 ORACLE MOJARRA JAVASERVER FACES 61
12.2.1.2 APACHE MYFACES CORE 61
12.2.1.3 APACHE MYFACES CORE AND HIBERNATE VALIDATOR 64

12.2.1.4 JSF COMPONENT FRAMEWORKS 66
12.2.2 GOOGLE WEB TOOLKIT 68
12.2.3 JAVA™ FOUNDATION CLASSES: SWING 74
12.2.3.1 JFC SWING: ACTION LISTENER APPROACH . . 74
12.2.3.2 SWING FORM BUILDER 75
12.2.4 THE STANDARD WIDGET TOOLKIT 78
12.2.4.1 JFACE STANDARD VALIDATION 79
12.2.4.2 JFACE BEAN VALIDATION 82
12.2.5 JAVAFX 85
12.2.5.1 FXFORM2 85
12.3 BUSINESS LAYER VALIDATION 87
12.4 DATA ACCESS LAYER VALIDATION 88
12.4.1 JSR 338: JAVA™ PERSISTENCE API 88
12.4.1.1 ECLIPSELINK 91
12.4.1.2 HIBERNATE ORM 93
12.4.1.3 DATANUCLEUS 95
12.4.2 JSR 317: JAVA™ PERSISTENCE API 97
12.4.2.1 APACHE OPENJPA 98
12.4.2.2 BATOO JPA 100
12.4.3 NON-STANDARD JPA PROVIDERS 101
12.4.3.1 HIBERNATE OGM 101
12.4.3.2 VERSANT JPA 105
12.4.3.3 OBJECTDB 106
12.4.3.4 KUNDERA 107

13 DATA TIER VALIDATION 113

14 TECHNOLOGY OVERVIEW 115

V APPROACH 120

15 DATA QUALITY MANAGEMENT FRAMEWORK 121
15.1 BASIS 121
15.2 FEATURES 123
15.3 PERSISTENCE 126
15.4 CONSTRAINTS AND DATA QUALITY DIMENSIONS 128
15.5 DEMO APPLICATION 128

16 ASSOCIATION CONSTRAINT 131
16.1 STATIC ASSOCIATION CONSTRAINT 132
16.1.1 SIMPLE `@Size` METHOD 132
16.1.2 SUBCLASSING METHOD 134
16.2 DYNAMIC ASSOCIATION CONSTRAINT 136
16.2.1 TYPE-LEVEL CONSTRAINT METHODS 137
16.2.1.1 HAND-CRAFTED ASSOCIATION CONSTRAINT METHOD 137

16.2.1.2 GENERIC ASSOCIATION CONSTRAINT METHOD 139
16.2.1.3 INTROSPECTIVE ASSOCIATION CONSTRAINT METHOD . . . 143
16.2.2 ASSOCIATION COLLECTION METHOD . . . 146

17 TEMPORAL CONSTRAINT 151
17.1 DATA STRUCTURE . . . 151
17.1.1 TEMPORAL INTERFACE . . . 151
17.1.2 PRIMITIVE TEMPORAL DATA TYPES . . . 152
17.1.3 TEMPORAL ASSOCIATION COLLECTION . . . 153
17.1.4 DATA STRUCTURE EXTENSION . . . 153
17.2 CONSTRAINTS . . . 154
17.2.1 `@Deadline` CONSTRAINT . . . 154
17.2.1.1 ANNOTATION . . . 154
17.2.1.2 VALIDATOR . . . 155
17.2.2 CONSTRAINT COMPOSITION . . . 156
17.2.2.1 `@AssertFalseOnDeadline` CONSTRAINT . . . 157
17.2.2.2 `@MinOnDeadline` CONSTRAINT . . . 158
17.2.3 TEMPORAL CONSTRAINT CREATION . . . 159

18 TIME-TRIGGERED VALIDATION COMPONENT 161
18.1 TTVC: SCHEDULERS . . . 161
18.2 TTVC: JOBS AND JOBDETAILS . . . 163
18.2.1 TTVC: BASIC JOB . . . 163
18.2.2 TTVC: ABSTRACT VALIDATION JOB . . . 163
18.2.3 TTVC: ABSTRACT JPA VALIDATION JOB . . . 164
18.2.4 TTVC: UNIVERSAL JPA VALIDATION JOB . . . 166
18.3 TTVC: TRIGGERS . . . 166
18.4 TTVC: JOB LISTENER . . . 167
18.5 TTVC: PERSISTENT VALIDATION REPORT . . . 168

19 HARD AND SOFT CONSTRAINTS 173
19.1 DEFINITION . . . 173
19.1.1 HARD CONSTRAINT . . . 173
19.1.2 SOFT CONSTRAINT . . . 175
19.1.3 SUMMARY . . . 176
19.2 IMPLEMENTATION . . . 176
19.2.1 HARD CONSTRAINT IMPLEMENTATION . . . 177
19.2.2 SOFT CONSTRAINT IMPLEMENTATION . . . 177
19.2.2.1 PAYLOAD-TRY-CATCH METHOD . . . 177
19.2.2.2 SOFT CONSTRAINTS VALIDATOR . . . 178
19.2.2.3 GROUP METHOD . . . 178
19.3 APPLICATION . . . 182

20 CONSTRAINTS AND DATA QUALITY DIMENSIONS 187

VI SUMMARY 190

21 CONTRIBUTION 191

22 CONCLUSION 193

23 FUTURE WORK 195
23.1 TECHNICAL EXTENSIONS 195
23.2 CONCEPTUAL EXTENSIONS 196

VII APPENDIX 198

A SOURCE CODE 199
A.1 SRC: CROSS-TIER VALIDATION 199
A.1.1 SRC: BEAN VALIDATION 199
A.1.1.1 SRC: JSR 349: BEAN VALIDATION 1.1 199
A.1.1.2 SRC: JSR 303: BEAN VALIDATION 1.0 200
A.2 SRC: LOGIC-TIER VALIDATION 200
A.2.1 SRC: PRESENTATION LAYER VALIDATION 200
A.2.1.1 SRC: JSR 314: JSF 200
A.2.1.2 SRC: GWT 201
A.2.1.3 SRC: JFC: SWING 202
A.2.1.4 SRC: SWT 203
A.2.1.5 SRC: JAVAFX 204
A.2.2 SRC: DATA ACCESS LAYER VALIDATION 204
A.2.2.1 SRC: JSR 338: JPA 2.1 204
A.2.2.2 SRC: JSR 317: JPA 2.0 206
A.2.2.3 SRC: NON-STANDARD JPA PROVIDERS 207
A.3 SRC: DATA QUALITY MANAGEMENT FRAMEWORK 207
A.4 SRC: ASSOCIATION CONSTRAINT 208
A.4.1 SRC: DYNAMIC ASSOCIATION CONSTRAINT 208
A.4.1.1 SRC: TYPE-LEVEL CONSTRAINT METHODS . . 208
A.4.1.2 SRC: ASSOCIATION COLLECTION METHOD . . 210
A.5 SRC: TEMPORAL CONSTRAINT 210
A.5.1 SRC: DATA STRUCTURE 210
A.5.1.1 SRC: TEMPORAL INTERFACE 210
A.5.1.2 SRC: PRIMITIVE TEMPORAL DATA TYPES . . . 210
A.5.1.3 SRC: TEMPORAL ASSOCIATION COLLECTION . 210
A.5.2 SRC: CONSTRAINTS 211
A.6 SRC: TIME-TRIGGERED VALIDATION COMPONENT 211
A.6.1 SRC: TTVC: SCHEDULERS 211
A.6.2 SRC: TTVC: JOBS AND JOBDETAILS 212
A.6.2.1 SRC: TTVC: BASIC JOB 212
A.6.2.2 SRC: TTVC: ABSTRACT VALIDATION JOB . . . 212
A.6.2.3 SRC: TTVC: ABSTRACT JPA VALIDATION JOB 212
A.6.2.4 SRC: TTVC: UNIVERSAL JPA VALIDATION JOB 212

A.6.3 SRC: TTVC: TRIGGERS 213
A.6.4 SRC: TTVC: JOB LISTENER 213
A.6.5 SRC: TTVC: PERSISTENT VALIDATION REPORT 213
A.7 SRC: HARD AND SOFT CONSTRAINTS 214
A.7.1 SRC: SOFT CONSTRAINT IMPLEMENTATION 214
A.7.1.1 SRC: PAYLOAD-TRY-CATCH METHOD 214
A.7.1.2 SRC: SOFT CONSTRAINTS VALIDATOR 214
A.7.1.3 SRC: GROUP METHOD 214
A.8 SRC: CONSTRAINTS AND DATA QUALITY DIMENSIONS 216

B LIST OF ABBREVIATIONS 217

C LIST OF FIGURES 221

D BIBLIOGRAPHY 225

PART I

INTRODUCTION

1 MOTIVATION

In information systems, constraints are tightly coupled with data validation. But why do we need validation at all? First, it can be used to check theses like it is usually done for business intelligence using a data warehouse. Moreover, we apply data validation because we distrust the user and want to avoid errors. This type of data validation happens a thousand times a day if you, for example, consider the registration process for an information system as depicted in figure 1.1. Lastly, we do data validation because we want to make sure that the quality of data meets a certain threshold.

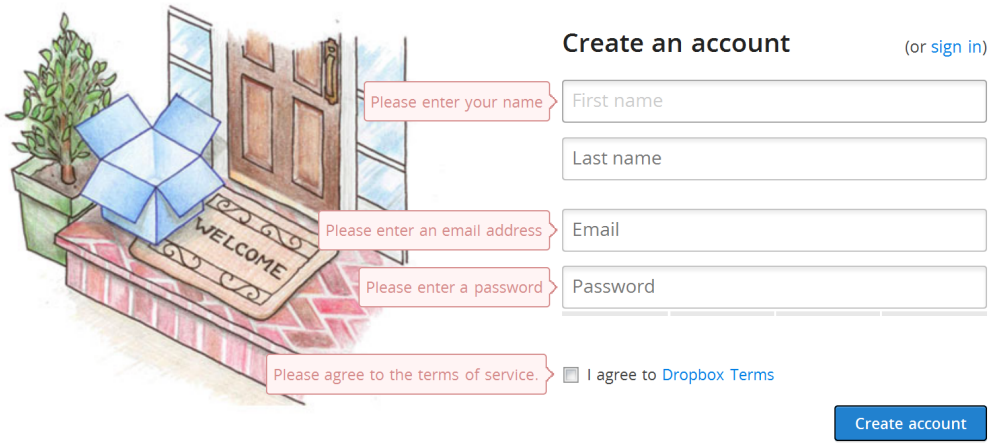


Figure 1.1: Screenshot of the Dropbox[1] registration process showing the violation errors if a user clicks on the 'Create account' button with an empty form.

[1] `https://www.dropbox.com/register`, [Online; accessed 06-October-2013]

Having considered the reasons for data validation one might ask how to implement constraints and a data validation process from a application developer viewpoint. An information system application is usually distributed across several tiers and layers as shown in figure 1.2. A developer has to be aware of several programming language constructs where each is usually applied to a specific layer. This can result in the definition of the same constraint in multiple layers leading to code duplication, inconsistency and redundancy because the same constraint may be checked more than once. Moreover, due to the layer and tier specific technologies the constraints and validation code will be distributed which makes it hard for a developer to have an overview of the defined constraints and there is a higher probability for an increased maintenance effort. These effects are even more increased because the constraints are most often not an independent set which can be reused in another application and hence must be implemented again. Finally, have you ever tried to define what a valid e-mail address is using for instance a regular expression? If so, compare your result with the regular expression[2] generate from the Request for Comments (RFC) 822 specification describing a valid e-mail format – I think you have got it wrong. This shows that defining a constraint can be very hard and that there is still room for supporting an application developer.

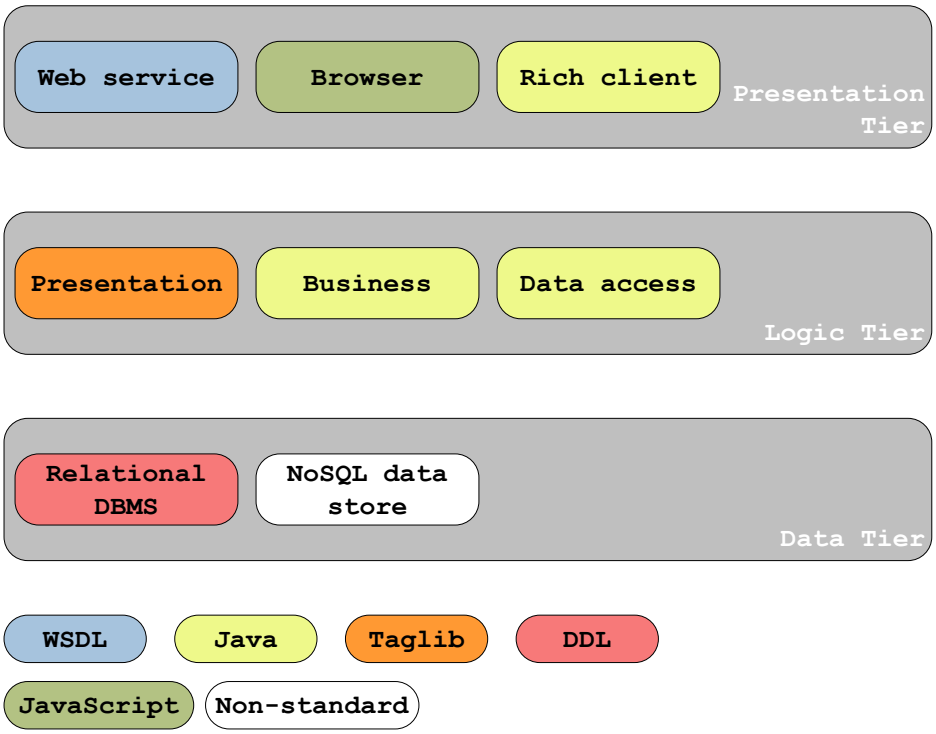


Figure 1.2: Tier and layer overview regarding constraint definition possibilities in a typical Java™ environment[3].

Concluding, within this master thesis we focus on ensuring data quality as the main reason

[2] `http://www.ex-parrot.com/pdw/Mail-RFC822-Address.html`, [Online; accessed 06-October-2013]

[3] Adapted from `http://alt.java-forum-stuttgart.de/jfs/2009/folien/F7.pdf`, [Online; accessed 06-October-2013]

for data validation because we think that if the data is of high quality the other reasons for data validation (e.g. avoidance of errors) are implicitly considered as well. The goal of this master thesis is the development of a data quality management framework which is based on constraints to validate data and ensure the data quality. The data quality management framework should support an application developer in the specification, management and usage of constraints using a single constraint model. The constraint model should not be coupled to a specific tier or layer nor to a specific technology and provide the possibility to validate data only once. Ultimately, the data quality management framework should support a developer in such a way that a constraint must be specified only once but can be used at different places of an application and offers the possibility for re-use in another application.

2 STRUCTURE

This thesis starts with the conceptual background (see part II) which discusses the definition of data quality in the first chapter (see chapter 3) of this part. Agreeing on the fact that data quality is a multidimensional concept, the second chapter within this part describes how to discover data quality dimensions and gives an overview about the most important dimensions with different definitions based on a research analysis. The concept part concludes with a technology independent overview of constraint types with respect to data validation.
The third part 'Research Background' (see part III and the subsequent part 'Technology Background' (see part IV) describe already existing solutions regarding constraint management and data validation. Part III focuses on publications within the research community and the fourth part presents technical solutions which are available in the Java™ environment.
The investigations were made in order to get an overview of already existing concepts and technologies and finally, taking a decision whether an existing concept and/or technology can be extend or an approach from scratch has to be done.
Both parts are organised in the same way: the first four chapters represent the common tiers in a three-tier architecture (presentation, logic and data tier) where the first chapter corresponds to a special tier which is called 'cross-tier'. Within the logic tier chapter, each section corresponds to a layer (presentation, business, data access) of a typical logic tier application running on a server with another special cross-layer section. Every publication and technology is categorised to the tier and/or layer according to the presented information with respect to constraints and data validation. The technology part contains a 'technology overview' chapter comparing the analysed technologies in a short and concise manner. Figure 2.1 visualises the structures of part three and four.
The conceptual and technical contribution to a data quality management framework is described in part V. This part begins with chapter about the conceptual and technical decisions considering the analysis of part III, IV and II. The following four chapters describe the decisions in more detail and moreover, they show alternative implementations to the individual concepts. The 'association constraint' described in chapter 16 shows possible implementations to constraint an association between entities. The third chapter (see chapter 17) within

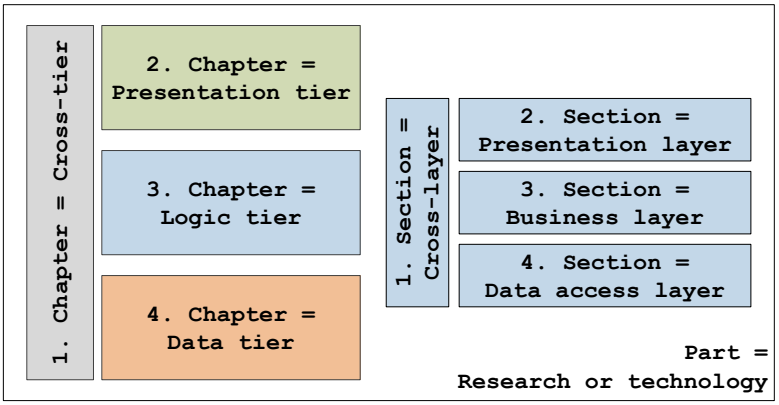


Figure 2.1: Visualisation of the thesis structure for part three III and four IV. Note that the chapter and section numbers are relative and not absolute references.

this part explains how to implement constraints which are coupled with a temporal dimension which means that a constraint does not have to hold immediately but at a certain point in time. This concept is followed by a chapter (see chapter 18) that shows an implementation of a time-triggered validation component which provides for instance the possibility to schedule validation jobs. Lastly, a definition of hard and soft constraints with a implementation suggestion is presented in chapter 19.
This master thesis completes with a part (see part VI) providing possible options for conceptual and technical future work, a conclusion chapter and summary of the contributions followed by the appendix which includes the list of source code examples, the list of abbreviations[1], the list of figures and the bibliography.

[1] In the digital version of this document you can click on almost every Three-letter acronym (TLA) (like this one) and you get the explanation for it. It works for abbreviations consisting of less or more than three letters ☺, too.

PART II

CONCEPT BACKGROUND

Data corresponds to real world objects which can be collected, stored, elaborated, retrieved and exchanged in information systems and that can be used in organisations to provide services to business processes [1]. Furthermore, there are three types of data according to [1]: structured (e.g. relational tables), semi structured (e.g. Extensible Markup Language (XML) files) and unstructured (e.g. natural language). In the following part, we present the results of our literature research regarding the conceptual background of this thesis. As [1] says

'Data quality is a multifaceted concept, as in whose definition different dimensions concur.'

the first chapter of this part summarises several definitions of data quality followed by a detailed study of data quality dimensions. Finally, we give an overview about different types of constraints.

3
DATA QUALITY

'What is data quality?' is the central question of this chapter. There is neither a precise nor a unique answer to this question. Nevertheless, the fact that data with a bad quality causes several problems is a common opinion in the research community (e.g. [2], [3] and [4]). Therefore, we present the results of our literature research to get a better insight about this term. In [1], a classification between the quality of data and the quality of the schema is done. Schema quality can for instance refer whether a given relational schema fulfils certain normal forms according to the theory of the relational model by Edgar F. Codd [5]. Within this master thesis, we solely focus on data quality.
In [2], the authors state that data without any defect is not possible or required in every situation (e.g. a correct postcode is sufficient, the city name does not have to be correct) and therefore a judgement about the data quality is useful. To do a judgement, they propose to tag the data with quality indicators which correspond to 'characteristics of the data and its manufacturing process'. A quality indicator would be for instance information about the collection method of the data. Based on these quality indicators a judgement can be done. Next, they motivate the definition of data quality with a comparison to the definition of product quality:

'Just as it is difficult to manage product quality without understanding the attributes of the product which define its quality, it is also difficult to manage data quality without understanding the characteristics that define data quality.'

After the definition of some data quality dimensions they conclude with the result that data quality is multi-dimensional and hierarchical illustrated by an example and a graphic which is depicted in 3.1. The hierarchy can be read for instance as follows [2]: the entry labelled with 'believable' expresses the dimension that data must be believable to the user so that decisions are possible based on this data. Next, consider the children of the node labelled with 'believable': a user can only say that data is believable if data is complete, timely and accurate (and maybe some other factors) which results in three child nodes. Finally, having

a detailed look at the timely dimension, we get another two child nodes: ‘Timeliness, in turn, can be characterized by currency (when the data item was stored in the database) and volatility (how long the item remains valid)’.

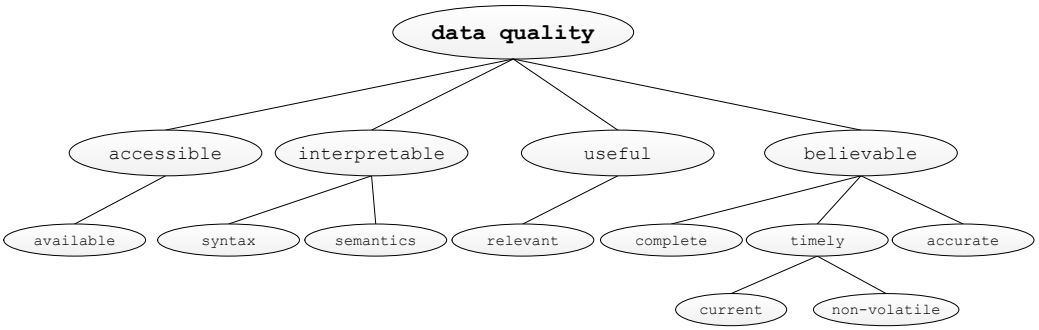


Figure 3.1: An example hierarchy of data quality dimensions adapted from [2].

Also in [3], they agree on the proposition that data quality is a multidimensional concept but they also state that there is no consistent view of the different dimensions. Furthermore, they think that the quality of data is subjective because it could depend on the application where the data is used. For instance, the amount of dollars could be measured in one application in units of thousands of dollars and in another one it is necessary to be more precise. Finally, they claim that data quality does also depend on the information system design which generates data.

Vassiliadis et al. [6] mention ‘the fraction of performance over expectancy’ and ‘the loss imparted to society from the time a product is shipped’ as definitions for data quality, but they think that the best definition for the quality of data is the ‘fitness for use’. This implies that data quality is subjective and varies from user to user as already stated in [2]. In addition, they think that data problems like non-availability and reachability can be measured in an objective way. Lastly, similar to [2], they state that data quality is information system implementation dependent and the measurement of data quality must be done in an objective way so that a user can compare the outcome with his expectations.

In [7], the definition of data quality is connected with the environment of decision-makers. They refer to the ‘classic’ definition of data quality as the ‘fitness for use’ (similar to [6]) or ‘the extent to which a product successfully serves the purposes of customers’. They agree on the claim that the quality of data depends on the purpose as already stated in [3] and [6]. Due to the fact of their research environment their dependency is the decision-task and they believe ‘that the perceived quality of the data is influenced by the decision-task and that the same data may be viewed through two or more different quality lenses depending on the decision-maker and the decision-task it is used for’.

Pipino et al. [8] criticise that the measurement for data quality is usually done ad hoc because elementary principles for usable metrics are not available which the paper tries to solve. They mention that studies have acknowledged that data quality is a multidimensional concept. Furthermore, the authors do also consider that data quality must be treated in two ways: subjective and objective as in [6]. Pipino et al. state that ‘subjective data quality assessments reflect the needs and experiences of stakeholders: the collectors, custodians, and consumers of data products’ and that there must be ‘objective measurements based on the data set in

question'. Objective assessments can be task-independent which means that they are not dependent on the context of an application or they can be task-dependent e.g. to reflect the business rules of a company. Besides, they mention that managers define information as processed data but often it is treated as the same.
Data quality is explained in [4] as a technical issue such as the first part of an Extract, Transform and Load (ETL) process and information quality is on the other hand described as a non technical issue (e.g. stakeholders have the appropriate information), but there is no common opinion about this distinction. Therefore, in [4] data quality is used for both and they mention the 'fitness for use' definition (also described by [6]). They think this definition changed the reasoning in the data quality research area because the quality of data is defined by the viewpoint of a consumer for the first time.
A completely different approach to define data quality was chosen by Liu and Chi [9]. They mention that the research community agrees that data quality is a multidimensional concept but the definition of the dimensions lack of a sound definition. Therefore, they try to specify data quality based on a clear and sound theory. Criticising the analogy between the quality of a product and the quality of data, they propose the data evolution life cycle depicted in figure 3.2. Within 3.2, data is usually passed through several stages. In the beginning, data is collected through observations in the real world (e.g. measurements of an experiment) and then stored in a data store. Afterwards, people use the stored data for analysis, interpretation and presentation which eventually is used within an application that can capture data again.

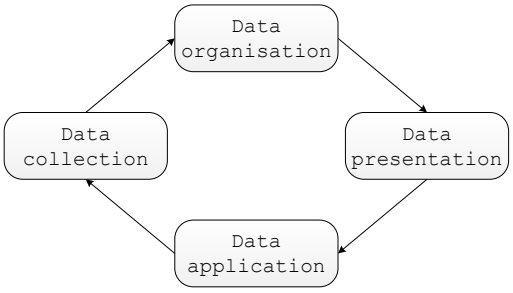


Figure 3.2: The data evolution life cycle as described in [9].

Based on this theory, they state that the definition of data quality depends on the stage within the data evolution life cycle where the measurement should take place. Furthermore, they propose that the data quality of the stages positively correlate and that there is a hierarchy (see figure 3.3) between the four stages of the data evolution life cycle which means for instance that organisation quality is more specific than collection quality. Finally, they explain how to measure the quality of data for each stage of the life cycle specifying the dimensions.
We conclude the research about the definition of data quality with the definition of [10] which mentions the agreement on the 'fitness for use' definition and propose another definition of data quality as 'data that fit for use by data consumers'.

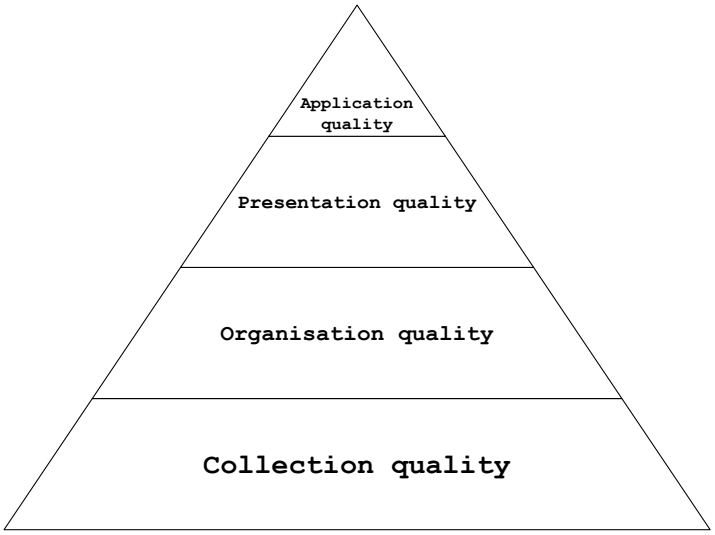


Figure 3.3: The hierarchy as described in [9] puts the stages of the data evolution life cycle 3.2 into a hierarchy regarding the specificity of the data quality contribution. The hierarchy should be read as follows: upper level is more specific than the lower level.

4
DATA QUALITY DIMENSIONS

Data quality dimensions are tightly coupled with data quality and as already mentioned for data quality (see chapter 3) there is no common definition of the data quality dimensions. According to [1], every dimension captures a specific aspect of data quality which means that a set of data quality dimensions are building the quality of data. Data quality dimensions can correspond to the extension of data which means that they refer to data values or they can correspond to the intension of data i.e. to the schema [1]. In this thesis we will only focus on data quality dimensions that refer to data values (extension) because they are more relevant according to [1] in real-life applications. We first describe how to figure out data quality dimensions followed by detailed description of individual data quality dimensions. The presented material is mainly based on the book 'Data Quality' by Batini and Scannapieco [1].

4.1 DATA QUALITY DIMENSION: DISCOVERY

In the following two sections three approaches to discover data quality dimensions are described: a theoretical, an empirical and an intuitive approach.

4.1.1 THEORETICAL APPROACH

Wand and Wang describe in [3] a theoretical approach to figure out some data quality dimensions which is summarised in [1]. They compare a real world system with an information system and the mapping between both systems. According to [1], a real world system is properly represented if the following two conditions hold:

1. Every state of the real world system is mapped to one or more states of the information system.
2. There are no two states of the real world system that map to the same state of the information system.

The analysis of the possible mistakes that can happen during the mapping, which means that the two conditions do not hold, lead to so called deficiencies. The paper uses a graphical representation of the mapping which is depicted in figure 4.1. Every mapping which does not correspond to the left one in the first row in figure 4.1 represents a deficiency. Using the discovered deficiencies, corresponding data quality dimensions are then derived.

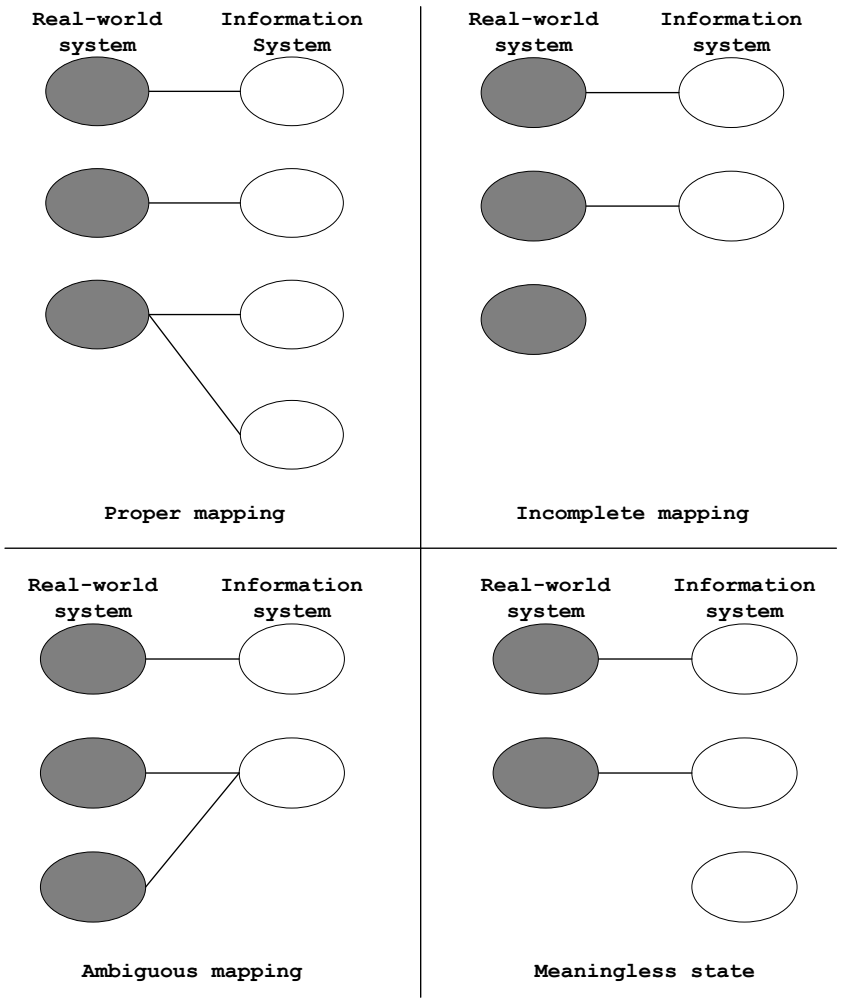


Figure 4.1: Graphical representation of different real world system state mappings to information system states taken from [1]. **Top row:** the left mapping shows a real-life system that is properly mapped to an information system and the right mapping shows an incomplete mapping. **Bottom row:** The left mapping shows some ambiguity and the right one a meaningless state in the information system

4.1.2 EMPIRICAL AND INTUITIVE APPROACH

In [10] a two-stage survey was done to figure out some data quality dimensions. The first survey was used to collect an exhaustive list of possible data quality dimensions. This was

done by asking practitioners and students who are consuming data. Within the second survey a set of alumni were asked to rate each data quality dimension of the first survey with respect to their importance. Afterwards, the authors did another study with the task to sort the data quality dimensions into different groups.
The intuitive approach mentioned in [1] is straightforward. The author of this approach mentions several data quality dimensions and put them into different categories.

4.2 DATA QUALITY DIMENSION: DESCRIPTION

This section describes several data quality dimensions in a qualitative way. As already mentioned, there is no precise and unique definition of every data quality dimension and therefore we present a choice of data quality dimension definitions and/or metrics. In addition, we do not focus on measurement methods which is also discussed in [1]. As mentioned in [1] some data quality dimensions are easier to detect than others. For instance, misspellings are often easier to tackle than an admissible but not correct value. Furthermore, some data quality dimensions are independent of the underlying data model while others are, for example, tightly coupled with the relational data model. Moreover, there is a trade-off in realising individual dimensions because they cannot be reached independently. For example, consider a huge amount of data with a lot of inconsistencies or less data but the consistency is high shows a trade-off between the completeness and consistency dimension. We use a running example which is also presented in [1] to explain certain dimensions in a more illustrative way. Figure 4.1 shows the running example which is a relation containing information about films.

Id	Title	Director	Year	#Remakes	Last Remake Year
1	Casablanca	Weir	1942	3	1940
2	Dead Poets Society	Curtiz	1989	0	$NULL$
3	Rman Holiday	Wylder	1953	0	$NULL$
4	Sabrina	$NULL$	1964	0	1985

Table 4.1: A relation containing information about films with several data quality issues with respect to different data quality dimensions. The example is adapted from [1]

We present the data quality dimensions accuracy, completeness, consistency and a group of time-related dimensions (timeliness, currency, volatility) in more detail because they are considered important in [1] and they belong to the seven most-cited data quality dimensions (except volatility which is not listed at all) as presented in [3]. The detailed description is followed by an exhaustive list of other data quality dimensions discovered during our literature research.

4.2.1 ACCURACY

Batini and Scannapieco define accuracy in [1] as follows:

Definition 1 (*Accuracy (Batini and Scannapieco)*)

> *Accuracy is defined as the closeness between a value v and a value v', considered as the correct representation of the real-life phenomenon that v aims to represent.*

In other words, v' is the true value and v is a given value which is compared to v'. For instance, considering the real-life phenomenon of a human kind with a first name 'John' we have $v' = John$. Furthermore, the authors categorise accuracy into syntactic and semantic accuracy which are defined as:

Definition 2 (*Syntactic accuracy (Batini and Scannapieco)*)

> *Syntactic accuracy is the closeness of a value v to the elements of the corresponding definition domain D.*

Definition 3 (*Semantic accuracy (Batini and Scannapieco)*)

> *Semantic accuracy is the closeness of the value v to the true value v'.*

Those definitions are best explained using our running example depicted in figure 4.1. Consider the tuple where the value of the attribute Id corresponds to 3. Every attribute has an associate set of applicable values which is called its domain. The domain of the attribute $Title$ is the set of all existing film titles. Since $Rman\ Holiday$ (light blue cell) is not in the set of possible film titles (it is a spelling error) it belongs to the category of a syntactic accuracy problem. Now, consider the two light red cells of the tuples where the attribute value of the attribute Id is set to 1 and 2 respectively. The attribute values $Weir$ and $Curtiz$ of the attribute $Director$ are assigned to the wrong films, because Weir is the actual director of the film called 'Dead Poets Society' and Curtiz is the director of 'Casablanca' resulting in a semantic accuracy problem. According to Batini and Scannapieco, the concept of semantic accuracy is in accordance with the correctness concept. Moreover, the accuracy of an attribute (attribute accuracy), of a relation (relation accuracy) or the whole database (database accuracy) can be defined next to the single value accuracy of a relation attribute as discussed above.

OTHER DEFINITIONS

In [2] accuracy is defined as 'the recorded value is in conformity with the actual value' and the authors of [3] mention that there is no exact definition, but they propose a definition according to their model as 'inaccuracy implies that information system represents a real world state different from the one that should have been represented. Therefore, inaccuracy can be interpreted as a result of garbled mapping into a wrong state of the information system.' This definition is illustrated in figure 4.2.

4.2.2 COMPLETENESS

In general, completeness can be defined as 'the extent to which data are of sufficient breadth, depth, and scope for the task at hand' according to [1]. Moreover, the authors focus on completeness based on relational data. Within this kind of model, they compare whether the relation matches with the real world. Therefore, they explain the usage and meaning of $NULL$ values with respect to the data quality dimension completeness. Considering a model where $NULL$ values are possible, Batini and Scannapieco state that a $NULL$ expresses the fact that a value exists in the real world but it is not present in the relation. Three cases must be analysed to match a $NULL$ value with a problem of the data quality dimension completeness in a correct way:

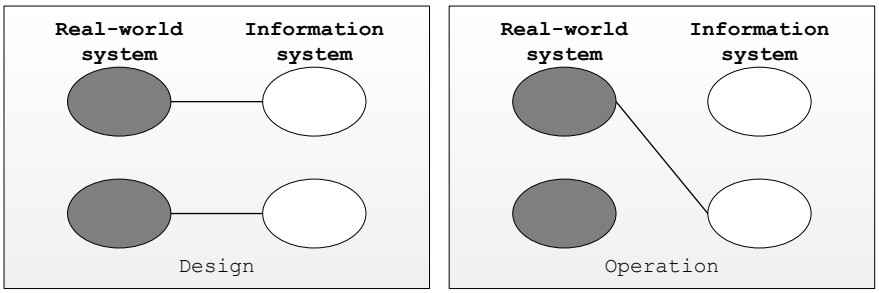


Figure 4.2: Although there is a proper design, at operation time the user could map a real-life state to the wrong information state (this is called garbling). The user could be able to infer a corresponding real-life state based on the information system state but the inference is not correct. This theory is connected with the data quality dimension accuracy as presented in [3]

1. $NULL$ means that no value in a real world exists (e.g. a person does not have an email address)
2. $NULL$ means that a value exists in the real world but it is not present in the relation (e.g. a person has an email address but the information system did not register it)
3. $NULL$ means that it is unknown whether a value exists or not (e.g. it is not known whether a person does have an email address or not)

According to the authors, only for the second case a completeness issue arises but not for the others. Consider our running example, we have a completeness problem because the value for the attribute called $Director$ (light yellow cell) is missing and we know that a film usually has a director.

OTHER DEFINITIONS

In [2] completeness is defined as 'all values for a certain variable are recorded' and the authors of [3] state that the literature defines the definition as a set of data with all necessary values but they propose a definition that is not related to data at all. As depicted in figure 4.1, the definition is based on the underlying theory model which says that 'completeness is the ability of an information system to represent every meaningful state of the represented real world system'[3]. Finally, [7] mentions the completeness definition of a data element 'as the extent to which the value is present for that specific data element'.

4.2.3 CONSISTENCY

Consistency is in [1] defined as 'the violation of semantic rules defined over (a set of) data items'. Data items can refer to tuples of a relational table and integrity constraints are an example of semantic rules with respect to the relational model. Due to the fact, that the authors use the concept of consistency simultaneously with the concept of semantic rules, we describe different types of semantic rules in chapter 5. Considering our running example

depicted in 4.1, a consistency problem arises for the tuple where the attribute Id has the value 1. Comparing the values of the attributes $Year$ and $Last\ Remake\ Year$ (grey cells) leads to a confusion because naturally the inequality $Last\ Remake\ Year \geq Year$ must hold. Moreover, the attribute values for the attributes $\#Remakes$ and $Last\ Remake\ Year$ (green cells) of the tuple where the attribute Id has the value 4 is not consistent. Either the number of remakes must be at least 1 because the last remake year is known or the attribute value for $Last\ Remake\ Year$ should be equal to $NULL$.

OTHER DEFINITIONS

Wang et. al mention in [2] 'the representation of the data value is the same in all cases' as a definition for consistency. In [3], the authors describe that consistency is multi-dimensional. It can refer to the values of data, the representation of data and to the physical representation of data. Based on their theory model, they can only consider consistency with respect to the values of data. Although, they mention consistency as a data quality dimension, their model does not consider inconsistencies as a deficiency, because inconsistency would disallow a one to many mapping which is not forbidden (see figure 4.1). Pipino et. al refer in [8] to the following definition of a consistent representation: 'the extent to which data is compactly represented'. But similar to [1] they also refer to integrity constraints (especially the referential integrity constraint) as a type of consistency. Finally, the authors in [9] define consistency as 'different data in a database are logically compatible'.

4.2.4 TEMPORAL DATA QUALITY DIMENSIONS

4.2.4.1 TIMELINESS

According to [1], timeliness belongs to the group of time-related dimensions and is defined as 'how current data is for the task at hand'. The importance for this dimension is justified by the authors by the possible scenario that data can be useless if they are late. The given example is taken from a university environment. A timeliness problem exists if the course catalogue does contain the most recent information but it is only accessible for the students after the start of a term.

OTHER DEFINITIONS

The authors of [2] refer to the definition 'the recorded value is not out of date' for timeliness. Moreover, they propose their own definition based on the observation that data quality is a hierarchical concept. Therefore, they state that timeliness can be defined by currency (see section 4.2.4.2) and volatility (see section 4.2.4.3). In [3], timeliness is analysed with respect to the theory model and therefore defined as 'the delay between a change of the real world state and the resulting modification of the information system state'. The authors also refer to other literature definitions such as 'whether the data is out of date' or the 'availability of output on time'. The definition 'how up-to-date the data is with respect to the task it's used for' described in [8] combines the definitions of [1], [2] and [3] and is also used in [9].

4.2.4.2 CURRENCY

This data quality dimensions does belong to the group of time-related dimensions as described in [1]. Currency is defined as 'how promptly data is updated' and within our running example the authors describe the problem that a remake of the film with the attribute value 4 of the attribute Id has been done but the relation does not consider this information because the number of remakes is equal to 0 (green cell of attribute $\#Remakes$). On the other hand, data is current if an information system stores the actual address of a person.

OTHER DEFINITIONS

In [2], the authors simply define currency as the time 'when the data item was stored in the database'. Similarly, Wand and Wang mention that this dimension can be interpreted 'as the time a data item was stored' but they also mention a definition of system currency with respect to their theory of a mapping between the real world system and information system. Within this model, they define system currency as 'how fast the information system state is updated after the real world system changes'.

4.2.4.3 VOLATILITY

This data quality dimension is the last member of the time-related dimension group according to [1]. The authors define volatility as 'the frequency with which data vary in time'. This definition becomes more clear with the following examples mentioned by Batini and Scannapieco: stable data (such as birth dates) have a volatility near or equal to zero and on the other side, data which changes a lot (like stock quotes) have a high volatility.

OTHER DEFINITIONS

Wang et al. define this dimension in [2] as 'how long the item remains valid' and in [3] the definition 'the rate of change of the real world system' is based on the underlying theory model described in section 4.1.1. Identical to Wang, in [8] is volatility defined as 'the length of time data remains valid'.

4.2.5 OTHER DATA QUALITY DIMENSIONS

The following list of dimensions and definitions are based on [1], [3] and [8].

- **Interpretability** is defined as 'the documentation and meta data that are available to correctly interpret the meaning and properties of data sources'
- 'The proper integration of data having different time stamps' belongs to the dimension **synchronization** (between different time series)
- **Accessibility** is a measurement about 'the ability of the user to access the data from his or her own culture, physical status/functions and technologies available'
- A group of three dimensions measure how 'trustable' an information source is

  - 'A certain source provides data that can be regarded as true, real and credible' is the definition for **believability**
  - **Reputation** concerns about 'how trustable is the information source'
  - The 'impartiality of sources in data provisioning' is the definition of **objectivity**

- The **appropriate amount of data** is defined as 'the extent to which the volume of data is appropriate for the task at hand'
- 'Whether the data can be counted on to convey the right information' or 'correctness of data' are definitions for **reliability**
- **Concise representation** is 'the extend to which data is compactly represented'
- The definition 'the extent to which data is easy to manipulate and apply to different tasks' belongs to the dimension **ease of manipulation**
- **Free-of-error** means 'the extent to which data is correct and reliable'
- The dimension **relevancy** can be defined as 'the extend to which data is applicable and helpful for the task at hand'
- **Security** is defined as 'the extent to which access to data is restricted appropriately to maintain its security'
- 'The extent to which data is easily comprehended' is the definition of **understandability**
- **Value-added** refers to 'the extent to which data is beneficial and provides advantages from its use'

An overview with more data quality dimensions (some are just mentioned without a definition) and the number of citations per dimension can be found in [3].

5 CONSTRAINT TYPES

This chapter describes different types of constraints in an technology independent way which means the considered constraints have a general applicability to data in an information system no matter what kind of technology is used. The presented material (including the citations) is taken from [11], unless otherwise stated.

5.1 DEFINITION

In general a constraint can be defined as

'one of a set of explicit or understood regulations or principles governing conduct or procedure within a particular area of activity. . . a law or principle that operates within a particular sphere of knowledge, describing, or prescribing what is possible or allowable.'

This definition is originally used in [11] for the term rule, but we can use it for constraint as well and will use both terms from now on interchangeably within the context of this master thesis. We can divide rules into two categories: definitional rules and operative rules. Definitional rules 'define various constructs created by the organization (or the industry within which it operates)' and operative rules are defined as 'what must or must no happen ion particular circumstances'. The following two examples point out the difference between the two definitions:

- **Definitional rule example:** 'An infant passenger is by definition a passenger whose age is less than 2 years at the time of travel.'
- **Operative rule example:** 'Each flight booking request for a return journey must specify the return date.'

We can create an operative rule which corresponds to a definitional rule ('mirroring') to avoid for example that a user enters invalid data. For example, the definitional rule 'pH is by

definition at least 0 and at most 14.' can be mirrored into the operative rule:'The pH specified in each water sample record must be at least 0 and at most 14.'.
In the following description we will only focus on operative rules because within this master thesis we are mainly interested in input data ('entered data') and how to decrease errors while entering data. Therefore, we can directly specify operative rules without definitional rules and the transformation step. The need for definitional rules is discussed in detail in [11].

5.2 TYPES

As mentioned in 5.1 we focus on operative rules which are divided into three categories in the rule taxonomy of [11]:

- Definitional rules
- **Operative rules**
  - **Data rules**
  - **Activity rules**
  - Party rules

Definitional rules and operative rules are defined in section 5.1. A data rule is a constraint on data which is included in a transaction or a (persistent) data store. Activity rules are defined as constraints on the operation of several business processes or activities and party rules are restrictions on the access to processes or activities (roles). A guide to categorise a given rule is described in [11]. As this master thesis is about data quality we will mainly present the different rule types for the category 'data rules' and one representative of the category 'activity rules'. An detailed description can be found in [11].

5.2.1 DATA RULES

The term 'data rules' (as defined in 5.2) can be synonymously used for 'integrity constraints', 'semantic integrity constraints' or 'system rules'. Semantic integrity constraints are referred to the specification of conditions on database records that must be fulfilled [12] to represent the real world in a correct way. Integrity constraints and system rules concern about the integrity of data (as against to business rules which consider the decisions of people) according to [13]. Although, (semantic) integrity constraints are mentioned in the same breath with databases the concept is not restricted to database records.
We now present the taxonomy for the subcategory 'data rules' by naming the individual types, their definition and a selected example as presented in [11].

1. **Data cardinality rules** require 'the presence or absence of a data item and/or places a restriction on the maximum or minimum number of occurrences of a data item'
   (a) *Mandatory data rules* mandate 'the presence of data'
      i. Mandatory data item rules require 'that a particular data item be present'
      Ex.: 'Each flight booking confirmation must specify exactly one travel class for each flight.'

ii. Mandatory option selection rules require 'that one of pre-defined options be specified'
Ex.: 'Each flight booking request must specify whether it is for a return journey, a one-way journey, or a multi-stop journey.'

iii. Mandatory group rules require 'that at least one of a group of data items be present'
Ex.: 'Each flight booking confirmation must specify a mobile phone number, an e-mail address, or both.'

(b) *Prohibited data rules* mandate 'the absence of some data item in a particular situation'
Ex.: 'A flight booking request for a one-way journey must not specify a return date.'

(c) *Maximum cardinality rules* place 'an upper limit (...) on how many instances of a particular data item there may be'
Ex.: 'A combination of departure date, flight number, and departure city must not be allocated more than one passenger for any one seat number.'

(d) *Multiple data rules* mandate 'the presence of two or more instances of a particular data item in a particular situation'
Ex.: 'Each flight booking confirmation for a return journey must specify at least two flights.'

(e) *Dependent cardinality rules* mandate 'how many of a particular data item must be present based on the value of another data item'
Ex.: 'The number of passenger names specified in each flight booking confirmation must be equal to the number of passengers specified in the flight booking request that gives rise to that flight booking confirmation.'

2. **Data content rules** place 'a restriction on the values contained in a data item or set of data items (rather than whether they must be present and how many there may or must be)'

(a) *Value set rules* require either 'that the content of a data item be (or not be) one of a particular set of values (fixed or not)' or 'that the content of a combination of data items match or not match a corresponding combination in a set of records'
Ex.: The travel class specified in each flight booking request must be 'first class', 'business class', (...) or 'economy class'

(b) *Range rules* require 'that the content of a data item be a value within a particular inclusive or exclusive single-bounded or double-bounded range'
Ex.: 'The number of passengers specified in each flight booking request must be at least 1 and at most 9.'

(c) *Equality rules* require 'that the content of a data item be the same as or not the same as that of some other data item'
Ex.: 'The destination city specified in each flight booking request must be different from the origin city specified in that flight booking request.'

(d) *Uniqueness constraints* require 'that the content of a data item (or combination or set of data items) be different from that of the corresponding data item(s) in the

same or other records or transactions'
Ex.: 'The record locator allocated to each flight booking confirmation must be different from the record locator allocated to any other flight booking confirmation.'

(e) *Data consistency rules* require 'the content of multiple data items to be consistent with each other, other than as provided for by a value set rule, range rule, or equality rule'
Ex.: 'The sum of the shares held by the proprietors of each real property parcel must be equal to 1.'

(f) *Temporal data constraints* constrain 'one or more temporal data items (data items that represent time points or time periods'

i. Simple temporal data constraints require 'that a particular date or time fall within a certain temporal range'
Ex.: 'The return date (if any) specified in each flight booking request must be no earlier than the departure date specified in that flight booking request.'

ii. Temporal data non-overlap constraints require 'that the time periods specified in a set of records (...) do not overlap each other'
Ex.: 'The time period specified in each employee leave record must not overlap the time period specified in any other employee leave record for the same employee.'

iii. Temporal data completeness constraints require 'that the time periods specified in a set of records be contiguous and between them completely span some other time period'
Ex.: 'Each day within the employment period specified in each employee record must be within the time period specified in any other employee leave record for the same employee.'

iv. Temporal data inclusion constraint require 'that the time periods specified in a set of records do not fall outside some other time period'
Ex.: 'Each day within the time period specified in each employee leave record must be within the time period specified in the employment record for the same employee.'

v. Temporal single record constraints require that data with a contiguous time period must not be split into chunks leaving the data except for the time period identical
Ex.: 'Each grade specified in an employee grade record must be different from the grade specified in the latest of the earlier employee grade records for the same employee.'

vi. Day type constraints require 'to restrict a date to a working day'
Ex.: 'The payment due date specified in each invoice must be a working day.'

(g) *Spatial data constraints* prescribe or prohibit 'relationships between data items representing spatial properties (points, line segments or polygons)'
Ex.: 'The polygon that constitutes each individual parcel in a real estate subdivision must not overlap the polygon that constitutes any other individual parcel in any real estate subdivision.'

(h) *Data item format rules* specify 'the required format of a data item'
Ex.: 'The mobile phone number (if any) specified in each flight booking confirmation must be a valid phone number.'

3. **Data update rules** either prohibit 'updates of a data item' or place 'restrictions on the new value of a data item in terms of its existing value'

   (a) *Data update prohibition rules* prohibit 'prohibit updates of a particular data item or set of data items'
   Ex.: 'A data item in a financial transaction must no be updated.'

   (b) *State transition constraints* limit 'the changes in a data item to a set of valid transitions'
   Ex.: 'The martial status of an employee may be updated to never married only if the marital status that is currently recorded for that employee is unknown.'

   (c) *Monotonic transition constraints* require 'that a numeric value either only increase or only decrease'
   Ex.: 'The hourly pay rate of an employee must not be decreased.'

The rules mentioned in the subcategory 'data cardinality' and 'data content' are static data constraints because they are 'concerned only with the presence or absence of a value or what that value is'. The 'data update rules' belong to the group of dynamic data constraints since they are 'concerned with allowed relationships between old and new values of a data item'.

5.2.2 ACTIVITY RULES

We present activity time limit rule as a member of the activity restriction rule group as we provide an implementation for those kind of rules as described in chapter 17. In our approach, we call those kind of rules 'temporal constraints' which should not be mixed up with the definitions given in section 5.2.1.

1. **Activity restriction rules** restrict 'a business process or other activity in some way.'

   (a) *Activity time limit rules* restrict 'a business process or other activity to within a particular time period'
   Ex.: 'Online check-in for a flight may occur only during the 24 h before the departure time of that flight.' or 'Acknowledgement of an order must occur during the 24 h after the receipt of that order.'

PART III

RESEARCH BACKGROUND

This part consists of relevant publications with respect to constraint specification and data validation that we found during our broad research. Every paper is assigned to a tier or layer of a typical three-tier architecture according to the presented material. The first chapter presents the publications that belong to the presentation tier, followed by a chapter about the logic tier. Within the logic tier, which is usually represented by application server, one can distinguish between several layers. Therefore, this chapter consists of four sections handling the presentation, business, data access and the cross-layer. Finally, the data tier is explained in a separate chapter. Every chapter starts with an explanation about the corresponding tier or layer. Although, the chapter and section headings do only mention the term 'validation' we always consider the specification of constraints as well because we believe that validation can only happen if constraints exist.

6 CROSS-TIER VALIDATION

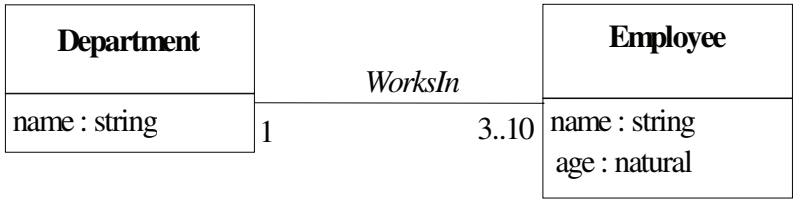
Cross-tier validation is a concept where data validation is not restricted to one tier but it provides the possibility to validate data at different tiers using the same concept. Publications describing such a concept are mentioned in this chapter.

6.1 CONSTRAINT SUPPORT IN MDA TOOLS: A SURVEY

In [14], the authors present a survey about existing tools to transform integrity constraints which are defined in a Platform-Independent Model (PIM) into running code that reflects the specified constraints. One of the reasons for the survey is that this semantic transformation is regarded as an open problem which must be solved to be able to use Model-Driven Development (MDD) approaches in general for building information systems. Cabot and Teniente define three criteria for the choice and evaluation of the considered tools:

- What kind of constraints can be defined? How expressive is the constraint definition language? They distinguish three levels:
  - Intra-object integrity constraints: one object, several attribute values
  - Inter-object integrity constraints: several objects and their relationship
  - Class-level integrity constraints: one class, several objects of the same class/type
- How efficient is the translated code ensuring the integrity constraints?
- What are the target technologies for the transformation process? They study two technologies in more detail:
  - Relational databases
  - Object-oriented languages

The last criteria is the reason, why we put this paper into the category of cross-tier validation. The authors use the running example (a PIM) depicted in figure 6.1 to show that even this simple model cannot be translated to code using one of the analysed tools. The tool survey is divide into four categories: Computer-Aided Software Engineering (CASE) tools, tools which follow a Model-Driven Architecture (MDA) approach, MDD methods and tools which can generate code from constraints defined in Object Constraint Language (OCL). Finally, they propose five desirable features that should be supported by a tool which translates PIM to code: expressivity, efficiency, technology-aware generation, technological independence and checking time.



context Employee **inv** ValidAge: self.age>16

context Department **inv** SeniorEmployees:
self.employee->select(el e.age>45)->size()>=3

context Employee **inv** UniqueName:
Employee.allInstances()->isUnique(name)

Figure 6.1: Running example used in [14] to analyse the transformation process of the individual tools.

6.2 INTERCEPTOR BASED CONSTRAINT VIOLATION DETECTION

Wang and Mathur explain in [15] a system which analyses the request/response messages between a client and a server to detect possible constraint violations based on the interfaces of the given domain model (i.e. 'interface level constraints'). They state four different kinds of interface categories which they consider in their work: constraints for return values, argument values and (class) attribute values ('value region'), constraints with respect to the response time of a request/response ('time region'), cross-attribute constraints, cross-argument constraints and constraints on the relationship between attributes and arguments ('spatial value relationship') and 'temporal value relationship' which is for example a constraint on the method invocation order. The main component is a monitor which is generated from an XML file that contains the constraints. Then, as depicted in figure 6.2, the monitor is embedded in the 'Interceptor Manager' which can analyse request/response messages between the client and a server (the message passing is 'intercepted'). Unmarshalling, constraint inspection, data validation and a modification of the message happens within this manager before the message is forwarded to the actual target.

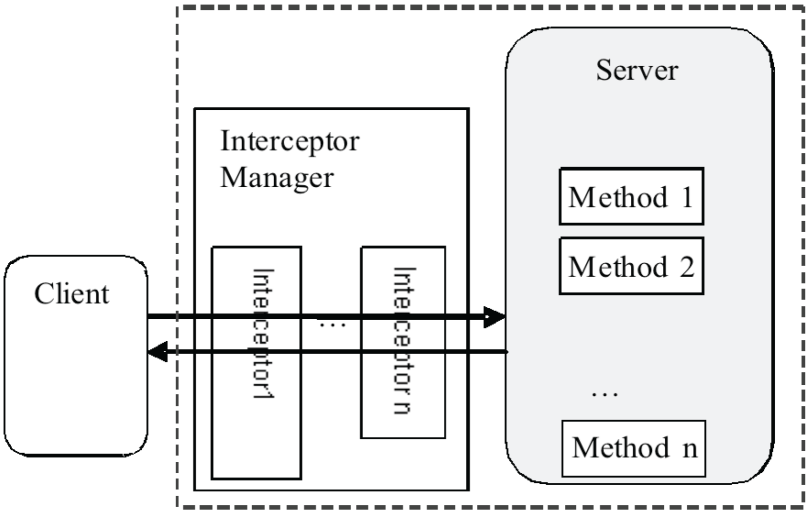


Figure 6.2: System structure of the interceptor based approach for constraint violation as described in cite1409949. The 'Interceptor Manager' acts between the server and the client analysing the request/response messages.

The authors mention four advantages of their approach:

- The monitoring code which handles the data validation and constraint inspection is independent of the functional code
- Easy XML based constraint specification which can be automatically translated to monitoring code
- The functional code remains unaffected
- Cleaner monitoring and functional code plus easier maintenance

We categorise this concept as a cross-tier validation method because the interceptor based approach could theoretically be applied to the request/response messages between the presentation tier and the logic tier or the logic tier and the database tier, i.e. not coupled to a specific tier which means cross-tier.

6.3 TOPES: REUSABLE ABSTRACTIONS FOR VALIDATING DATA

Scaffidi et al. propose in [16] a sophisticated concept to validate input data using the idea of a 'tope' which is an abstraction of a data category that can be used for data validation. Within this paper an abstraction is a pattern which reflects the valid values of the considered data category. For instance, the data category mentioned in the paper is e-mail addresses and a corresponding tope (i.e. abstraction/pattern) could be that 'a valid e-mail address is a user name, followed by an @ symbol and a host name'. This abstraction is transformed into executable code and a similarity function checks the actual value against the defined constraint (i.e. tope). The result can be either valid, invalid or questionable depending on

the outcome of the similarity function (the image of the similarity function is between zero and one, where one means valid and zero invalid). The authors state the possibility to re-use this concept among different applications without any modification and therefore suggest a transformation scheme of the data into a general format. Mentioning that the concept is not coupled to a specific application type gives us the reason to classify this concept under this chapter.

7 PRESENTATION TIER VALIDATION

Papers which propose validation methods that are used at the presentation tier are explained in this chapter. Presentation tier validation is a method were validation is done at the client-side (e.g. a thin client such as a browser) and no data is sent to the server for validation purposes.

7.1 POWERFORMS

PowerForms [17] is a declarative and domain specific language for client-side form field validation for Hypertext Markup Language (HTML). Constraints can be declared for the HTML element `input` (`text`, `password`, `radio` or `checkbox`) and `select` using a regular expression. The regular expressions are described for strings using XML and get then translated to a deterministic finite-state automaton using JavaScript and HTML which checks if the input data is valid (i.e. accept state of the finite-state automaton stands for valid data). An example declaration and the usage is depicted in listing 7.1 taken from [17].

Listing 7.1: Declaration of a regular expression using PowerForms and its usage for an 'input' element as presented in [17].

```
<regexp id="isbn">
  <regexp exp="([0-9]([ -]?)){9}[0-9X]"/>
</regexp>

Enter ISBN number: <input type="text" name="isbn" size=20 />
<format name="isbn" help="Enter an ISBN number"
    error="Illegal ISBN format">

  <regexp idref="isbn" />
</format>
```

Validation happens while editing data and on the submit of a form. While editing a form field, the data is checked against the specified regular expression and small traffic lights show the validation status in three phases. 'Green' stands for valid data, 'Yellow' for data that is a valid

prefix and ‘Red’ for invalid data. For instance, consider the left part of figure 7.1. The text box entry in this example must be ‘Hello World’. The first traffic lights are yellow because ‘Hello’ is a valid prefix of the required string ‘Hello World’. The lower traffic lights is red because the entered data (‘Hello Orld!’) is not a valid prefix. On submit, validation violations are displayed within a JavaScript alert box. Finally, PowerForms can express interdependencies which means that the validation of form fields can be made dependent on already entered data. For instance, the following dependency ‘if a person attended to a conference, an assessment could be made’ is implemented in PowerForms as depicted in the right part of figure 7.1. Technically, the second question can only be answered if ‘yes’ is chosen for the first one.

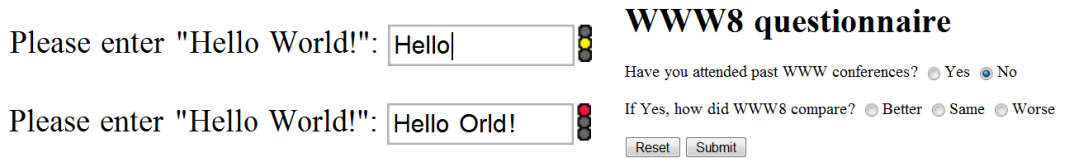


Figure 7.1: **Left:** Rendering example of a validation violation with the default icons (traffic lights). **Right:** A form with an interdependency. It is not possible to answer the second question if ‘no’ is chosen. These examples can be tested on the project website of PowerForms[1].

[1] `http://www.brics.dk/bigwig/powerforms`, [Online; accessed 19-August-2013]

8 LOGIC TIER VALIDATION

We define the logic tier (which is sometimes also called the application tier) as a bundling of three layers: the presentation layer, business layer and data access layer. The logic tier is the connection piece between the presentation tier and the data tier. The data access layer is responsible for managing the Create, Read, Update, Delete (CRUD) operations (e.g. persist an entity into a data store) that have to be made to the persistence tier. Data coming from the data tier is transformed by the presentation layer (e.g. generates HTML code) which makes the data ready to be displayed by the presentation tier (e.g. a browser). Finally the business layer provides the glue code between the presentation and data access layer where individual data processing happens. Furthermore, the term cross-layer is used where one cannot specify a single layer but several. In this chapter, we present publications which present material that belong to one of the layers within this tier.

8.1 CROSS-LAYER VALIDATION

Papers which propose validation methods that are not coupled to a specific layer are described in the following section.

8.1.1 INTEGRATION OF DATA VALIDATION AND USER INTERFACE CONCERNS IN A DSL FOR WEB APPLICATIONS

Groenewegen and Visser present in [18] a sub-language of WebDSL to handle data validation rules. WebDSL is a domain-specific programming language to create web applications which gets then translate to a running Java™ web application relying on a customised Java™ framework. The validation component allows a developer to specify rules in a declarative way using four different types of data validation rules:

- **Value well-formedness:** Checks if the input value can be converted to the appropriated underlying type. This happens automatically using the declared type of the underlying

model (e.g. `Email`). Rules and error messages of built-in types can be customised.

- **Data invariants:** Using `validate(e: Boolean, s: String)` one can define a constraint for a domain model specifying an expression `e` that returns a `Boolean` and an error message `s`. The rules are checked whenever an entity consisting of data invariants is saved, updated or deleted.
- **Input assertions:** Some data validation rules are not coupled with the underlying domain model. Therefore, input assertions can be directly specified in the form declaration (e.g. double password check)
- **Action assertions:** The `validate(e: Boolean, s: String)` function taking an expression returning a Boolean and an error message can be used at arbitrary execution points. If the validate function returns false, the processing is stopped and the error message is displayed.

An example of some data validation rules is given in listing 8.1. It shows an entity definition with a value well-formedness constraint given by the declared types such as `Email`, data invariants for the `password` property and in the page definition an input assertion for the password field and an action assertion for the `save` event.

Listing 8.1: A WebDSL example using the validation sub-language. It shows the four different data validation rules (value well-formedness, data invariants, input assertions, action assertions) within a small user management application. The code fragments are taken from the WebDSL project website[1].

```
entity User {
  username :: String (id)
  password :: Secret (
    validate(password.length() >= 8,
      "Password needs to be at least 8 characters"),
    validate(/[a-z]/.find(password),
      "Password must contain a lower-case character"),
    validate(/[A-Z]/.find(password),
      "Password must contain an upper-case character"),
    validate(/[0-9]/.find(password),
      "Password must contain a digit"))
  email :: Email
}

define page editUser(u:User) {
  var p: Secret;
  form {
    group("User") {
      label("Username") { input(u.username) }
      label("Email") { input(u.email) }
      label("New Password") {
        input(u.password)
      }
      label("Re-enter Password") {
        input(p) {
          validate(u.password == p, "Password does not match")
        }
```

```
      }
      action("Save", save() )
    }
  }
  action save() {
    validate(email(newGroupNotify(ug)),
      "Owner could not be notified by email");
    return userGroup(ug);
  }
}
```

Error messages are either shown directly at the input field if the input is not well-formed or causes a violation of a data invariant or at the form element that triggered the execution process (e.g. a submit button) if a data invariant was violated during execution. The validation process is embedded into the 'WebDSL request processing lifecycle' consisting of five phases which handles the different data validation rules and the appropriate rendering of the response page. The declarative validation rules are transformed to a normalized WebDSL code which gets then translated to Java™.
We put this technology in the section of cross-layer validation because it uses the constraints for input validation at the presentation layer and in addition the validation mechanism checks the validity of the entities before saving, updating or deleting them into a persistence unit at the data access layer. Note that the technology focuses on web application and therefore cannot be used to create desktop applications.

8.2 PRESENTATION LAYER VALIDATION

Presentation layer validation refers to the concept were the server produces the front-end that is shipped to the client-side which includes components to initiate the validation process. This section gives an overview about published concepts regarding validation within the presentation layer.

8.2.1 MODEL-DRIVEN WEB FORM VALIDATION WITH UML AND OCL

Escott et al. focus in [19] on the creation of constraints from Unified Modeling Language (UML) and OCL for web forms. Therefore, they propose three categories that are needed for web form validation: the 'single element' validation category is used for a single web form field like a text field, the 'multiple element' category is similar to the concept of interdependencies mentioned in section 7.1 and 'entity association' is a category which handles the constraints of the underlying domain model and its relationships.
Furthermore, the goal is to translate the specified constraints in the model into code for a specific web application framework. Therefore, they analyse four different web application frameworks (Spring Model View Controller (MVC)[2], Ruby on Rails®[3], Grails[4], ASP.NET MVC[5]) and explain the transformation from the model to code for the Spring MVC frame-

[1] http://webdsl.org/selectpage/Manual/Validation, [Online; accessed 07-October-2013]
[2] http://www.springsource.org/, [Online; accessed 19-April-2013]
[3] http://rubyonrails.org/, [Online; accessed 19-April-2013]
[4] http://grails.org/, [Online; accessed 19-April-2013]
[5] http://msdn.microsoft.com/de-de/asp.net, [Online; accessed 19-April-2013]

work using Java™ while arguing that the transformation could be done with any of those web application frameworks. The transformation is made for each validation category separately. For the single elements, a UML diagram has to be enriched with a stereotype (see figure 8.1) which gets then translated to Java™ code (see listing 8.2) using JSR 303 (Bean Validation 1.0) constraints (see section 10.1.2) and Java Emitter Template (JET)[6].

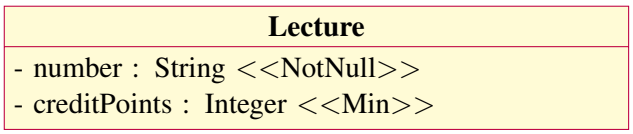


Figure 8.1: UML class diagram (adapted from [19]) with stereotypes that are translated to JSR 303 (Bean Validation 1.0) constraints.

Listing 8.2: A Java™ class (adapted from [19]) produced by the transformation step from the UML class diagram 8.1 using JET.

```
public class Lecture {
  @NotNull
  private String number;

  @Min(1)
  private int creditPoints;

  // Getter and setter methods

}
```

Constraints for multiple elements are expressed using OCL (see listing 8.3) which are then translated using Eclipse OCL[7] and JET. The generated code corresponds to a validator class which implements the `Validator` interface in case of the web application framework Spring MVC. According to the authors, the transformation for entity associations is application specific but their basic idea is that a validator class checks for the multiplicities of an association.

Listing 8.3: An OCL constraint expressing that either the email address or the phone number (or both) must be present (taken from [19]).

```
  email.size() > 0 or phone.size() > 0
```

8.3 BUSINESS LAYER VALIDATION

The business layer is responsible for every data processing that has to be done between the presentation layer and the data access layer. This section describes the papers which provide a way to validate data within the business layer, i.e. business layer validation.

[6] `http://www.eclipse.org/modeling/m2t/?project=jet#jet`, [Online; accessed 19-August-2013]

[7] `http://www.eclipse.org/modeling/mdt/?project=ocl`, [Online; accessed 19-August-2013]

8.3.1 OVERVIEW AND EVALUTION OF CONSTRAINT VALIDATION APPROACHES IN JAVA

In [20], the authors do a survey about different validation techniques within the Java™ programming language environment coupled with a benchmark test. The studied techniques range from hand-crafted constraints using if-then-else statements and exception mechanism over code instrumentation and compiler-based approaches to explicit constraint classes and interceptor mechanisms. Hand-crafted if-then-else statements are usually used in the business layer in combination with the functional code as depicted for example in listing 8.2. In-place validation code and wrapper-based constraint validation are techniques which belong to the category of code instrumentation. The first approach injects validation code at the place where the validation should happen (e.g. at the beginning of a method) while the latter one introduces a wrapper method which includes the validation code and calls the original method inside the wrapping as depicted in listing 8.3. Java™ Modeling Language (JML) enriches a Java™ class with preconditions, postconditions and invariants using a special syntax within a comment block. This is recognised by the compiler and translated into executable byte code, i.e. a compiler-based approach for data validation. Finally, explicit constraint classes separates the constraint definition from functional code and is most often implemented using a validate method which takes an object as an argument and depending on the validation outcome returns `True` or `False`. An example for an interceptor mechanism is given in section 8.3.2.

Listing 1. Simple constraint implementation

```
class A {
  void someMethod() {
    if ( ( (...) OR (...) ) AND (...) ) {
      //business logic code
    } else if (! ...) {
      throw SomeException();
    } else {
      if (...) {
        //exception handling code
        printErrorMessage(...);
      } else {
        //business logic continued
      }
    }
    //further business logic
  }
}
```

Figure 8.2: Coding example that shows how to implement a constraint validation mechanism using if-then-else statements and exceptions as described in [20].

8.3.2 LIMES: AN ASPECT-ORIENTED CONSTRAINT CHECKING LANGUAGE

Limes [21] is a programming language which allows the developer to specify constraints for a model using Aspect-Oriented Programming (AOP). Within this kind of programming language the central concept is the definition of an aspect which is according to [21] a 'modular unit that explicitly captures and encapsulates a cross-cutting concern'. The goal of Limes is the platform independent declaration of constraints for a domain model and the specification

Listing 3. Wrapper-based validation

```
public int countChar (char c) {
  //--> code for validation of invariant
  //    constraints and preconditions
  //--> Call the original method
  int result = countChar_wrapped(c);
  //--> code for validation of invariant
  //    constraints and postconditions
  return result;
}

public int countChar_wrapped(char c) {
  int result = 0;
  char[] chars = toCharArray();
  for (int i=0; i<length(); i++) {
    if (chars[i] == c) result++;
  }
  return result;
}
```

Figure 8.3: Original method and a wrapper method which includes the validation logic and a call to the original method (taken from [20]).

of the execution point (constraint checking time) and the manner of constraint checking.

ASPECT-ORIENTED PROGRAMMING (AOP)

As in [21], we describe the core concepts of AOP because we think not every reader is familiar with those concepts. An aspect is a module which contains one or more advices. An advice has a type, an argument list, a pointcut expression and an advice block. An advice is never called directly but the advice type and the pointcut expression specify the execution point of the advice block which contains the logic of an advice. A pointcut expression is a predicate which can match a joinpoint. A joinpoint can be almost any execution point of a program (e.g. a method call). The advice type specifies when the advice block should be executed with respect to the pointcut expression. For example, if a pointcut expression selects the method `foo` of a program then the advice type `before` specifies that the advice block is executed before the call to the `foo` method happens. Alternative advice types are `after`, `before after` and `around` which means instead of the code of the matched join point. Finally, the aspects and the 'normal' program are combined which is done by a 'weaver' and depicted in 8.4.

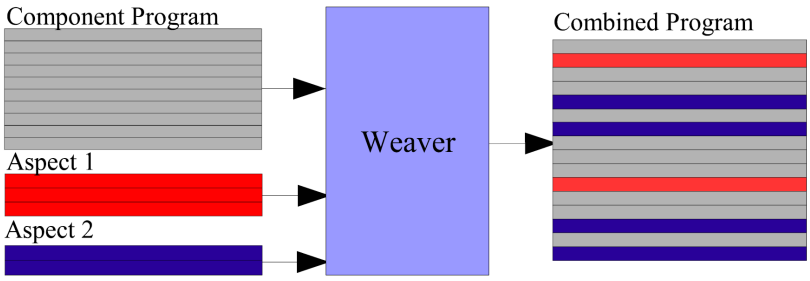


Figure 8.4: Visualising the weaving process with a regular program and two aspects (taken from [21]).

An example of an aspect is given in listing 8.4. The aspect is called `PersonConstraints`

and the pointcut is called `myPointcut`. It specifies a pointcut expression which selects the execution of the method `divorce` of the class `Person`. The advice definition has the advice type `before` and references the pointcut denoted as `myPointcut`. The advice body is a precondition which checks if the person is married (which is necessary for a divorce) adding some textual information in case of a failure. Concluding, the considered joinpoint in this example is the method call `divorce` of the class `Person` which is filtered by the pointcut `myPointcut`. The advice body is executed before a method call to the joinpoint specified by the pointcut expression happens, i.e. in our case before the call of the `divorce` method happens.

Listing 8.4: An example adapted from [21] showing the basic AOP constructs.

```
aspect PersonContraints [Person] {

  pointcut myPointcut(const Person self)
    [[ this(self) && execution(void Person.divorce()) ]];

  before(const Person self) [[ myPointcut(self) ]] : {
    precondition(self.spouse != null, "divorce()");
  }

}
```

In the remaining part of the paper, the authors explain language specific constructs to define constraints in detail. The show how to formulate a precondition, postcondition and an invariant using Limes. Lastly, the explain the requirements for a translation into a general purpose language like Java™.

8.3.3 VALIDATION APPROACHES USING OCL

The publications mentioned in this chapter have all one thing in common: the analyse or propose validation approaches using OCL.
In [22], the authors compare different approaches for constraint validation using OCL followed by a case study including performance tests. The considered approaches range from hand-crafted if-then-else statements and Java™ assertions, over Design by Contract®based approaches as described in [23] and [24] using JML constructs, to validation concepts using AOP as described for instance in 8.3.2 or AspectJ.
Verheecke and Van Der Straeten describe in [25] an approach which translates OCL constraints on UML diagrams into code using object-oriented mechanisms to represent constraints (i.e. classes) and the validation trigger points in the source code is automatically recognised by a path analysis of declarative constraint expressions.

8.4 DATA ACCESS LAYER VALIDATION

The communication between the persistence tier which represents a data store and the logic tier is done in the data access layer. This layer takes data from the business layer and initiates CRUD operations to the persistence tier. Data from the persistence tier is first processed by the data access layer and then shipped to the business layer for further processing. This

section considers publications which present material about data validation within the data access layer.

8.4.1 CONSTRAINT-BASED DATA QUALITY MANAGEMENT FRAMEWORK FOR OBJECT DATABASES

In [26], framework is presented which uses constraints that are defined in a configuration file using a special constraint definition language. Afterwards, the constraints are parsed and translated to executable Java™ classes which are stored in a separated constraint repository. The validation process follows the Event Condition Action (ECA) where an application can register for certain events e.g. at a commit event of the database. An event listener checks the actions of the object-oriented database and activates the corresponding validator instance which has access to the database to validate the relevant objects. Furthermore, it is possible to define hard and soft constraints where soft constraints are defined by the way they are validated, i.e. if a hard constraint is violated a exception is triggered and a warning is thrown if a soft constraint is violated. The approach is evaluated using a sample scenario with students and lectures based on the object-oriented database Database for Objects (db4o)[8].

[8] `http://www.db4o.com/`, [Online; accessed 02-September-2013]

9 DATA TIER VALIDATION

The data tier comprises all concepts which provide a way that data is stored in a non-volatile way such as relational databases or Not only SQL (NoSQL) data stores. It processes requests from the logic tier and responses to the logic tier with some data. This section presents different papers which propose an approach to validate data in the data tier. In general, the book written by Graham Witt [11] gives concise overview about the history of constraints/rules in the third chapter (e.g. database constraints, XML schemas).

9.1 OBJECT-ORIENTED DATABASES

Within the area of object-oriented databases several approaches exist for the data validation and constraint specification process. In [27], the authors specify constraints in a global way (inter-object constraints) and transform them to constraints which can be applied to the individual classes (intra-object constraints). Oakasha et al. propose in [28] a constraints catalogue which is used to check the validation of objects at well-defined points (e.g. when an update occurs). Lastly, Medeiros and Pfeffer explain in [29] the creation of first order logic rules which are similar to Structured Query Language (SQL) and based on rule objects (i.e. rules are first-class citizens). The activation of the rules follows the ECA paradigm.

9.2 RELATIONAL DATABASES

Relational Database Management System (DBMS) usual provide the constraints that are specified in the SQL standard. For instance, MySQL (a DBMS) provides (among others) a unique constraint, a non-null constraint, a foreign key and primary key constraint. A detailed discussion of the SQL:1999 standard is available in [12]. Moreover, Demuth et al. describe in [30] and [31] the translation of constraints expressed in OCL to SQL using built-in integrity constraints, SQL queries and SQL views. In case of SQL views, different evaluation approaches are possible. Either the data access layer checks if the executed SQL view returns

a result or an SQL trigger is defined which handles the validation directly in the DBMS.

PART IV

TECHNOLOGY BACKGROUND

In this part, we present the results of our research regarding the existing technologies to define and manage constraints for the data validation process in informations systems. The part is organised as a usual three-tier architecture. The first chapter presents the results that belong to the presentation tier, followed by a chapter about the logic tier. Within the logic tier, which is usually represented by application server, one can distinguish between several layers. Therefore, this chapter consists of four sections handling the presentation, business, data access and the cross-layer. Finally, the data tier is explained in a separate chapter. Every chapter starts with an explanation about the corresponding tier or layer. An overview about this kind of architecture is depicted in figure 9.1 and shows the most important standards which are considered in this part next to their corresponding tier or layer.
The analysis includes programming language features, libraries, frameworks and language specifications with a strong focus on the Java™ programming language because its popularity is really high[1]. The technology background part is rounded off with an overview about the individual technology features and their benefits and drawbacks. Although, the chapter and section headings do only mention the term 'validation' we always consider the specification of constraints as well because we believe that validation can only happen if constraints exist.

[1] `http://www.tiobe.com/index.php/content/paperinfo/tpci/index.html`, [Online; accessed 08-October-2013]

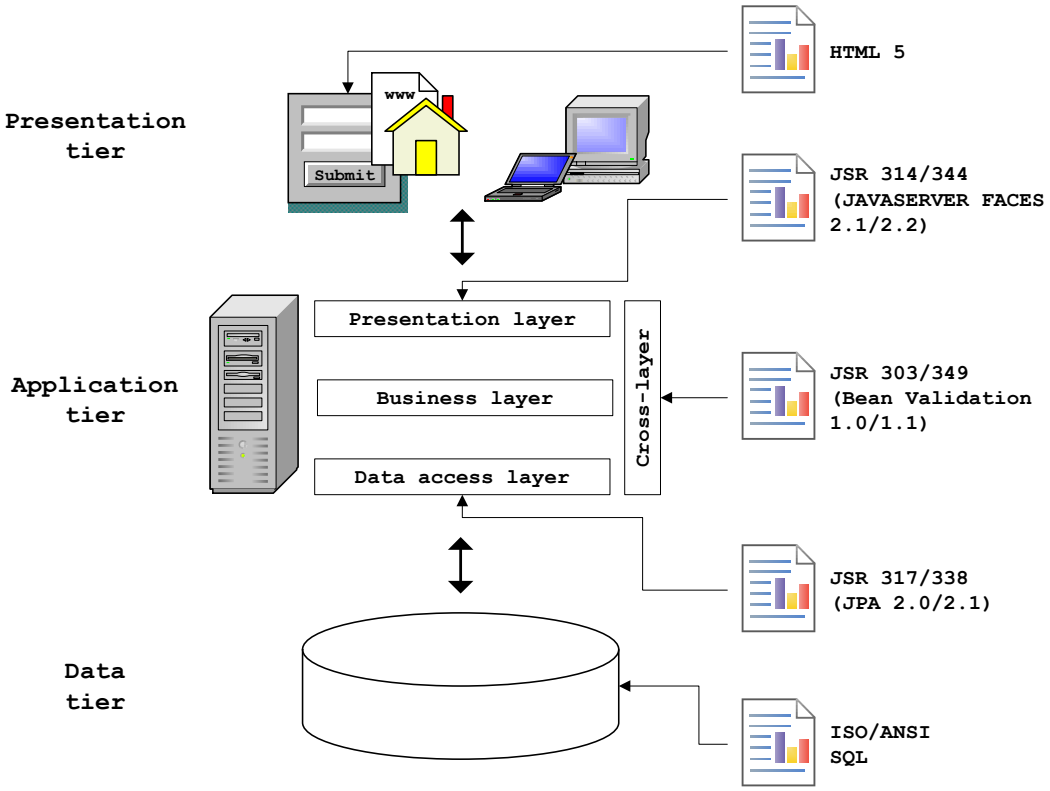


Figure 9.1: Overview of the discussed tiers and layers with respect to possible validation technologies and the most important standards/specifications which are connected to a specific layer or tier.

10 CROSS-TIER VALIDATION

A technology realising cross-tier validation is not coupled to a specific tier such as presentation or data tier but can handle several tiers simultaneously. Technologies supporting constraint definition and data validation for several tiers are presented in this chapter.

10.1 BEAN VALIDATION

The Java™ Community Process (JCP) published two JSR regarding validating data which belongs to the category cross-tier validation as they describe the specification goal in chapter 1.2 of the JSR 349 (Bean Validation 1.1) [32] as follows:

'The validation API developed by this JSR is not intended for use in any one tier or programming model. It is specifically not tied to either the web tier or the persistence tier, and is available for both server-side application programming, as well as rich client Swing application developers.'

Following the well-known metaphor 'a picture is worth a thousand words' the top part in figure 10.1 describes a well known problem regarding data validation and the lower part the solution proposed by the JSR 349 (Bean Validation 1.1).

10.1.1 JSR 349: BEAN VALIDATION 1.1

The JSR 349 (Bean Validation 1.1) [32] has the goal to centralise the definition of the validation logic. Therefore, it introduces a meta data model and an Application Programming Interface (API) to validate JavaBeans™ [33]. The idea is to enrich JavaBeans™ with annotations to express constraints within the domain model. The meta data model is not restricted to a specific layer (such as presentation, logic or persistence) and can be used with other tech-

[1] See footnote 3

nologies. Furthermore, the JSR 349 introduces constraints for method arguments and return values, but does not want to provide a full 'Programming by Contract' framework.
A summary of the most important features followed by a detailed description is given below.

FEATURE OVERVIEW

- **Constraint definition:** Java™ annotation or XML file
- **Constraint target:** Field, method, constructor, argument (method, constructor), type (class), annotation type
- **Constraint composition:** Possible if constraint definition target includes 'annotation type'; constraints are combined with logical 'AND'
- **Built-in constraint definitions:** 13 constraint definitions are described in the standard which every reference implementation must provide
- **Custom constraint:** New constraint definitions are allowed
- **Validation methods:**
  - Whole object: `validateProperty(object)`
  - Field or getter of an object:
    `validateProperty(object, field/getter name)`
  - Field or getter of a class set to a specific value:
    `validateValue(class, field/getter name, value)`
  - Static fields and static methods cannot be used
- **Validation violations:** Exception model with violation messages (customisable)
- **Integration:** Java™ Platform, Enterprise Edition (Java EE), Context and Dependency Injection (CDI), Java™ Persistence API (JPA) 2, JavaServer™ Faces (JSF) 2, Java™ API for RESTful Web Services (JAX-RS)
- **Additional specifications:** Graph validation, group validation, method validation, meta data API
- **JSR status:** Active (Final Release started on 24/05/2013)[2]

DETAILED DESCRIPTION

The JSR 349 (Bean Validation 1.1) includes a detailed description of every feature which can be found on the website of the JCP [32]. An important specification detail is that the methods `validateProperty` and `validateValue` do not consider the `@Valid` constraint annotation which has the semantic to validate the annotated object references recursively (i.e. object graph validation).

[2]Checked on 05/06/2013

10.1.1.1 HIBERNATE VALIDATOR

Hibernate Validator 5[3] is a project of the Hibernate project collection. It is the reference implementation for the JSR 349 (Bean Validation 1.1) [32]. It has passed all the tests of the Technology Compatibility Kit (TCK). Therefore, all the features of the JSR 349 are supported by Hibernate Validator 5. Next, the framework introduces some specific features which are not mentioned in the specification.
A summary of the additional features, a detailed description of the Hibernate Validator and a small example are given below.

ADDITIONAL FEATURE OVERVIEW

Every feature of the JSR 349 (Bean Validation 1.1), see section 10.1.1, is supported by Hibernate Validator. In addition, the following features are implemented in Hibernate Validator:

- **Constraint definition:** Programmatic (fluent API)
- **Constraint composition:** Logical 'OR' and 'ALL_FALSE'
- **Built-in constraints:** 10 additional constraints e.g. `@Email` constraint which checks for a valid email address
- **Validation methods:** Fail fast mode
- **Validation violations:** Additional Service Provider Interface (SPI) to retrieve violation messages
- **Documentation:** Detailed documentation, examples, forum, detailed Java™ API available
- **Licence:** Apache Licence 2.0
- **Latest release:** Version 5.0.1 Final (30/04/2013)[4]

DETAILED DESCRIPTION

The Programmatic constraint definition extension of Hibernate Validator 5 allows the client to add constraints at runtime with the help of the fluent API. Also, the composition of constraints can be connected with a logical 'OR' in addition to the logical 'AND' which is the default operator. The 'ALL_FALSE' operator implements the logical 'NOT' in this framework. For convenience, the project team has added some built-in constraints (e.g. email address validator) which are included in the Hibernate Validator 5 distribution. Next, the fail fast mode offers the possibility to cancel the validation process when the first constraint violation occurs which is not enabled by default. All the other basic features are described in

[3] `http://www.hibernate.org/subprojects/validator.html`, [Online; accessed 23-April-2013]
[4] Checked on 13/05/2013

detail in the JSR 349 (Bean Validation 1.1), see section 10.1.1, or on the Hibernate Validator 5 project website[5].

EXAMPLE

This small example 10.1 shows how to get the default validator and validate a simple person object. The whole example can be found in the appendix section A.1.1.1.

Listing 10.1: JUnit test suite which shows how to instantiate a validator (see `@BeforeClass`) and a test case which validates a simple person object using the `validate` method.

```
public class SimpleBeanValidationTestCases {

  private static Validator validator;

  @BeforeClass
  public static void setUpBeforeClass() throws Exception {
    ValidatorFactory factory =
      Validation.buildDefaultValidatorFactory();

    validator = factory.getValidator();
  }

  @Test
  public void testPersonNameNull() {
    SimplePerson p = new SimplePerson();
    p.setName(null);
    Set<ConstraintViolation<SimplePerson>> constraintViolations =
      validator.validate(p);

    assertEquals("There is a Bean Validation violation.", 1,
      constraintViolations.size());

    assertEquals("The name must not be null.", "name is mandatory",
      constraintViolations.iterator().next().getMessage());
  }

}
```

10.1.2 JSR 303: BEAN VALIDATION 1.0

The JSR 303 (Bean Validation 1.0) [34] is the predecessor of the JSR 349 (Bean Validation 1.1), see section 10.1.1, which implies that not all features of the JSR 349 are also specified in the JSR 303. The main differences of the JSR 303 compared to the JSR 349 are the missing method validation and the group conversion. However, the basic features for data validation were already defined in the JSR 303. A detailed comparison can be found in the changelog of the JSR 349.
A feature overview and a detailed description are given below.

[5] See footnote 3

FEATURE OVERVIEW

The main features of the JSR 303 (Bean Validation 1.0) are the same as listed in section 10.1.2 with some exceptions mentioned in [32] chapter 2 and appendix D.

DETAILED DESCRIPTION

The JSR 303 (Bean Validation 1.0) includes a detailed description of every feature which can be found on the website of the JCP [34].

10.1.2.1 APACHE BVAL

Apache BVal 0.5[6] is a project of the Apache Software Foundation which implements the JSR 303 (Bean Validation 1.0) (see section 10.1.2). This implies that not all features of the JSR 349 (Bean Validation 1.1) are implemented in Apache BVal. However, the basic features for data validation are also implemented in BVal 0.5 and therefore this implementation can be used for all applications using the general data validation mechanisms. According to the Apache BVal project page[7], the activity status of March 2013, says that the project road map includes the implementation of the JSR 349.
A summary of the additional features (compared to the Bean Validation 1.0 and 1.1) followed by a detailed description of the Apache BVal are given below.

ADDITIONAL FEATURE OVERVIEW

Every feature of the JSR 303 (Bean Validation 1.0), see section 10.1.2, is supported by Apache BVal. In addition, the following features are implemented in Apache BVal:

- **Built-in constraints:** 18 (additional library required)
- **Documentation:** No detailed documentation, only a simple example
- **Licence:** Apache Licence 2.0
- **Latest release:** Version 0.5 (21/09/2012)[8]

DETAILED DESCRIPTION

The Apache BVal project provides an additional library for some useful constraints (e.g. IBAN validator) which has to be included when using the additional annotations. All the other basic features are described in detail in the JSR 303 (Bean Validation 1.0) [34]. In addition, the description of the JSR 349 (Bean Validation 1.1), see section 10.1.1, could be used with an additional check, if the feature of the JSR 349 (Bean Validation 1.1) was already included in the Bean Validation 1.0 specification.

[6] `http://bval.apache.org/index.html`, [Online; accessed 25-April-2013]
[7] Checked on 13/05/2013
[8] Checked on 13/05/2013

EXAMPLE

The same domain model and test cases as described in section 10.1.1.1 can be used for Apache BVal 0.5. The library dependencies must be adapted when using this Bean Validation implementation (see appendix section A.1.1.2).

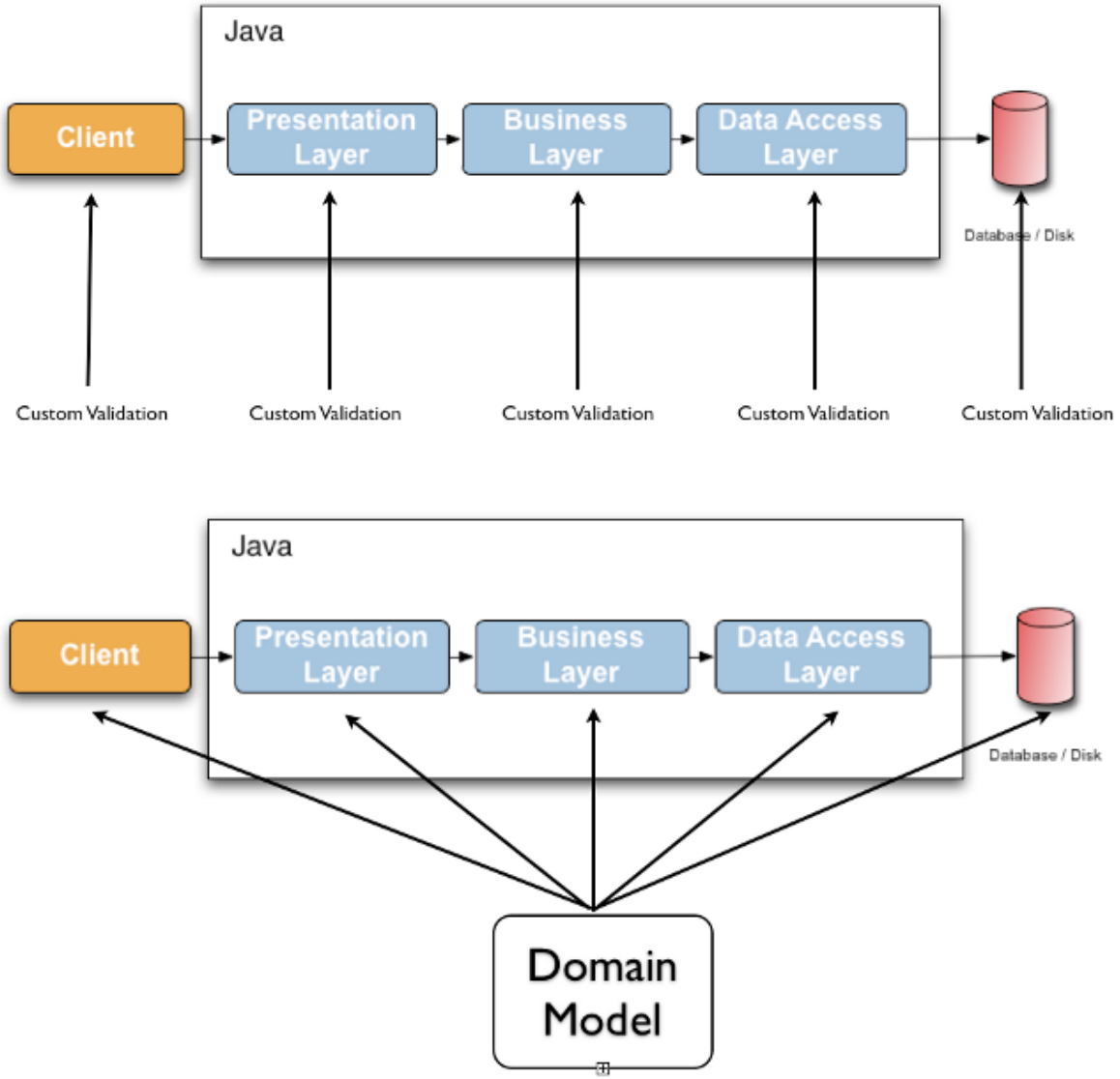


Figure 10.1: **Top part:** Validation happens at different tiers and layers resulting in individual validation mechanisms for each tier and layer. **Lower part:** JSR 349 (Bean Validation 1.1) centralises the validation mechanism using constraint annotations within the domain model. Both pictures are taken from the reference implementation (Hibernate Validator) of the JSR 349 (Bean Validation 1.1) project website[1].

11 PRESENTATION TIER VALIDATION

Within this master thesis, the presentation tier is regarded as the place where a client can interact with the information system. This description includes thin clients (such as a browser) and fat clients (such as a desktop application) which can communicate with the logic tier which usually consists of a server. In this chapter, we present validation mechanisms for the presentation tier which are often used for user input data validation.

11.1 HTML 5

The World Wide Web Consortium (W3C) [35] defines several standards for the web such as the HTML standard which defines features to build web pages. Currently[1], they are working on the fifth version of this standard which is called HTML5 [36] and the status of the specification is 'Candidate Recommendation'. The specification includes the possibility for client-side form validation providing special elements. The following list gives an overview about the features followed by an example.

FEATURE OVERVIEW

- **Constraint definition:** Special elements and attributes
- **Constraint target:** Form fields
- **Built-in constraint definitions:**
  - `input` elements of type `email` and `url`
  - `required` attribute for `input`, `select` or `textarea` elements
  - `pattern` and `maxlength` attribute for `input` element of type `text`, `search`, `url`, `tel` (telephone), `email` and `password`

[1] Checked on 01/08/2013

  - `min`, `max` and `step` attribute for `input` element of type `date`, `time`, `month`, `week`, `number` and `range`

- **Custom constraint:** Possible with the constraint validation API but limited
- **Validation methods:** Whole form or nothing (validation on submit can be turned on/off)
- **Validation violations:** Errors are displayed next to the input field and messages can be customised
- **Additional specifications:** Constraint validation API, validation on form submit and while editing is possible
- **W3C status:** Candidate Recommendation (06/08/2013)[2]

Most browsers support the form validation feature of HTML5 already.[3]

EXAMPLE

The example shows an HTML5 form with the validation feature 'required' and the Cascading Style Sheets (CSS) feature 'invalid' to render inputs that are not valid (while typing) with a red border (see listing 11.1). A rendering using Opera™ 12.16.
is shown in figure 11.1.

Listing 11.1: HTML5 code using the form field validation features combined with CSS.

```
<!doctype html>
  <head>
    <meta charset="utf-8">
    <title>HTML5 form validation example</title>
    <style type="text/css">
      <!-- Basic CSS here -->
      :invalid {
        border: 2px solid #ff0000;
      }
    </style>
  </head>
  <body>
    <h2>HTML5 Form Validation Example</h2>
      <form>
        <label for="name">Please enter your name:</label>
        <input id="name" type="text" required>
        <input type="submit" value="Login">
      </form>
  </body>
</html>
```

[2] Checked on 18/08/2013
[3] `http://caniuse.com/#feat=form-validation`

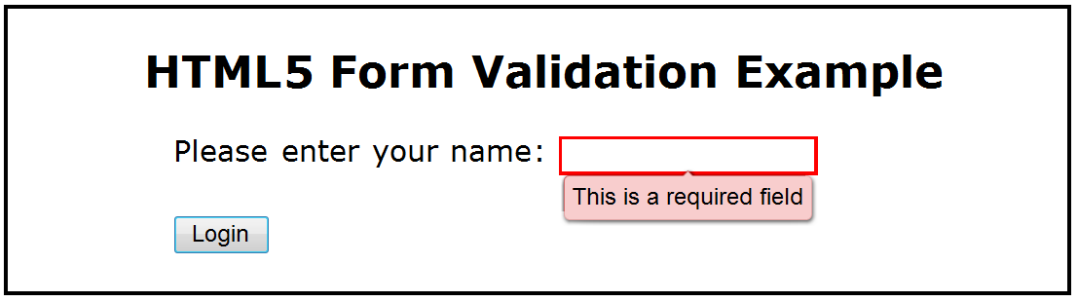


Figure 11.1: HTML5 rendering example using Opera™ 12.16 and the form validation example of listing 11.1.

12 LOGIC TIER VALIDATION

In this chapter, we present validation mechanisms for the logic tier. A definition for the term 'logic tier' is given in the beginning of chapter 8.

12.1 CROSS-LAYER VALIDATION

Cross-layer validation means that a technology is not coupled to a specific layer (such as presentation or data access) and can therefore be used in several layers. No technology which is explicitly designed to supported this kind of validation has been found. Of course, Bean Validation which is described in section 10.1 does support this kind of validation because it is a cross-tier validation concept which supports cross-layer as well.

12.2 PRESENTATION LAYER VALIDATION

The presentation layer is responsible for the generation of the front-end (e.g. a graphical user interface) which is afterwards visible to the user. For example, in a web application the presentation layer produces the HTML code which is then rendered by a browser at the presentation tier (see chapter 11). The validation process for this layer is done on the server-side (or partially on the client-side). This section gives an overview how data validation can be integrated in the presentation layer using different technologies and concepts.

12.2.1 JSR 314: JAVASERVER™ FACES

The JSR 314 (JSF 2.1) [37] defines a standard to develop a (graphical) User Interface (UI) for Java™ web applications. Every JSF 2.1 web application relies on the Java™ Servlet technology [38]. Easy creation of non-standard UI components and the reuse of those components are main objectives of this JSR. In addition, the standard defines the 'Request Processing Lifecycle' with several phases to process the user input and render the response. One of those phases is the 'Process Validation' phase where data validation can happen.

A summary of the validation features defined in the JSR 314 followed by a detailed description are given below.

FEATURE OVERVIEW

- **Validation model:** Zero or more validators register on each `EditableValueHolder` component
- **Built-in validators:** 5 standard validators (`DoubleRangeValidator`, `LengthValidator`, `LongRangeValidator`, `RegexValidator`, `RequiredValidator`)
- **Bean Validation support:** Supported
- **Bean Validation integration:** Smoothly integrated without any implementation overhead (automatic implementation detection)
- **Validation location:** Server-side[1]
- **Violation rendering:** Error messages can be displayed automatically
- **JSR status:** Maintenance (22/11/2010)[2]; JSF 2.2 (JSR 344) is currently worked on

DETAILED DESCRIPTION

JSF 2.1 allows to add zero or more validators to an `EditableValueHolder` component (e.g. a text box) where each validator has to implement the `javax.faces.validator.Validator` interface. The user can specify the default validators that are automatically registered to every `UIInput` component (for example the five standard validators as mentioned in the feature overview or the bean validator). During the 'Process Validation' phase the validation mechanism is triggered for every `EditableValueHolder` component in the component tree. If a validation violation occurs, a `ValidatorException` is thrown which includes a `FacesMessage` with the error message. The `FacesMessage` is used to render the error message for the user feedback. If a Bean Validation (see section 10.1.2 and 10.1.1) implementation is available in the runtime environment of a JSF 2.1 application (and not explicitly disabled), the standard validator of the Bean Validation implementation must be used (maybe in addition to the standard validators). The JSR 314 does only mention the Bean Validation 1.0 integration according to JSR 303 (see section 10.1.2), but Bean Validation 1.1 according to the JSR 349 (see section 10.1.1) does provide all the features of the JSR 303 and therefore, the implementations of the JSR 349 can be used. It is not explicitly mentioned where the data validation happens, but the 'Request Processing Lifecycle' runs on the server-side which includes the 'Process Validation' phase. The error messages are rendered if the element `<h:message>` or `<h:messages>` is included in the web page. Furthermore, it is not explicitly described if the full Bean Validation functionality must be supported. The specification mentions a 'field level approach' that should be supported.

[1] Server-side validation means that the validation process takes place within the logic tier that usually runs on the server. Data is sent to the server, validated and the response is sent back (e.g. validation violations).

[2] Checked on 14/05/2013

A more detailed description of the JSF 2.1 concept can be found in the JSR 314. Especially, chapter 3.5 of the 'JSR-000314 JavaServer™ Faces 2.1 Maintenance Release 2' which can be found on the website of the Java™ Community Process [37] describes the validation model.

12.2.1.1 ORACLE MOJARRA JAVASERVER FACES

Mojarra 2.1[3] is the reference implementation of the JSR 314 (see section 12.2.1). Therefore, every feature mentioned in the JSR 314 is supported by Mojarra 2.1.
A summary of the additional features (not mentioned in the JSR 314) followed by a detailed description of the Mojarra 2.1 implementation are given below.

ADDITIONAL FEATURE OVERVIEW

- **Documentation:** Java™ Faces API, forum, no examples or tutorials (just links to third party websites)
- **Licence:** GNU General Public Licence (GPL) + 'Classpath Exception' and Common Development and Distribution Licence (CDDL) 1.0
- **Latest release:** Version 2.1.22 (08/05/2013)[4]

DETAILED DESCRIPTION

Mojarra 2.1 is 'just' the reference implementation of the JSR 314 without any additional features. All the features are described in the JSR 314 (see section 12.2.1) or on the project web site of Mojarra[5].

12.2.1.2 APACHE MYFACES CORE

Apache MyFaces[6] is a collection of libraries providing features for developing JSF applications. The Apache MyFaces Core library implements the JSR 314 [37] (see section 12.2.1). The Apache MyFaces Core 2.1 supports all the features mentioned in the JSR 314 (which includes the possibility to use the Bean Validation 1.0 defined in the JSR 303).
A summary of the additional features (not mentioned in the JSR 314), a detailed description of the Apache MyFaces Core implementation followed by an example are given below.

FEATURE OVERVIEW

- **Documentation:** Detailed documentation, examples, forum, detailed Java™ API available
- **Licence:** Apache Licence 2.0
- **Latest release:** Version 2.1.11 (01/04/2013)[7]

[3] `https://javaserverfaces.java.net/`, [Online; accessed 15-May-2013]
[4] Checked on 15/05/2013
[5] See footnote 3
[6] `http://myfaces.apache.org/`, [Online; accessed 24-April-2013]
[7] Checked on 15/05/2013

DETAILED DESCRIPTION

The Apache MyFaces core supports all features of the JSR 314, but no additional features are shipped with the core distribution. According to the project website of Apache MyFaces (see section 12.2.1.2) not all features of the Bean Validation 1.0 specification JSR 303 are supported in Apache MyFaces Core which means that the Bean Validation concept is only partially applicable for JSF applications. The following features are not applicable:

- Type-safe group validation
- Support of @Valid
- Model validation
- Severity aware validation
- Client-side validation[8]
- Sorted violation messages
- Dependency injection support for constraint validators
- Mapped constraint source

Not all features which are not supported by JSF are defined in the Bean Validation specification (e.g. client-side validation), but they are mentioned as non-supported because they are tightly coupled with data validation. On the other hand, some basic functionalities (e.g. object graph validation) are defined in the JSR 303/349 (Bean Validation 1.0/1.1) but don't work in the standard JSF 2.1 implementations (e.g. Apache MyFaces Core) because they are not integrated.
For a more detailed description see section 12.2.1 or the documentation of the project web site of Apache MyFaces[9].

EXAMPLE

A small example how the JSF standard validators can be used is given below. An example of the Bean Validation integration in JSF can be found in section 12.2.1.3.
We use a simple person object to validate its name with the help of one JSF standard validator (`RequiredValidator`). Next, we have to implement a Managed Bean which handles the logic between our domain model and the Graphical User Interface (GUI) (see listing 12.1).

Listing 12.1: Interface view[10] of the class `PersonBean` which is a Managed Bean that handles the logic between the domain model and the front-end.

```
@ManagedBean
@RequestScoped
```

[8] Client-side validation is a mechanism where the validation process directly starts the client without sending any data to the server. For instance, data is entered in a web form and the validation takes place within the browser (e.g. using JavaScript) without asking the server for validating data (in contrast to server-side validation, see footnote 1).

[9] See footnote 6

```
public class PersonBean {

  public PersonBean() {
  }

  public Person getPerson() {
    return this.person;
  }

  public void setPerson(Person person) {
    this.person = person;
  }

  public String send() {
    return "welcome.xhtml";
  }

}
```

Furthermore, we have to enrich the front-end file to activate the validation during the validation phase. Therefore, we add the `validateRequired` element and the `message` element to display the (error) messages (see listing 12.2).

Listing 12.2: Parts of the front-end file of a JSF web application with enriched HTML elements using JSF standard validators. The standard required validator is added with the help of the `<f:validateRequired/>` element and the data binding is done by referencing the corresponding Managed Bean property (e.g. `value="#personBean.person.name"`)

```
...
<h:form id="mainForm">
  <h:panelGrid columns="3">
    <h:outputLabel for="name" value="Please enter your name:" />
      <h:inputText id="name" label="Name"
          value="#{personBean.person.name}">
        <f:validateRequired/>
      </h:inputText>
      <h:message for="name"/>
      <h:commandButton value="Login" action="#{personBean.send}" />
      <h:outputText value="" />
      <h:outputText value="" />
  </h:panelGrid>
</h:form>
...
```

The error messages are automatically rendered if the `message` element is used. It is possible to render all the messages in a single place or to render each error message separately. In our example (see listing 12.2), only the error messages corresponding to the component with `id` equal to `name` (which is bound to the `name` field of a person object) is rendered. A rendering example is shown in figure 12.1.
The complete example can be found in the appendix section A.2.1.1.

[10] See footnote 3

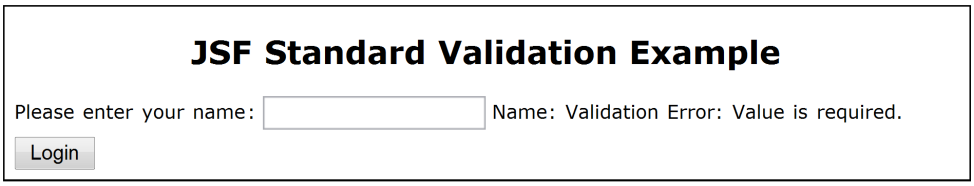


Figure 12.1: GUI example with a rendered error message using the JSF standard validator.

12.2.1.3 APACHE MYFACES CORE AND HIBERNATE VALIDATOR

Now, we describe how the validation process works when using Apache MyFaces Core 2.1.11 (see section 12.2.1.2) and Bean Validation. For Bean Validation we use the Hibernate Validator 5.0.1 (see section 10.1.1.1) implementation. We restrict the analysis to those two implementations because other implementations (e.g. Mojarra or Apache BVal) that fulfil the standard should allow the same approach for data validation.
Web applications using JSF 2.1 work on the basis of a special servlet in a event-driven way. After every event (e.g submitting a form) the server is notified via a Hypertext Transfer Protocol (HTTP) request. This special servlet is implemented in the class `javax.faces.webapp.FacesServlet` of the Apache MyFaces Core library and therefore called the 'Faces servlet'. Within this class, the HTTP request is analysed and an object of type `javax.faces.context.FacesContext` is created which stores all the necessary information of the HTTP request (e.g. the user input data corresponding to the model). Next, the 'Faces servlet' which is the central unit of the JSF application starts the JSF life cycle which is depicted in figure 12.2. In the context of this master thesis the 'Process Validation' phase is the most important one.

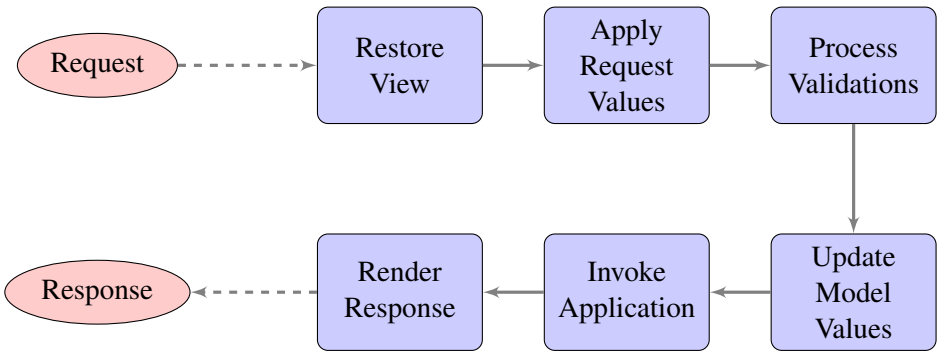


Figure 12.2: Life cycle of a HTTP request in a JSF application.

If the HTTP request should trigger the data validation automatically, then these three conditions must be fulfilled:

1. Hibernate Validator 5 and its dependencies are included in the runtime environment of the Apache MyFaces Core web application.

2. The domain model of the JSF application is annotated with constraints as described in section 10.1.1.1.

3. The GUI component is bound to the correct property of the domain model.

The GUI component of a small example using Apache MyFaces Core and Hibernate Validator is shown in the listings 12.1 and 12.3. If we compare the validation process using Bean Validation 1.1 to the standard validation mechanism of JSF 2.1 (see section 12.2.1.2), we can see that the domain model and the Managed Bean are exactly the same and the front-end file gets rid of the standard validator element. The validation process is automatically triggered against the constraint which is annotated in the domain model.

Listing 12.3: Code excerpt of the front-end file of a JSF web application with enriched HTML elements. This front-end file uses Bean Validatio to validate the input data. Compared to the front-end file depicted in listing 12.2 we do not need to add special constraint tags because validation is automatically done using a JSR 349 (Bean Validation 1.1) implementation and the annotated domain model.

```
...
<h:form id="mainForm">
  <h:panelGrid columns="3">
    <h:outputLabel for="name" value="Please enter your name:" />
    <h:inputText id="name" value="#{personBean.person.name}">
    </h:inputText>
    <h:message for="name" />
    <h:commandButton value="Login" action="#{personBean.send}" />
    <h:outputText value="" />
    <h:outputText value="" />
  </h:panelGrid>
</h:form>
...
```

The ‘Process Validation’ phase for Bean Validation is implemented with the help of the class `javax.faces.validator.BeanValidator` which delegates the validation process to the Hibernate Validator. Within the ‘Process Validation’ phase, the ‘Process Validation Executor’ implemented in class `org.apache.myfaces.lifecycle.ProcessValidationsExecutor` starts the validation process by traversing the component tree starting at the component of type `UIViewRoot`. For every component, the corresponding value stored in an object of type `FacesContext` is converted (e.g. converting the string ‘21.02.1988’ to an object of type `java.util.Date`) and then all registered validators are invoked with the help of the static method `callValidators(FacesContext, UIInput, Object)` of the class `javax.faces.component._ComponentUtils`. For Bean Validation, the validation process uses the method `validateValue(beanType, propertyName, value, groups)` of the Bean Validation 1.1 API (see section 10.1.1.1) which is implemented by the Hibernate Validator.
The result of the validation process (e.g. violations) can be displayed using the `<h:message/>` element. A JSF 2.1 application uses messages (represented by instances of the class `FacesMessage`) for different internal notices (e.g. constraint violations). The message attribute of the constraints in the domain model (i.e. the Java™ annotation attributes named ‘message’ e.g. `@NotNull(message = "name is mandatory")`) are automatically used to display any constraint violation messages if necessary. A sample GUI is shown in figure 12.3.
The full example is available in the appendix section A.2.1.1.

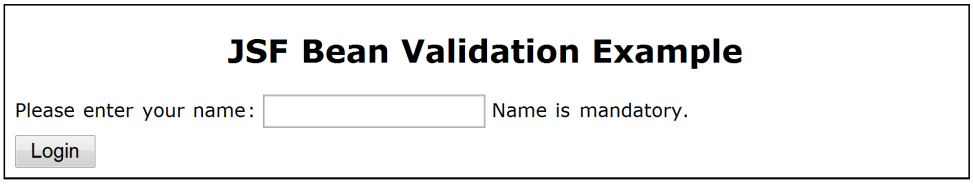


Figure 12.3: GUI example with a rendered error message using Hibernate Validator.

12.2.1.4 JSF COMPONENT FRAMEWORKS

The standard implementation of JSF 2.1 (see section 12.2.1) does only provide a limited amount of UI components and functionalities to build a web application. Therefore, a lot of different component frameworks exist to enhance the standard distributions with additional features (e.g. client-side validation). First, an overview of some JSF component frameworks and a selection of their contributions are given. Next, one component framework is described in more detail.

OVERVIEW

- **Apache MyFaces Trinidad:** Client-side validation[11], components for barrier-free web applications.
- **Apache MyFaces Tobago:** Building web applications without directly writing HTML, CSS or JavaScript.
- **Apache MyFaces Extensions Validator (ExtVal):** Client-side validation, object graph validation (@Valid).
- **JBoss RichFaces:** Client-side validation, object validation, object graph validation (@Valid), advanced Asynchronous JavaScript and XML (AJAX) capabilities, additional components and skins, components for mobile devices
- **ICEsoft ICEfaces:** Advanced AJAX features, additional components, components for mobile devices
- **Prime Teknolojili PrimeFaces:** Advanced AJAX features, additional components, components for mobile devices, skins, partial validation

JSF COMPONENTN FRAMEWORK: JBOSS RICHFACES

RichFaces 4[12] is a project of JBoss which provides a framework that can be used on top of a JSF 2.1 implementation. One of the main objectives of the framework are the enhanced AJAX capabilities and the client-side validation with the help of Bean Validation. Therefore, the framework introduces its own elements to use their own components and new capabilities. Client-side validation is provided for the standard JSF validators and the JSR 303/349 (Bean Validation 1.0/1.1) validators (see section 12.2.1). This validation mechanism

[11] See footnote 8

[12] `http://www.jboss.org/richfaces`, [Online; accessed 22-May-2013]

is activated by adding the <rich:validator/> element to the input field that has to be validated. The JavaScript and AJAX mechanism for the validation process is automatically generated for Bean Validation and the standard JSF validation and the developer does not have to write a single line of JavaScript. Unfortunately, not all validators defined in the JSR 303/349 are implemented for client-side validation in the RichFaces version 4.3.2, but according to the website, the road map includes the extension of the client-side validation to support more built-in and custom validators. Nevertheless, if no client-side validation mechanism is provided for a validator, the RichFaces framework uses an AJAX mechanism to validate the data via a server call which works seamlessly to the user and without an additional overhead for the developer. Furthermore, the framework provides the possibility for object graph validation (e.g. the use of the annotation @Valid) which is not defined in the JSF standard. RichFaces provides the elements <rich:message>, <rich:messages> and <rich:notify> to visualise the error messages in an advanced way compared to the standard implementation (e.g. different icons depending on the severity of the validation violation).
The front-end file in listing 12.4 shows how to use the RichFaces components. The domain model and the Managed Bean are exactly the same as in the example of section 12.2.1.3. In figure 12.4 the rendered example is depicted. In this example, the validation mechanism is triggered when the event keyup occurs. Submitting a from (e.g. pressing the command button) does always initialise a server call and therefore server-side validation happens.

Listing 12.4: Source code part of the front-end file of a JSF web application using RichFaces components and Bean Validation. The client-side validation is done (if possible) automatically. The elements using the namespace rich belong to the JBoss RichFaces framework.

```
...
<h:form id="mainForm">
  <h:panelGrid columns="3">
    <h:outputLabel for="name" value="Please enter your name:" />
    <h:inputText id="name" value="#{personBean.person.name}">
      <rich:validator event="keyup" />
    </h:inputText>
    <rich:message for="name" />
    <h:commandButton value="Login" action="#{personBean.send}" />
    <a4j:status>
      <f:facet name="start">
        <h:graphicImage
            value="/resources/images/ajax-loader.gif" alt="ai"
          style="height:12px;width:12px;" />
      </f:facet>
    </a4j:status>
    <h:outputText value="" />
  </h:panelGrid>
</h:form>
...
```

According to the project website, RichFaces is compatible with Oracle Mojarra version 2.1.5 or higher and Apache MyFaces version 2.1.5 or higher. The example uses Apache MyFaces 2.1.11, Hibernate Validator 5.0.1 and JBoss RichFaces 4.3.3. In addition, the following built-in constraints defined in the JSR 349 (Bean Validation 1.1), see section 10.1.1, were tested with this combination regarding client-side validation support.

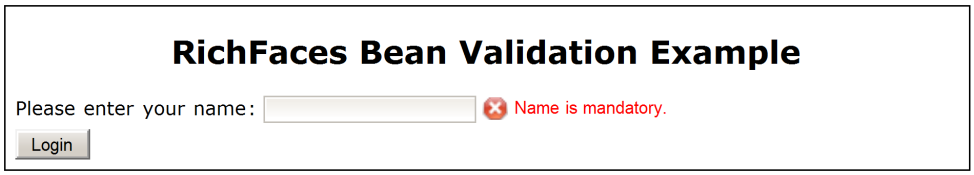


Figure 12.4: RichFaces rendering example of an error message.

- Supported client-side validation constraints
  - **@NotNull**
  - **@AssertTrue**
  - **@AssertFalse**
  - **@Min**
  - **@Max**
  - **@Size**
  - **@Pattern**
- Not supported client-side validation constraints
  - **@Null**
  - **@DecimalMin**
  - **@DecimalMax**
  - **@Digits**
  - **@Past**
  - **@Future**
  - **@Valid**

s The code to test if a constraint definition supports client-side validation and the whole source code for the example can be found in the appendix section A.2.1.1.

12.2.2 GOOGLE WEB TOOLKIT

The Rich Internet Application (RIA) framework of Google is called Google Web Toolkit (GWT)[13] and provides functionalities to build web applications with almost pure Java™. Therefore, GWT provides some graphic widgets (e.g. an input field) and Java™ APIs (for features like AJAX , HTML 5, data validation) that are finally translated to JavaScript for e.g. cross-browser compatibility. Since GWT version 2.5, the framework supports validation with almost all features according to JSR 303 (Bean Validation 1.0). Supporting client-side and server-side validation is one of the main goals of GWT.
A summary of the main features regarding data validation of GWT, a detailed description and a small example are given below.

[13] `https://developers.google.com/web-toolkit/`, [Online; accessed 24-May-2013]

FEATURE OVERVIEW

- **Validation model:** Validation mechanism automatically translated to JavaScript (client-side) or manual validation (server-side)
- **Built-in validators:** GWT does not support its own validation mechanism out-of-the-box
- **Bean Validation support:** Bean Validation 1.0 supported
- **Bean Validation integration:** Small coding overhead (some conventions must be followed) to use validation, JavaBeans™ binding must be activated manually (GWT configuration files must be adapted)
- **Validation location:** Client-side[14] and server-side[15]
- **Violation rendering:** Error messages must be displayed manually
- **Documentation:** Tutorial with small examples, forum (Google groups), GWT API
- **Licence:** Apache Licence Version 2.0
- **Latest release:** Version 2.5.1 (11/03/2012)[16]

DETAILED DESCRIPTION

GWT does not ship any validation framework by its own, but provides the possibility to use the features of Bean Validation (see section 10.1.2 and 10.1.1). Unfortunately, they do not provide a full implementation of the JSR 303 (Bean Validation 1.0) and therefore some features are missing. According to the project website, the following features are not applicable (but they are defined in the JSR 303 (Bean Validation 1.0)):

- XML configuration
- `ConstraintValidatorFactory`. Constraint validators are generated using `GWT.create()` instead.
- Validation providers other than the default provider
- Validation provider resolvers

Furthermore, Java™ classes that are not compatible with GWT (e.g `Java.util.Calender`) cannot be used for client-side validation (but server-side validation works). It is the responsibility of the developer to implement validators that can be translated to JavaScript if client-side validation is desired.
The validation model of GWT is divided into two parts. For validation on the client-side, the constraints and the validators are translated to JavaScript which is done automatically. For the server-side validation process, a validator object has to be used and an appropriate

[14] See footnote 8
[15] See footnote 1
[16] Checked on 24/05/2013

validation method can be called to check the constraints against an object. The framework provides the flexibility to decide where (client-side, server-side or both) and at what position in the control flow (e.g. after the click on a button) the validation process should be triggered. In addition, all built-in constraint definitions of the JSR 303 (see section 10.1.2) and every custom constraint definition which can be translated to JavaScript by GWT can be used for client-side validation. If the domain model correspond to JavaBeans™, it is possible to use the GWT `Editor` and `UiBinder` framework to enable the binding between the model objects and the front-end. Constraint violations can occur on client-side and server-side. The processing (e.g. rendering at the front-end) must be implemented manually.
GWT validation does only work with an implementation of the JSR 303 (Bean Validation 1.0) and the validation API 1.0.0 (already included in the GWT distribution).

EXAMPLE

The following example illustrates the main import steps to use Bean Validation 1.0 (JSR 303) within GWT. The example uses GWT 2.5.1 and Hibernate Validator 4.1.0 which is the reference implementation of the JSR 303 and the precursor of Hibernate Validator 5 (see section 10.1.1.1).
We use a person model (see appendix section A.2.1.2) and add a type/class constraint (see listing 12.5) to show client-side and server-side validation. Next, we introduce two groups to manage client-side and server-side validation. The `NotNull` constraint of the person's name is added to the `ClientConstraints` group which is checked at the client-side. The `MagicName` constraint of the class belongs to the `ServerConstraints` group which is handled on the server-side.

Listing 12.5: The annotation `@interface MagicName` is used for a constraint that is checked at the server only using a custom validation group. The corresponding validator class is depicted below the annotation declaration and checks if the name of a person contains the string `"magic"`.

```
@Target({ METHOD, FIELD, ANNOTATION_TYPE, TYPE })
@Retention(RUNTIME)
@Documented
@Constraint(validatedBy = { MagicNameValidator.class })
public @interface MagicName {

    String message() default "not a magic name";

    Class<?>[] groups() default {};

    Class<? extends Payload>[] payload() default {};

}

public class MagicNameValidator implements
  ConstraintValidator<MagicName, Person> {

  public void initialize(MagicName constraintAnnotation) {
  }

  public boolean isValid(Person person,
```

```
    ConstraintValidatorContext context) {

    if (person == null) {
      return true;
    }

    String name = person.getName();
    return name == null || name.contains("magic");
  }

}
```

The binding between the JavaBean™ (the person model) and the front-end file which contains a text field to enter the name is implemented with the help of the `Editor` and `UiBinder` framework of GWT. Therefore, we create a new class which implements the `Editor<Person>` interface and extends the class `Composite` to generate a text box (see listing 12.6). Furthermore, we have to provide an XML file, where we declare the text box which is afterwards included in the front-end file (see listing 12.7).

Listing 12.6: Data binding in GWT is done via JavaBean™ properties: the declared `TextBox` which is called `name` must correspond to a getter method denoted as `getName()` in the underlying data model. The corresponding data model is passed as an actual generic parameter to the interface `Editor` which is in our case `Person`.

```
public class PersonEditor extends Composite
  implements Editor<Person> {

  private static PersonEditorUiBinder uiBinder =
    GWT.create(PersonEditorUiBinder.class);

  interface PersonEditorUiBinder extends
    UiBinder<Widget, PersonEditor> {
  }

  @UiField
  TextBox name;

  public PersonEditor() {
    initWidget(uiBinder.createAndBindUi(this));
  }

}
```

Listing 12.7: XML snippet which defines a front-end text box in a declarative way using GWT templates. Afterwards, the text box with the `ui:field` set to `name` is connected with the declaration depicted in 12.6.

```
<ui:UiBinder xmlns:ui="urn:ui:com.google.gwt.uibinder"
  xmlns:g="urn:import:com.google.gwt.user.client.ui">

  <g:TextBox ui:field="name" />

</ui:UiBinder>
```

Finally, we have to implement a validator factory and the logic (server-side and client-side). A validator factory can be obtained by extending the `AbstractGwtValidatorFactory` and adding the class to be validated and the different groups to the `@GwtValidation` annotation (see appendix section A.2.1.2). The important parts of the validation process are depicted in listing 12.8 and 12.9.
For the client-side validation, the code is executed if the user clicks on the 'Login' button. Then, a validator object is created and the object `user` of type `Person` is validated. If a validation violation occurs, an error label is used to display the message. This part of the code is completely executed on the client-side, because the GWT will translate it to JavaScript. Then, the AJAX callback function is implemented, which handles the error messages of the server-side validation by implementing the functions `onSuccess` and `onFailure`.
On the server-side, again a validator object is created and used to validate the `user` object of type `Person` which was populated with the data entered by the user (due to the binding implementation). If a validation violation occurs, a corresponding exception is thrown and handled by the callback function of the client-side. If no violation occurs, the response is created.
It is important that a 'dummy' method with return type `ValidationSupport` is included in the class implementing the `RemoteService` to ensure that the Hibernate Validator implementations are serialisable and the module configuration file of GWT inherits the Hibernate Validator.

Listing 12.8: Parts of the `GwtBeanValidation` class which realises the client-side validation (see comment `Do client-side validation`). The relevant source code is automatically translated to JavaScript which can then run in the browser. and error message handling (for client-side and server-side). Server-side validation is initiated with an AJAX request followed by a result handling on the client-side (`onFailure`/`onSuccess`).

```
...
// With the help of binding, we get the value of the text box.
Person user = driver.flush();

// Do client-side validation
Validator validator =
  Validation.buildDefaultValidatorFactory().getValidator();
Set<ConstraintViolation<Person>> violations =
  validator.validate(user, ClientConstraints.class);
if (!violations.isEmpty()) {
  errorLabel.setText(
    violations.iterator().next().getMessage());
  return;
}
loginButton.setEnabled(false);

// Client-side validation passed so check server-side
welcomeService.welcomeUser(user, new AsyncCallback<SafeHtml>() {

  public void onFailure(Throwable caught) {

    // Check want kind of exception occurred
    if (caught instanceof ConstraintViolationException) {
      ConstraintViolationException cvex =
```

```
        (ConstraintViolationException) caught;
      errorLabel.setText(cvex.getConstraintViolations()
        .iterator().next().getMessage());
      return;
    } else {
      errorLabel.setText("general error");
    }
  }

  public void onSuccess(SafeHtml result) {
    responseLabel.setText(result.asString());
  }
});
...
```

Listing 12.9: Server-side validation is started by an AJAX request from the client. Next, the server manually instantiate a validator (as for the client-side) using the `ServerConstraints.class` group. Depending on the outcome a `ConstraintViolationException` is thrown or a response message is returned which is then handled by the client in the corresponding method (`onFailure`/`onSuccess`).

```
...
public SafeHtml welcomeUser(Person user)
  throws IllegalArgumentException, ConstraintViolationException {

  // Do server side-validation
  Set<ConstraintViolation<Person>> violations =
    validator.validate(user, ServerConstraints.class);

  // Throw exception if a violation occurred,
  // checked by onFailure part
  if (!violations.isEmpty()) {
      HashSet<ConstraintViolation<?>> temp =
        new HashSet<ConstraintViolation<?>>(violations);
      throw new ConstraintViolationException(temp);
  } else {
      // Return user name processed by onSuccess
      ...
      return response;
  }
}
...
```

The full source code of the code excerpts above and any additional files to run the example can be found in the appendix section A.2.1.2. Figure 12.5 and 12.6 show the front-end when validating on the client-side and on the server-side, respectively.

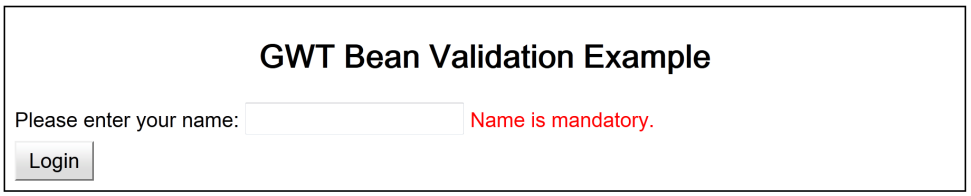


Figure 12.5: GWT rendering example of a client validation violation.

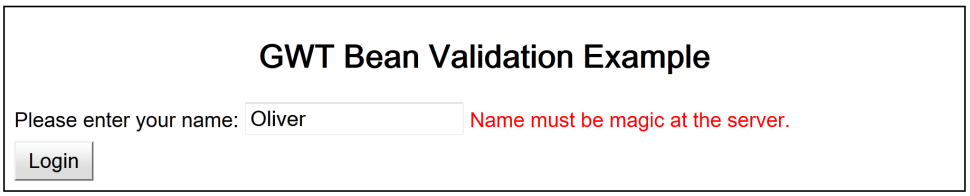


Figure 12.6: GWT rendering example of a server validation violation. 'Name must be magic at the server' means that the entered name does not contain the word 'magic' as defined in the constraint depicted in 12.5.

12.2.3 JAVA™ FOUNDATION CLASSES: SWING

The Java™ Foundation Classes (JFC) are a collection of APIs to develop GUIs in Java™. One of the central components of the JFC is the Swing API[17] to create desktop applications with GUI components that do not depend on the underlying Operating System (OS) where the application is running. Swing builds on top of the Abstract Window Toolkit (AWT) API which is also included in the JFC. While AWT does only allow to use the graphical components of the underlying OS, the Swing components are fully implemented in Java™ and therefore can be used on every OS. Furthermore, Swing provides some look and feel components to simulate the characteristics of the different OS.
The Swing API does not provide any built-in framework for input validation nor it defines a standard for the integration with Bean Validation (see 10.1.2). Nevertheless, we present two possibilities to enable data validation in the following sections because Swing is a common way of building graphical applications.

12.2.3.1 JFC SWING: ACTION LISTENER APPROACH

The straightforward way, which we call 'action listener approach', to implement data validation in form of input validation, is with the help of some Java™ action listeners that will be triggered if the user clicks on a button or enters some data in a text field. This kind of approach makes use of a cross-layer validator like Hibernate Validator (see section 10.1.1.1). The following basic steps are necessary to enable a validation mechanism for input validation (e.g. validation of a form):

1. Define a domain model
2. Design and implement a graphical form

[17] `http://docs.oracle.com/javase/7/docs/technotes/guides/swing/`, [Online; accessed 27-May-2013]

3. Implement the desired action listeners
4. Trigger the validation process for all relevant event types

The following example shows how to validate an input form, where the user can login with his name. We use the domain model described in the appendix section A.2.1.3 for our user who wants to login. Next, we use Hibernate Validator (see section 10.1.1.1) to validate our model. The validation mechanism should happen if the user clicks on a login button or if the value of the text box changes.
In listing 12.10 the listener implementation for the login button is shown. The validation process of Hibernate Validator is triggered manually by calling the `validatePerson` method which wraps the validator creation and validation process with Hibernate Validator. In addition, the error messages are implemented by simple label changes. The implementation for the text box listener and the detailed code of the input form can be studied in the appendix section A.2.1.3. Figure 12.7 shows the GUI part of the input form when an error occurs.

Listing 12.10: Code excerpt of the 'action listener approach' implementation which shows the validation process if a user clicks on the button denoted with `login` and labelled with 'Login' as depicted in figure 12.7. The actual validation takes place in the `validatePerson` method which instantiates a validator and calls the `validate` method.

```
...
login.addActionListener(new ActionListener() {
  public void actionPerformed(ActionEvent e) {
    Person per = new Person();
    per.setName(nameField.getText());

    // Validate when user clicks on the login button
    String result = validatePerson(per);
    if (result != null) {
      labelErrorMessage.setText(result);
    } else {
      labelErrorMessage.setText("Hello " + per.getName()
          + "! We hope you enjoy Java Swing!");
    }
  }
});
...
```

With this kind of approach, all features of the underlying validator (e.g. Hibernate Validator) can be used, but everything has to be implemented manually.

12.2.3.2 SWING FORM BUILDER

Swing Form Builder[18] is a library which can create Java™ Swing applications based on JavaBeans™. The library is specialised in creating user forms out of the domain models (JavaBeans™). In addition, the library automatically validates the user input with Bean Validation 1.0 (see section 10.1.2) using the reference implementation Hibernate Validator (see section 10.1.1.1).
A summary of the most important features followed by a detailed description are given below.

[18] `http://code.google.com/p/swing-formbuilder/`, [Online; accessed 28-May-2013]

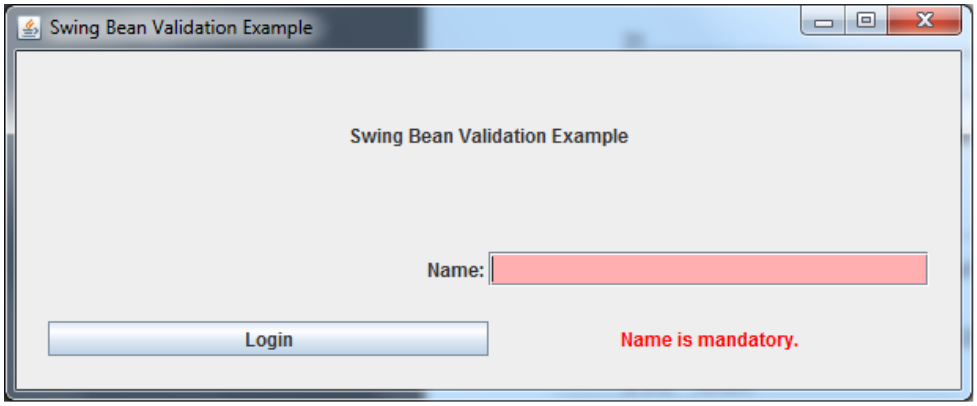


Figure 12.7: GUI of the input form using Java™ Swing.

FEATURE OVERVIEW

- **Validation model:** Validation process is coupled with Java™ Swing components, the corresponding JavaBeans™ and a `ChangeHandler`
- **Built-in validators:** Swing Form Builder does not support its own validation mechanism out-of-the-box
- **Bean Validation support:** Yes, but only a subset of the Bean Validation 1.0 features
- **Bean Validation integration:** Smoothly integrated without any implementation overhead for a set of Java™ Swing components (automatic implementation detection)
- **Validation location:** Stand-alone (i.e. code runs completely at client-side)
- **Violation rendering:** Java™ Swing component is marked red and the error message is shown as a tool tip
- **Documentation:** Only some tiny examples are provided
- **Licence:** Apache Licence Version 2.0
- **Latest release:** Version 1.1 (31/08/2010)[19]

DETAILED DESCRIPTION

The primary goal of Swing Form Builder is the alleviation of creating Java™ Swing components for input forms. Therefore, the library provides a service to create input forms on the basis of JavaBeans™ (a domain model) by mapping Java™ types to different Swing components. Furthermore, Swing Form Builder does not only bind the JavaBeans™ to the form components, but it also uses the Bean Validation 1.0 (see section 10.1.2) mechanism to automatically validate JavaBeans™ if a implementation like Hibernate Validator (see section 10.1.1.1) is available in the runtime environment. According to the project website Swing Form Builder needs Hibernate Validator to use Bean Validation, but running the examples

[19] Checked on 29/05/2013

with Apache BVal (see section 10.1.2.1) showed that Swing Form Builder does only need a implementation of the JSR 303 (Bean Validation 1.0), see section 10.1.2, because everything worked as using Hibernate Validator. Unfortunately, if we do not include a Bean Validation provider, the Swing Form Builder throws an exception and the application does not run. The developer has to explicitly disable the validation process with the help of the method `doValidation`.
Next, Swing Form Builder does not provide the full spectrum of Bean Validation. It only validates a value of a user form input field against the corresponding JavaBean™ property (i.e. field or getter method) using the annotated property constraints. This behaviour can be seen in listing 12.11 which is a code excerpt of the Swing Form Builder implementation. It is not described how to change this restriction or how to provide another implementation.

Listing 12.11: Parts of the source code of Swing Form Builder[20] that shows how JSR 303/349 (Bean Validation 1.0/1.1) is integrated into the library. The variable `beanType` corresponds to the underlying domain class which is bound to the form, the `propertyName` variable is the corresponding field of the domain model which is bound to a form element (e.g. a text box) and the variable `newValue` is the data provided by the user.

```
...
@Nonnull
protected Set<ConstraintViolation<B>> doValidation(
  @Nonnull final Validator validator,
  @Nullable final V newValue ) {
  final PropertyDescriptor descriptor =
    propertyEditor.getDescriptor();
  final Class<B> beanType = (Class<B>)
    descriptor.getReadMethod().getDeclaringClass();
  final String propertyName = descriptor.getName();
  return
    validator.validateValue( beanType, propertyName, newValue );
}
...
```

In addition, the point of data validation is predefined by the library. The validation mechanism is only triggered if the value of the Swing component changes. Again, no description is available which provides further information about possible extensions to this behaviour.
If validation violations occur, an automatic tool tip is generated. If several error messages per input field exist, then they are concatenated and displayed as a single tool tip. Furthermore, the corresponding Swing component is highlighted with a red colour. The behaviour of a validation violation can be specified when the Swing component to JavaBean™ mapping is implemented (maybe some built-in mappings have to be refactored).
Finally, it is possible to define an own mapping between Java™ types and Swing components to customise the design of a form.

EXAMPLE

The following example validates the name of a person defined using the domain model described in 12.2.3.1. The annotation `@UITitle("Name: ")` is needed to display a

[20] See footnote 18

proper label in the form. The example uses the default settings of the Swing Form Builder in combination with Hibernate Validator 5 (see section 10.1.1.1). No problems occurred while using Hibernate Validator 5 although it is the reference implementation of the JSR 349 (Bean Validation 1.1), see section 10.1.1, because all features of Bean Validation 1.0 are included in Hibernate Validator 5. Within this setup, building a form is a '2-liner' which can be seen in listing 12.12. Bean validation is automatically added and triggered, no additional code is needed.

Listing 12.12: Building the Swing components for a form based on the person domain model using Swing Form Builder is done using two instructions.

```
final Form<Person> form =
  FormBuilder.map(Person.class).buildForm();

JComponent formComponent = form.asComponent();
```

The validation mechanism for the Swing button is implemented in the same way as described in section 12.2.3.1 because Swing Form Builder does not support automatic data validation for other events than `ChangeEvents`.
The complete example can be found in the appendix section A.2.1.3 and a rendering example is depicted in figure 12.8.

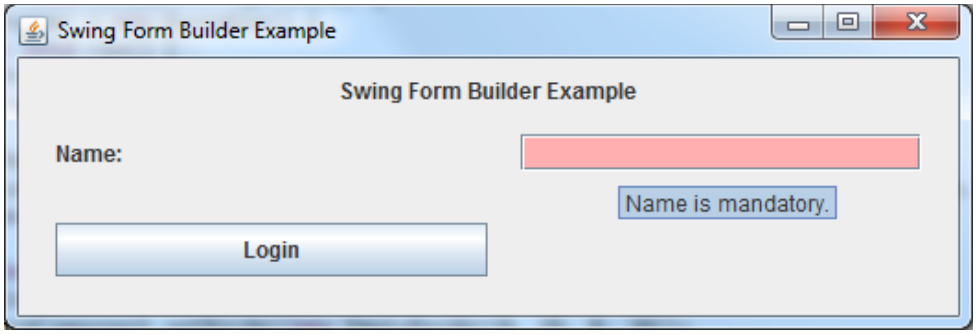


Figure 12.8: GUI of the input form using Swing Form Builder.

12.2.4 THE STANDARD WIDGET TOOLKIT

The Standard Widget Toolkit (SWT)[21] is a collection of GUI APIs to implement Java desktop implementations or Eclipse plug-ins. SWT uses the native graphic components of the underlying OS like AWT. The library provides some wrapper methods to make the use of the native graphic components more convenient for the developer. Furthermore, the look and feel of the underlying OS is automatically used in an SWT application. Therefore, the SWT library cannot be ported to a new OS automatically without adaptation.
SWT does not provide any data validation mechanism out-of-the-box. A similar approach as described in section 12.2.3.1 could be done. Instead of 'action listeners' the SWT environment provides 'event listeners'. The appendix section A.2.1.4 shows a small example and figure 12.9 a sample rendering. The example uses SWT version 4.2.1. Writing complex applications with a lot of graphical elements can be complicated when using SWT. Thus,

[21] `http://www.eclipse.org/swt/`, [Online; accessed 29-May-2013]

the following two sections describe how to overcome the issue of non-existing data validation mechanisms in SWT. Moreover, SWT and the following presented technologies are the basics for writing a rich client using a Rich Client Platform (RCP).

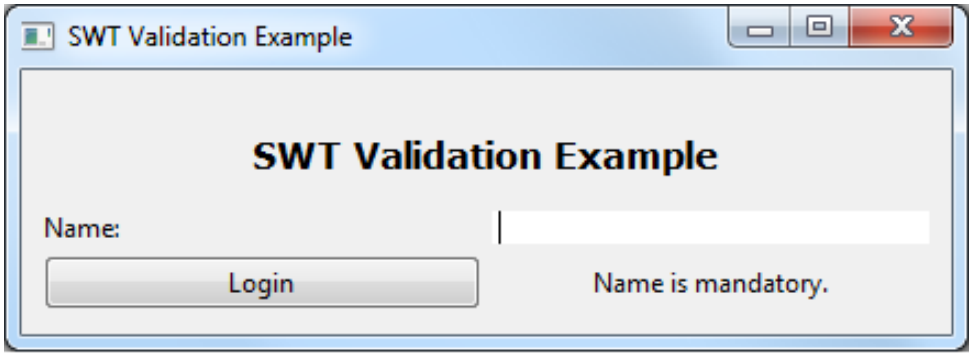


Figure 12.9: Rendered form using SWT and Hibernate Validator under Windows 7.

12.2.4.1 JFACE STANDARD VALIDATION

Eclipse introduced JFace[22] to solve some of the issues which can occur when using SWT (see section 12.2.4) like building complex graphical components (e.g. tables or trees). Furthermore, since Eclipse 3.2 the JFace widget toolkit provides a framework to bind domain models and widget components (e.g. text field) together for automatic data propagation between the front-end and the domain model objects in the business logic. Finally, the library provides an interface to implement validators and combine them with the principle of data binding. JFace does not replace the need of the underlying SWT library nor does it hide its API, but it extends the features of SWT.
The following sections give a feature overview and a detailed description regarding the data validation mechanism using JFace.

FEATURE OVERVIEW

- **Validation model:** Binding of validators with the domain model to a widget
- **Built-in validators:** None
- **Bean Validation support:** No. Manual usage is possible
- **Bean Validation integration:** Not integrated
- **Validation location:** Stand-alone (i.e. code runs completely at client-side)
- **Violation rendering:** Must be defined manually (e.g. field decorators exist)
- **Documentation:** Good Documentation, API (provided by Eclipse), examples
- **Licence:** Eclipse Public Licence (EPL)
- **Latest release:** Version 3.8.0 (22/05/2012)[23]

[22] `http://wiki.eclipse.org/JFace`, [Online; accessed 30-May-2013]
[23] Checked on 30/05/2013

DETAILED DESCRIPTION

Building the graphical part of a JFace application works basically similar as if one uses SWT only. Data validation can be integrated following few steps. First, a validator has to be defined by implementing the interface `IValidator` provided by the JFace data binding API. An example implementation of a validator (`StringRequiredValidator`) can be found in appendix section A.2.1.4. Furthermore, a `MultiValidator` class exists which can be extended to implement cross-field validation.
Next, the data binding between the domain model and the widget has to be specified. The small code snippet in listing 12.13 shows the steps in detail.

Listing 12.13: Code excerpt which shows how to bind a text field (`nameField`) with a field of the underlying domain model (`"name"`, `person`) using a standard validator implementation (`StringRequiredValidator`).

```
...
//Step 1
final DataBindingContext dataBindingContext =
  new DataBindingContext();
//Step 2
final StringRequiredValidator stringRequiredValidator =
  new StringRequiredValidator
      ("Name is mandatory.", nameControlDecoration);
//Step 3
dataBindingContext.bindValue(
  //Step 3a
  SWTObservables.observeText(nameField, SWT.Modify),
  //Step 3b
  BeansObservables.observeValue(person, "name"),
  //Step 3c
  new UpdateValueStrategy
  (UpdateValueStrategy.POLICY_UPDATE).
    setBeforeSetValidator(stringRequiredValidator),
  //Step 3d
  null);
...
```

- **Step 1**: Create an object of type `DataBindingContext` provided by the JFace data binding API.
- **Step 2**: Use a validator which was previously defined. In our case, we have to pass the error message and its decorator as constructor arguments.
- **Step 3**: Use the method `bindValue` to bind the graphical component, the domain model and the validator.
  - **Step 3a)**: The first argument is an observable for SWT widgets. The `SWTObservables` is a factory that provides several strategies to observe a widget (e.g. observe the text attribute of a widget). The method `observeText` requires an SWT control object (e.g. a text box) and an event type which specifies

what kind of actions are observed (e.g. `SWT.Modify`). The code in listing 12.13 defines that every text modification in the text box `nameField` is handled.

- **Step 3b)**: The second argument is an observable for the domain model. Depending on the domain model implementation one can either use the `PojoObservables` factory or the `BeansObservables` factory to create an observable. Using the `BeansObservables` one has to add additional methods to the domain model to handle the property change events. Similar as in step 3a), there are different strategies to observe a domain model (e.g. observe the value of a property). The method `observeValue` expects an object of the domain model which should be observed and the name of the attribute as a string which gets connected to the widget. For the example code in listing 12.13 we have defined that the widget has to be connected to the `name` field of our person domain model (see appendix section A.2.1.4).
- **Step 3c)**: The third argument defines when and how the updates of the SWT control (e.g. a text box) are propagated to the domain model. Therefore, a new object of type `UpdateValueStrategy` has to be created with the update policy as an argument (e.g. `UpdateValueStrategy.POLICY_UPDATE`). Next, a validator can be specified which is then automatically triggered according to the update policy and the event type. Furthermore, the time when the validation should happen within the update phase of a domain model can be specified. The update phase includes the following phases:

  1. Validate after get (use method `setAfterGetValidator`)
  2. Conversion
  3. Validate after conversion (use method `setAfterConvertValidator`)
  4. Validate before set (use method `setBeforeSetValidator`)
  5. Value set

  Independently of the chosen validation moment, one has to pass the corresponding validator instance as an argument, otherwise no validation will happen (there are no default validators). For our example, the validation will happen on every change of the source control widget (e.g. `nameField`) and is done right before the value of the `nameField` is set to the corresponding property of the domain model (e.g. the attribute `name` of the object of type `Person`).
- **Step 3d)**: The last argument can be used to specify the update strategy when a value is shipped from the domain model to the SWT control widget (e.g. text box). In our case we do not specify this case, because we do only save the values entered by the user with the help of the GUI.

It is important to note that the validation mechanism does not update the underlying domain model if a validation violation occurred. The JFace API defines this moment in the following way:

'Statuses returned from validators are aggregated into a `MultiStatus` until a status of `ERROR` or `CANCEL` is encountered. Either of these statuses will abort the update process.'

Therefore, if the underlying domain model is checked manually (i.e. without the use of data binding), for example after a click on a button, the domain model could still store some old values which are not displayed in the front-end.

EXAMPLE

The complete example can be found in the appendix section A.2.1.4. It shows a small form which asks the user to enter a name. The steps described in the detailed description were applied and the example shows a possible way to overcome the issue if the model is not up to date due to some violation errors. In addition, the example code shows how to use the decorator interface and how to use the feature of severity aware validation. Figure 12.10 shows a rendering example.

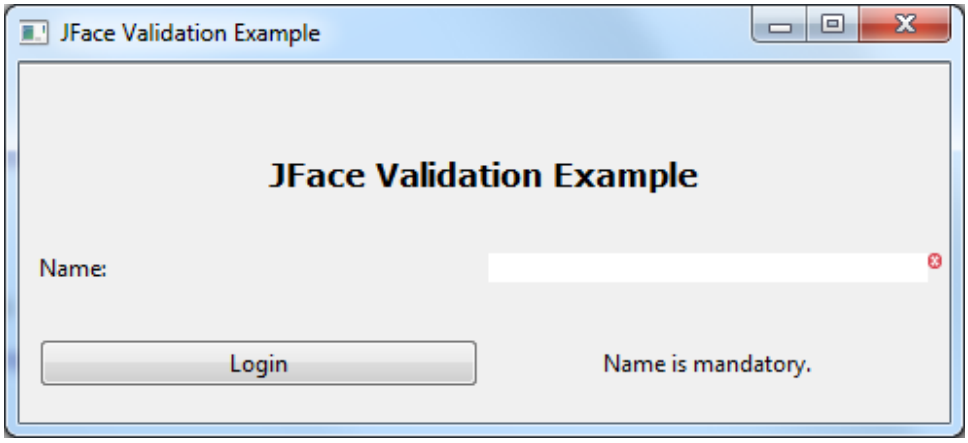


Figure 12.10: JFace rendering example of a validation violation.

12.2.4.2 JFACE BEAN VALIDATION

JFace does not provide any API to provide Bean Validation out-of-the-box but the JFace Data Binding[24] project website, which is an additional library for JFace, provides some experimental work to integrate Bean Validation. The code which implements the integration of Bean Validation is provided by two third party developers. The contribution is not included in any official release of Eclipse. The extension is provided via some Java™ classes that have to be added to the project manually (no jar library exists).
The following sections give an overview of the features and a detailed description.

FEATURE OVERVIEW

- **Validation model:** Binding of a Bean Validator with the domain model to a widget
- **Built-in validators:** None
- **Bean Validation support:** Yes, but only a subset of the Bean Validation 1.0 features

[24] `http://wiki.eclipse.org/JFace_Data_Binding`, [Online; accessed 31-May-2013]

- **Bean Validation integration:** No automatic detection or validation. Validator must be created and bound to the widget
- **Validation location:** Stand-alone (i.e. code runs completely at client-side)
- **Violation rendering:** Must be defined manually (e.g. field decorators exist)
- **Documentation:** Tiny documentation, one small running example
- **Licence:** EPL
- **Latest release:** No library published. Just the Java™ classes are available (2011)[25]

DETAILED DESCRIPTION

The basic idea of JFace Bean Validation is the implementation of the `IValidator` interface which is provided by the JFace data binding API (see section 12.2.4.1). Within this implementation the method `validate` has to be implemented which simply delegates the validation process to a Bean Validation implementation like Hibernate Validator. The project website focuses on providing Bean Validation implementations according to the JSR 303 (Bean Validation 1.0), see section 10.1.2, but an implementation of the JSR 349 (Bean Validation 1.1) could also be used (see section 10.1.1 and the example which uses Hibernate Validator 5). The code excerpt in listing 12.14 shows this delegation process.

Listing 12.14: Code excerpt that shows the delegation of the validation process to an implementation of Bean Validation. A default validator is instantiated and the `validateValue` method with the appropriate arguments is called. The status is adapted according to the number of constraint violations.

```
...
public IStatus validate(Object value) {

  // Delegation to an implementation of Bean Validation
  Set<ConstraintViolation> violations = validateConstraints(value);
  if (violations.size() > 0) {
    ...
    return ...
  }
  return Status.OK_STATUS;
}
...
protected Set<ConstraintViolation>
    validateConstraints(Object value) {

  // Get Bean Validation validator and use its validate mechanism
  return (Set<ConstraintViolation>) getValidatorFactory().
    getValidator().validateValue(beanType, propertyName, value);
}
```

The class `JSR303BeanValidator` defines what kind of features of the JSR 303 (Bean Validation 1.0) are applicable. Unfortunately, the implementation does only provide a validation by using the method `validateValue` of the Bean Validation implementation which

[25] Checked on 31/05/2013

means that only a property set to a specific value can be checked against the defined constraints (e.g. no object graph validation, type validation, group validation). But the validation mechanism allows to specify custom constraint definitions, because the validation process is just delegated. Furthermore, five additional classes are provided to facilitate the configuration of binding and validation. A special `UpdateValueStrategyFactory` is provided which manages the binding of JavaBeans™ to the Bean Validation implementation automatically. Listing 12.15 shows an example of its usage.

Listing 12.15: Code excerpt that shows how to use a Bean Validation implementation with the help of the `JSR303BeansUpdateValueStrategyFactory`. Compared to the standard validation described in section 12.2.4.1 there are some minor differences: no separate validation class has to be implemented because the validation process is delegated to a Bean Validation implementation and the property to validate is automatically detected using the variable `modelNameObserveValue`.

```
...
final DataBindingContext dataBindingContext =
  new DataBindingContext();

IObservableValue modelNameObserveValue =
  BeansObservables.observeValue(person, "name");

Binding binding = dataBindingContext.bindValue(
    SWTObservables.observeText(nameField, SWT.Modify),
    modelNameObserveValue,

    //The property to validate is automatically
    //recognized by the object modelNameObserveValue
    //of type IObservableValue
    JSR303BeansUpdateValueStrategyFactory.create(
        modelNameObserveValue,
        true,
        UpdateValueStrategy.POLICY_UPDATE
      ),
    null
  );
...
```

The display of validation violations must be handled manually (for example with field decorators). Compared to the JFace Standard Validation (see section 12.2.4.1) the mechanism does not differ that much. The basic steps are the same, just the final validation step is done by a library (i.e. a Bean Validation implementation).

EXAMPLE

The complete example is depicted in the appendix section A.2.1.4 and it is almost the same as for the standard validation approach (see section 12.2.4.1. The only difference is that the validation process is delegated to a Bean Validation implementation (in this case Hibernate Validator). Figure 12.11 shows a rendering example.
The classes for the JSR 303/349 (Bean Validation 1.0/1.1) integration are available in the appendix section A.2.1.4.

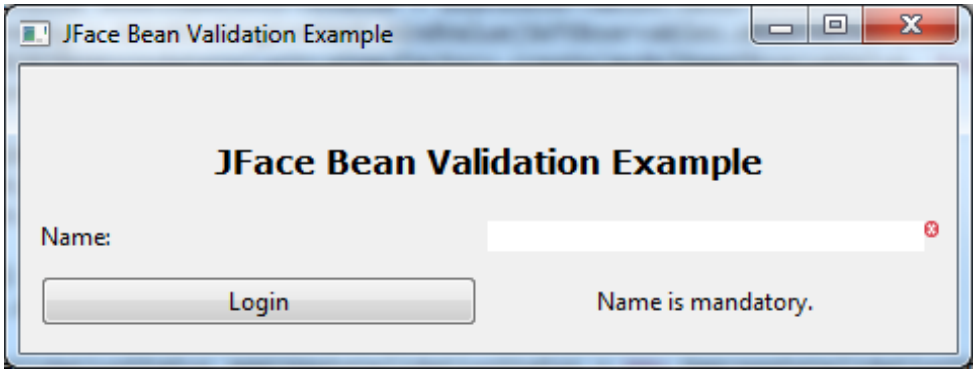


Figure 12.11: JFace rendering example of a validation violation using Bean Validation.

12.2.5 JAVAFX

JavaFX[26] is an API which provides a set of graphic, audio and video components to build RIAs. Applications written with JavaFX can run on several devices and can be used as a web or desktop application. Up to JavaFX 1.3.1, a special scripting language was used to build an application with code translation to pure Java™. Since version 2.0, JavaFX applications can be written using only Java™. The library does not provide any built-in validation mechanisms which work out-of-the-box. A simple listener and binding approach as described in section 12.2.3.1 and 12.2.4.1 is possible. The following section describes how to use an external library to integrated Bean Validation into JavaFX applications. We focus on libraries using Java FX 2.2 because for JavaFX Scripting (version 1.3.1) the 'end of life announcement'[27] was published in May 2013.

12.2.5.1 FXFORM2

The FXForm2 library[28] provides an API to generate graphical forms automatically using JavaFX 2. The generation process is based on JavaBeans™ which defines the matching between the underlying data model and its types and the graphical components. In addition, the library integrates Bean Validation according to the JSR 303 (Bean Validation 1.0) to validate user input.
The following sections give an overview of the main features regarding validation followed by a detailed description.

FEATURE OVERVIEW

- **Validation model:** Binding of the domain model and graphical component combined with a listener that triggers validation
- **Built-in validators:** None
- **Bean Validation support:** Yes, but only a subset of the Bean Validation 1.0 features

[26] `http://docs.oracle.com/javafx/index.html`, [Online; accessed 04-June-2013]
[27] `https://blogs.oracle.com/javafx/entry/javafx_1_3_1_runtime`
[28] `http://dooapp.github.io/FXForm2/`, [Online; accessed 04-June-2013]

- **Bean Validation integration:** Smoothtly integrated without any implementation overhead (automatic implementation detection)
- **Validation location:** Stand-alone (i.e. code runs completely at client-side)
- **Violation rendering:** Violation messages are automatically displayed
- **Documentation:** Tiny documentation, some small examples, no API documentation, no forum
- **Licence:** GNU Lesser General Public Licence (LGPL)
- **Latest release:** Version 0.2.1 (April 2013)[29]

DETAILED DESCRIPTION

Using FXForm2 to create default forms in JavaFX 2.2 can be done with few additional instructions which are shown in listing 12.16. The binding between the graphical components and the underlying model is done automatically by the library. In addition, it is possible to customise the design process of the form by implementing form factories to define how the model is mapped to the graphical components and by providing CSS files it is possible to adapt the look and feel of the form.

Listing 12.16: Code excerpt that shows how to build a form in JavaFX 2.2 using the FXForm2 library. The domain model (`person`) is passed as the source to the variable `fxForm` which is of type `FXForm<Person>`. The `FXForm` class takes as the actual generic parameter the type of the underlying domain model. Finally, the form is added to a `GridPane`.

```
...
final Person person = new Person();
final FXForm<Person> fxForm = new FXForm<Person>();
fxForm.setSource(person);
grid.add(fxForm, 0, 1, 2, 1);
...
```

If a JSR 303 (Bean Validation 1.0) (see section 10.1.2) implementation (e.g. Hibernate Validator) is added to the runtime environment of the application, the validation process is activated. The constraints must be annotated on the getter methods (see section 10.1.1) so that the library and the Bean Validation implementation can make use of it. Unfortunately, Bean Validation is only partially supported which means that only a field set to a specific value can be checked against the defined constraints. This behaviour is determined by the validation method that is called during the validation process. In case of FXForm2, the validation process is delegated to a Bean Validation implementation by calling the method `validateValue` which is depicted in listing 12.17. This behaviour means that, among other, no group validation, object graph validation or type validation is possible.

Listing 12.17: Code excerpt of the FXForm2 library that shows how the validation mechanism happens. A validator instance calls the `validateValue` with the appropriate arguments: the underlying domain model class, the field name/getter method name and the actual input value.

[29] Checked on 04/06/2013

```
...
if (validator != null) {
  Set<ConstraintViolation<Object>> constraintViolationSet =
    validator.validateValue((Class<Object>)
      (getElement().getBean().getClass()),
      getElement().getName(),
      newValue
    );
  constraintViolations.clear();
  constraintViolations.addAll(constraintViolationSet);
}
if (constraintViolations.size() == 0) {
  ((PropertyElement) getElement()).setValue(newValue);
}
...
```

If a validation violation occurs, the corresponding graphical component is automatically enriched with a graphical notification including the defined error message. It is important to notice, that if any validation violation occurs, the underlying domain model is not updated (see listing 12.17). The documentation of FXForm2 does not mention how to change this behaviour. Furthermore, it is not possible to trigger the validation process without manually creating a separate validator instance of the Bean Validation implementation. This makes a validation after certain events (e.g. a click on a button) more complicated. A workaround can be found in the example source code (see appendix section A.2.1.5).
The project website recommends to use a JSR 303 (Bean Validation 1.0) implementation (e.g. Hibernate Validator 4.1.0) but it is also possible to use a JSR 349 (Bean Validation 1.1) implementation such as Hibernate Validator 5 (see section 10.1.1.1) if the runtime environment is adapted. If no Bean Validation implementation is found, the validation process is not activated and the form can still be used (no exception is thrown).

EXAMPLE

The complete example can be found in the appendix section A.2.1.5 and it shows the main steps to build a form for a domain model. Moreover, the example shows a possible solution to validate data after a click on a button. Figure 12.12 shows a rendering example using FXForm2 library version 0.2.2, Hibernate Validator 5.0.1 and JavaFX 2.2.

12.3 BUSINESS LAYER VALIDATION

The business layer is responsible for every data processing that has to be done between the presentation layer and the data access layer. Data validation within this layer is usually done by hand-crafted if-then-else statements, exception mechanisms or special validator classes. There is no technology which addresses this kind of validation explicitly. For sure, Bean Validation which is described in section 10.1 can be used for business layer validation by using a validator instance (no automatic validation – the instance has to be created manually).

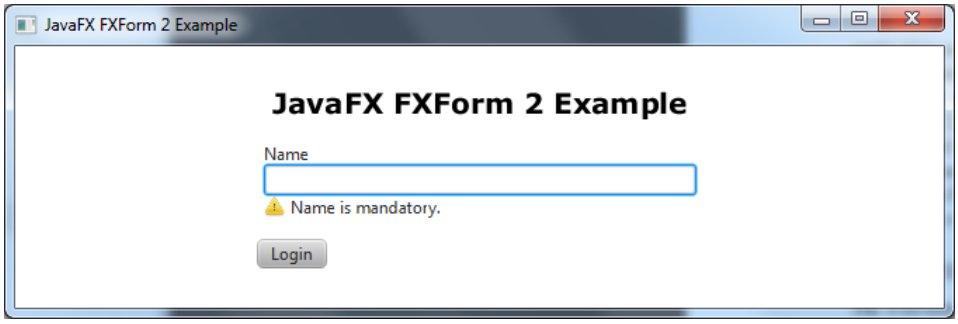


Figure 12.12: FXForm2 using JavaFX 2.2 to render a form with a validation violation using Hibernate Validator.

12.4 DATA ACCESS LAYER VALIDATION

The communication between the persistence tier which represents a data store and the logic tier is done in the data access layer. This layer takes data from the business layer and initiates CRUD operations to the persistence tier. Data from the persistence tier is first processed by the data access layer and then shipped to the business layer for further processing. This section considers validation within the data access layer, i.e. data access layer validation.

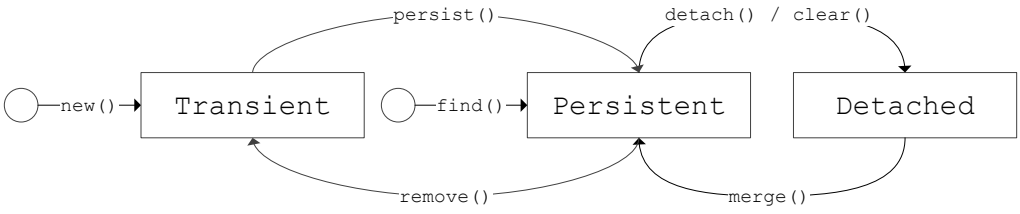


Figure 12.13: An entity can be in three possible states: transient, persistent and detached. The flow-chart shows the state transitions that can happen if the corresponding method is called. The figure was adapted from [39]

12.4.1 JSR 338: JAVA™ PERSISTENCE API

The JSR 338 (JPA 2.1) [40] defines a standard to manage the mapping between Java™ objects and relational databases. Therefore, the standard provides annotations which can be used within a domain model to specify the Object/Relational Mapping (ORM). Furthermore, the specification defines the Java™ Persistence Query Language (JPQL), a criteria query interface, an interface to the native query language, a meta model API (e.g. to access the annotations), stored procedure invocation and the support for data validation. Finally, the ORM can be defined using XML instead of annotations. The standard explicitly mentions that its purpose is only the mapping from objects to relational databases, but some implementations of the standard also support NoSQL databases.
The JSR 338 defines the behaviour when Bean Validation is used in a JPA application which is presented in the feature overview and a detailed description below.

FEATURE OVERVIEW

- **Validation model:** Validation delegation when a pre-persist, pre-update or pre-remove event[30] occurs
- **Built-in validators:** None
- **Bean Validation support:** Yes, for all managed classes
- **Bean Validation integration:** Smoothly integrated without implementation overhead (automatic implementation detection, potential configuration file adaptation)
- **JSR status:** Active (Final Release started on 22/05/2013)[31]

DETAILED DESCRIPTION

The JSR 338 includes a detailed description of every feature which can be found on the website of the JCP [40]. The basic concept of the JSR 338 and Bean Validation (see section 10.1.2 and 10.1.1) is that data validation happens before an object is persisted or removed from the underlying data store. Therefore, no data validation happens directly in the data store. Within the validation process, the validation mechanisms of a Bean Validation implementation, e.g. Hibernate Validator (see section 10.1.1.1) should be used, by calling its `validate` method as specified in the JSR 338 [40]. If a validation violation occurs a `ConstraintViolationException` should be thrown which describes during which event type (e.g. pre-persist) the validation error was detected. This behaviour is independent of the underlying technology and therefore the validation mechanism works for relational DBMS as well as for NoSQL solutions.
The specification defines that automatic Bean Validation should occur after the pre-persist, pre-update and pre-remove life cycle events. The default settings do not include the pre-remove event for automatic Bean Validation. If automatic validation should occur for pre-remove events the `persistence.xml` file has to be adapted. Furthermore, the specification leaves some details regarding the pre-update event dependent on the implementation. The JSR 338 in chapter 3.6.1 ('Automatic Validation Upon Lifecycle Events') says:

'In the case where an entity is persisted and subsequently modified in a single transaction or when an entity is modified and subsequently removed in a single transaction, it is implementation dependent as to whether the pre-update validation event occurs. Portable applications should not rely on this behaviour.' [40]

Inspired by a blog entry[32], which compares two JSR implementations regarding this behaviour, we present the results for every JSR 339 implementation presented in the sections

[30] A pre-persist, pre-update or pre-remove event occurs immediately before the actual event persist, update or remove occurs, respectively. The persist event stores an entity in the database, the update event flushes changes to the database and the remove event deletes an entity.

[31] Checked on 05/06/2013

[32] `http://michael.hoennig.de/2013/05/08/bean-validation-in-jpa2-0/`, [Online; accessed 01-July-2013]

below. In addition, we also check the following two specification aspects according to their correct implementation mentioned in the JSR 338 chapter 3.6.1.2 ('Requirements for Automatic Validation upon Lifecycle Events')[40]:

- 'Validation cascade (@Valid) must not occur for entity associations (single- or multi-valued).'
- 'Embeddable attributes must be validated only if the Valid annotation has been specified on them.'

An embedded attribute is a property of a class annotated with `@Embedded`. The property type is a reference to a class annotated with `@Embeddable`. Compared to classes annotated with `@Entity`, the ORM does not create a separate table for an embeddable, but puts the data into the table of the enclosing entity declaration. The behaviour regarding 'Automatic Validation Upon Lifecycle Events' as mentioned above is presented for each implementation in its corresponding section in a table. The table uses the following abbreviations:

- **`begin:`** Begin of a transaction
- **`persist:`** Persist an object which fulfils every constraint
- **`persist(null):`** Persist an object with an attribute set to null, i.e. violating a constraint (i.e. `@NotNull`)
- **`... = null:`** Change an object's attribute to a value violating a constraint (i.e. `@NotNull`)
- **`flush:`** Forcing the entity manager to write the data immediately to the data store
- **`remove:`** Remove an object from the data store
- **`commit:`** End of transaction, writing all the data to the data store

The first column of the table refers to a test case checking the behaviour regarding the different life cycle events. The detailed test cases are depicted in the appendix section A.2.2. The second column, which is called 'event type' shows whether the last modification of an object (which makes it invalid with respect to the specified constraints) caused an event. For instance, in the second row of table 12.1 no event occurred when updating the object but a pre-persist event occurred of course before. The column 'Exception type' mentions the Java™ exception type that occurred and in the last column you can find the information about the conformance to the corresponding JSR. It is also worth taking a look into the caching behaviour of the implementations of the JSR 338 (JPA 2.1) because it is possible that invalid objects can be queried from the cache.
A more detailed specification of the Bean Validation integration can be found in chapter 3.6 (Bean Validation) of the JSR 338 (JPA 2.1).

12.4.1.1 ECLIPSELINK

The EclipseLink[33] project is part of the Eclipse Foundation and provides a framework to persist data. A general functionality to support several data stores is one of the main goals of this project. Furthermore, EclipseLink is the reference implementation of the JSR 338 (JPA 2.1) [40] and therefore, supports all the features defined in this specification. In particular, EclipseLink supports the Bean Validation integration according to the JSR 338 (JPA 2.1) and some additional, non-relational data stores which are officially not required by the standard. The following sections give an additional feature overview, a detailed description and an example regarding Bean Validation.

ADDITIONAL FEATURE OVERVIEW

Every feature of the JSR 338 (JPA 2.1), see section 12.4.1, is supported by EclipseLink. In addition, EclipseLink implements some additional features and some of those (the most relevant ones) are listed below (the complete list is applicable on the project website[34]).

- **JPQL:** Additional query statements, for instance `UNION`, `INTERSECT` and `EXCEPT`
- **ORM extensions:** `@Struct` and `@Array` to use the specific Database (DB) types
- **Suppported relational DBMS:** Oracle Database, MySQL, PostgreSQL, IBM DB2, Microsoft SQL Server, Sybase Advantage Database Server (ADS), Sybase SQL Anywhere, IBM Informix, Apache Derby, Firebird, H2 Database Engine, Hyper Structured Query Language Database (HSQL), SAP MaxDB, Fujitsu Symfoware Server, Microsoft Access, Attunity, Apache Cloudscape, dBASE, PointBase, Oracle TimesTen
- **Supported NoSQL DBs:** MongoDB, Oracle NoSQL, XML files, JMS, Oracle AQ, SAP HANA, Pervasive PSQL
- **Miscellaneous:** Multi-database integration, data partitioning and sharding, connection pooling
- **Documentation:** Good and detailed library description, some examples, Java™ API, forum
- **Licence:** Dual licenced under EPL and Eclipse Distribution Licence (EDL) (Berkeley Software Distribution (BSD))
- **Latest release:** Version 2.5.0 (28/05/2013)[35]

DETAILED DESCRIPTION

As being the reference implementation of the JSR 338 (JPA 2.1), EclipseLink does smoothly integrate the JSR 303 (Bean Validation 1.0), see section 10.1.2, as defined in the

[33] `http://www.eclipse.org/eclipselink/`, [Online; accessed 05-June-2013]

[34] `http://wiki.eclipse.org/EclipseLink/FAQ/JPA#What_features_beyond_JPA_does_EclipseLink_support.3F`

[35] Checked on 07/06/2013

specification. Furthermore, it is possible to use any implementation of the JSR 349 (Bean Validation 1.1), if the features that were already specified in the JSR 303 (Bean Validation 1.0) are used (see section 10.1.1).
Table 12.1 shows the automatic validation behaviour for the different life cycle events using EclipseLink. After a new object is persisted without any validation violations and then changed to a state that violates a constraint, EclipseLink does not throw an exception, if the `flush` method of the entity manager is not called. The `flush` method forces the application to write the data immediately to the data store. This method is also implicitly called when committing a transaction. Therefore, it is possible that an object which violates some constraints is stored in the data store at commit time. We recommend to use the `flush` method before committing the transaction, because calling this method on a new object marks it as 'used' and then the validation mechanism is triggered on the next pre-update event such as another call of the `flush` method or at committing the transaction.
As defined in the specification, EclipseLink does not consider the `@Valid` annotation for entity associations, but the library checks all references separately. The annotation is considered only for embedded properties during the validation process and if not present (i.e. an embedded property is not annotated), no validation will be done.[36] In addition, EclipseLink does not validate the entity associations if a validation violation occurred in one of the single-valued properties but it validates all annotated embedded properties in any case.
EclipseLink does support the distribution of entities to different data stores with the help of composite persistence units. Especially when relational data stores and NoSQL data stores are mixed, it is useful to check which features are applicable. Not all of the features defined in the JSR 338 (JPA 2.1) are applicable to NoSQL solutions. EclipseLink provides the annotation `@NoSql` to map objects to a NoSQL data store and an overview of the applicable annotations according to the JSR 338 (JPA 2.1) on the project website[37].
It is also possible that the library throws a `RollbackException` which contains a `ConstraintViolationException` when a validation violation was detected at commit time which is not explicitly mentioned in the specification.

EXAMPLE

The example consists of several JUnit test cases defined in the appendix section A.2.2.1 which cover several validation cases (e.g. `@NotNull` constraint) and life cycle validation event types (e.g. pre-persist) based on the domain models depicted in the appendix section A.2.2.1. Every test case has been successfully tested with the following configurations:

- **Bean Validation implementation:** Hibernate Validator 5.0.1 Final, Apache BVal 0.5
- **Data stores:** MySQL Java™ driver 5.1.26, MongoDB Java™ driver 2.11.2
- **JPA 2.1 implementation:** EclipseLink 2.5.0

It is important to set up the `persistence.xml` file correctly. Note, that there are no spaces allowed between the list of validation groups, otherwise the groups are not recog-

[36] `http://wiki.eclipse.org/EclipseLink/Development/JPA_2.0/validation_api_integration`
[37] `http://www.eclipse.org/eclipselink/documentation/2.4/concepts/nosql003.htm`

nised correctly by the library. The appendix section A.2.2.1 shows the complete test cases including the correct `persistence.xml` file for MySQL and MongoDB. Note that the `@NoSql(dataFormat = DataFormatType.MAPPED)` annotation has to be added to every class (entity or embeddable) in the domain model which will be persisted when using a NoSQL solution. Moreover, the unique identifier (defined with the `@Id` annotation) has to be of type `String` when using MongoDB and EclipseLink. Finally, the 'drop-and-create-tables' command is not supported for NoSQL solutions.

12.4.1.2 HIBERNATE ORM

Hibernate is a collection of projects which provides functionalities to store and retrieve objects with the help of ORM. One of the core projects is the Hibernate ORM[38] library implementing the ORM. The library provides its own, native API together with an SQL like query language which is called Hibernate Query Language (HQL) to store and retrieve data. In addition, Hibernate provides a JSR 338 (JPA 2.1) implementation which is called Hibernate EntityManager. Both ORM solutions integrate the JSR 303 (Bean Validation 1.0) to enable automatic data validation (see section 10.1.2). According to the project website, all features of the JSR 338 (JPA 2.1) are implemented with the current version except for the entity graph. An additional feature overview followed by a detailed description are given below.

ADDITIONAL FEATURE OVERVIEW

Every feature of the JSR 338 (JPA 2.1), see section 12.4.1, except the entity graph is supported by Hibernate ORM (EntityManager). In addition, Hibernate ORM offers some additional features which are listed below.

- **JPQL:** HQL as an alternative query language which can be mixed with JPQL but breaks portability
- **ORM extensions:** None
- **Suppported relational DBMS (certified by Hibernate):** Oracle Database, MySQL, PostgreSQL, IBM DB2, Microsoft SQL Server, Sybase Adaptive Server Enterprise (ASE)
- **Suppported relational DBMS (tested by the community):** IBM Informix, Apache Derby, Firebird, PointBase Embedded, Progress 9, Mckoi SQL, Ingres, FrontBase
- **Suppported relational DBMS (requires a dialect from Hongxin Technology and Trade Ltd. of Xiangtan City (HXTT)):** Microsoft Access, Corel Paradox, flat text, CSV file, TSV file, fixed-length and variable-length binary file, Xbase databases, Microsoft Excel, Cobol data files, XML 1.0 and XML 1.1
- **Suppported relational DBMS (requires dialect from HP):** HP NonStop SQL/MX 2.0
- **Supported NoSQL DBs:** Hibernate ORM supports only relational DBMS; Hibernate Object/Grid Mapper (OGM) (see section 12.4.3.1) supports NoSQL data stores

[38] `http://www.hibernate.org/`, [Online; accessed 10-June-2013]

- **Miscellaneous:** Full text search module (Hibernate Search), API to analyse historical data, NHibernate (Hibernate for .NET)
- **Documentation:** Good and detailed library description, examples, Java™ API, forum
- **Licence:** LGPL
- **Latest release:** Version 4.3.0.Beta3 (29/05/2013)[39]

DETAILED DESCRIPTION

Using Hibernate ORM and its JSR 338 (JPA 2.1) implementation works almost identical compared to the reference implementation (see section 12.4.1.1) regarding Bean Validation. Furthermore, the native API of Hibernate ORM and the HQL works similar to the API of JPA 2.1. It does support Bean Validation like the JSR 338 (JPA 2.1) defines. Therefore, we do only consider the Bean Validation mechanism regarding the JSR 338 (JPA 2.1), because it works similar for the native API.
Table 12.2 shows the automatic validation behaviour for the different life cycle events using Hibernate ORM. Hibernate ORM is not in line with the JSR 338 (JPA 2.1) because it does not always validate if a pre-persist or pre-remove event occurs (i.e. method `persist` or `remove`). The behaviour depends on the definition of the entity's primary key, denoted with `@Id`. The `@Id` is mapped to the primary key in the data store and can be created in different ways. If the generation type `AUTO` or `IDENTITY` is used, then the validation happens for pre-persist events. If `SEQUENCE` or `TABLE` is used, then no validation happens for pre-persist events. The behaviour for pre-remove events is not influenced by these settings. This behaviour is also reported as a bug[40], but has still the status 'open'[41]. The implementation always validates in case of a pre-update event (i.e. method `flush`), so no invalid data is written to the data store because at commit time, the `flush` method is always called if necessary. All the other behaviour regarding Bean Validation is identical to EclipseLink (see section 12.4.1.1).
Hibernate ORM does not support any NoSQL solutions. It is also possible that the library throws a `RollbackException` instead of a `ConstraintViolationException` if a validation violation occurs at commit time.

EXAMPLE

The example uses the same domain model and JUnit test cases as in section 12.4.1.1 without the example for the NoSQL solution because Hibernate ORM does not support this kind of data store. The test cases has been successfully tested with the following configurations:

- **Bean Validation implementation:** Hibernate Validator 5.0.1 Final, Apache BVal 0.5
- **Data stores:** MySQL Java™ driver 5.1.26

[39] Checked on 10/06/2013
[40] https://hibernate.atlassian.net/browse/HHH-8028
[41] Checked on 02/07/2013

- **JPA 2.1 implementation:** Hibernate ORM 4.3.0.Beta2

It is important to set up the `persistence.xml` file with the Hibernate ORM specific properties. The relevant properties are depicted in listing 12.18. The other configurations are identical to the one using EclipseLink (see section 12.4.1.1).

Listing 12.18: Property elements for the Hibernate ORM persistence file using MySQL.

```
<property name="hibernate.dialect"
  value="org.hibernate.dialect.MySQLDialect" />
<property name="hibernate.show_sql"
  value="true"/>
<property name="hibernate.hbm2ddl.auto"
  value="create"/>
```

The full example is available in the appendix section A.2.2.1.

12.4.1.3 DATANUCLEUS

The main product of DataNucleus[42] is a persistence and retrieval framework which is called DataNucleus Access Platform. The library implements the JPA, Java™ Data Objects (JDO) and the Representational State Transfer (REST) API for data management. The purpose of JDO which is defined in the JSR 243 [41] is similar to the one of JPA (see section 12.4.2 and 12.4.1), but they do not specify any data validation mechanisms like Bean Validation (see section 10.1.2 and 10.1.1) integration. The library supports the JPA 2 standard and hence Bean Validation 1.0 according to JSR 303 is integrated. Furthermore, the DataNucleus Access Platform provides a rich product line-up of supported data stores (relational and NoSQL).
The following sections give a feature overview, a detailed description and an example.

FEATURE OVERVIEW

Every feature of the JSR 317 (JPA 2.0), a predecessor of the JSR 338 (JPA 2.1), is supported by the DataNucleus Access Platform. The JSR 317 (JPA 2.0) is explained in section 12.4.2). In addition and up to our knowledge, the DataNucleus Access Platform supports every feature of the JSR 338 (JPA 2.1), see section 12.4.1, except the `TREAT` function using JPQL. Although, the project does not provide a complete JSR 338 (JPA 2.1) implementation we categorise this implementation into this section because only some minor features are missing and the relevant features (e.g. Bean Validation integration) are supported. The following list gives an overview about the additional features provided by the DataNucleus Access Platform and the supported data stores.

- **JPQL:** None
- **ORM extensions:** Additional DataNucleus specific annotations (e.g. `@JdbcType`)
- **Suppported relational DBMS:** Oracle Database, MySQL, MariaDB, PostgreSQL, PostGIS, IBM DB2, HSQLDB, SQLite, Microsoft SQL Server, Sybase SQL Anywhere, IBM Informix, Apache Derby, Firebird, H2 Database Engine, HSQL, SAP MaxDB, SAPDB, Virtuoso, Pointbase, Oracle TimesTen, Mckoi SQL Database

[42] `http://www.datanucleus.org/`, [Online; accessed 18-June-2013]

- **Supported NoSQL DBs:** Microsoft Excel files (`*.xls`, `*.xlsx` (OOXML)), Open Document Format (ODF) files, XML, HBase, MongoDB, Neo4j, JavaScript Object Notation (JSON), Amazon S3, GoogleStorage, Lightweight Directory Access Protocol (LDAP), NeoDatis
- **Supported NoSQL DBs (third party):** Google BigTable (with DataNucleus Access Platform 3.0), Cassandra, OrientDB and VMForce (with DataNucleus Access Platform 2.x)
- **Miscellaneous:** Plugin mechanism for extensions, Eclipse plugin, multi-database integration
- **Documentation:** Very good and detailed library description, a lot of examples, Java™ API, forum
- **Licence:** Apache Licence Version 2.0
- **Latest release:** Version 3.3.1 (09/06/2013)[43]

DETAILED DESCRIPTION

As mentioned above the project website does not specify which JSR is implemented. According to some research in the issue tracking system of the project, there are some features of the JSR 338 (JPA 2.1) that are not implemented but the JSR 317 (JPA 2.0) is fully implemented. Therefore, DataNucleus Access Platform 3.2.3 integrates Bean Validation 1.0 according to JSR 303 (see section 10.1.2) and works similar to EclipseLink (see 12.4.1.1) and Hibernate ORM/OGM (see section 12.4.1.2 and 12.4.3.1). The library requires an enhancement step which enriches the Java™ classes via byte-code injection. This step has to be started manually, but DataNucleus provides several tools to facilitate this step (e.g. an Eclipse plugin which automatically enriches the classes).
Table 12.3 shows the automatic validation behaviour for the different life cycle events using DataNucleus Access Platform. Validation is done on every event type also for those scenarios in which the JSR 317 (JPA 2.0) does not force the implementation to apply validation (compare with the reference implementation EclipseLink in section 12.4.1.1). Therefore, no invalid data can be stored in the underlying data store. Furthermore, if a validation violation occurs, a `ConstraintViolationException` will be thrown in any case independent of the event type. The `ConstraintViolationException` message contains the life cycle event type that caused the validation plus the validation groups besides the original violations.
The DataNucleus Access Platform does not conform to the standard regarding object graph validation. It is not possible to validate properties annotated with `@Embedded` in a recursive way using the `@Valid` annotation. The specification mentions that this validation mechanism should be possible. DataNucleus Access Platform simply ignores the `@Valid` annotation. On the other hand, DataNucleus Access Platform does validate entity references separately but stops the validation process if an error occurred for a basic type (single-valued property).

[43] Checked on 18/06/2013

The DataNucleus Access Platform does support a lot of NoSQL solutions. According to the project website it is possible to adapt the library to integrate a NoSQL solution which is not present, but a correct and complete integration could take several weeks according to the project website[44]. Validation violations work as described in the standard.

EXAMPLE

The example is identical to the one presented in section 12.4.1.1 using DataNucleus Access Platform except for the underlying data stores. It is possible that not every annotation is supported by every data store. For instance, the Neo4J data store does not support embedded properties. One test case has failed using the `@Valid` annotation because it is not considered during validation as described above. All the test cases have been checked with the following configurations:

- **Bean Validation implementation:** Hibernate Validator 5.0.1 Final, Apache BVal 0.5
- **Data stores:** MySQL Java™ driver 5.1.26, MongoDB Java™ driver 2.11.2
- **JPA 2.0/2.1 implementation:** DataNucleus Access Platform Core 3.2.4, DataNucleus Access Platform relational DBMS 3.2.3, DataNucleus Access Platform API JPA 3.2.3

Note, the test cases have presented the same result regarding MongoDB and MySQL. Using a relational DBMS it is possible to create and drop the schema automatically using a property element with name equal to `"datanucleus.generateSchema.database.mode"` and value set to `"drop-and-create"` within the `persistence.xml` file (not possible for MongoDB). The complete example (test cases, domain model, configuration files) can be found in the appendix section A.2.2.1.

12.4.2 JSR 317: JAVA™ PERSISTENCE API

The JSR 317 (JPA 2.0) [42] is the predecessor of the JSR 338 (JPA 2.0), see section 12.4.1, which means that some features were introduced with JPA version 2.1 and therefore are not applicable in implementations using JPA 2.0. The main features that are not applicable in JPA 2.0 are the missing schema generation, type conversion methods, the use of entity graphs in queries and the stored procedure invocation. The basic features have been already defined in the JSR 317 and a comprehensive list of changes made in the JSR 338 (JPA 2.1) compared to the JSR 317 can be found in the specification of the JSR 338 (see section 12.4.1)
A feature overview and a detailed description are given below.

FEATURE OVERVIEW

The main features of the JSR 317 are the same as listed in section 12.4.1 with some exceptions mentioned in [40] chapter 1 and appendix A.

[44] http://www.datanucleus.org/servlet/wiki/display/ENG/HOWTO+Support+a+new+datastore

DETAILED DESCRIPTION

The JSR 317 includes a detailed description of every feature which can be found on the website of the JCP [42].

12.4.2.1 APACHE OPENJPA

Apache OpenJPA 2.2.0[45] is an implementation of the JPA 2.0 according to the JSR 317 (see section 12.4.2) and passes the TCK JPA 1.0b which means that it is backwards compatible with JPA 1.0. According to the project website it passes the TCK JPA 2.0 which means that all features described in the specification are supported by Apache OpenJPA. Especially, OpenJPA supports Bean Validation 1.0 according to JSR 303 (see section 10.1.2) for the supported relational DBMS. The project does not offer a NoSQL data store for persistence, but mentions the possibility to adapt the library for those data stores.
The following sections give a feature overview, a detailed description and an example.

ADDITIONAL FEATURE OVERVIEW

Every feature of the JSR 317 (JPA 2.0), see section 12.4.2, is supported by Apache OpenJPA. In addition, the library implements some additional features which are depicted below.

- **JPQL:** MethodQL is an extension to formulate a query with the help of a static Java method
- **ORM extensions:** None
- **Suppported relational DBMS (verified against the latest version):** Apache Derby, IBM DB2, IBM Informix Dynamic Server, Microsoft SQL Server, MySQL, Oracle Database, PostgreSQL, IBM solidDB, Sybase Adaptive Server Enterprise
- **Suppported relational DBMS (compatible with an older version):** Borland Interbase, Borland JDataStore, Empress, Firebird, H2 Database Engine, Hypersonic Database Engine, Ingres Database, InterSystems Cache, Microsoft Access, Microsoft Visual FoxPro, Pointbasem
- **Suppported relational DBMS (reported by the community):** SAP MaxDB
- **Miscellaneous:** Entity enhancement tool, reverse mapping tool, tool to create/synchronise domain model to a DB schema, runtime extensions
- **Documentation:** Very detailed with JPA description, running examples, Java™ API, forum
- **Licence:** Apache Licence Version 2.0
- **Latest release:** Version 2.2.2 (22/04/2013)[46]

[45] `http://openjpa.apache.org/`, [Online; accessed 15-June-2013]
[46] Checked on 15/06/2013

DETAILED DESCRIPTION

Apache OpenJPA 2.2.2 works as any implementation of the JSR 338 (JPA 2.1) just without the newly defined features. The library uses an enhancer to automatically add code as a default setting to optimise the runtime performance. This enhancement must be started manually compared to EclipseLink and Hibernate ORM/OGM which transparently enhance the classes. It is possible to skip the manually configuration by adding a property element in the `persistence.xml` file which is depicted in listing 12.19. Like with Hibernate ORM and EclipseLink, the library provides the possibility to create the schema automatically by adding another custom property element (see listing 12.19).
The automatic validation behaviour for the different life cycle events using Apache OpenJPA is identical to the behaviour of the reference implementation EclipseLink as described in 12.4.1.1. Unfortunately, Apache OpenJPA does not display the life cycle event type if a violation occurs. Therefore, the life cycle event type has to be figured out manually (e.g. with a debugger). Furthermore, the life cycle event types are named differently in Apache OpenJPA and mapped when needed. The following list shows the relevant life cycle event types and their mapping:

- Pre-persist ⇔ Before-persist ⇔ integer constant `0`
- Pre-update ⇔ Before-update ⇔ integer constant `2 0`
- Pre-remove ⇔ Before-remove ⇔ integer constant `7`

The full internal life cycle event type list is available in the `org.apache.openjpa.event.LifecycleEvent` class of the Apache OpenJPA implementation.
The behaviour while validating an object graph is different when using Apache OpenJPA than for EclipseLink or Hibernate ORM/OGM. The specification says that the `@Valid` constraint is only applicable for embeddables and if not present no validation occurs for the embeddable. Apache OpenJPA does not conform to that specification, because the annotation can also be used for entity references to achieve a proper exception behaviour. If an entity reference is checked without a `@Valid` constraint and a validation violation occurs the implementation will throw a `PersistenceException` instead of a `ConstraintViolationException`. Annotating an entity reference with `@Valid` changes the behaviour of Apache OpenJPA by now throwing a `ConstraintViolationException`. Furthermore, it does not stop validation for an entity reference if a validation violation occurs for a single-valued property and the entity reference is annotated with `@Valid`.
Apache OpenJPA does not support NoSQL solutions. According to the project website it is possible to adapt the library to integrate a NoSQL solution by extending the `AbstractStoreManager`.

EXAMPLE

The example uses the same domain model and JUnit test cases as in section 12.4.1.1 without the example for the NoSQL solution because Apache OpenJPA does not provide any implementation for this kind of data store. One test

case has failed because a `PersistenceException` is thrown instead of a `ConstraintViolationException` as described above. Every other test case has been successfully tested with the following configurations:

- **Bean Validation implementation:** Hibernate Validator 5.0.1 Final, Apache BVal 0.5
- **Data stores:** MySQL Java™ driver 5.1.26
- **JPA 2.0 implementation:** Apache OpenJPA 2.2.2

To set up the DB schema automatically and disable the enhancement step it is possible to add the properties shown in listing 12.19 to the `persistence.xml` file.

Listing 12.19: Property elements for the Apache OpenJPA persistence file using a relational DBMS.

```
<!-- This property disables the need for the manual enhancement.
 It enriches the classes with another technique which is not
 that efficient. -->
<property name="openjpa.RuntimeUnenhancedClasses"
  value="supported" />

<!-- This property automatically creates the schema in the DB. -->
<property name="openjpa.jdbc.SynchronizeMappings"
  value="buildSchema(ForeignKeys=true,schemaAction='dropDB,add')" />
```

The complete runnable example is available in the appendix section A.2.2.2.

12.4.2.2 BATOO JPA

We describe Batoo JPA[47] not only due to its full JSR 317 (JPA 2.1) implementation but also because it claims that 'it is 10 to 20 times faster than the leading JPA providers'[48]. Although this implementation has not yet passed the TCK we put it into this section because it fulfils Bean Validation 1.0 according to JSR 303 (see section 10.1.2) which is most important for this thesis. The project does not support NoSQL data stores and the L2 Cache implementation. The following sections give a feature overview, a detailed description and an example.

ADDITIONAL FEATURE OVERVIEW

Every feature of the JSR 317 (JPA 2.0), see section 12.4.2, is supported by Batoo JPA except for the L2 Cache. The following list summarises the most important points.

- **JPQL:** No additional features
- **ORM extensions:** None
- **Suppported relational DBMS (tested by Batoo JPA):** Apache Derby, HSQLDB, H2, Microsoft SQL Server, MySQL, Oracle Database, PostgreSQL, Sybase SQL Anywhere

[47] `http://batoo.org/`, [Online; accessed 04-September-2013]
[48] `https://github.com/BatooOrg/BatooJPA`

- **Miscellaneous:** Benchmark sources, Java Transaction API (JTA) support, connection pool, prepared statement cache
- **Documentation:** Tiny configuration description, link to official JPA description
- **Licence:** GNU Lesser GPL
- **Latest release:** Version 2.0.1.2 (28/08/2013)[49]

DETAILED DESCRIPTION

Compared to Apache OpenJPA (see section 12.4.2.1), Batoo JPA does not mention any enhancement steps and therefore we think it either does it automatically or it is not needed. Furthermore, the automatic validation does not work as described in the standard (see section 12.4.2) because one has to explicitly set the validation mode to `CALLBACK` to activate validation. The standard stipulates that the validation mode is set to `AUTO` which means that the JPA implementation should automatically detect if a valid JSR 303 (Bean Validation 1.0) implementation is available and if so, it should enable automatic validation.
The behaviour regarding the automatic validation behaviour shows that Batoo JPA does only but always validate when a flush to the data store happens (i.e. a calling the `flush()` method or at commit time) which is not correct according to the specification. Therefore, it is for instance possible to persist a valid object, change it to an invalid state and then remove it within the same transaction without a validation violation.
The behaviour of the `@Valid` constraint is handled as described in the standard and Batoo JPA does not stop validation of entity references if an error occurred for a single-valued property. Moreover, some tests showed that Batoo JPA has some serious bugs regarding the key generation. A workaround is possible by setting the primary keys manually. We put this JSR 317 (JPA 2.0) implementation nonetheless into this category because the implementation works in general. Source code examples are depicted in the appendix section A.2.2.2.

12.4.3 NON-STANDARD JPA PROVIDERS

In this section, we present several JPA providers which do not fulfil the JSR 317/338 (JPA 2.0/2.1). Some providers mention that their implementation is still in progress while others claim that they are offering an implementation that is fully compliant to the standard, but we think it is not. One of the reasons why we think some providers are not compliant with the standard is the absence of the JSR 303/349 (Bean Validation 1.0/1.1) integration. We mention those JPA providers, because they either do support Bean Validation but miss some other important JPA features or they support relational and NoSQL data stores without Bean Validation. In both cases, we think it is worthwhile to observe the development. The following sections give an overview of some non-standard JPA providers.

12.4.3.1 HIBERNATE OGM

Hibernate OGM[50] is a project within the Hibernate project collection. It has the goal to provide a JPA 2.1 according to JSR 338 (see section 12.4.1) for NoSQL data stores. The

[49] Checked on 04/09/2013
[50] `http://www.hibernate.org/subprojects/ogm.html`, [Online; accessed 14-June-2013]

project is still in an early development phase and therefore it does only support a subset of the JSR 338. The road map of the project mentions the proper implementation of all annotations that are used in a relational environment to the corresponding concept in a NoSQL technology, the complete support of the JPQL and the extension of supported data stores. In addition, the native Hibernate API which is called the Hibernate Session API can be used. The documentation of Hibernate OGM does not describe whether Bean Validation is already integrated as defined in the JSR 338 (JPA 2.1) but some tests showed that it is already implemented and works identical to Hibernate ORM (see section 12.4.1.2).
The following section shows the feature spectrum which the project provides at the moment. This overview is followed by a detailed description and an example.

FEATURE OVERVIEW

The following list shows the features that are supported by Hibernate OGM. Not all the features defined in the JSR 338 are supported.

- **JPQL:** Does partially work with Hibernate Session API (JPA not supported yet) and a limited set of operators (e.g. the `find` method)
- **ORM extensions:** None.
- **Supported ORM features:** CRUD operations for entities, polymorphic entities, embeddable entities, associations, collections, basic types
- **Suppported NoSQL DBs:** Infinispan, Ehcache, MongoDB
- **Miscellaneous:** Full text search module (Hibernate Search) is supported, JPA and native Hibernate ORM API are supported (for the implemented features)
- **Documentation:** Good and detailed library description, small examples, Java™ API
- **Licence:** LGPL
- **Latest release:** Version 4.0.0.Beta2 (14/01/2013)[51]

DETAILED DESCRIPTION

The Hibernate OGM project does not fulfil the JSR 338 (JPA 2.1) because the project has just started. Nevertheless, the features that are already available can be used as in the case of using a relational DB and Hibernate ORM (see section 12.4.1.2). Currently, it is possible to create, read, update and delete entities using one of the supported NoSQL solutions as a data store. The general idea is to reuse components of Hibernate ORM which delegates the CRUD operations to a specific data store provider component (e.g. one for MongoDB) and ships queries to a special query engine. The components then directly interact with the underlying NoSQL data store. The specific data store provider is a library which must be included in the runtime environment. Figure 12.14 shows an architecture overview of Hibernate OGM from the project website. Hibernate OGM tries to achieve that

[51] Checked on 13/06/2013

every annotation of Hibernate ORM can be used and that they are appropriately mapped to the underlying NoSQL data store concept. For example, Hibernate OGM takes care of the `_id` field which is mandatory for MongoDB and automatically maps the property annotated with `@Id` to this field.

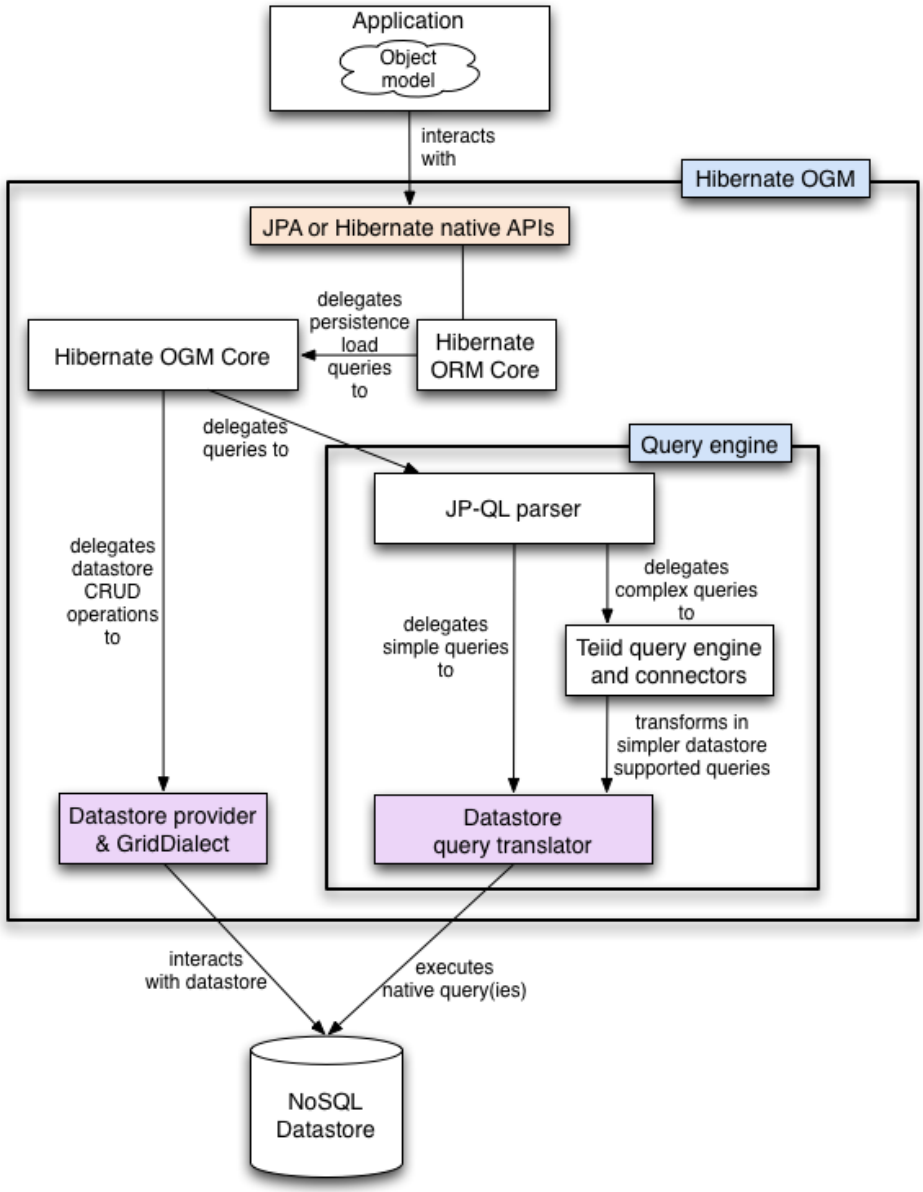


Figure 12.14: Hibernate OGM architecture as presented on their project website[52].

Reading data with the help of JPQL is not possible with the current version of Hibernate OGM. According to the project website it is the short term target to implement simple JPQL queries. At the moment, it is possible to formulate some simple queries (no join or aggregation) using the native Hibernate ORM API (Hibernate Session API). Furthermore, it is

[52] See footnote 50

possible to use Hibernate Search, which is a full text search engine, or the `find` method of the entity manager to retrieve data.
Regarding Bean Validation, the Hibernate OGM project does not mention any support on their project website, but all the test cases defined in the appendix section A.2.2.3 were successfully tested with the supported data stores. We assume that the library integrates Bean Validation 1.0 according to JSR 303 (see section 10.1.2). The data validation delegation is identical to Hibernate ORM (which is described in section 12.4.1.2). As Hibernate ORM, Hibernate OGM is not in line with the JSR 338 (JPA 2.1) but is different in the behaviour than Hibernate ORM. The exception and event type do not depend on the generation type of the entity (see Hibernate ORM description, section 12.4.1.2), but Hibernate OGM does only validate during the pre-update event which is not correct according to the JSR 338 (JPA 2.1). Furthermore, validation violations in entity references do not prohibit to store a valid entity referencing the invalid entity in the underlying data store (e.g. a valid person is stored but has an invalid address which is not stored). The reason for this behaviour could be the missing (or wrong) transaction management of Hibernate OGM when rolling back a transaction using MongoDB ('But do not consider rollback as a possibility, this won't work.'[53]). Table 12.4 summarises the behaviour regarding automatic Bean Validation.

EXAMPLE

The example is identical to the one presented in section 12.4.1.2 using Hibernate ORM except for the underlying data stores. The test cases have been tested with the following configurations:

- **Bean Validation implementation:** Hibernate Validator 5.0.1 Final, Apache BVal 0.5
- **Data stores:** MongoDB Java™ driver 2.11.2, Infinispan 5.1.7, Ehcache 2.5.1
- **JPA 2.1 implementation:** Hibernate OGM 4.3.0.Beta2

Every test case except one has been successful. The test case regarding the invalid entity reference (an address without a street) does not work as expected. The person object referencing the invalid address should not be stored in the underlying data store but using Hibernate OGM and the data stores mentioned above store the person object without the address.
It is important to set up the `persistence.xml` file with the Hibernate OGM specific properties and its provider. The relevant elements are depicted in listing 12.20 and the full source code is available in the appendix section A.2.2.3.

Listing 12.20: Property elements for the Hibernate OGM persistence file using MongoDB, Infinispan or Ehcache.

```
<provider>org.hibernate.ogm.jpa.HibernateOgmPersistence</provider>

<property name="hibernate.ogm.datastore.provider"
  value="org.hibernate.ogm.datastore.
  mongodb.impl.MongoDBDatastoreProvider" />
```

[53] See footnote 50

```
<property name="hibernate.ogm.datastore.provider"
  value="org.hibernate.ogm.datastore.
  infinispan.impl.InfinispanDatastoreProvider"/>

<property name="hibernate.ogm.datastore.provider"
  value="org.hibernate.ogm.datastore.
  ehcache.impl.EhcacheDatastoreProvider"/>
```

12.4.3.2 VERSANT JPA

The Versant Cooperation[54] provides several NoSQL data stores like db4o[55]. Among their portfolio the Versant JPA[56] provides a NoSQL data store using the JPA 2.0 (JSR 317). In a blog post presenting the Versant JPA implementation which was published on 25/10/2012 they claim the following:

'You can now break free of relational database engines and proprietary NoSQL solutions when architecting Java applications while still leveraging your JPA skills (Versant JPA includes a fully compliant JPA 2.0 implementation).'[57]

Testing the implementation regarding Bean Validation 1.0/1.1 support (i.e. JSR 303/349 integration) showed that the constraints are ignored and the Bean Validation implementation which was applicable in the runtime environment was not used. In addition, we asked whether Bean Validation is supported in the customer forum.[58] A user answered that Versant JPA does not support Bean Validation which reinforces our suspicion that the standard is not met.

FEATURE OVERVIEW

The following list shows the features that are supported by Versant JPA.

- **JPQL:** Fully supported with additional cursor queries
- **ORM extensions:** Index definition
- **Supported data stores:** Built-in object-oriented database of Versant
- **Miscellaneous:** Multiple database support, runtime enhancement of entity classes, automatic schema evolution, R-Integration, Hadoop Connector
- **Documentation:** Not well documented, only installation guides; referring to the JPA standard
- **Licence:** Commercial

[54] http://actian.com/products/versant, [Online; accessed 02-September-2013]
[55] http://www.db4o.com/, [Online; accessed 02-September-2013]
[56] http://community.versant.com/Home.aspx, [Online; accessed 02-September-2013]
[57] http://community.versant.com/Blogs/VOD/tabid/216/entryid/1075/Default.aspx
[58] http://community.versant.com/Forums/tabid/98/aft/12240/Default.aspx

- **Latest release:** Version 1.0.9.1904 (Server), Version 1.0.10.2007 (Java Development Kit (JDK)) (14/08/2013)[59]

12.4.3.3 OBJECTDB

ObjectDB is an object-oriented database developed by ObjectDB Software[60]. One of the main advantages of ObjectDB is that they rely on the standard API defined in JSR 317/338 (JPA 2.0/2.1) to provide the CRUD operations. Therefore, a developer does not need to learn another (proprietary) API to use ObjectDB. One the main page of their website, they claim that JPA 2 is supported, but a closer look at the features[61] gives the following result:

'Most features of JPA 2 are supported. Support of remaining features is in progress - see issue tracking (requires login).'

One can find among those remaining issues the support for Bean Validation[62] which makes ObjectDB not fully compliant with JSR 317/338 (JPA 2.0/2.1). A user asked for Bean Validation support in 2011 in the support forum, but the team answered that this feature has still not been implemented. We replied to ask whether Bean Validation is on the roadmap and the answer in 2012 is that they have it in their plans but do not know when it will be implemented.

FEATURE OVERVIEW

The following list shows the features that are supported by ObjectDB.

- **JPQL:** No subqueries, case statements and maps
- **ORM extensions:** None.
- **Supported data stores:** Built-in object-oriented database ObjectDB (i.e. ObjectDB uses JPA as its CRUD API)
- **Miscellaneous:** Client-Server mode, no embedded database mode, JDO support, additional database tools
- **Documentation:** Detailed documentation about installation and JPA usage
- **Licence:** Commercial
- **Latest release:** Version 2.5.2_05 (29/08/2013)[63]

[59] Checked on 02/09/2013
[60] `http://www.objectdb.com/`, [Online; accessed 02-September-2013]
[61] http://www.objectdb.com/object/db/database/features
[62] http://www.objectdb.com/database/issue/14
[63] Checked on 02/09/2013

12.4.3.4 KUNDERA

Usually, every NoSQL data store has its own API for the CRUD operations. Kundera[64] is a JPA provider which has the main goal to make the CRUD operations of NoSQL data stores as easy as for relational data stores. Therefore, the implementation offers several NoSQL data stores (and relational data stores) which can be used with JPA 2.0. Unfortunately, Kundera does not integrate Bean Validation although they mention on the project website[65]:

'A JPA 2.0 compliant Object-Datastore Mapping Library for NoSQL Datastores.'

In addition, we commented on the issue request[66] about integrating Bean Validation, but we have not received an answer whether it is on the roadmap or not. Therefore, we think Kundera does not fulfil the JSR 317/338 (JPA 2.0/2.1).

FEATURE OVERVIEW

The following list shows the features that are supported by Kundera.

- **JPQL:** Most features of the JPQL are supported
- **ORM extensions:** None.
- **Supported data stores:** Cassandra, MongoDB, HBase, Redis, OracleNoSQL, Neo4j, relational databases
- **Miscellaneous:** Horizontal scaling is possible (i.e. adding another data store), Polyglot persistence
- **Documentation:** Good documentation with code examples
- **Licence:** Apache Licence Version 2.0
- **Latest release:** Version 2.6 (06/07/2013)[67]

[64] `https://github.com/impetus-opensource/Kundera`, [Online; accessed 02-September-2013]
[65] `https://github.com/impetus-opensource/Kundera`
[66] `https://github.com/impetus-opensource/Kundera/issues/208`
[67] Checked on 02/09/2013

	Behaviour		
	Event type	**Exception type**	**According to JSR?**

`begin`
`persist(null)`
`commit`
	Pre-persist	Constraint Violation	Yes

`begin`
`persist`
`... = null`
`commit`
	No event	None	Yes

`begin`
`persist`
`... = null`
`flush`
`commit`
	Pre-update	Rollback	Yes

`begin`
`persist`
`... = null`
`flush`
`flush`
`commit`
	Pre-update	Constraint Violation	Yes

`begin`
`persist`
`commit`
`... = null`
`begin`
`flush`
`commit`
	Pre-update	Constraint Violation	Yes

`begin`
`persist`
`... = null`
`remove`
`commit`
	Pre-remove	Constraint Violation	Yes

`begin`
`persist`
`commit`
`... = null`
`begin`
`remove`
`commit`
	Pre-remove	Constraint Violation	Yes

Table 12.1: Behaviour of EclipseLink regarding the automatic Bean Validation within the life cycle event types.

	Behaviour		
	Event type	**Exception type**	**According to JSR?**

`begin`
`persist(null)`
`commit`
	Pre-persist	Constraint Violation or Rollback	No

`begin`
`persist`
`... = null`
`commit`
	Pre-persist or pre-update	Rollback	No

`begin`
`persist`
`... = null`
`flush`
`commit`
	Pre-persist or Pre-update	Constraint Violation	Yes

`begin`
`persist`
`... = null`
`flush`
`flush`
`commit`
	Pre-persist or pre-update	Constraint Violation	Yes

`begin`
`persist`
`commit`
`... = null`
`begin`
`flush`
`commit`
	Pre-update	Constraint Violation	Yes

`begin`
`persist`
`... = null`
`remove`
`commit`
	Pre-persist or pre-remove	Rollback	No

`begin`
`persist`
`commit`
`... = null`
`begin`
`remove`
`commit`
	Pre-remove	Rollback	No

Table 12.2: Behaviour of Hibernate ORM regarding the automatic Bean Validation within the life cycle event types. The exception and event type depends on the definition of the entity.

	Behaviour		
	Event type	**Exception type**	**According to JSR?**

`begin`
`persist(null)`
`commit`
	Pre-persist	Constraint Violation	Yes

`begin`
`persist`
`... = null`
`commit`
	Pre-update	Constraint Violation	Yes

`begin`
`persist`
`... = null`
`flush`
`commit`
	Pre-update	Constraint Violation	Yes

`begin`
`persist`
`... = null`
`flush`
`flush`
`commit`
	Pre-update	Constraint Violation	Yes

`begin`
`persist`
`commit`
`... = null`
`begin`
`flush`
`commit`
	Pre-update	Constraint Violation	Yes

`begin`
`persist`
`... = null`
`remove`
`commit`
	Pre-remove	Constraint Violation	Yes

`begin`
`persist`
`commit`
`... = null`
`begin`
`remove`
`commit`
	Pre-remove	Constraint Violation	Yes

Table 12.3: Behaviour of DataNucleus Access Platform regarding the automatic Bean Validation within the life cycle event types.

	Behaviour		
	Event type	**Exception type**	**According to JSR?**

`begin`
`persist(null)`
`commit`
	Pre-persist	Rollback	No

`begin`
`persist`
`... = null`
`commit`
	Pre-persist	Rollback	No

`begin`
`persist`
`... = null`
`flush`
`commit`
	Pre-persist	Constraint Violation	Yes

`begin`
`persist`
`... = null`
`flush`
`flush`
`commit`
	Pre-persist	Constraint Violation	Yes

`begin`
`persist`
`commit`
`... = null`
`begin`
`flush`
`commit`
	Pre-update	Constraint Violation	Yes

`begin`
`persist`
`... = null`
`remove`
`commit`
	Pre-persist	Rollback	No

`begin`
`persist`
`commit`
`... = null`
`begin`
`remove`
`commit`
	Pre-remove	Rollback	No

Table 12.4: Behaviour of Hibernate OGM regarding the automatic Bean Validation within the life cycle event types.

13
DATA TIER VALIDATION

The data tier comprises all technologies which provide a way that data is stored in a non-volatile way such as relational databases or NoSQL data stores. It processes requests from the logic tier and responses to the logic tier with some data. This kind of validation depends on the data store provider. Relational DBMS usually support the integrity constraints defined in the SQL standard (see section 9) with some provider specific extensions. In general, there is no standard which defines validation mechanisms or constraint specification possibilities for NoSQL data stores.

14 TECHNOLOGY OVERVIEW

This chapter summarises the investigated technologies with respect to their provided features, their benefits and drawbacks.
Table 14.1 shows each technology and some selected characteristics which were important within this master thesis. The first two rows mention the tier and layer affiliation of the technology followed by an information whether it is standardised (e.g. a JSR specification exists) or not. The row denoted as 'Built-in validation?' expresses the existence of a validation possible which is integrated into the technology and different from Bean Validation 1.0/1.1 (see section 10.1). If such a feature exist, the characteristics of the following three rows make sense: Does the validation mechanism support the definition of custom constraints besides some built-in functionality? How are constraints (built-in and custom) implemented and how are they used? Finally, what is the purpose of the data validation? The row denoted as 'Bean Validation integration?' gives information about the integration status of Bean Validation with respect to the considered technology. The last two rows specify the quality of the existing documentation and the refresh period which expresses how common the considered technology nowadays is.
We do not include the cross-tier validation technology 'Bean Validation' (see section 10.1) as a separate column in this comparison because it is not coupled to a specific technology and tier/layer and therefore could be combined with any technology mentioned in table 14.1. Hence, we provide separate tables for a Bean Validation integration comparison with other technologies. A Bean Validation integration is always possible by manually invoking the validator but we try to focus on automatic validation mechanisms in the comparison. Therefore, we do only compare such technologies which provide some way of automatic Bean Validation mechanisms.
For completeness, we mention the individual characteristics (see rows of table 14.1): Bean Validation according to JSR 303/349 is a cross-tier validation technology with built-in validation mechanisms and the possibility to add custom constraints. Constraints are implemented with annotation interfaces and validator classes and can be applied to a domain model using Java™ annotations or an XML configuration file. The documentation quality is very good

and the community is currently[1] working on a new release.
Table 14.2 shows an overview of the considered JSR within this master thesis and the corresponding implementations (reference implementation and alternative providers).

	Technology							
	HTML5	JSF	GWT	JFC: Swing	SWT	JFace	JavaFX	JPA
Tier	Present.	Logic	Logic	Logic	Logic	Logic	Logic	Logic
Layer	-	Present.	Present.	Present.	Present.	Present.	Present.	Data access
Standard?	Yes	Yes	No	No	No	No	No	Yes
Built-in validation?	Yes	Yes	No	No	No	Yes	No	No
Custom constraints	Limited extent	Feasible	-	-	-	Feasible	-	-
Impl. method	API	Validator classes	-	-	-	Validator classes	-	-
Usage	Elements and attributes	Elements and attributes	-	-	-	Method call	-	-
Validation target?	HTML5 forms	HTML forms	-	-	-	Objects	-	-
Bean Validation integr.?	No	Yes	Yes	Partially	Partially	Partially	Partially	Yes
Document. quality	+	++	++	++	+	+	++	++
Refresh period	++	++	++	–	o	o	++	++

Table 14.1: Comparing the analysed technologies with selected characteristics that are relevant within this master thesis. A comparison with respect to Bean Validation is depicted in table 14.3. Assessment scale: of – (very poor) over o (neutral) to ++ (very good).

The tables 14.3 and 14.4 show the benefits and drawbacks of a Bean Validation integration with respect to the considered technologies (columns) and the Bean Validation features (rows). The first table shows an overview about the technologies regarding the presentation layer (see section 12.2) and the second table describes the pros and cons for the data access layer technologies (see section 12.4). The first two rows show what kind of specification is supported followed by a subjective assessment how well the standard is integrated in the considered technology. The row labelled with 'Location' specifies where the actual validation happens with subsequent row about the actual validation target, i.e. what should be validated, what is the purpose. The rows '@Valid?', 'Type?' and 'Group?' show if object graph validation, class-level constraints and group validation are possible (only for table 14.3). The following row specifies the possibility for an XML configuration file instead of Java™ an-

[1] Checked on 05/10/2013

	JSR				
	303	349	314	317	338
Title	Bean Validation	Bean Validation	JavaServer™ Faces (JSF)	Java™ Persistence API (JPA)	Java™ Persistence API (JPA)
Version	1.0	1.1	2.1	2.0	2.1
Reference impl.	Hibernate Validator	Hibernate Validator	Oracle Mojarra JavaServer™ Faces	EclipseLink	EclipseLink
Version	4.3.1.Final	5.0.1.Final	2.1.11	2.4.2	2.5.0
Alternative providers	Apache BVal	-	Apache MyFaces Core	Apache OpenJPA, Batoo JPA	Hibernate ORM, DataNucleus

Table 14.2: Overview of the considered Java™ Specification Request (JSR) within this master thesis.

notations. Table 14.4 shows in the row labelled with 'Exception mechanism', 'Automatic validation', 'Relational DBMS' and 'NoSQL data store' if an exception is thrown for a constraint violation, the validation mechanism is automatically triggered if a Bean Validation implementation is available in the runtime environment, an assessment about the supported relational and NoSQL data stores, respectively. The row 'Custom collection type' specifies the possibilities for an integration of a custom data structure for associations. The presentation layer table 14.3 does not show those information but an assessment about the rendering process of constraint violations and the data binding to the underlying domain model (see rows 'Rendering' and 'Binding', respectively). The last four rows are the same for both tables. They describe the possibility of a custom validation provider integration, the coding overhead regarding the Bean Validation integration, the documentation quality and the activity with respect to validation which expresses the concerns of the technologies regarding Bean Validation integration (top to bottom).
Closing, the studied technologies are a selection of existing Java™ technologies and further analysis regarding constraint definition and data validation is possible. For instance, the web application framework overview of Wikipedia[2] shows different projects and their feature list which includes 'Form validation framework(s)'. The JavaServer™ Faces framework which is discussed in section 12.2.1 is an example of a mentioned project but others seem to be relevant too (e.g. the Play[3] framework which uses also Java™ annotations for defining constraints but it is not clear whether they fully integrated Bean Validation).

[2] `http://en.wikipedia.org/w/index.php?title=Comparison_of_web_application_frameworks&oldid=575790818#Java_2`, [Online; accessed 05-October-2013]
[3] `http://www.playframework.com/`, [Online; accessed 05-October-2013]

	Technology					
	JSF 2.1: MyFaces	JSF 2.1: RichFaces	GWT	Swing Form Builder	JFace	FXForm2
JSR 303?	✓	✓	✓	✓	✓	✓
JSR 349?	✓	✓	×	✓	✓	✓
Integration	+	++	+	o	o	o
Location	Server-side	Server-side and client-side	Server-side and client-side	Stand-alone	Stand-alone	Stand-alone
Target	HTML forms	HTML forms	Model	JFC: Swing forms	JFace forms	JavaFX 2 forms
`@Valid`?	×	✓	✓	×	×	×
Type?	×	✓	✓	×	×	×
Group?	✓ type-safety lost	✓ type-safety lost	✓	×	×	×
XML?	✓	✓	×	✓	✓	✓
Rendering	++	++	–	+	o	+
Binding	++	++	++	+	+	+
Provider?	✓	✓	×	✓	✓	✓
Overhead	minimal	minimal	small	minimal	medium	minimal
Doc.	++	+	++	–	-	–
Activity w.r.t. Validation	o	+	o	–	-	o

Table 14.3: Overview of the technology features regarding Bean Validation and the presentation layer (see section 10.1 and 12.2, respectively). Symbol explanation: ✓ means that the feature is supported and × means it is not supported; Assessment scale: of – (very poor) over o (neutral) to ++ (very good).

	Technology					
	EclipseLink	Hibernate ORM	DataNucleus	Apache OpenJPA	Batoo JPA	Hibernate OGM
JSR 303?	✓	✓	✓	✓	✓	✓
JSR 349?	✓	✓	✓	✓	✓	✓
Integration	++	++	++	++	+	++
Location	Server-side	Server-side	Server-side	Server-side	Server-side	Server-side
Target	Model	Model	Model	Model	Model	Model
XML?	✓	✓	✓	✓	✓	✓
Exception mechanism	✓	✓	✓	✓	✓	✓
Automatic validation	✓	✓	✓	✓	✓	×
Relational DBMS	++	++	++	++	+	–
NoSQL data stores	+	–	++	–	–	+
Custom collection type	++	o	–	–	–	–
Provider?	✓	✓	✓	✓	✓	✓
Overhead	minimal	minimal	minimal	minimal	minimal	minimal
Doc.	++	+	++	–	-	–
Activity w.r.t. Validation	o	+	o	–	-	o

Table 14.4: Overview of the technology features regarding Bean Validation and the data access layer (see section 10.1 and 12.4, respectively). Symbol explanation: ✓ means that the feature is supported and × means it is not supported; Assessment scale: of – (very poor) over o (neutral) to ++ (very good).

PART V

APPROACH

This part explains the main contribution of this master thesis: the concepts and implementation of a data quality management framework. We describe in the first chapter the components of our framework and justify every design decision in detail. This chapter is followed by four chapters which explain the concepts and implementations used by our framework in detail. Moreover, we do also present implementation alternatives where applicable. Finally, we propose in chapter 20 that the quality of data with respect to specific data quality dimensions can be increased using appropriate constraint types by giving some examples.

15 DATA QUALITY MANAGEMENT FRAMEWORK

We present in this chapter the components of our data quality management framework using a constraint-based approach. One of the main aims of this framework is the provision of the following options for an application developer:

- Specify constraints for a data model using a single constraint model
- Define constraints only once to avoid redundancy
- Use constraints to validate data
- Single validation to avoid unnecessary, multiple validation
- Provide advanced constraints and features to control data (e.g. time-triggered validation)

In the following sections, we describe and justify the design decisions of the data quality management framework. First, we present the basic concept followed by a feature description. Next, we explain how to persist data when using our data quality management framework. Finally, we describe the relation between constraints and data quality dimensions and provide the source code for a demonstrator application.

15.1 BASIS

After our broad research regarding already existing technologies for constraint specification and data validation we decided to build a data quality management framework using Java™ because the popularity of this programming language is really high[1] and the focus of our

[1] `http://www.tiobe.com/index.php/content/paperinfo/tpci/index.html`, [Online; accessed 08-October-2013]

research has been within the Java™™ environment as well. Furthermore, we think that the Java™ programming language offers widespread technologies for constraint management and data validation as shown in the technology part (see part IV). Next, we decided to use the existing framework defined in the JSR 349 which is called Bean Validation 1.1 (see section 10.1) as the basis for our data quality management framework. We will explain now how this concepts solve the problems mentioned above and in the introduction.

SINGLE CONSTRAINT MODEL

If we consider the figure 1.2 of the introduction again, we see that most often more than one language is used to specify constraints and validate data in a typical three-tier architecture although, the underlying data model is almost the same for every tier and/or layer. Therefore, Bean Validation 1.1 offers the possibility to specify constraints in a uniform way using a single language and constraint model. Conceptually, we annotate a domain model with constraints using Java™ annotations in a declarative way achieving our desired goal. Bean Validation constraints are layer/tier independent because they can be used for validation at different places within a three-tier architecture (see for example the GWT section 12.2.2). Specifying constraints in the domain model therefore avoids redundancy because there is no need to replicate constraints in the database or at the client-side. Besides the advantage that we get rid of the multiple languages within our constraint definitions, Java™ annotations are a well-known programming language construct according to [43] and hence a constraint definition using annotations seems to be a feasible and comprehensible way to specify constraints. Moreover, Java™ annotations are close to the source code which is a benefit if you compare constraints defined in the Java™ documentation (which nobody reads) or a configuration file consisting of constraint declarations (which is most often not close enough to the code to be practical) with a Java™ annotation on top of a class field or getter method. If a developer really wants to separate the constraint definitions in a separate configuration file, the JSR 349 (Bean Validation 1.1) offers still the possibility to create a corresponding XML file. Furthermore, the constraint definitions will be more consistent because we do not have to duplicate a constraint nor do we have to scatter constraint definitions to different tiers and/or layers. Summarising, Bean Validation provides a constraint declaration framework with a single constraint model using a single language (i.e. Java™ annotations in the domain model).

CONSTRAINT VALIDATION

Choosing Bean Validation as the basis for our data quality management framework gives us the possibility to validate data using the reusable validation API specified in the JSR 349 and the declared constraints in the domain model. Therefore, our third goal is implicitly given if we use the provided validation API which is another reason why we choose this concept as the basis for our framework.

INTEGRATION

If we consider table 14.1, we can immediately observe that almost every technology under study (except HTML5) integrates the concept of Bean Validation at least partially.

In addition, table 14.3 and table 14.4 clearly show that the integration process (i.e. how to enable Bean Validation) does not cause much overhead. Due to the analysis of the existing technologies, we know that it is possible to specify constraints only once using Bean Validation. Then, we can apply them within different tiers and/or layers avoiding redundancy. Although, HTML5 does not integrate Bean Validation, it is still possible to generate HTML5 code within a Bean Validation environment, i.e. not supporting a direct integration is no argument against. Again, due to the fact that Bean Validation allows the integration with other technologies we achieve our goal that we want to specify a constraint only once but use it within different tiers and/or layers using different technologies. Moreover, it is possible to validate data only once avoiding unnecessary and multiple validation. This can be realised by a validator which is located in the logic tier, managing the validation for the presentation and data tier.
Finally, Bean Validation 1.1 is a standardised concept which runs in an Java™ Standard Edition (SE) and Enterprise Edition (EE) environment which also supports our decision because we think that more features and an even better integration could be launched in the near future if a technology is a standard. Furthermore, it allows a developer to create custom constraints introducing new annotations. A detailed description of the JSR 349 (Bean Validation 1.1) and corresponding integration with other technologies is available in the technology part (see part IV).

15.2 FEATURES

This section gives an overview about the features of our data quality management framework with additional references to a detailed description. First, we define what kind of implementation library we use because in section 15.1 we explained that we use the JSR 349 (Bean Validation 1.1) without mentioning what kind of implementation provides this standard. We decided to choose Hibernate Validator 5 (see section 10.1.1.1) which is the reference implementation for the JSR 349 (Bean Validation 1.1). The implementation does also support the (JSR 303 (Bean Validation 1.0) which is the predecessor specification but we mainly choose this implementation because there is no other[2] implementation which supports Bean Validation 1.1 and the reference implementation provides additional features which we use for the implementation of temporal constraints (see below).

BASIC FEATURES

Our data quality management framework consists of every feature which is provided by Hibernate Validator 5 (see section 10.1.1.1) because our framework is build on top of this library. Hibernate Validator provides a couple of built-in constraints where some are required by the specification and others are Hibernate Validator specific. The following list shows every available constraint which we think is an important information to avoid the creation of redundant custom constraints.

- **Required by the JSR 349 (Bean Validation 1.1):** `@AssertFalse`, `@AssertTrue`, `@DecimalMax`, `@DecimalMin`, `@Digits`, `@Future`, `@Max`, `@Min`, `@NotNull`, `@Null`, `@Past`, `@Pattern`, `@Size`

[2]Checked on 08/10/2013

- **Hibernate Validator 5 specific:** `@CreditCardNumber`, `@Email`, `@Length`, `@ModCheck`, `@NotBlank`, `@NotEmpty`, `@Range`, `@SafeHTML`, `@ScriptAssert`, `@URL`

A detailed explanation can be found in section 10.1.1.1.

ASSOCIATION CONSTRAINT

An association constraint restricts the cardinality between entities. For instances, the number of students attending a lecture cannot be larger than the room size. Using our data quality management framework, a developer can specify this kind of constraint in an easy way using the association collection method described in section 16.2.2. Although, the Bean Validation provides a way to constraint the cardinalities of an association, we provide a more flexible solution. The section 16.2.2, describes the encountered problems in detail and provides several solutions mentioning their advantages and drawbacks. Finally, we decided to include the solution called 'association collection method' because it is a statically type safe and reusable solution which allows the specification of the cardinalities at runtime while requiring only a tiny overhead in form of a data structure. The corresponding annotation is denoted as `@VariableSize`. The complete list of pros and cons is described in the association collection method section (see section 16.2.2).

TEMPORAL CONSTRAINT

In chapter 17 the idea of a temporal constraint is introduced. A temporal constraint is a usual constraint coupled with a deadline which has the semantic that the actual constraint does not have to hold right now but at a certain point in time specified by the deadline. We provide such a constraint type because we think there are situations were an immediate constraint validation leads to a undesired behaviour. For instance, considered the situation were a student has to register for a certain number of courses at the beginning of the semester. Suppose, that the university has the rule that every student has to be registered for at least two courses. When a student enrols for the new semester the constraint cannot be satisfied at all leading to a undefined state with respect to the declared constraints. Introducing a deadline, where each student has to be registered for at least two courses solves the problem because it is totally legal to violate the constraint before the deadline. The implementation of this constraint type uses a special functionality (see section 17.2.1) of the Hibernate Validator 5 library which makes it necessary to use this reference implementation for the basis of our data quality management framework. A detailed description of the data structure which realises this kind of constraint and step by step explanation how to create a custom temporal constraint is given in the chapter 17. The following list shows the constraints which are already realised in our data quality management framework. The prefix of the constraint name (i.e. the string before the `OnDeadline`) always refers to the original constraint (see for instance the paragraph about the basic features). The temporal constraint has the same semantic as the actual constraint but with an additional deadline which specifies when the actual constraint must be fulfilled.

- **Temporal constraints** (based on JSR 349 constraints)**:**

  - `@AssertFalseOnDeadline`, `@AssertTrueOnDeadline`
  - `@DecimalMaxOnDeadline`, `@DecimalMinOnDeadline`
  - `@DigitsOnDeadline`
  - `@MaxOnDeadline`, `@MinOnDeadline`
  - `@NotNullOnDeadline`, `@NullOnDeadline`
  - `@PatternOnDeadline`
  - `@SizeOnDeadline`

- **Temporal constraints** (based on Hibernate Validator 5 specific constraints)**:**
  - `@CreditCardNumberOnDeadline`
  - `@EmailOnDeadline`
  - `@LengthOnDeadline`
  - `@ModCheckOnDeadline`
  - `@NotBlankOnDeadline`
  - `@NotEmptyOnDeadline`
  - `@RangeOnDeadline`
  - `@SafeHTMLOnDeadline`
  - `@ScriptAssertOnDeadline`
  - `@URLOnDeadline`
  - `@VariableSizeOnDeadline`

TIME-TRIGGERED VALIDATION COMPONENT

Closely connected to temporal constraints the time-triggered validation component described in chapter 18 offers a developer the possibility to schedule validation jobs at certain points in time or within a specified interval. The data quality management framework consists of predefined interfaces to create validation jobs and a universal validation job which can access objects in a data store using JPA.

SOFT AND HARD CONSTRAINTS

Our data quality management framework allows an application developer to categorise into hard and soft constraints. The first section of chapter 19 defines what a hard and soft constraint is followed by implementation suggestions. No additional implementation is necessary to realise hard constraints using Bean Validation because every declared constraint is automatically a hard constraint (the validation result of Bean Validation is always valid or invalid). Hence, we have to simulate soft constraints using the provided technology defined in the JSR 349 (Bean Validation 1.1). The soft constraint implementation section (see section 19.2.2) describes three possible implementations to realise them in a Bean Validation environment. We decided to include the ‘group method’ described in section 19.2.2.3 in our data quality management framework because a constraint can be easily categorised as a soft

constraint using the group attribute specified by the JSR 349 (Bean Validation 1.1) while leaving every constraint that should be a hard constraint untouched. Moreover, we describe how to you use the payload attribute which is also specified by the JSR 349 (Bean Validation 1.1) as a way to penalise soft constraint violations. Lastly, we describe an application to use the concept of hard and soft constraints to make a quantitative statement about the quality of data.

15.3 PERSISTENCE

Integrating a data access layer in combination with a data tier is possible with our data quality management framework. Especially, if the JPA is used to persist objects because this kind of approach is standardised, integrating an automatic validation mechanism based on Bean Validation. A very detailed description can be found in section 12.4. The complete list of features of our data quality management framework in combination with a persistence tier can only be used if the JPA 2.1 provided by the reference implementation EclipseLink is used. We need the reference implementation to persist the special data structure of our association constraint because no other implementation allows the persistence of a relationship using a custom implementation of the Java™ collection interface as we do in the data structure for the association constraint. When using an association constraint, we have to make sure that the cardinality information is stored as well if we persist the object that is taking part in the association. Moreover, we have to make sure to save the specified deadline when persisting an object that contains a temporal constraint.
Listing 15.1 shows an example how to persist an association with an association constraint. First, we have to declare the association denoted as `students` using the custom collection type `AssociationCollection<PersistentStudent>`. We have to use the fetch type `EAGER` otherwise EclipseLink would not be able to store a relationship that uses a custom collection implementation. Next, we define the join columns of the relationship, using the primary key (`LECTURE_NUMBER`) of the lecture class and the primary key (`STUDENT_NUMBER`) of the student class depicted in listing 15.2. Moreover, we have to add two fields that store the cardinalities of the corresponding association (i.e. the fields `minimumNumberOfStudents` and `maximumNumberOfStudents`). This seems to be an unnecessary overhead but we are not able to store the cardinalities directly combined with the declared collection. The getter method of the field `students` is annotated with the association constraint (`@VariableSize`) to restrict the number of elements in the collection. This would be enough to save a lecture object with the corresponding cardinality information. Unfortunately, EclipseLink is not able to proper initialise the fields if we load an object from the database that contains a custom collection because it is treated as a usual collection. Therefore, we use the JPA construct `@PostLoad` to set the cardinalities of the association after the object is completely load.

Listing 15.1: Annotation example of a lecture class that can be persisted using an association constraint.

```
@Entity
@Table(name = "LECTURES")
public class PersistentLecture {

  @Id
```

```
  @Column(name = "LECTURE_NUMBER")
  private String number;

  @Column(name = "TITLE")
  private String title;

  @ManyToMany(cascade = CascadeType.ALL, fetch = FetchType.EAGER)
  @JoinTable(
    name = "LECTURES_STUDENTS",
    joinColumns = @JoinColumn(name = "LECTURE_NUMBER"),
    inverseJoinColumns = @JoinColumn(name = "STUDENT_NUMBER")
  )
  private AssociationCollection<PersistentStudent> students;

  @Column(name = "MINIMUM_NUMBER_OF_STUDENTS")
  private int minimumNumberOfStudents;

  @Column(name = "MAXIMUM_NUMBER_OF_STUDENTS")
  private int maximumNumberOfStudents;

  ...

  @Valid
  @VariableSize
  public AssociationCollection<PersistentStudent> getStudents() {
    return students;
  }

  @PostLoad
  private void setCardinalities() {
    students.setMinimum(this.minimumNumberOfStudents);
    students.setMaximum(this.maximumNumberOfStudents);
  }

}
```

Listing 15.2 shows a student class that can be persisted using the temporal constraint `@MinOnDeadline(value = 18)` annotated next to the getter method of the field `age`. The class uses the special data structure for temporal constraints. In this example, the field `age` is of type `TemporalNumber` which corresponds to the usual number that can be coupled with a deadline. The data type `TemporalNumber` is annotated with `@Embeddable` which makes is possible to store the reference type within another table as described in section 12.4. If we annotate the declared field with `@Embedded` the relevant information (i.e. the number and the deadline) are automatically persisted. Furthermore, we should specify the column names as depicted for the field denoted as `age` at least if we would like to use more than one field having the data type `TemporalNumber` because if we do not specify the column names, a default column name is taken, which is the same for every attribute declared with the type `TemporalNumber` which results in a schema that should have two columns with the same name which is not possible.

Listing 15.2: Student class having a temporal constraint that can be persisted using additional annotation elements.

```
@Entity
@Table(name = "STUDENT")
public class PersistentStudent {

  @Id
  @Column(name = "STUDENT_NUMBER")
  private String studentNumber;

  @Column(name = "NAME")
  private String name;

  @Embedded
  @AttributeOverrides({
    @AttributeOverride(name = "deadline",
      column = @Column(name = "AGE_DEADLINE")),
    @AttributeOverride(name = "value",
      column = @Column(name = "AGE"))
  })
  private TemporalNumber age;

  @ManyToMany(mappedBy = "students")
  private List<PersistentLecture> lectures;

  @MinOnDeadline(value = 18)
  public TemporalNumber getAge() {
    return this.age;
  }

  public void setAge(TemporalNumber age) {
    this.age = age;
  }

}
```

Parts of the database structure for the annotated classes 15.2 and 15.1 are depicted in figure 15.1 using MySQL as the persistence unit within EclipseLink.

15.4 CONSTRAINTS AND DATA QUALITY DIMENSIONS

We believe that constraints can significantly improve data quality and therefore present in chapter 20 the relation between constraints and data quality dimensions.

15.5 DEMO APPLICATION

The complete source code for the examples mentioned above and a demonstrator application can be found in the appendix section A.3.

#	Name	Typ	Kollation	Attribute	Null	Standard
1	LECTURE_NUMBER	varchar(255)	latin1_swedish_ci		Nein	*kein(e)*
2	MAXIMUM_NUMBER_OF_STUDENTS	int(11)			Ja	*NULL*
3	MINIMUM_NUMBER_OF_STUDENTS	int(11)			Ja	*NULL*

#	Name	Typ	Kollation	Attribute	Null	Standard
1	STUDENT_NUMBER	varchar(255)	latin1_swedish_ci		Nein	*kein(e)*
2	LECTURE_NUMBER	varchar(255)	latin1_swedish_ci		Nein	*kein(e)*

#	Name	Typ	Kollation	Attribute	Null	Standard
1	STUDENT_NUMBER	varchar(255)	latin1_swedish_ci		Nein	*kein(e)*
2	NAME	varchar(255)	latin1_swedish_ci		Ja	*NULL*
3	AGE_DEADLINE	date			Ja	*NULL*
4	AGE	decimal(38,0)			Ja	*NULL*

Figure 15.1: Parts of the MySQL database structure which are able to store mapped objects of type `PersistentStudent` and `PersistentLecture`.

16 ASSOCIATION CONSTRAINT

The structure and relationship between entities of the underlying domain model is defined by associations. An association usually exists between two entities with additional multiplicities (also called cardinalities). For instance, a lecture in an university environment is usually attended by several students. The maximum number of students per lecture could be limited because the room where the lecture takes places has a certain number of seats. Furthermore, a professor does not hold a lecture if too few students are registered for a lecture. In this scenario, which is shown in figure 16.1, we have two entities (lecture and student) which are connected by a unidirectional association (called attended by) with multiplicities (min and max) and an association role (students). We will use this model as a running example within this chapter.

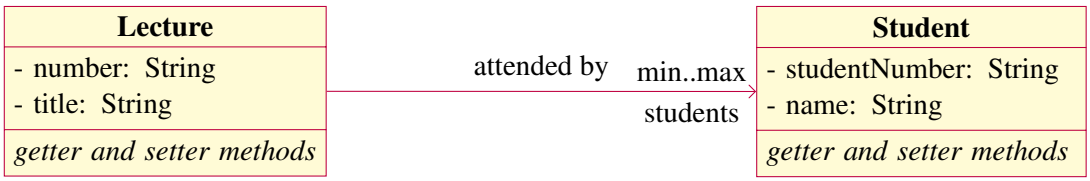


Figure 16.1: UML class diagram within the university domain showing an association between lectures and students.

Associations are usually implemented using a reference type or a collection of reference types. The Java™ classes in 16.1 show a possible implementation of the model defined by the UML diagram depicted in 16.1.

Listing 16.1: Java™ classes implementing an association between lectures and students.

```
public class Lecture {

  private String number;

  private String title;
```

```
  private Collection<Student> students;

  public Lecture() {
    students = new ArrayList<Student>();
  }

  // Additional methods (e.g. getter and setter methods)
  ...

}

public class Student {

  private String studentNumber;

  private String name;

  public Student() {
  }

  // Additional methods (e.g. getter and setter methods)
  ...

}
```

The goal is to control an association between two entities by controlling the multiplicity i.e. we would like to specify the minimum and maximum number of elements of a collection which represents an association. Furthermore, we would like to have the possibility to validate an association whether it fulfils its specified multiplicity. The following sections describe solutions to achieve the specified goals by implementing an association constraint.

16.1 STATIC ASSOCIATION CONSTRAINT

If we can only specify the multiplicity of an association constraint at compile time and it is not possible to change the value at runtime, we say that this kind of association constraint is a 'static association constraint'. The following sections present solutions to control an association with the help of a static association constraint.

16.1.1 SIMPLE `@Size` METHOD

The JSR 349 (Bean Validation 1.1) defines some built-in constraints as described in section 10.1.1. The annotation `@Size(min=, max=)` can be used for properties of type `CharSequence`, `Collection< E >`, `Map<K, V>` and arrays. The corresponding validator checks if the number of elements are between the specified minimum (`min`) and maximum (`max`) i.e. $collection.size \in [min, max]$. A class using this kind of constraint is shown in 16.2

Listing 16.2: Java™ lecture class using the built-in constraint @Size.

```
public class Lecture {

  private String studentNumber;
  private String name;

  @Size(min = 10, max = 50)
  private Collection<Student> students;

  ...

}
```

According to the standard only constant expressions can be specified as an attribute value. If we would like to use the `@Size(min=, max=)` annotation to control the multiplicity of an association, we have to determine the `min` and `max` values at compile time. It is not possible to use variables and pass some values at runtime. We can only change the values if we recompile and restart our application. Therefore, we define the multiplicity (i.e. `min` and `max`) once for a class `A` at compile time and every object conforming to type `A` has the same multiplicity at runtime. The following list summaries the advantages and disadvantages using `@Size(min=, max=)`:

- Pros
  - Built-in constraint `@Size(min=, max=)` as specified in the JSR 303/349 (Bean Validation 1.0/1.1) (see section 10.1.2 and 10.1.1)
  - No additional implementation needed (e.g. additional annotations, validators)
  - Works for associations implemented by an array or conforming to the Java™ type `Collection<T>` or `Map<K, V>`
  - Can be validated using a JSR 303/349 (Bean Validation 1.0/1.1) implementation e.g. Hibernate Validator (see section 10.1.1.1)
  - Optional minimum and/or maximum attribute value
  - Relationship between two entities can be controlled
  - Statically type safe
- Cons
  - Fixed constraint definition at compile time, the multiplicity cannot be changed at runtime
  - Every object conforming to type `A` must fulfil the constraints defined in class `A`

If a fixed size of objects will be created of a class that specifies a `@Size(min=, max=)` constraint and the multiplicities vary between the objects, one could apply the following rule:

- Let the value of the attribute `min` be the maximum of all known minima
- Let the value of the attribute `max` be the minimum of all known maxima

For instance, we have our lecture class and we know that we will create three objects representing different lectures. The name `o1` denotes the object of type `Lecture` with the field `name` set to `"Web Engineering"`. At least 12 and at most 89 students should attend the lecture. For short, $o1 : [Lecture, WebEngineering, min = 12, max = 89]$ will denote our first lecture. Furthermore, we have $o2 : [Lecture, InformationSystems, min = 5, max = 135]$ and $o3 : [Lecture, ObjectDatabases, min = 1, max = 42]$. Now we specify the multiplicity in our lecture class with the following values:

- $min = \max\{o1.min, o2.min, o3.min\} = \max\{12, 5, 1\} = 12$
- $max = \min\{o1.max, o2.max, o3.max\} = \min\{89, 135, 42\} = 42$

Of course, we lose the detailed requirements of the individual lecture objects, but we can at least set the multiplicity of the lecture class to a value which satisfies the requirement of the considered objects. Unfortunately, this does not always work. If we set the minimum number of students required for the 'Object Databases' lecture to 50, we will end up in a contradicting situation with $min = 50$ and $max = 42$. Furthermore, we cannot handle the case if another lecture will be introduced without recalculating the multiplicity, recompiling and restarting our application.

16.1.2 SUBCLASSING METHOD

The subclassing method does also use the `@Size(min=, max=)` constraint as described in section 16.1.1 to control an association but also uses Java™ subclassing (which is subtyping and inheritance[1]) and constraint inheritance (which is described in the JSR 303/349 (Bean Validation 1.0/1.1), see 10.1.2 and 10.1.1). The main idea is the usage of a base class `B` which defines an association and the constraints using `@Size(min=, max=)`. To solve the problem that every object of type `B` has the same constraints, we can introduce a subclass `B1` which extends the base class `B` and use constraint inheritance. Therefore, we can override the getter method in our subclass `B1` and add another `@Size(min=, max=)` constraint. Constraint inheritance will use the annotations of the base class and the subclass for validation and aggregates the result, which means that the validation result of the individual annotations is combined with a logical 'AND'. Therefore, we should make sure that the constraint of our subclass always fulfils the constraint of our base class. Listing 16.3 shows an implementation of our running example within the university domain.

Listing 16.3: Two classes showing the subclassing method to control the association between lectures and students.

```
public class Lecture {

  private String studentNumber;
  private String name;

  @Size(min = 1, max = 500)
  private Collection<Student> students;
```

[1] `http://www.pm.inf.ethz.ch/education/courses/coop/archive/2012/Lectures/lecture_03_-_inheritance.pdf`

```
  public Lecture(String title, String number) {
    this.number = number;
    this.title = title;
    students = new ArrayList<Student>();
  }

  public Collection<Student> getStudents() {
    return students;
  }

  ...

}

public class WebEngineering extends SimpleLecture {

  public WebEngineering() {
    super("Web Engineering", "252-0374-00L");
  }

  @Override
  @Size(min = 12, max = 89)
  public Collection<Student> getStudents() {
    return super.getStudents();
  }

}
```

A constraint that is annotated on a field and on the corresponding getter method, which is overridden in the subclass, is handled as if both constraints are annotated on a field or getter method. Another possibility is the creation of an abstract class without any constraints and only the subclass introduces the constraints.
Now we can create different objects which conform to the base type but specify individual constraints. Unfortunately, if a new subclass is needed we have to recompile and restart our application. A summary of the pros and cons for the subclassing method is given below:

- Pros
  - Built-in constraint `@Size(min=, max=)` as specified in the JSR 303/349 (Bean Validation 1.0/1.1) (see section 10.1.2 and 10.1.1) can be used
  - Works for associations implemented by an array or conforming to the Java™ type `Collection<T>` or `Map<K, V>`
  - Can be validated using a JSR 303/349 (Bean Validation 1.0/1.1) implementation e.g. Hibernate Validator (see section 10.1.1.1)
  - Optional minimum and/or maximum attribute value
  - Relationship between two entities can be controlled
  - Statically type safe
  - Individual constraints for subclasses using constraint inheritance and subclassing is possible

- Cons
  - Fixed constraint definition at compile time, the multiplicity cannot be changed at runtime
  - Subclasses have to be added at compile time (without reflection) leading to a recompile and restart of the application

16.2 DYNAMIC ASSOCIATION CONSTRAINT

The approaches described in 16.1 do not allow to specify the multiplicity of an association at runtime. Ideally, we would like to declare in a class `A` an association constraint but the multiplicity should be set when an object of type `B` that conforms to type `A` is created. The class depicted in 16.4 shows the desired scenario. As already mentioned in 16.1, a value of an annotation attribute must be a constant expression and therefore the solution presented in 16.4 does not compile. In fact, what we have to solve is the problem of cross-field validation which can be defined as the validation of a constraint depending on several fields of a class.

Listing 16.4: Java™ class showing how one could use variables within annotations. Caution, this Java class does not compile because we cannot use variables within annotations.

```
public class Lecture {

  private String number;
  private String title;

  @Size(min = minimumNumberOfStudents,
    max = maximumNumberOfStudents)
  private Collection<Student> students;

  private int minimumNumberOfStudents;
  private int maximumNumberOfStudents;

  public Lecture(String title, String number,
    int minimumNumberOfStudents, int maximumNumberOfStudents) {

    this.minimumNumberOfStudents = minimumNumberOfStudents;
    this.maximumNumberOfStudents = maximumNumberOfStudents;
    this.number = number;
    this.title = title;
    students = new ArrayList<Student>();

  }

  ...

}
```

The following sections present several possibilities to define the multiplicity of an association constraint at runtime. We call this kind of constraint a ‘dynamic association constraint’. At the same time the solutions show how cross-field validation can be done.

16.2.1 TYPE-LEVEL CONSTRAINT METHODS

Type-level constraints (which are also called class-level constraints) are specified in the JSR 303/349 (Bean Validation 1.0/1.1), see section 10.1.2 and 10.1.1 and they are based on annotations that are declared on top of a class. The corresponding validator can access the full public interface of the object that is validated. Next, we present several possibilities to realise a dynamic association constraint using type-level constraints.

16.2.1.1 HAND-CRAFTED ASSOCIATION CONSTRAINT METHOD

The basic idea of the Hand-crafted Association Constraint (HAC) method is a custom annotation and custom cross-field validator for every association. For every association a programmer has to code a validator and an annotation from scratch. Furthermore, every class implementing an association has to introduce two fields which correspond to the minimum and maximum size of the association. A sample implementation for our running example is depicted in 16.5. The association constraint is implemented with a custom annotation called `StudentAssociation` and the corresponding validator `StudentAssociationValidator`.

Listing 16.5: Sample implementation of the hand-crafted association constraint (HAC) method with the custom type-level annotation `@StudentAssociation` and its validator.

```
@Target({ METHOD, FIELD, ANNOTATION_TYPE, TYPE })
@Retention(RUNTIME)
@Documented
@Constraint(validatedBy = { StudentAssociationValidator.class })
public @interface StudentAssociation {

  String message() default
    "wrong multiplicity for the association role students";

  Class<?>[] groups() default {};

  Class<? extends Payload>[] payload() default {};

}

public class StudentAssociationValidator
  implements ConstraintValidator<StudentAssociation, Lecture> {

  @Override
  public void initialize(StudentAssociation constraintAnnotation) {
  }

  @Override
  public boolean isValid(Lecture value,
    ConstraintValidatorContext context) {

    int numberOfStudents = value.getStudents().size();
    int minimum = value.getMinimumNumberOfStudents();
    int maximum = value.getMaximumNumberOfStudents();
```

```
    return minimum <= numberOfStudents &&
      numberOfStudents <= maximum;
  }

}
```

Now, we can use the association constraint in our lecture class depicted in 16.6. The class introduces two fields to control the multiplicity of the association implemented by the field `students`. Furthermore, we can set the value of the multiplicity at runtime using the constructor arguments.

Listing 16.6: Usage example of the association constraint `@StudentAssociation`.

```
@StudentAssociation
public class Lecture {

  private String number;
  private String title;

  private ArrayList<Student> students;
  private int minimumNumberOfStudents;
  private int maximumNumberOfStudents;

  public Lecture() {
    students = new ArrayList<Student>();
  }

  public Lecture(String number, String title,
    int minimumNumberOfStudents, int maximumNumberOfStudents) {

    this.minimumNumberOfStudents = minimumNumberOfStudents;
    this.maximumNumberOfStudents = maximumNumberOfStudents;
    students = new ArrayList<Student>();
    this.number = number;
    this.title = title;
  }

  ...
}
```

This method achieves our goal to control a relationship between two entities and in addition it provides a mechanism to define the multiplicity for individual objects at runtime. The following list shows the benefits and drawbacks of the presented method:

- Pros
  - Works for associations implemented by an array or conforming to the Java™ type `Collection<T>` or `Map<K, V>`
  - Can be validated using a JSR 303/349 (Bean Validation 1.0/1.1) implementation e.g. Hibernate Validator (see section 10.1.1.1)
  - Relationship between two entities can be controlled
  - Statically type safe

  - Runtime specification of the multiplicity for every object of type `A` (and every object with type `B` that conforms to class `A`) that declares the association
  - No subclassing for individual object requirements needed
  - Static association constraint can still be simulated by field initialisation

- Cons
  - Optional minimum and/or maximum attribute value has to be implemented
  - No built-in constraint of the JSR 303/349 (Bean Validation 1.0/1.1) (see section 10.1.2 and 10.1.1) is used
  - No re-use: every association requires a new annotation and validator
  - Additional implementation needed: annotation, validator and the model class has to be modified

The complete source code can be found in the appendix section A.4.1.1.

16.2.1.2 GENERIC ASSOCIATION CONSTRAINT METHOD

The Generic Association Constraint (GAC) method reduces the implementation work which is necessary to control the multiplicity of an association compared to the method described in 16.2.1.1. Still, this method requires that a programmer does some implementation work, but only once for an application. The main idea is the creation of a small data structure which is accessed by the validator to fetch the multiplicity and the actual association size. The data structure consists of a `HashMap` which maps keys of type `AssociationRole` to values of type `AssociationDefinition` which is depicted in the left part of figure 16.2.

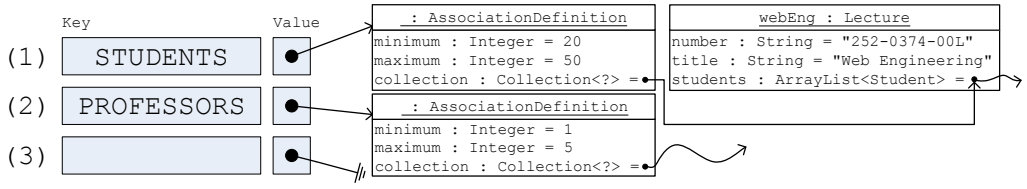


Figure 16.2: Snapshot of a possible application state showing a `HashMap` in the left part and the `AssociationDefinition` which stores the multiplicity and a reference to the corresponding association implementation in the right part.

The type `AssociationRole` is a Java™ enumeration which must contain a constant for every possible association role in the application. The enumeration corresponding to our running example and figure 16.2 is depicted in 16.7.

Listing 16.7: Sample enumeration containing the constants for the association roles.

```
public enum AssociationRole {
  STUDENTS,
  PROFESSORS
}
```

Next, the class `AssociationDefinition` stores the multiplicity (minimum and maximum) and a reference to the association implementation. In case of figure 16.2, the association role `STUDENTS` maps to the object with `minimum = 20`, `maximum = 50` and a reference to the association implementation `students`, which conforms to the type `Collection<?>`. A class which wants to use this association constraint method must extend the `Associations` class (see code fragment 16.8) to use the hashing function and the `AssociationDefinition`.

Listing 16.8: The `Associations` class builds the data structure to store and retrieve the association information like the multiplicity of an association.

```
public class Associations {

  private Map<AssociationRole, AssociationDefinition> associations;

  public Map<AssociationRole, AssociationDefinition>
    getAssociations() {

    return this.associations;
  }

  public void addAssociationDefinition(
    AssociationRole role,
    AssociationDefinition associationDefinition) {

    this.associations.put(role, associationDefinition);
  }

  ...

  public class AssociationDefinition {

    private int minimum;
    private int maximum;
    private Collection<?> collection;

    ...

    }

}
```

Finally, we can create a custom annotation and the corresponding validator (see code fragment 16.9) to realise the association constraint. Therefore, we define an annotation `@Association(role=)` which specifies the attribute `role` and it must take a value of type `AssociationRole`. The corresponding validator class `AssociationValidator` is initialised with the attribute value which corresponds to the underlying association implementation and makes use of it within the `isValid` method. The validator uses the value of the field `role` as the key to retrieve the mapped object of type `AssociationDefinition` (see figure 16.2). Next, it can access the multiplicity and the actual size of the association to calculate the validation result.

Listing 16.9: The custom annotation `@Association` and the corresponding validator `AssociationValidator` are combined and build a dynamic association constraint.

```
@Target({ METHOD, FIELD, ANNOTATION_TYPE, TYPE })
@Retention(RUNTIME)
@Documented
@Constraint(validatedBy = { AssociationValidator.class })
public @interface Association {

    String message() default
      "wrong multiplicity for the association role {role}";

    Class<?>[] groups() default {};

    Class<? extends Payload>[] payload() default {};

    AssociationRole role();

    @Target({ FIELD, METHOD, TYPE, ANNOTATION_TYPE })
    @Retention(RUNTIME)
    @Documented
    @interface List {
      Association[] value();
    }

}

public class AssociationValidator
  implements ConstraintValidator<Association, Associations> {

  private AssociationRole role;

  @Override
  public void initialize(Association constraintAnnotation) {
    this.role = constraintAnnotation.role();
  }

  @Override
  public boolean isValid(
    Associations value,
    ConstraintValidatorContext context) {

    AssociationDefinition ad = value.getAssociations().get(role);
    int collectionSize = ad.getCollection().size();
    return (ad.getMinimum() <= collectionSize)
            && (collectionSize <= ad.getMaximum());
  }

}
```

Now, we are ready to use the dynamic association constraint `@Association(role=)` for our running example. To see the benefit compared to the hand-crafted association constraint (see section 16.2.1.1), we extend the lecture with an association to professors which hold a

lecture. The code snippet 16.10 shows the usage of the new constraint. The example uses a list of `@Association(role=)` constraints which is defined by the JSR 303/349 (Bean Validation 1.0/1.1) in section 10.1.2 and 10.1.1.

Listing 16.10: Sample usage of the new type-level association constraint `@Association` with an extended university model.

```
@Association.List({
  @Association(role = AssociationRole.STUDENTS),
  @Association(role = AssociationRole.PROFESSORS)
})
public class Lecture extends Associations {

  private String number;
  private String title;

  private ArrayList<Student> students;
  private ArrayList<Professor> professors;

  public Lecture() {
    students = new ArrayList<Student>();
    professors = new ArrayList<Professor>();
  }

  ...
}

public class Main {
  public static void main(String[] args) {
    Lecture wE =
      new Lecture("252-0374-00L", "Web Engineering");
    wE.addAssociationDefinition(
      AssociationRole.STUDENTS,
      wE.new AssociationDefinition(20, 50, wE.getStudents()));
    wE.addAssociationDefinition(
      AssociationRole.PROFESSORS,
      wE.new AssociationDefinition(1, 2, wE.getProfessors()));
  }
}
```

To control a new association we simply have to add a constant to the `AssociationRole` enumeration, add the type-level constraint `@Association(role=)` and set up the data structure with the multiplicity. We do not have to create another custom annotation or validator, nor do we have to introduce additional fields to our class. We just have to implement the data structure once and extend every class by `Associations` which wants to use the `@Association(role=)` constraint. Unfortunately, it is not possible to provide the code in form of a library, because Java™ does not allow to extend an enumeration (enumeration inheritance is not possible), but the list of constants is application specific and must therefore be defined by the programmer of the application. Nevertheless, we provide the full source code in the appendix section A.4.1.1 to be able to use the functionality. The classes `Associations`, `AssociationValidator` and the annotation implementation `Association` do not have to be modified. Sole, the enumeration `AssociationRole`

has to be created and filled with the appropriated constants. We summarise the GAC method by listing the advantages and disadvantages:

- Pros
  - Works for associations using the Java™ type `Collection<T>`
  - Can be validated using a JSR 303/349 (Bean Validation 1.0/1.1) implementation e.g. Hibernate Validator (see section 10.1.1.1)
  - Relationship between two entities can be controlled
  - Statically type safe
  - Runtime specification of the multiplicity for every object of type `A` (and every object with type `B` that conforms to class `A`) that declares the association
  - No subclassing for individual object requirements needed
  - Static association constraint can still be implemented by some initialisation code e.g. within the constructor
  - Re-use is possible: not every association requires a new annotation and validator, the data structure can be used for all associations
- Cons
  - Additional code is needed to support arrays or the Java™ type `Map<K, V>`
  - Optional minimum and/or maximum attribute value has to be implemented
  - No built-in constraint of the JSR 303/349 (Bean Validation 1.0/1.1) (see section 10.1.2 and 10.1.1) is used
  - Additional implementation for each application needed: one custom annotation and the corresponding validator, a utility class and an enumeration
  - Data structure cannot be provided by a library (but can be copied from the appendix)
  - Small overhead when initialising the data structure to provide the validation information

16.2.1.3 INTROSPECTIVE ASSOCIATION CONSTRAINT METHOD

Inspired by Stack Overflow question[2] and a blog post[3], the Introspective Association Constraint (IAC) method combines the simplicity of the HAC method (see section 16.2.1.1) with the genericity of the GAC method (see section 16.2.1.2) by using introspection. The basic idea is the specification of the minimum, maximum and collection field name as strings within the constraint and then access the values with the help of introspection. Following this kind of approach, the programmer does not have to create a custom annotation or validator because the implementation can be shipped within a library. The main drawback is

[2] `http://stackoverflow.com/questions/1972933/cross-field-validation-with-hibernate-validator-jsr-303`, [Online; accessed 12-August-2013]

[3] `http://www.app-solut.com/blog/2011/03/cross-field-validation-with-the-bean-validation-api/`, [Online; accessed 12-August-2013]

that we lose the static type safety due to introspection and casting. The dynamic association constraint consists of a custom annotation and the corresponding validator which is depicted in 16.11. The custom annotation `@IntrospectiveAssociation` defines three non-optional attributes: the name of the minimum, maximum and collection field. The corresponding validator uses the Apache Commons BeanUtils[4] library to access the fields with the corresponding name.

Listing 16.11: The `@IntrospectiveAssociation` annotation and the validator class `IntrospectiveAssociationValidator` use introspection to control the multiplicity of an association.

```
@Target({ METHOD, FIELD, ANNOTATION_TYPE, TYPE })
@Retention(RUNTIME)
@Documented
@Constraint(validatedBy =
  { IntrospectiveAssociationValidator.class })
public @interface IntrospectiveAssociation {

  String message() default
    "wrong multiplicity for the
    association role {collectionFieldName}";

  Class<?>[] groups() default {};

  Class<? extends Payload>[] payload() default {};

  String minimumFieldName();

  String maximumFieldName();

  String collectionFieldName();

  @Target({ FIELD, METHOD, TYPE, ANNOTATION_TYPE })
  @Retention(RUNTIME)
  @Documented
  @interface List {
    IntrospectiveAssociation[] value();
  }

}

public class IntrospectiveAssociationValidator
  implements ConstraintValidator<IntrospectiveAssociation, Object> {

  private String minimumFieldName;
  private String maximumFieldName;
  private String collectionFieldName;

  @Override
  public void initialize(IntrospectiveAssociation annotation) {
```

[4] `http://commons.apache.org/proper/commons-beanutils/`, [Online; accessed 12-August-2013]

```
    minimumFieldName = annotation.minimumFieldName();
    maximumFieldName = annotation.maximumFieldName();
    collectionFieldName = annotation.collectionFieldName();
  }

  @Override
  public boolean isValid(Object value,
    ConstraintValidatorContext context) {

    if (value == null) {
      return true;
    } else {
      int minimum;
      int maximum;
      Collection<?> collection;
      try {
        minimum = (int) PropertyUtils.getProperty(
          value, minimumFieldName);

        maximum = (int) PropertyUtils.getProperty(
          value, maximumFieldName);

        collection = (Collection<?>) PropertyUtils.getProperty(
          value, collectionFieldName);

        if (collection == null) {
          return true;
        } else {
          int size = collection.size();
          return minimum <= size && size <= maximum;
        }
      } catch (Exception e) {
        throw new IllegalArgumentException(
          "Field does not exist.", e);
      }
    }
  }

}
```

The constraint is designed in such a way, that it is possible to specify several association constraints for the same type using the annotation list concept of the JSR 303/349 (Bean Validation 1.0/1.1) which is described in section 10.1.2 and 10.1.1. It is also possible to get rid of the Apache Commons BeanUtils dependency using solely the Java™ Reflection API[5]. The following list shows the pros and cons for the IAC method:

- Pros
  - Works for associations using the Java™ type `Collection<T>`
  - Can be validated using a JSR 303/349 (Bean Validation 1.0/1.1) implementation e.g. Hibernate Validator (see section 10.1.1.1)

[5] See footnote 3

  - Relationship between two entities can be controlled
  - Runtime specification of the multiplicity for every object of type `A` (and every object with type `B` that conforms to class `A`) that declares the association constraint is possible
  - No subclassing for individual object requirements needed
  - Static association constraint can still be simulated by field initialisation
  - Easy re-use: one custom annotation and the corresponding validator can be used for every association
  - No special data structure needed

- Cons
  - Optional minimum and/or maximum attribute value has to be implemented
  - No built-in constraint of the JSR 303/349 (Bean Validation 1.0/1.1) (see section 10.1.2 and 10.1.1) is used
  - Additional code is needed to support arrays or the Java™ type `Map<K, V>`
  - Not statically type safe
  - Model class has to introduce additional fields for the multiplicity information

The complete source code for the IAC method is available in the appendix section A.4.1.1.

16.2.2 ASSOCIATION COLLECTION METHOD

The association collection method tries to solve the association control problem with the help of a custom data structure. Therefore, the Java™ `Collection<T>` interface and the `Association` interface is implemented by the class `AssociationCollection<T>` as depicted in figure 16.3. The `Association` interface defines the methods to access the multiplicity and the actual size of an association implementation. The Java™ `Collection<T>` interface defines the queries and commands (see [44]) to modify a collection (e.g. add or remove an element) or to retrieve the status (e.g. the number of elements in the collection). The class `AssociationCollection<T>` represents a custom collection with two additional features called association collection. Implementing the `Collection<T>` interface with the help of delegation makes the association collection to be a usual Java™ `Collection<T>`. The delegation mechanism allows the client to use any concrete implementation of the Java™ `Collection<T>`interface like `ArrayList<T>`. The two additional fields `minimum` and `maximum` can be set when constructing an object of type `AssociationCollection<T>`. Together with the `Association` interface implementation the association collection can control the multiplicity of an association.
Next, we define a custom constraint which makes use of the data structure association collection. Therefore, we create the custom annotation `@VariableSize` without an additional argument. This annotation is the counterpart of the static association constraint `@Size` (see section 16.1) but without the attributes `min` and `max`. The custom annotation is shown in listing 16.12 coupled with the validator class `VariableSizeValidator`. Note that we use the Java™ Expression Language [45] within our default message definition. This only works for JSR 349 (Bean Validation 1.1) implementations as JSR 303 (Bean Validation 1.0)

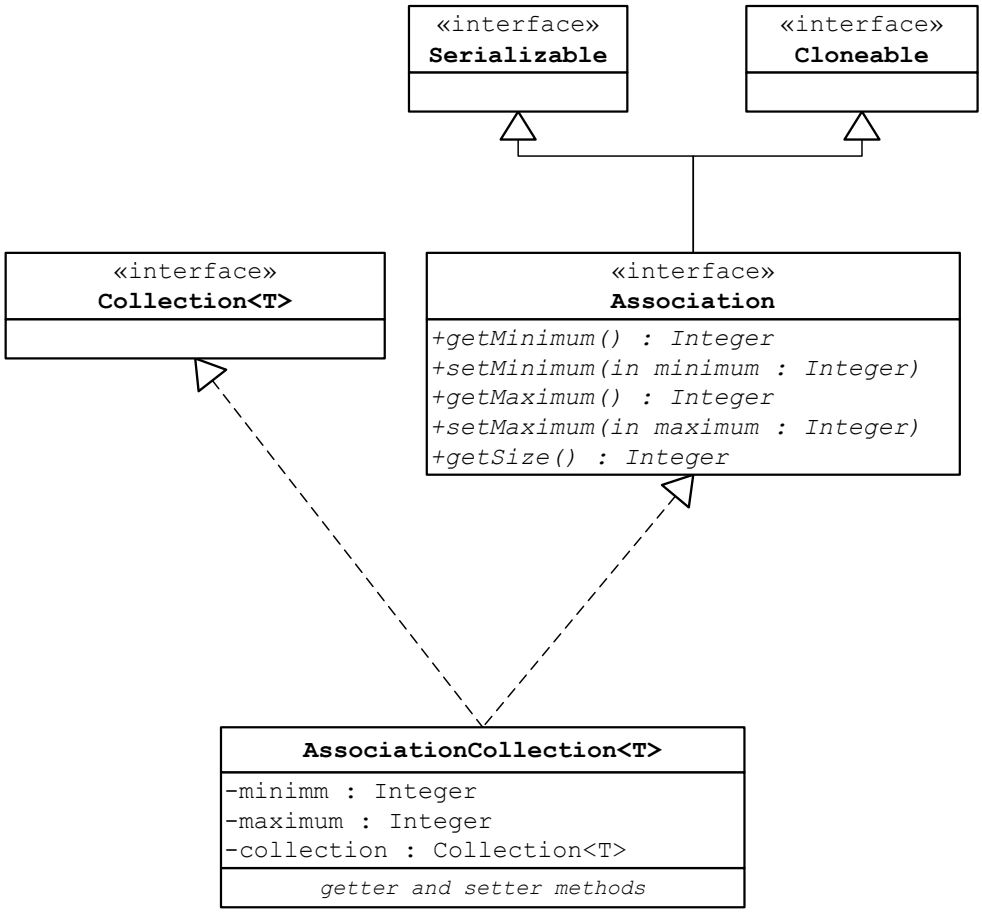


Figure 16.3: UML diagram of the data structure for the association collection method.

does not support the Java™ Expression Language within messages (see section 10.1.1 and 10.1.2). An implementation[6] for the Java™ Expression Language must be included in the runtime environment to make use of it. If a JSR 303 (Bean Validation 1.0) implementation is used the expression within the message will not be evaluated however no exception is thrown and the expression is simply printed as a string (e.g. `\${validatedValue.size()}` is printed as it is).

Listing 16.12: The custom annotation called `@VariableSize`.

```
@Target({ METHOD, FIELD, ANNOTATION_TYPE })
@Retention(RUNTIME)
@Documented
@Constraint(validatedBy = { VariableSizeValidator.class })
```

[6]For instance the reference implementation available at `https://uel.java.net/`, [Online; accessed 03-October-2013]

```
public @interface VariableSize {

    String message() default
      "\${validatedValue.size() < validatedValue.getMinimum() ?
      'minimum' : 'maximum'}
      multiplicity of the association role is not fulfilled";

    Class<?>[] groups() default {};

    Class<? extends Payload>[] payload() default {};

}
```

The validator class for the `@VariableSize` annotation defines that the custom annotation can only be used on data types conforming to `Association`. Therefore, we make sure that the validator has access to the methods of the association collection i.e. the multiplicity and the actual size of the association implementation. The object to be validated is passed as an argument in the `isValid` method as depicted in listing 16.13.

Listing 16.13: The validator class `VariableSizeValidator` which corresponds to the `@VariableSize` annotation depicted in 16.12.

```
public class VariableSizeValidator implements
  ConstraintValidator<VariableSize, Association> {

  @Override
  public void initialize(VariableSize constraintAnnotation) {
  }

  @Override
  public boolean isValid(Association value,
    ConstraintValidatorContext context) {

    if (value == null) {
      return true;
    } else {
      int minimum = value.getMinimum();
      int maximum = value.getMaximum();

      if (minimum < 0 && maximum < 0) {
        throw new IllegalArgumentException(
          "Minimum and maximum argument cannot be negative.");
      } else if (minimum < 0) {
        throw new IllegalArgumentException(
          "Minimum argument cannot be negative.");
      } else if (maximum < 0) {
        throw new IllegalArgumentException(
          "Maximum argument cannot be negative.");
      } else if (maximum < minimum) {
        throw new IllegalArgumentException(
          "Minimum argument cannot be larger
          than maximum argument.");
      } else {
```

```
        int collectionSize = value.size();
        return collectionSize >= minimum
          && collectionSize <= maximum;
      }
    }
  }
}
```

The dynamic association constraint `@VariableSize` is implemented in order that it can be directly used at the corresponding field or getter method where the association is implemented with the association collection. Furthermore, it is possible to ship this association constraint within a library and the client does not have to implement anything in addition. The following list mentions the benefits and drawbacks of the association collection method:

- Pros
  - Can be validated using a JSR 303/349 (Bean Validation 1.0/1.1) implementation e.g. Hibernate Validator (see section 10.1.1.1)
  - Relationship between two entities can be controlled
  - Runtime specification of the multiplicity for every object of type `B` which conforms to the class type `A` that declares the association constraint is possible
  - No subclassing for individual object requirements needed
  - Static association constraint can still be simulated using initialisation within a constructor
  - Easy re-use: one custom annotation and the corresponding validator can be used for every association
  - Statically type safe
  - No type-level annotation needed, the constraint can be directly annotated at the association implementation
  - Custom collection can be provided by a library, no further implementation has to be done
  - Custom collection behaves almost like a usual Java™ `Collection<T>`
  - Model class does not need additional fields for the multiplicity
  - No additional annotations or validators needed
- Cons
  - Optional minimum and/or maximum attribute value has to be implemented
  - No built-in constraint of the JSR 303/349 (Bean Validation 1.0/1.1) (see section 10.1.2 and 10.1.1) is used
  - Additional code is needed to support arrays or the Java™ type `Map<K, V>`
  - Special data structure (association collection) has to be used for the association implementation
  - Tiny initialisation overhead of the data structure

The usage of dynamic association constraint `@VariableSize` is a one-liner depicted in the code fragment 16.14.

Listing 16.14: Code fragment showing the usage of the `@VariableSize` constraint for our running example.

```
@VariableSize
AssociationCollection<Student> students = new
  AssociationCollection<Student>(new ArrayList<Student>(), 0, 1);
```

The interface `Association` depicted in figure 16.3 extends the interfaces `Serializable` and `Cloneable` to ensure that the association implementation can be persisted. The complete source code can be found in appendix section A.4.1.2.

17 TEMPORAL CONSTRAINT

Sometimes, a declared constraint must not hold immediately but at a certain point in time. For instance, within a university environment every student must register for a minimum number of courses and furthermore there is a limit on the number of exams a student can take. But, if a student enrols in a new term it is not necessary that he immediately registers for some courses and therefore a constraint on the minimum number of courses need not to be fulfilled right away. In this scenario, a constraint should hold only at a certain point in time specified by a deadline. Moreover, the number of exams a student will take should not be limited right at the start of a term but only when the registration for the exam is done.
We call this kind of constraint a 'temporal constraint' because the constraint need not hold until a certain point in time is reached and therefore the constraint is extended by a temporal dimension. The following sections present a solution to support a temporal dimension for constraints.

17.1 DATA STRUCTURE

We present a solution which is based on a small data structure that handles the definition and access of a deadline. A deadline is a date which specifies a certain point in time and if reached, a corresponding constraint should hold. In section 17.1.4 we explain how to extend the data structure.

17.1.1 TEMPORAL INTERFACE

The Java™ interface `Temporal` which is depicted in 17.1 defines a method to specify and access a deadline. A deadline is declared with the Java™ type `Date` which allows a client to specify a deadline up to a precision of milliseconds as described in [46].
An interface instead of an (abstract) class declaration was chosen to offer a data structure client the possibility to inherit from another class while using the data structure which is not possible if we declare `Temporal` as an (abstract) class due to the absence of multiple

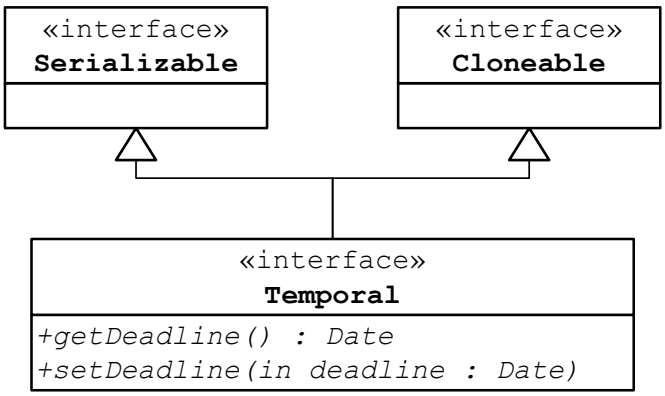


Figure 17.1: UML class diagram of the `Temporal` interface.

inheritance in Java™. The interface `Temporal` extends the interfaces `Serializable` and `Cloneable` to make sure that we can persist the specified deadline. The source code is available in the appendix section A.5.1.1.

17.1.2 PRIMITIVE TEMPORAL DATA TYPES

A temporal constraint always consists of a deadline and a value. The value is validated against a constraint with respect to the specified deadline. For example the value of a temporal constraint could be the minimum number of courses, the deadline is set to the start of a term and the constraint is defined as 'the minimum number of courses are at least 3'. This is a cross-field validation problem because we have to retrieve the value of two fields (the deadline and the actual value) and check against an appropriate constraint. Cross-field validation can be implemented with a type-level constraint (see section 10.1.2 and 10.1.1) or a data structure combining the value and deadline with a special custom constraint as presented below. We do not use type-level constraints because a generic design needs an initialisation overhead for the client (analogous to 16.2.1.2) or we lose static type safety (analogous to 16.2.1.3).
Therefore, we present a special data structure for the primitive Java™ types which combines a primitive Java™ type with a deadline to handle cross-field validation. The small data structure is depicted in figure 17.2 and shows that every primitive temporal data type class implements the `Temporal` interface described in 17.1.1 and adds the corresponding primitive Java™ type.
It is not possible to extend the wrapper classes for the primitive Java™ types because the classes are declared as `final` which disallows any specialisation. But we can use other classes to make the existing constraints applicable to our data structure (see section 17.2.3. First, the abstract `Number` class represents a supertype of the wrapper classes representing a number like `Integer` and `Double`. We use the abstract `Number` class to represent a value of a temporal constraint which is a number. In addition, the `TemporalNumber` class provides features to define integers and decimals by implementing the abstract methods of

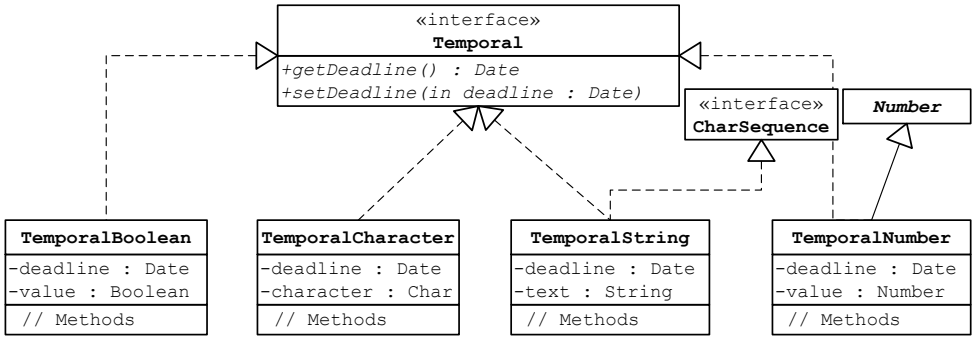


Figure 17.2: UML class diagram of the primitive temporal data types which implement the `Temporal` interface to handle the temporal dimension.

`Number` as defined in [47]. Second, we use the interface `CharSequence` and implement the methods described in [48]. The implementation of this interface makes it possible to treat a `TemporalString` as a `CharSequence` that makes it possible to apply some built-in constraints like `@Size` to it. The source code of the primitive temporal data types can be found in the appendix section A.5.1.2.

17.1.3 TEMPORAL ASSOCIATION COLLECTION

An association constraint as described in chapter 16 has a multiplicity which controls the minimum and the maximum number of entities building the association. There are scenarios where an association constraint might hold not right away but at a certain point in time. For instance, the lectures of an university are published two months before the next term starts. Each lecture defines a minimum and maximum number of students which must hold at the start of the semester. Consequently, there is a time slot where the constraints of a lecture need not hold which adds a temporal dimension to the constraints of the multiplicity of an association, i.e. for an association constraint.
There are several possibilities to handle the temporal aspect of an association. We could extend the type-level constraint methods described in section 16.2.1 or we could extend the association collection (see section 16.2.2). We do not use type-level constraints for the same reasons as described in 17.1.2 and we therefore present the extension of the Association Collection using the `Temporal` interface of section 17.1.1 to build the temporal association collection. Technically, we extend the `AssociationCollection<T>` class described in 16.2.2 and implement the `Temporal` interface at the same time resulting in the class `TemporalAssociationCollection` which is shown in figure 17.3 and the full source code is available in the appendix section A.5.1.3.

17.1.4 DATA STRUCTURE EXTENSION

We summarise the most important steps to extend the data structure:

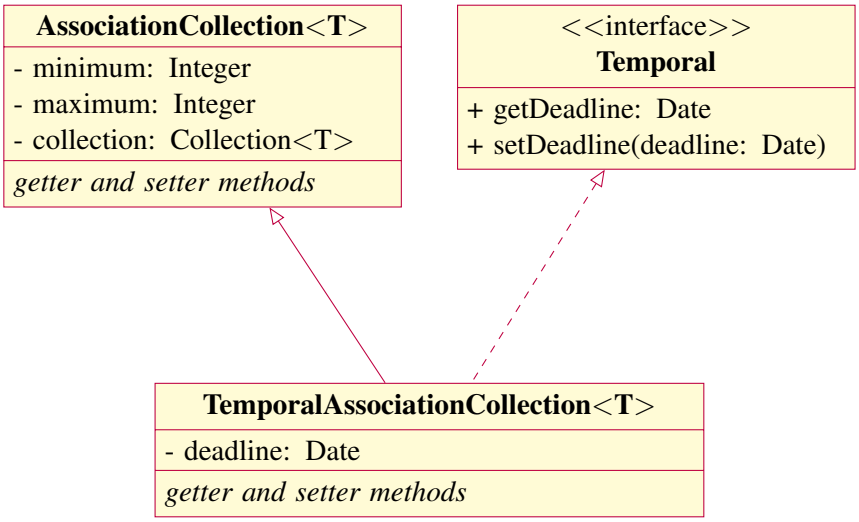


Figure 17.3: UML class diagram showing how the temporal association collection is designed.

1. Choose a type T which should be extended to be able to handle a temporal dimension
2. Create a new class `TemporalT` which implements the interface `Temporal` described in 17.1.1
   (a) Add a private field of type `java.util.Date` called deadline
   (b) Add a private field of type T with a meaningful name
   (c) Add getter and setter methods for both fields (implementing the methods of `Temporal`)
   (d) Add an argumentless public constructor (optional: any additional constructor)

17.2 CONSTRAINTS

A temporal constraint is built using the data structure presented in section 17.1 and consists of an annotation and its corresponding validator. The basic idea is the definition of a deadline constraint (i.e. an annotation and a validator) handling the temporal part which can then be combined with any other constraint checking whether the actual value is valid. We first explain the deadline constraint, then we describe how to combine the deadline constraint with other constraints and finally we explain how to add a new temporal constraint.

17.2.1 `@Deadline` CONSTRAINT

17.2.1.1 ANNOTATION

The annotation denoted as `@Deadline` is a standard Java™ annotation without any additional attributes connected with a validator described in section 17.2.1.2 except that the scope of the annotation usage is restricted. The annotation can only be used on top of other annotations as defined by the `@Target` annotation depicted in listing 17.1. This means that it can only be used in composition with another annotation where only a composition within

the annotation interface makes sense. We introduce this restriction because we think using a deadline constraint (i.e. a `@Deadline` annotation) without another constraint does not make sense. For instance, what would be the meaning of a deadline constraint for the age of a person? It becomes clear, if we say that the minimum age of a person should be valid after a deadline which leads to the composition with another constraint.

Listing 17.1: The annotation denoted with `@Deadline` which can be combined with other constraints using constraint composition.

```
@Target({ ANNOTATION_TYPE })
@Retention(RUNTIME)
@Documented
@Constraint(validatedBy = { DeadlineValidator.class })
public @interface Deadline {

  String message() default "deadline has expired";

  Class<?>[] groups() default {};

  Class<? extends Payload>[] payload() default {};

}
```

The annotation class can be found in the appendix section A.5.2.

17.2.1.2 VALIDATOR

The validator for the `@Deadline` annotation (see section 17.2.1.1) denoted with `DeadlineValidator` is designed based on the following observation: If the specified deadline has not yet expired the corresponding value need not be valid but afterwards. Using this observation, the `DeadlineValidator` simply compares the specified deadline with today's date and returns `true` if the deadline has not yet expired and `false` otherwise (see listing 17.2). In addition, we return `false` if no deadline is specified.

Listing 17.2: The deadline validator for the `@Deadline` annotation which compares the specified date with today's date.

```
public class DeadlineValidator
  implements ConstraintValidator<Deadline, Temporal> {

  @Override
  public void initialize(Deadline constraintAnnotation) {
  }

  @Override
  public boolean isValid(Temporal value,
    ConstraintValidatorContext context) {

    if (value == null || value.getDeadline() == null) {
      return false;
    } else {
      Date deadline = value.getDeadline();
      if (new Date().before(deadline)) {
```

```
            return true;
        } else {
            return false;
        }
    }
  }

}
```

This design is sufficient for our purpose because if we combine the result of the deadline validator with the result of another constraint with a logical 'OR' we get the desired behaviour: A constraint can be violated before the deadline and when the deadline is reached the outcome is determined solely by the validation of the other constraint. In table 17.1 we list the four possible combinations of the validation outcome of the deadline validator and any other validator to prove the correctness of our approach. Constraint composition is explained in section 17.2.2.

D	A	$D \vee A$
0	0	0
0	1	1
1	0	1
1	1	1

Table 17.1: All possible outcomes of a deadline constraint combined with another constraint where $D = 0$ stands for 'deadline expired' and $D = 1$ means 'deadline not expired'. Furthermore, A stands for any other constraint not equal to the deadline constraint where $A = 0$ means constraint A is not fulfilled and $A = 1$ has the meaning that A is fulfilled.

Finally, the deadline validator specifies that only objects of type `Temporal` can be validated (see interface implementation in listing 17.2) which means that the `@Deadline` annotation and any composition including it can only be used on the declared type `Temporal`. This mechanism ensures that the deadline validator has access to the specified deadline with the help of the getter method defined in the interface `Temporal` which must be implemented by a client class in a correct way. The complete source code is available in the appendix section A.5.2.

17.2.2 CONSTRAINT COMPOSITION

Constraint composition is a feature of the JSR 303/349 (Bean Validation 1.0/1.1) which allows a developer to create a new constraint based on existing ones (see section 10.1.2). The main idea is the composition of existing constraints under a new name to avoid a lot of annotations on a single field and make the reuse of constraints easier. Composed constraints are always validated individually and the result is combined with a logical 'AND'. As explained in section 17.2.1.2 we would like to combine the validation results with a logical 'OR' to make a reuse of the `@Deadline` constraint possible. Therefore, we use the additional functionality of Hibernate Validator (see section 10.1.1.1) which allows the Boolean composition of constraints which is not part of the standard. We explain with the help of two examples how to support temporal constraints using constraint composition.

17.2.2.1 `@AssertFalseOnDeadline` CONSTRAINT

The constraint `@AssertFalse` has according to Hibernate Validator (see 10.1.1.1) the behaviour to 'check(s) that the annotated element is false'. Now we try to combine this behaviour with a temporal dimension so that the annotated element must be false only after a specified deadline. Therefore, we create a new constraint using a custom annotation that is called `@AssertFalseOnDeadline` which is depicted in listing 17.3. We use the Boolean constraint composition feature of Hibernate Validator with the attribute value `CompositionType.OR` to combine the result of the individual constraints with a logical 'OR'.

Listing 17.3: The `@AssertFalseOnDeadline` constraint using constraint composition.

```
@ConstraintComposition(CompositionType.OR)
@Deadline
@Target({ METHOD, FIELD, ANNOTATION_TYPE })
@Retention(RUNTIME)
@Documented
@Constraint(validatedBy = { AssertFalseOnDeadlineValidator.class })
@ReportAsSingleViolation
public @interface AssertFalseOnDeadline {

  @OverridesAttribute(constraint = Deadline.class, name = "payload")
  Class<? extends Payload>[] payload() default {};

  @OverridesAttribute(constraint = Deadline.class, name = "message")
  String message() default "must be false after the deadline";

  @OverridesAttribute(constraint = Deadline.class, name = "groups")
  Class<?>[] groups() default {};

}
```

Next, we specify that the `@Deadline` constraint should be comprised in the `@AssertFalseOnDeadline` constraint by simply adding the annotation. We cannot reuse the built-in constraint `@AssertFalse` because it can only be applied to the types `boolean` and `Boolean` but the `@AssertFalseOnDeadline` constraint should be applicable for the types conforming to `TemporalBoolean` which is not a subtype of `Boolean`. In general, a constraint comprising several other constraints can only be applied to a type conforming to every individual constraint. Therefore, we create a validator class `AssertFalseOnDeadlineValidator` to implement the behaviour of the `AssertFalse` constraint which is depicted in listing 17.4. Finally, we specify that violations of the individual constraints should be reported as a single violation using the `@ReportAsSingleViolation` annotation and override the individual attributes to share attribute values among the comprising constraints.

Listing 17.4: The `AssertFalseOnDeadlineValidator` class copies the behaviour of the `@AssertFalse` validator which checks if the passed value is equal to false.

```
public class AssertFalseOnDeadlineValidator implements
  ConstraintValidator<AssertFalseOnDeadline, TemporalBoolean>{
```

```
  @Override
  public void initialize(
    AssertFalseOnDeadline constraintAnnotation) {
  }

  @Override
  public boolean isValid(
    TemporalBoolean value, ConstraintValidatorContext context) {

    return value == null || !value.isValue();
  }

}
```

Finally, we can use the new constraint on every type conforming to `TemporalBoolean` as specified in the `AssertFalseOnDeadlineValidator` class. During the validation process, the individual constraints are validated with the corresponding values (i.e. the deadline and the Boolean value) and the results will be combined using the logical 'OR'. If the result is equal to `false` a constraint violation is generated with the corresponding information (e.g. violation message).

Listing 17.5: Usage example of the `@AssertFalseOnDeadline` constraint.

```
public class Person {

  @AssertFalseOnDeadline
  private TemporalBoolean isRegistered;

  public Person() {
  }

  // Getter and setter methods

}
```

The complete source code of this constraint can be found in appendix section A.5.2.

17.2.2.2 **@MinOnDeadline** CONSTRAINT

The composition of the `@Min` constraint with the `@Deadline` constraint is much easier than for the `AssertFalseOnDeadline` constraint described in 17.2.2.1 because our data structure representing numbers inherits from the class `Number` (see section 17.1.2) and the constraint `Min` is applicable to types conforming to `Number`. Therefore, we compose the `@Deadline` and `Min` constraint to build the constraint `@MinOnDeadline` using the composition type 'OR' and the single report annotation as described in 17.2.2.1. We do not need a validator class and therefore the `validatedBy` attribute value can be set to an empty array as depicted in listing 17.6.

Listing 17.6: The `@MinOnDeadline` constraint using constraint composition without any additional validator class.

```
@ConstraintComposition(CompositionType.OR)
@Deadline
```

```
@Min(value = 0)
@Target({ METHOD, FIELD, ANNOTATION_TYPE })
@Retention(RUNTIME)
@Documented
@Constraint(validatedBy = { })
@ReportAsSingleViolation
public @interface MinOnDeadline {

  ...

  @OverridesAttribute.List({
    @OverridesAttribute(
      constraint = Deadline.class, name = "message"
    ),
    @OverridesAttribute(
      constraint = Min.class, name = "message"
    )
  })
  String message() default "may not be null after the deadline";

  ...

  @OverridesAttribute(constraint=Min.class, name="value")
  long value();

}
```

As for the `@AssertFalseOnDeadline` constraint we overwrite the attributes using a list of `@OverrideAttribute` annotations to share the attribute values among both comprising constraints. Finally, we delegate the non-optional attribute value specification of the `@Min` constraint by overriding it to the `@MinOnDeadline` constraint. Note, the 'dummy' value `@Min(value = 0)` in the upper part of the constraint definition does not have any effect, because we override the attribute but a value must be specified to be able to compile. The complete source code of this constraint can be found in appendix section A.5.2.

17.2.3 TEMPORAL CONSTRAINT CREATION

We summarise the most important steps to create a temporal constraint:

1. Take an already existing constraint C (not equal to the `@Deadline` constraint) that should be extend by a temporal dimension
2. Check the applicable type of C and check whether it conforms to one of the types of the data structure described in 17.1
    - **Yes**: Compose constraint C as described in section 17.2.2.2 with the `@Deadline` constraint (e.g. applicable type of `@Min` constraint is `Number` which conforms to `TemporalNumber`)
    - **No**: Check if a primitive temporal data type T corresponds to the applicable type of C (e.g. the applicable type of `@AssertFalse` is `Boolean` and `boolean` which does not conform but corresponds to `TemporalBoolean`) exists

- **Yes**: Compose constraint C as described in section 17.2.2.1 with the `@Deadline` constraint and create a validator class restricting the applicability of the composed constraint to T
- **No**: Extend the data structure as described in 17.1.4 and follow the instructions at point 2

18 TIME-TRIGGERED VALIDATION COMPONENT

Constraint validation usually happens at a certain execution point within an application as defined by the developer such as the dispatch of form data. But how to handle a nightly based validation task? The Time-Triggered Validation Component (TTVC) offers the possibility to define a validation job with a certain trigger point by a developer. This functionality gives a developer the opportunity to schedule tasks at various points in time without explicitly starting the task because the TTVC will check for the right start time. Furthermore, the developer can integrate this component to build a front-end where the user can manage time-triggered validation tasks. The implementation of the TTVC is based on Bean Validation 1.1 (see section 10.1), JPA 2.1 to store the optional validation report and the open source job scheduling library Quartz[1].
This chapter describes the individual subcomponents of the TTVC in detail and shows some examples how to use it.

18.1 TTVC: SCHEDULERS

The scheduler is the central subcomponent which manages all relevant information to start jobs with individual triggers. Our implementation provides the class `JPAValidationScheduler` which uses the default scheduler of Quartz[2] for the main work and provides some additional features. First, it is possible to specify the name of a JPA persistence unit (see section 12.4) which is then used to communicate with the underlying data store. The declaration of a persistence unit is mandatory if the JPA validation job is used. Next, the `JPAValidationScheduler` provides a method to set up the underlying data store with all relevant tables and indexes by passing the path to the corresponding SQL

[1] `http://quartz-scheduler.org/`, [Online; accessed 24-September-2013]
[2] See footnote 1

script. The interface view[3] is depicted in listing 18.1 and an example SQL script for MySQL can be found in the appendix section A.6.1.

Listing 18.1: The interface view[4] of the `JPAValidationScheduler` class which handles the execution of planned jobs.

```
public class JPAValidationScheduler {

  public static String persistenceUnitName;

  public JPAValidationScheduler(String persistenceUnitName)
    throws SchedulerException {
  }

  public Scheduler getScheduler() {
  }

  public void setScheduler(Scheduler scheduler) {
  }

  public void setupQuartzDB(String sqlScript)
    throws FileNotFoundException, IOException, SQLException {
  }

}
```

A usage example can be found in the appendix section A.6.1 and a more detailed description of the Quartz scheduler can be found on the project website[5], [Online; accessed 27-September-2013].
Note that the execution of the SQL scripts for MySQL must be done with care: the table creation script is idempotent but the index script is not. Running the index script twice will generate exceptions because the indexes do already exist and an instruction to check for existence is not available for MySQL (but a workaround[6] is possible but cumbersome).
Finally, the Quartz scheduler must be configured before launching. This can be done with the help of a property file as depicted in listing 18.2 or with the help of the API as shown in the source code which can be found in the appendix section A.6.1.

Listing 18.2: The content of a property file with some configuration examples. It is possible to store the jobs in RAM using the value `org.quartz.simpl.RAMJobStore` or in a data store when the value `org.quartz.impl.jdbcjobstore.JobStoreTX` is set. If a data store is used further configuration is necessary (e.g. data store connection information)

```
org.quartz.scheduler.instanceName = dqmfScheduler
org.quartz.threadPool.threadCount = 3
#org.quartz.jobStore.class = org.quartz.simpl.RAMJobStore
org.quartz.jobStore.class = org.quartz.impl.jdbcjobstore.JobStoreTX
org.quartz.jobStore.driverDelegateClass =
```

[3] A view of a class that shows only fields and methods that are accessible by clients of that class without any implementation details.

[4] See footnote 3

[5] `http://quartz-scheduler.org/documentation/quartz-2.2.x/cookbook/`

[6] `http://dba.stackexchange.com/questions/24531/mysql-create-index-if-not-exists`, [Online; accessed 27-September-2013]

```
  org.quartz.impl.jdbcjobstore.StdJDBCDelegate
org.quartz.jobStore.tablePrefix = QRTZ_
org.quartz.jobStore.dataSource = dqmf

org.quartz.dataSource.dqmf.driver = com.mysql.jdbc.Driver
org.quartz.dataSource.dqmf.URL =
  jdbc:mysql://localhost:3306/jpa_examples
org.quartz.dataSource.dqmf.user = root
org.quartz.dataSource.dqmf.password = Hc25HwztzNtcwDVE
org.quartz.dataSource.dqmf.maxConnections = 20
```

Configuration details can be found on the project website[7] of Quartz and the complete source code with examples showing the usage of the `JPAValidationScheduler` and default Quartz scheduler can be found in the appendix section A.6.1.

18.2 TTVC: JOBS AND JOBDETAILS

18.2.1 TTVC: BASIC JOB

When using Quartz[8], a job can be defined as a unit of work which can be executed. A job is realised by implementing the `Job` interface of Quartz directly or by extending one of the abstract classes as shown in figure 18.1. The `Job` interface as described on the Quartz project website[9] is the basic way to define a job. An example is given in the appendix section A.6.2.1. Job instances are indirectly created using a `JobDetail` which can be created with the help of the builder class `JobBuilder` as depicted in listing 18.3 for the basic job.

Listing 18.3: The `JobBuilder` class implements the builder pattern to create objects of type `JobDetail` which represent a job instance (e.g. `BasicValidationJob`).

```
JobDetail basicValidationJob =
  JobBuilder.newJob(BasicJob.class)
    .withIdentity("bj-01", "ttvc-tests-jobs")
    .storeDurably(false)
    .usingJobData(persons)
    .build();
```

18.2.2 TTVC: ABSTRACT VALIDATION JOB

The `AbstractValidationJob` class provides basic features to realise a job which can validate objects using a JSR 303/349 (Bean Validation 1.0/1.1) implementation like Hibernate Validator (see section 10.1.1.1). The main feature is the `getValidationViolations` method which uses the Bean Validation implementation to validate the object passed as an argument and returns a set of constraint violations. The complete source code and an example is available in the appendix section A.6.2.2.

[7] `http://quartz-scheduler.org/documentation/quartz-2.2.x/configuration/`, [Online; accessed 27-September-2013]

[8] See footnote 1

[9] `http://quartz-scheduler.org/documentation/quartz-2.2.x/cookbook/DefineJobWithData`, [Online; accessed 27-September-2013]

18.2.3 TTVC: ABSTRACT JPA VALIDATION JOB

To validate objects in a data store one can extend the `AbstractJPAValidationJob` class which provides basic methods to access objects in a data store using JSR 317/338 (JPA 2.0/2.1). There are three central methods which can be used: the `executeFetchOperation`, the `getConstraintViolations` and the `getFetchedObjects`. The first method executes the retrieval operation using either the entity class and the id or the query. It stores the retrieved objects and the constraint violations (which might occur during the JSR 317/338 (JPA 2.0/2.1) validation process, see section 12.4.2 and 12.4.1) which can then be accessed using the other two methods. Calling the `executeFetchOperation` method several times will overwrite the previous results every time. Figure 18.2 shows the usual steps that have to be done to use the functionality of this job.
Jobs which extend the `AbstractJPAValidationJob` class can make use of the `JPAValidationJobBuilder` class which uses the builder pattern[10] to create objects. The `JPAValidationJobBuilder` class can create objects with the following properties:

- **Job detail id:** Unique job identifier among all accessible jobs in the RAM or data store within a job detail group (mandatory)
- **Job detail group:** Specify a group to arrange jobs (default: `"SingleObjectValidation"`)
- **Job description:** Textual explanation (default: `"Validate objects fetched with a given entity class + id or with a JPQL query using the validate() method of a Bean Validation implementation."`)
- **Job persistence:** Boolean value which defines whether a job is stored after completion (default: `true` which means that the job is stored)
- **Validation job class:** Class name which extends the `AbstractJPAValidationJob` class (default: `UniversalJPAValidationJob.class`)

Moreover, another builder class can be used to generate objects which store entity data to retrieve objects from the data store and validation data to configure validation method. The `EntityValidationData` class stores the following information:

- **Entity id:** Identity of an object stored in a data store (i.e. the primary key of an object)
- **Entity class:** The class name of the object which is stored in the data store (i.e. the table name)
- **JPQL query:** A query (i.e. `SELECT` statement) specifying a set of objects which should be fetched from the data store
- **Validation groups:** Bean Validation groups for constraint selection

[10] `http://en.wikipedia.org/wiki/Builder_pattern`, [Online; accessed 28-September-2013]

- **Property name:** Name of a getter method to validate only this property/getter method and not the whole object

Note that the builder pattern of `EntityValidationData` is designed in such a way that is only possible to use either an entity class and an entity id or a JPQL query but not both at the same time. There are no default values for any information. The complete source code of the `AbstractJPAValidationJob` class, the `JPAValidationJobBuilder` class and the `EntityValidationData` can be found in the appendix section A.6.2.3.
The code expert in listing 18.4 shows a usage example which is visualised in figure 18.3. A `JPAValidationJobBuilder` instance stores information about the job such as the job description and the corresponding job class (e.g. `BasicJPAValidationJob.class`) which is used to build a Quartz `JobDetail`[11]. Furthermore the instances stores a list of `EntityValidationData` objects which define what and how objects are retrieved from the data store. There are two possibilities: either one retrieves a single object by defining the entity class and a corresponding id (see `entityIdbuilder` and first list entry in the bottom part of figure 18.3) or it is possible to define a JPQL query (`SELECT` statement) to retrieve one or more objects (see `queryBuilder` and second list entry in the bottom part of figure 18.3). Afterwards, when calling the `build()` method of the `JPAValidationJobBuilder` class a `JobDetail` object with all the defined information is generated. The `EntityValidationData` is stored in the `JobDataMap`[12] with the key `"EntityValidationData"` (see left bottom corner in figure 18.3) which can then be used in the `execute` method of a `Job` definition.

Listing 18.4: Code excerpt of a usage example of the `JPAValidationJobBuilder`, the `EntityValidationData` builder to create a `BasicJPAValidationJob`.

```
...
EntityValidationData.Builder entityIdbuilder =
  new EntityValidationData.Builder(PersistentAddress.class);

EntityValidationData.Builder queryBuilder =
  new EntityValidationData.Builder(
    "SELECT a FROM PersistentAddress a WHERE a.id = " + adr.getId()
  );

basicJPAValidationJob = new JPAValidationJobBuilder(
  "bjpavj-01",
  entityIdbuilder.withEntityId(address.getId()).build())
  .addEntityValidationData(queryBuilder.build())
  .withJobPersistence(false)
  .withJobDescription(
    "Validate the same object twice: ...")
  .withJobDetailGroup("ttvc-tests-jobs")
  .withValidationJobClass(BasicJPAValidationJob.class)
  .build();
...
```

[11] `http://quartz-scheduler.org/documentation/quartz-2.2.x/tutorials/tutorial-lesson-03,[Online;accessed28-September-2013]`
[12] See footnote 9

The complete source code including a usage example is available in the appendix section A.6.2.3.

18.2.4 TTVC: UNIVERSAL JPA VALIDATION JOB

The class `UniversalJPAValidationJob` is an example implementation of an abstract JPA validation job and therefore extends the `AbstractJPAValidationJob` class. This job implementation can be used out of the box with the following features:

- Process a list of `EntityValidationData` as described in 18.2.3
- Fetch data from a data store using JPA and the information of the `EntityValidationData`: retrieve a single object with an entity class and id or use JPQL to specify a query
- Validate the whole object or just a specific property (i.e. field or getter method)
- Specify constraint groups which are considered during validation
- Persistent validation results (see section 18.5)

The only requirement for the usage is that the correct persistence unit name of the `JPAValidationScheduler` is set. Automatic Bean Validation (as described in section 12.4.2 and 12.4.1) must be activated within the persistence unit. A full example and the complete source code for the universal JPA validation job can be found in the appendix section A.6.2.4.

18.3 TTVC: TRIGGERS

A trigger within the Quartz[13] library is defined as follows:

'Triggers are the "mechanism" by which Jobs are scheduled. Many Triggers can point to the same Job, but a single Trigger can only point to one Job.'

There are two possibilities how to connect a trigger and a job: Either one passes a `JobDetail` instances to a `Trigger` definition or the scheduler instance combines a `JobDetail` with a `Trigger`. Listing 18.5 shows both possibilities with one `JobDetail` instance, two `Trigger` instances and the default scheduler instance. Note that a trigger using the `forJob` method can only access jobs that are durable and already stored (which happens if `scheduleJob(JobDetail, Trigger)` was executed).

Listing 18.5: Code excerpt which shows how to connect a `JobDetail` (e.g. `basicJob`) with a `Trigger` (e.g. `basicTrigger`).

```
...

basicTrigger = newTrigger()
  .withIdentity("bt-01", "ttvc-tests-triggers")
```

[13] See footnote 1

```
  .build();

basicTrigger2 = newTrigger()
  .withIdentity("bt-02", "ttvc-tests-triggers")
  .forJob(basicJob)
  .build();

...

sched.scheduleJob(basicJob, basicTrigger);
sched.scheduleJob(basicTrigger2);

...
```

A full example showing both possibilities can be found in the appendix section A.6.3. Moreover, we have defined two specific triggers which can be used out of the box by creating an instance with the two builder classes `UnrepeatableTriggerBuilder` and `DailyTriggerBuilder`. The first builder produces a trigger which fires a job exactly once at a certain point in time and the second trigger fires every day at a specific time. A usage example can be found in the appendix section A.6.3 and a detailed description is available on the project website[14] of Quartz.

18.4 TTVC: JOB LISTENER

A job listener observes the state of a job and is called if certain events occur. For Quartz[15] it is possible to define a job listener by implementing the `JobListener` interface which has the following abstract methods:

- `public String getName()`
- `public void jobToBeExecuted(JobExecutionContext context)`
- `public void jobExecutionVetoed(JobExecutionContext context)`
- `public void jobWasExecuted(JobExecutionContext context, JobExecutionException jobException)`

The most important ones are the second and fourth method which are called if a job has just started or has just finished its work respectively. A detailed description of the interface is available on the project website of Quartz[16] and an example can be found in the appendix section A.6.4 which shows how to use a job listener with a scheduler instance.

[14] `http://quartz-scheduler.org/documentation/quartz-2.2.x/tutorials/`, [Online; accessed 30-September-2013]

[15] See footnote 1

[16] `http://www.quartz-scheduler.org/documentation/quartz-2.2.x/tutorials/tutorial-lesson-07`, [Online; accessed 30-September-2013]

18.5 TTVC: PERSISTENT VALIDATION REPORT

The persistent validation report is a data structure which offers the possibility to store the result of a validation job in a persistent way using JPA. Therefore, the UML class diagram depicted in 18.4 shows the most important parts of the data structure. The class `ValidationJobInformation` can store the job id, the job group and the validation date time of a validation job. Next, a `ValidationJobInformation` instance is composed of one or more `QueryInformation` instances which stores the retrieval mode. Either the objects are fetched with an entity class and and entity id or with a JPQL query from the data store. Finally, each `QueryInformation` object is made up of one or more `ConstraintViolationInformation` instances. A `ConstraintViolationInformation` instance stores the error message, the invalid value, the element name (i.e. field or getter method which was annotated) and a list of constraint groups.
The universal JPA validation job described in section 18.2.4 uses the persistent validation report. The source code is available in the appendix section A.6.2.4 and a screenshot of a persistent validation result is depicted in figure 18.5.
The full source code can be found in the appendix section A.6.5 including a usage example.

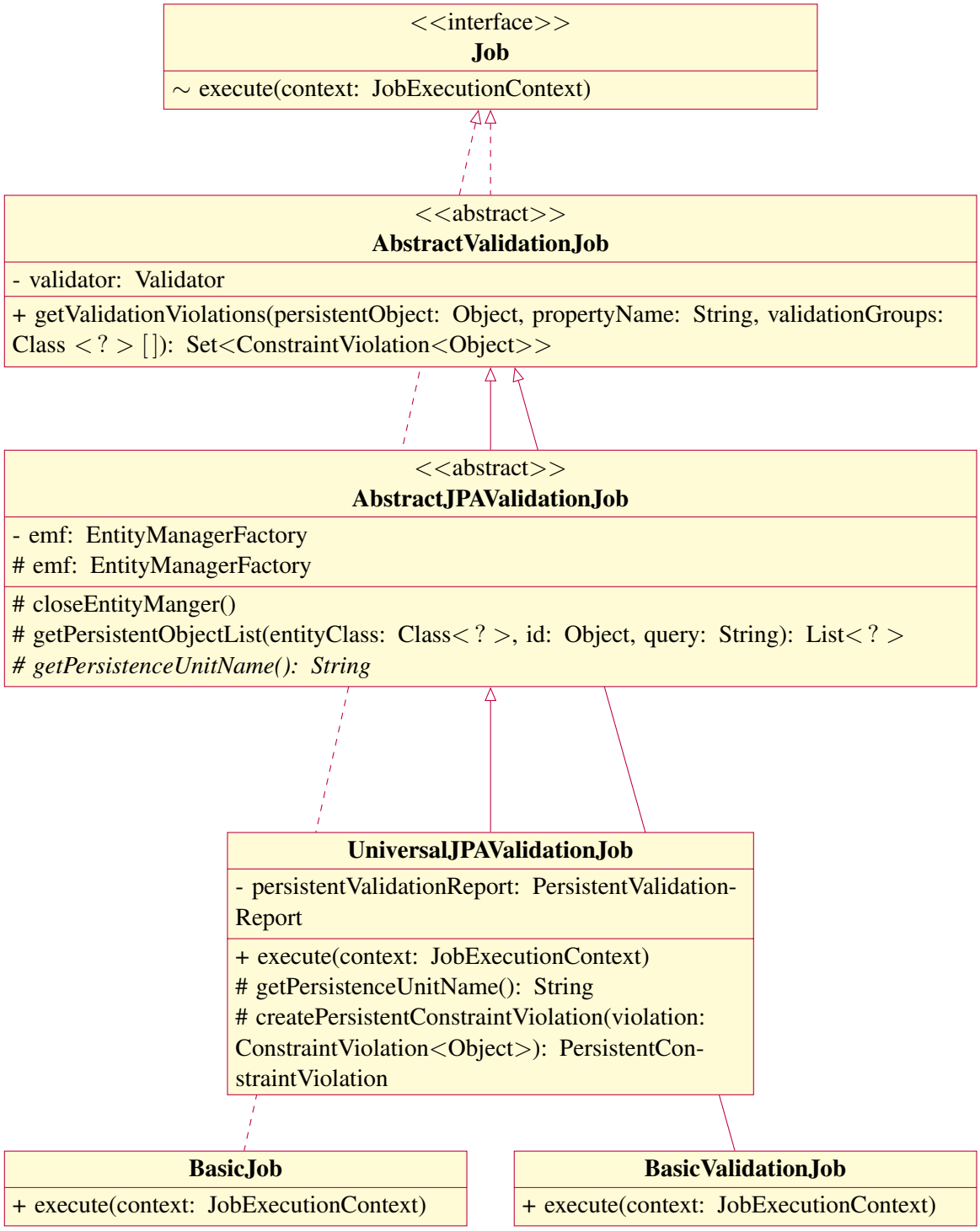


Figure 18.1: UML class diagram showing the different possibilities to define a job. The `UniversalJPAValidationJob` class is a concrete example of a job implementation.

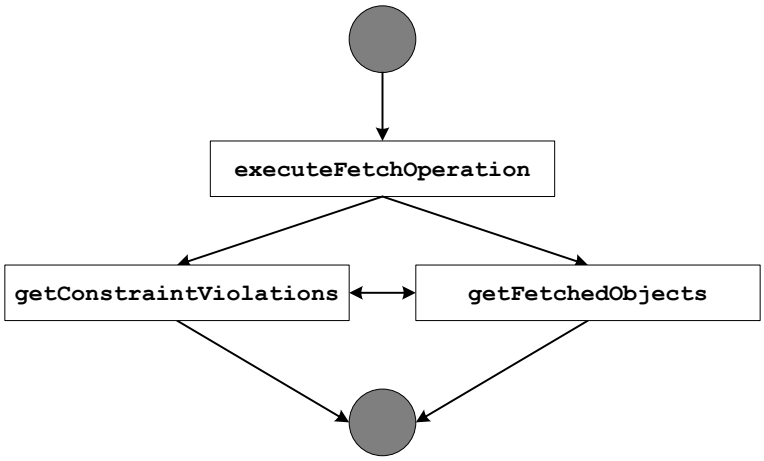


Figure 18.2: The first step is executed by calling the `executeFetchOperation` method. Next, it is possible to call the `getConstraintViolations` method to retrieve the set of constraint violations or the `getFetchedObjects` method to access the retrieved objects.

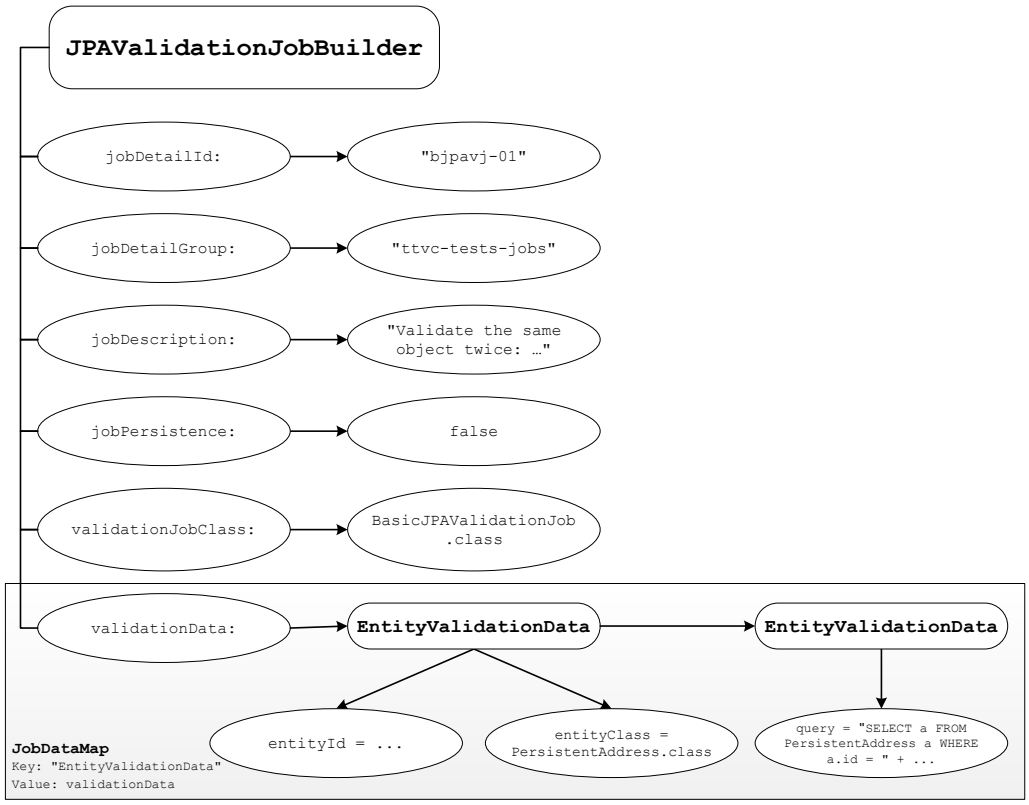


Figure 18.3: Instance of a possible `JPAValidationJobBuilder` state storing all relevant information and a list of `EntityValidationData` objects which represent the information to retrieve object from the data store using JPA features.

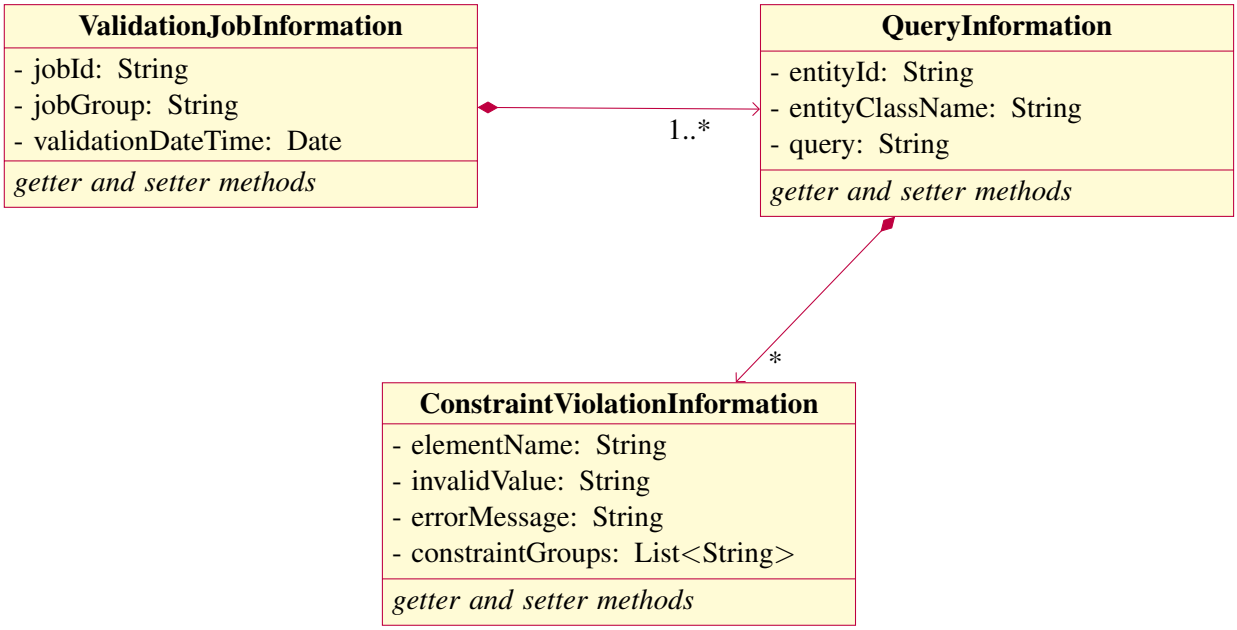


Figure 18.4: UML class diagram showing the validation report data structure which can be used to persist a validation result using JPA.

RVJI_ID	JOB_GROUP	JOB_ID	VALIDATION_DATE_TIME
51	ttvc-tests-jobs	bjpavj-01	2013-10-01 13:37:47
60	ttvc-tests-jobs	bjpavj-02	2013-10-01 13:37:47
53	ttvc-tests-jobs	bjpavj-03	2013-10-01 13:37:48
56	ttvc-tests-jobs	bjpavj-04	2013-10-01 13:37:48

QI_ID	ENTITY_CLASS_NAME	ENTITY_ID	QUERY	VALIDATION_JOB_INFORMATION
52	PersistentAddress	1	NULL	51
54	PersistentPerson	2	NULL	53
57	PersistentPerson	2	NULL	56
59	PersistentAddress	1	NULL	56
61	NULL	NULL	SELECT a FROM PersistentAddress a WHERE a.id = 1 O...	60

RCVI_ID	ELEMENT_NAME	ERROR_MESSAGE	INVALID_VALUE	CONSTRAINT_VIOLATION_INFORMATION
55	address	not specified, but not mandatory	NULL	54
58	address	not specified, but not mandatory	NULL	57
62	street	street is mandatory	NULL	61
63	streetNumber	must be greater than or equal to 1	-1	61

RCVI_ID	CONSTRAINTGROUPS
55	Info
58	Info
62	Default
63	Default

Figure 18.5: Screenshot after the execution of the universal JPA validation job example of the persistent validation report using MySQL as a data store.

19 HARD AND SOFT CONSTRAINTS

According to the JSR 349 (Bean Validation 1.1), which is described in section 10.1.1, a constraint is defined as 'a restriction on a bean instance, the value of a field or the value of a JavaBean property (i.e. a getter method)' and a constraint validation is a 'constraint logic algorithm used to determine whether a given value passes a constraint or not' (see table A.1 in [32]). In this chapter, we draw on these definitions and integrate them into the context of a hard and soft constraint. Furthermore, we show how to implement hard and soft constraints using Bean Validation.

19.1 DEFINITION

19.1.1 HARD CONSTRAINT

Looking into the area of Linear Programming (LP) we can find the concept of a hard and soft constraint as described in [49]. Within LP we are given a set of linear inequalities (usually written as $\mathbf{Ax} \leq \mathbf{b}$) and an objective function (denoted as $\mathbf{c}^\intercal\mathbf{x}$) which have to be maximised over all possible $\mathbf{x}$. As stated in [49], every linear inequality defines a feasible region of possible solutions.
For instance, if the input space is $\mathbb{R}^2$ then the linear inequalities will build a two dimensional convex region with possible solutions as depicted in figure 19.1[1]. These kind of constraints are usually called 'hard constraints'.
Summarising, as described in [50], hard constraints must be satisfied and often occur in real-life scenarios. An example from [50] about time-tabling problems ('teacher can teach only one class at a time') shows the existence of hard constraints in real-life scenarios. Having these insights, we can formulate the definition of hard constraints in the scope of this thesis

[1] `http://en.wikipedia.org/wiki/File:Linear_Programming_Feasible_Region.svg`, [Online; accessed 04-September-2013]

[2] `http://en.wikipedia.org/wiki/Linear_programming`, [Online; accessed 04-September-2013]

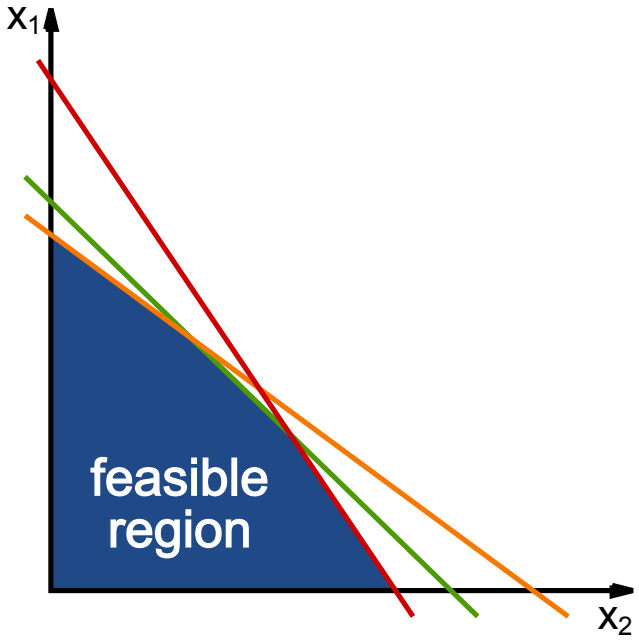


Figure 19.1: 'In a linear programming problem, a series of linear constraints produces a convex feasible region of possible values for those variables. In the two-variable case this region is in the shape of a convex simple polygon.'[2]

as:

Definition 4 (*Hard Constraint*)

A hard constraint (HC) is a restriction on a bean instance, the value of a field or the value of a JavaBean property (i.e. a getter method) that can be validated against a given value with the two possible outcomes of the validation. The outcome is `True` *if the given value fulfils the hard constraint and* `False` *otherwise. For short:*

$$validate\,(HC, value) = \begin{cases} True, & \text{if } value \text{ fulfils } HC \\ False, & \text{otherwise} \end{cases}$$

19.1.2 SOFT CONSTRAINT

Similar as for the hard constraint (see 19.1.1), we take a look at the area of solving a Constraint Satisfaction Problem (CSP) and LP. As stated in [50] a soft constraint in a CSP environment 'should be satisfied, but may be violated'. Like for a hard constraint, a soft constraint is a real-life incident and [50] gives an example in the area of time-tabling problems:

'A request such as "the schedule of teacher should be concentrated in two days" is simply a preference, but not essential for the solution.'

In LP, the notes of [49] define soft constraints as 'a penalty on certain assignments rather than prohibiting them'. This means, that we have a set of functions (called penalty functions) which put a penalty on the solutions. The object function in LP (see 19.1.1) is modified including the sum of the penalties. Therefore, we can view a hard constraint as a special case of a soft constraint where the penalty function assigns to a solution which violates a hard constraint a very high penalty (i.e. ∞) and no penalty if the solution fulfils the hard constraint.
With these observations, we are now ready to formulate the following definition:

Definition 5 (*Soft Constraint*)

A soft constraint (SC) is a restriction on a bean instance, the value of a field or the value of a JavaBean property (i.e. a getter method) that can be validated against a given value. A value can violate a soft constraint making the whole validation process still valid. Violating a soft constraint can be penalised. For short:

$$validate\,(SC, value) = \begin{cases} True, & \text{if } value \text{ fulfils } SC \\ False, & SC \text{ is violated and no penalty} \\ P, & SC \text{ is violated and penalty } P \text{ added} \end{cases}$$

19.1.3 SUMMARY

We can summarise the concept of hard and soft constraints with a metaphor and a figure (depicted in 19.2)[3]:

'Hard constraints are those which we definitely want to be true. These might relate to the successful assembly of a mechanism. Soft constraints are those we would like to be true - but not at the expense of the others. These might say that a mechanism must follow a given path. There is not point in trying to match every point exactly if this can only be done by breaking the assembly of the links.'

In addition to this metaphor, soft constraints are not only constraints which are nice to be fulfilled but a penalty score can be assigned to every soft constraint telling how much we have to 'pay' if we do not fulfil the constraint.

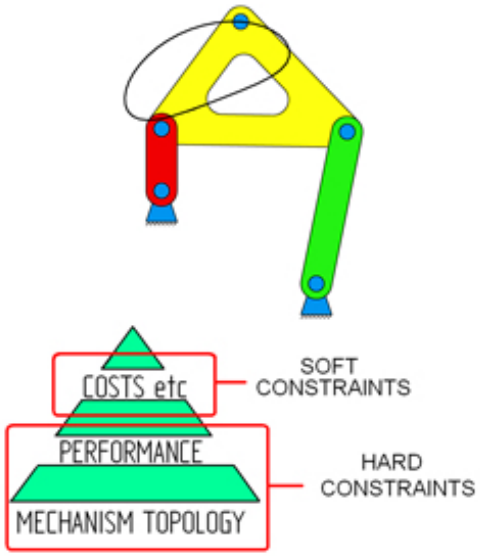


Figure 19.2: Visualising the concept of hard and soft constraints in the area of mechanical engineering.[4]

19.2 IMPLEMENTATION

This section shows a possible implementation of the definitions given in section 19.1 within the context of this thesis using Bean Validation 1.1 (see section 10.1.1).

[3] `http://www.bath.ac.uk/mech-eng/constraintmodelling/HardSoft.html`, [Online; accessed 05-September-2013]

[4] See footnote 3

19.2.1 HARD CONSTRAINT IMPLEMENTATION

The definition for hard constraints as given in 19.1.1 can be smoothly adapted to the concept of constraints in the JSR 349 (Bean Validation 1.1). Declaring a constraint using Bean Validation 1.1 and result only in two possible outcomes during validation. If the value fulfils the constraint, no validation violation occurs and the corresponding Bean Validation method returns `True`. Otherwise, it returns `False`. This behaviour is identical to the one we defined in section 19.1.1 and therefore, every constraint that is declared using Bean Validation is automatically a hard constraint.

19.2.2 SOFT CONSTRAINT IMPLEMENTATION

Due to the fact that every declared constraint using Bean Validation 1.1 is automatically a hard constraint (see section 19.2.1) we propose three solutions to simulate a soft constraint using Bean Validation 1.1.

19.2.2.1 PAYLOAD-TRY-CATCH METHOD

According to the JSR 349 (Bean Validation 1.1) [32] every constraint declaration must have a `payload` element which has the type `Class<? extends Payload>[]`. The purpose for the payload element according to the standard is the association of meta data which can be used for instance to add a severity to a constraint which can be displayed to the user in the front-end. Using this idea, we can simulate a soft constraint using a severity class with two levels as depicted in 19.1.

Listing 19.1: Severity class defining the two levels `Info` and `Warning` by extending the `Payload` interface.

```
public class Severity {

  public interface Info extends Payload {
  };

  public interface Warning extends Payload {
  };

}
```

Then we can categorise each constraint on whether it belongs to a hard or soft constraint. Adding a payload that is not equal to `Info` or `Warning` means that the constraint is a hard constraint (this definition includes the case where no payload is added) and otherwise it is a soft constraint. An example is given in listing 19.2. Using this concept, it is possible to add more severity levels by adding fields which extend the `Payload` interface in the `Severity` class.

Listing 19.2: The domain model `Person` uses a hard constraint for the field `name` and a soft constraint (`Info`) for the field `address`.

```
public class Person {

  @NotNull(message = "name is mandatory")
  private String name;
```

```
  @NotNull(message = "address is nice to have",
    payload = {Severity.Info.class})
  private String address;

  // Getter and setter methods

}
```

Now we describe how to use this concept to make the validation process successful even if some soft constraints are violated. We can use the usual validation process provided by a Bean Validation 1.1 implementation (e.g. Hibernate Validator, see section 10.1.1.1) which may return a set of constraint violations. Next, we can check every constraint violation if the violated constraint is a soft constraint (read out the payload) and if the set only contains violations of soft constraints we can either map each severity level to a penalty score, sum up and return it or we simply mark the validation process as successful. Afterwards, we can check if the validation process was successful, display the penalty score and execute further steps depending on the outcome (e.g. persisting the entity if validation was successful).
Furthermore, within a JPA 2.0/2.1 environment (see section 12.4.2 and 12.4.1) it is possible to catch the `ConstraintViolationException` and do the same case distinction. Afterwards, we can disable the automatic validation during persistence, storing the entities of the successful validation and then turn on the automatic validation again. A detailed example and the full source code can be found in the appendix section A.7.1.1.

19.2.2.2 SOFT CONSTRAINTS VALIDATOR

To get rid of the analysis of the constraint violation set (see section 19.2.2.1) in every validation situation in an application, it is possible to integrate the analysis in a custom validator (e.g. a soft constraints validator) which can then be used in the whole application. We omit any details, because the implementation can depend on the chosen Bean Validation implementation (e.g. Hibernate Validator, see section 10.1.1.1) and can be cumbersome. Furthermore, within a JPA 2.0/2.1 environment (see section 12.4.2 and 12.4.1) the delegation process to the appropriate Bean Validation implementation has to be adapted with respect to soft constraints. The source code in the appendix section A.7.1.2 shows a custom soft constraints validator for the JSR 338 (JPA 2.1) reference implementation (EclipseLink) and a usage example.

19.2.2.3 GROUP METHOD

The group method relies on the fact that every constraint has a group element of type `Class<?>[]` with an empty array as the default value according to the JSR 349 (Bean Validation 1.1) chapter 3.1.1.2. Furthermore, if no group is specified the Bean Validation implementation considers the constraint to be part of the `Default` group. The basic idea is the usage of a special group for soft constraints. For the validation process, the validation API described in the chapter 5.1.3 of the JSR 349 (Bean Validation 1.1) offers the possibility to pass an optional group array argument which specifies the validation of all constraints that belong to one of the passed groups. If no argument is passed the `Default` group is automatically used. To summarise the idea, we do the following steps:

1. Define a group data structure to represent soft constraints
2. All constraints which are hard constraints are not declared with a group from (1) (e.g. no group i.e. the `Default` group)
3. The validation process is done in two ways:
   (a) Validate objects with respect to hard constraints by calling the validation API without a group from (1) as an argument
   (b) Validate objects with respect to soft constraints by calling the validation API with a group from (1) as an argument

The group data structure should consist of a 'main' group `SoftConstraint` which represents every type of soft constraint. To differentiate how important the fulfilment of a soft constraint is we provide the two subgroups `Info` and `Warning`. Providing a certain level of flexibility and comfort we design the data structure in a way to pass the 'main' group `SoftConstraint` as an argument to the validation API which should then has the effect that every constraint belonging to the group `SoftConstraint`, `Info` or `Warning` should be considered. The straightforward solution depicted in 19.3 in which we make the interfaces `Info` and `Warning` a subclass of the interface `SoftConstraint` does not work. Subclassing means that we specialise a class and this structure can be shown best with a Venn diagram (see left part of figure 19.3). If we now validate an entity passing the `SoftConstraint` group as an argument it will not validate the group `Info` nor the group `Warning` because the group does not fully cover one of the other groups.

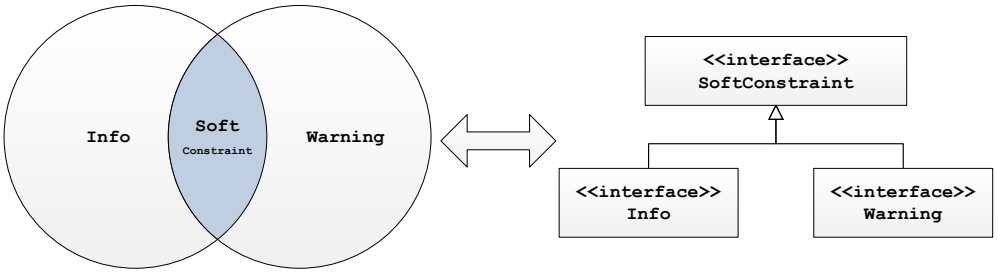


Figure 19.3: The interface `SoftConstraint` is extended by the interfaces `Info` and `Warning` which can be viewed as two intersecting sets sharing some properties. Validating both interfaces with the common ancestor is not possible with this design, because the `SoftConstraint` set does not fully cover both sets.

Therefore, as described in chapter 4.4.1 of the JSR 349 (Bean Validation 1.1), we implement the group data structure the other way round as shown in figure 19.4 which is sometimes counter intuitive using inheritance. Now, the `SoftConstraint` interface extends the interfaces `Info` and `Warning` which in addition explains the usage of interfaces instead of classes. With this data structure we get the desired behaviour, the groups `Info` and `Warning` are subsets of the group `SoftConstraint` as visualised in the left part of figure 19.4. With this data structure we are able to validate every subgroup of the

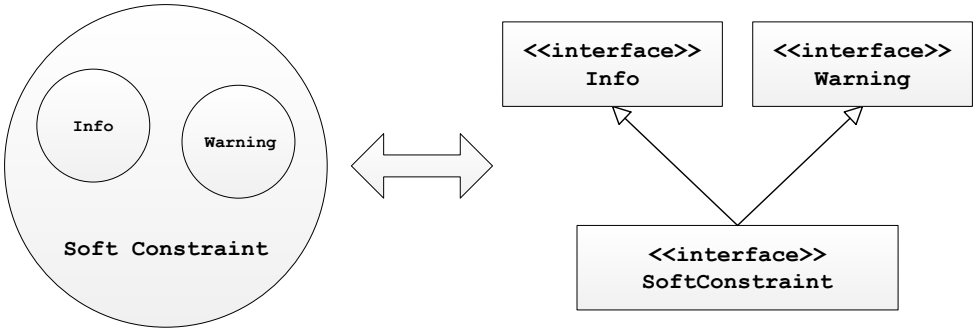


Figure 19.4: The `SoftConstraint` interface extends every special soft constraint group and therefore inherits the properties of `Info` and `Warning`. This way it is possible to validate the soft constraint subgroups with a single call to the validation API using the subinterface `SoftConstraint`.

`SoftConstraint` group by passing the corresponding superinterface to the validation API.
Using the right group design we are able to regulate the validation process regarding hard and soft constraints. For instance, we can validate an entity using the validation API without any group arguments (i.e. passing the `Default` group) and check if some constraint violations occurred. If not, we are ready for further processing of the entity, not caring about the status of the soft constraints as they are allowed to be violated. The complete source code and a detailed example is available in the appendix section A.7.1.3.
Within the JPA environment it is possible to activate automatic Bean Validation (see section 12.4.2 and 12.4.1) for the CRUD operations. By default, JPA 2.0/2.1 does only proceed with a CRUD operation if no `ConstraintViolation` occurred. To enable further processing even if a soft constraint is not fulfilled one can specify the groups which should be considered during the validation phase. Leaving out any of the soft constraint groups it is possible to store an entity violating soft constraints because they are not considered during validation. An usage example within a JPA environment is given in the next paragraph.

PENALTY IMPLEMENTATION

To meet the requirements of the soft constraint definition given in section 19.1.2 we have to provide the possibility to penalise soft constraints if they are not fulfilled. We apply the same idea as described in section 19.2.2.1 using a case distinction for the different payload types or we can now use the payload element for different penalty levels because the group element now handles the hard constraint/soft constraint distinction. A penalty level defines different penalty scores which can be assigned to a soft constraint. If a soft constraint is violated one can use the penalty score of the corresponding penalty level to make a quantitative analysis about the fulfilment of the soft constraints.
Therefore, listing 19.3 shows the `PenaltyLevel` interface which extends the `Payload` class defined by the JSR 303/349 (Bean Validation 1.0/1.1) and defines the method `value`.

Listing 19.3: The `PenaltyLevel` interface contains the `value` method to return the corresponding penalty score. Each penalty level must implement this interface to represent a penalty level.

```
public interface PenaltyLevel extends Payload {

    public int value();

}
```

A sample data structure containing three penalty levels is depicted in figure 19.5. The static attribute `value` is used to be able change the value of the penalty level in a static way as depicted in listing 19.4. The penalty score is set for each penalty level and is the same for the whole application. The additional (non-static) method `value()` simply returns the value of the attribute `value`. This seems redundant at the first sight but the method is needed to access the penalty level value at runtime using introspection (see listing 19.4).

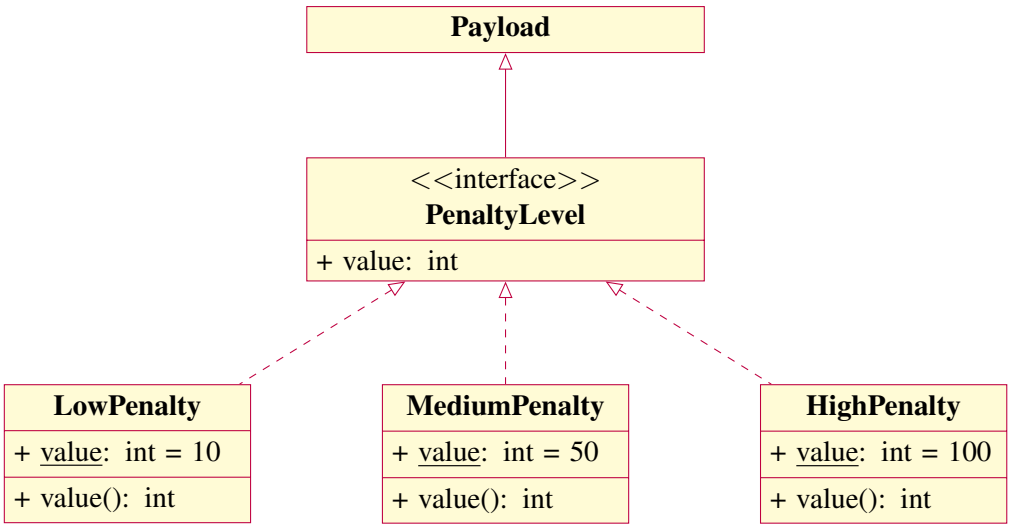


Figure 19.5: UML class diagram showing three penalty level implementations: low, medium and high.

It is possible to add a penalty level by implementing the `PenaltyLevel` interface. If the implementation provides a static field `value` and the `value()` method returns its value the same functionality as described for the sample data structure in 19.5 and shown in listing 19.4 can be used. Using this concept, we can support a more fine-grained way of coupling a soft constraint with a penalty score by adding more penalty levels to the penalty class.

Listing 19.4: This code fragment shows how to change the penalty score of the low penalty level in a static way. This adjustment could be used for instance during the setup time of an application. Moreover, the second part shows how to access the penalty level value at runtime using introspection (exception handling omitted).

```
LowPenalty.value = 20;

Class<? extends PenaltyLevel> penaltyLevel = LowPenalty.class;
int value = penaltyLevel.newInstance().value();
```

An example and the complete source code for the presented concept is available in the appendix section A.7.1.3 and consists of several test cases using soft constraints in a JPA envir-

onment and in a non-permanent way.

19.3 APPLICATION

We now provide an idea how to use the concept of soft and hard constraints described in section 19.1.2 and 19.1.1. The main goal of the idea is that we can make a quantitative statement about how much constraints are fulfilled, i.e. how 'good' the data is with respect to the constraints that are defined. For hard constraints, data is assumed to be 'good' if a constraint is fulfilled and otherwise not. Soft constraints are 'nice to be fulfilled' but are not mandatory (see section 19.1.2). Moreover, the penalty score can be used to assess our needs that a soft constraint should be fulfilled. For instance, a higher penalty score expresses a more important soft constraint than a lower penalty score. Using this insight, we assume that data is 'good' with respect to soft constraints if the sum of penalty scores is low where we sum over all soft constraints that are not fulfilled.
More formally, we do the following calculations to get a quantitative statement. We divide the calculations for hard and soft constraints to be able to make a clear statement for both constraint types. Matching violated hard constraint to a very high penalty score and taking it into the sum of penalty scores for soft constraints would be possible but then we lose the exact information about the actual number of fulfilled hard constraints which should be usual equal to the declared number of hard constraints because they are mandatory.
Let C_A denote the set of all constraints that are declared in class A of the domain model and let T denote the set of all possible constraint types. Then, ct denotes a function which maps constraints to constraint types, i.e. $ct : C_A \to T$. Within this master thesis we consider T to be the set $\{hc, sc\}$ where hc means hard constraint and sc means soft constraint and the image of ct can be defined as follows:

$$ct\,(c) \coloneqq \begin{cases} sc, & \text{if } c \text{ is declared as a soft constraint} \\ hc, & \text{otherwise} \end{cases} \tag{19.1}$$

where $c \in C_A$ for all classes A which declare at least one constraint. Furthermore, let $\chi_{\hat{T}}$ be the characteristic function defined as:

$$\chi_{\hat{T}}\,(t) \coloneqq \begin{cases} 1, & \text{if } t \in \hat{T} \\ 0, & \text{otherwise} \end{cases} \tag{19.2}$$

where $t \in T$ and $\hat{T} \subseteq T$. After the definition for the declared constraints in the classes of the domain model we provide the analogous definitions for the corresponding data objects. Let O_B be the set of instances of a class B that declares at least one constraint and let $violations\,(o, t)$ be a function which counts the number of constraint violations for an object $o \in O_B$ and a constraint type $t \in T$.

HARD CONSTRAINTS FULFILMENT

Now we are ready to do to the following calculations to be able to make a quantitative statement about the quality of data regarding hard constraints:

$$hcf \coloneqq 1 - \frac{\sum_{i=1}^{n} \sum_{o \in O_{A_i}} violations\,(o, hc)}{\sum_{i=1}^{n} |O_{A_i}| \sum_{c \in C_{A_i}} \chi_{\hat{T}=\{hc\}}\,(ct\,(c))} \tag{19.3}$$

with n being the total number of classes that have at least one constraint (i.e. there are classes $A_1, A_2, \ldots, A_n$) and $hc \in T$.
The definition 19.3 denoted as hcf (which means 'hard constraints fulfilment') is a ratio which has the lower bound zero and the upper bound one (i.e. $0 \leq hcf \leq 1$) and is simply the total number of violated hard constraints of all objects (that declare at least one constraint) divided by the total number of objects times the declared constraints of all classes (that declare at least one constraint) subtracted from one (because we count the violations but we are interested in the fulfilment). If we multiply this ratio with 100 we are able to make a statement what percentage of the hard constraints are met.
Consider the domain model in listing 19.5 which represents a simple person. We use the implementation method for hard and soft constraints as described in section 19.2.1 and 19.2.2.3, respectively. The field `name` has the constraint `@NotBlank` and belongs to the `Default` constraint group which means it is a hard constraint and we denote it with $hc1$.

Listing 19.5: Domain model representing a person with a mandatory name (hard constraint) and two soft constraints (address and age). The soft constraint for `address` has a low penalty level with a value of 10 and `age` is a soft constraint with a high penalty level and a value of 50.

```
public class Person {

  @NotBlank
  private String name;

  @NotBlank(
    payload = LowPenalty.class,
    groups = SoftConstraint.class
  )
  private String address;

  @Min(
    value = 25,
    payload = MediumPenalty.class,
    groups = SoftConstraint.class
  )
  private int age;

  // Additional methods (e.g. getter and setter methods)
  ...

}
```

Suppose that we have created two objects of type `Person` denoted as `o1` and `o2`. Moreover, assume that `o1` fulfils $hc1$ and `o2` does not. Table 19.1 summarises this scenario.
Applying definition 19.3, we have $n = 1$ (i.e. $A_1 =$ `Person.class`) and $O_{A_1} = \{$`o1`, `o2`$\}$. Therefore, the numerator of 19.3 reduces to

Identifier	Type	$hc1$	$sc1$	$sc2$
o1	Person	✓	✓	×
o2	Person	×	×	×

Table 19.1: Sample set of objects that are both of type Person. The object denoted as o1 does not fulfil the soft constraint $sc2$ and object o2 does not fulfil any constraint.

$$\begin{aligned}\sum_{i=1}^{1}\sum_{o\in O_{A_i}} violations\,(o,hc) &= \sum_{o\in O_{A_1}} violations\,(o,hc)\\ &= \sum_{o\in\{\mathtt{o1},\mathtt{o2}\}} violations\,(o,hc)\\ &= violations\,(\mathtt{o1},hc) + violations\,(\mathtt{o2},hc)\\ &= 1+0=1\end{aligned}$$

and for the denominator we have $|O_{A_1}| = 2$ and $C_{A_1} = \{hc1, sc1, sc2\}$ and we get

$$\begin{aligned}\sum_{i=1}^{1}|O_{A_i}|\sum_{c\in C_{A_i}}\chi_{\hat{T}=\{hc\}}\,(ct\,(c)) &= |O_{A_1}|\sum_{c\in C_{A_1}}\chi_{\hat{T}=\{hc\}}\,(ct\,(c))\\ &= 2\cdot\sum_{c\in\{hc1,sc1,sc2\}}\chi_{\hat{T}=\{hc\}}\,(ct\,(c))\\ &= 2\cdot\Big[\chi_{\hat{T}=\{hc\}}\,(ct\,(hc1)) + \chi_{\hat{T}=\{hc\}}\,(ct\,(sc1)) +\\ &\chi_{\hat{T}=\{hc\}}\,(ct\,(hc1))\Big]\\ &= 2\cdot[1+0+0]\\ &= 2\end{aligned}$$

Calculating the ratio as depicted in 19.3 we get $1-\frac{1}{2}=0.5$, i.e. 50% of the hard constraints are fulfilled.
Similar calculations as for the hard constraints fulfilment can be done for soft constraints. Due to the fact that soft constraints have a variable penalty score we have to introduce some formalism before we can calculate the scf (which means soft constraints fulfilment).
Let $penalty\_score$ be a function from constraints to the set of all positive numbers, i.e. $penalty\_score : C_A \to \mathbb{N}^+$. In addition, let $violation\,(o,c)$ be a function which returns one if the object denoted with $\circ$ violates constraint c and zero otherwise, i.e.

$$violation\,(o,c) := \begin{cases}1, & \text{if } \circ \text{ does not fulfil } c\\ 0, & \text{otherwise}\end{cases} \tag{19.4}$$

SOFT CONSTRAINTS FULFILMENT

The following calculations (see 19.5) are basically the same as for the hard constraints

fulfilment with the additional condition that the individual penalty scores of the soft constraints must be included. The following ratio can be used to make a quantitative statement about the quality of data regarding soft constraints:

$$scf \coloneqq 1 - \frac{\sum_{i=1}^{n} \sum_{o \in O_{A_i}} \sum_{c \in C_{A_i}} \chi_{\hat{T}=\{sc\}}\left(ct\left(c\right)\right) \cdot violation(o,c) \cdot penalty\_score\left(c\right)}{\sum_{i=1}^{n} |O_{A_i}| \sum_{c \in C_{A_i}} \chi_{\hat{T}=\{sc\}}\left(ct\left(c\right)\right) \cdot penalty\_score\left(c\right)} \tag{19.5}$$

with n being the total number of classes that have at least one constraint (i.e. there are classes $A_1, A_2, \ldots, A_n$).
Definition 19.5 has the same bounds as definition 19.3, i.e. $0 \leq scf \leq 1$ and means that we simply sum up the penalty score over all violated soft constraints and divide this by the maximum possible penalty score over all objects. Subtracting this ratio from one yields 19.5. This definition is the weighted adaptation of definition 19.3 and if we multiply it with 100 we get the percentage of soft constraints that are fulfilled.
Consider the domain model depicted in listing 19.5 again. In addition to the hard constraint, the domain model also defines two soft constraints. The field `address` has the soft constraint `@NotBlank` and belongs to the low penalty level since the payload is set to `LowPenalty.class`. Moreover, the `age` field belongs to the group `SoftConstraint.class` and is set to a medium penalty level for the constraint `@Min(value = 25)`. The soft constraints express that they are optional and the lower penalty level of the `@NotBlank` constraint for the field `address` means that this soft constraint is less important than the one for the field `age`. Similar to the calculations for the hard constraint fulfilment, we now present the calculations for the soft constraints fulfilment using the domain model depicted in listing 19.5 and the instances described in table 19.1.
Hence, we define the values of the penalty levels by specifying the image of the $penalty\_score$ as follows:

$$penalty\_score\left(c\right) \coloneqq \begin{cases} 10, & \text{if } c \text{ has a low penalty level} \\ 50, & \text{if } c \text{ has a medium penalty level} \\ 100, & \text{if } c \text{ has a high penalty level} \\ 0, & \text{otherwise} \end{cases} \tag{19.6}$$

Applying definition 19.5, we have $n = 1$ (i.e. $A_1 = \texttt{Person.class}$) and $O_{A_1} = \{\texttt{o1}, \texttt{o2}\}$. Therefore, the numerator of 19.5 reduces to

$$\begin{aligned} & \sum_{i=1}^{1} \sum_{o \in O_{A_i}} \sum_{c \in C_{A_i}} \chi_{\hat{T}=\{sc\}}\left(ct\left(c\right)\right) \cdot violation(o,c) \cdot penalty\_score\left(c\right) \\ = & \sum_{o \in O_{A_1=\{\texttt{o1},\texttt{o2}\}}} \sum_{c \in C_{A_1}} \chi_{\hat{T}=\{sc\}}\left(ct\left(c\right)\right) \cdot violation(o,c) \cdot penalty\_score\left(c\right) \\ = & \sum_{c \in \{hc1, sc1, sc2\}} \chi_{\hat{T}=\{sc\}}\left(ct\left(c\right)\right) \cdot violation(\texttt{o1},c) \cdot penalty\_score\left(c\right) \\ & + \sum_{c \in \{hc1, sc1, sc2\}} \chi_{\hat{T}=\{sc\}}\left(ct\left(c\right)\right) \cdot violation(\texttt{o2},c) \cdot penalty\_score\left(c\right) \end{aligned}$$

$$
\begin{aligned}
= \quad & \chi_{\hat{T}=\{sc\}}\left(ct\left(hc1\right)\right) \cdot violation(\texttt{o1}, hc1) \cdot penalty\_score\left(hc1\right) \\
& + \chi_{\hat{T}=\{sc\}}\left(ct\left(sc1\right)\right) \cdot violation(\texttt{o1}, sc1) \cdot penalty\_score\left(sc1\right) \\
& + \chi_{\hat{T}=\{sc\}}\left(ct\left(sc2\right)\right) \cdot violation(\texttt{o1}, sc2) \cdot penalty\_score\left(sc2\right) \\
& + \chi_{\hat{T}=\{sc\}}\left(ct\left(hc1\right)\right) \cdot violation(\texttt{o2}, hc1) \cdot penalty\_score\left(hc1\right) \\
& + \chi_{\hat{T}=\{sc\}}\left(ct\left(sc1\right)\right) \cdot violation(\texttt{o2}, sc1) \cdot penalty\_score\left(sc1\right) \\
& + \chi_{\hat{T}=\{sc\}}\left(ct\left(sc2\right)\right) \cdot violation(\texttt{o2}, sc2) \cdot penalty\_score\left(sc2\right) \\
= \quad & 0 \cdot 0 \cdot 0 + 1 \cdot 0 \cdot 10 + 1 \cdot 1 \cdot 50 \\
& + 0 \cdot 1 \cdot 0 + 1 \cdot 1 \cdot 10 + 1 \cdot 1 \cdot 50 \\
= \quad & 110
\end{aligned}
$$

and for the denominator we have $|O_{A_1}| = 2$ and $C_{A_1} = \{hc1, sc1, sc2\}$ and we get

$$
\begin{aligned}
& \sum_{i=1}^{1} |O_{A_i}| \sum_{c \in C_{A_i}} \chi_{\hat{T}=\{sc\}}\left(ct\left(c\right)\right) \cdot penalty\_score\left(c\right) \\
= \quad & |O_{A_1}| \sum_{c \in C_{A_1}} \chi_{\hat{T}=\{sc\}}\left(ct\left(c\right)\right) \cdot penalty\_score\left(c\right) \\
= \quad & 2 \sum_{c \in \{hc1, sc1, sc2\}} \chi_{\hat{T}=\{sc\}}\left(ct\left(c\right)\right) \cdot penalty\_score\left(c\right) \\
= \quad & 2 \cdot \Big[\chi_{\hat{T}=\{sc\}}\left(ct\left(hc1\right)\right) \cdot penalty\_score\left(hc1\right) + \chi_{\hat{T}=\{sc\}}\left(ct\left(sc1\right)\right) \cdot penalty\_score\left(sc1\right) \\
& + \chi_{\hat{T}=\{sc\}}\left(ct\left(sc2\right)\right) \cdot penalty\_score\left(sc2\right)\Big] \\
= \quad & 2 \cdot [0 \cdot 0 + 1 \cdot 10 + 1 \cdot 50] \\
= \quad & 120
\end{aligned}
$$

The ratio according to 19.5 is therefore $1 - \frac{110}{120} = 0.08$, i.e. 8% of the soft constraints are fulfilled. A more sophisticated and realistic example is given in the appendix section A.7.1.3. To combine the evaluation of hard and soft constraints one just has to adapt the concept slightly: let $\hat{T}$ contain the hard constraint as well, i.e. $\hat{T} = \{hc, sc\}$ and adapt the penalty score function accordingly. For instance, let the adapted $penalty\_score$ function be

$$
penalty\_score\left(c\right) := \begin{cases} 10, & \text{if } c \text{ has a low penalty level} \\ 50, & \text{if } c \text{ has a medium penalty level} \\ 100, & \text{if } c \text{ has a high penalty level} \\ \infty, & \text{otherwise} \end{cases} \tag{19.7}
$$

where ∞ should be replaced with the largest machine-representable number. Note that the quantitative statement concerning both types of constraints have to be carefully analysed. The complete source code can be found in the appendix section A.7.1.3.

20 CONSTRAINTS AND DATA QUALITY DIMENSIONS

Data quality is a multidimensional concept which is widely accept in the research community as already stated in the data quality chapter (see chapter 3). Therefore, we described several data quality dimensions in the concept background (see part II). Now, we try to answer the question

'How do constraints improve data quality?'

The quality of data can be measured with respect to a individual data quality dimension using an appropriate metric. The definition of a metric for individual data quality dimension is a non trivial task and beyond of the scope of this master thesis. Nevertheless, we believe that the data quality can be increased with respect to an individual dimension by applying constraints which can be categorised as an improver for this category. For instance, if we would like to improve data quality with respect to the completeness dimension we could think of a group of data constraints which help to fulfil the completeness characteristic. Hence, we try to give some examples how to assign constraints to different data quality dimensions.

ACCURACY

The definition mentioned in 4.2.1 distinguishes between syntactic and semantic accuracy. We cannot mention a constraint of our data quality framework which should increase the data quality with respect to semantic accuracy because such a constraint needs some ground truth[1] which is hardly always available. For instance, if a director is assigned to the wrong film, another data set (the ground truth) needs to be considered to check if the assignment is

[1] See `http://en.wikipedia.org/wiki/Ground_truth`

right or wrong.
Constraints which help to increase the data quality in terms of syntactic accuracy are possible using our data quality framework. For instance, checking whether a post code of the United Kingdom is valid or not could be done with the following constraint:

```
@Pattern(
regexp="^([A-PR-UWYZ0-9][A-HK-Y0-9][AEHMNPRTVXY0-9]?
[ABEHMNPRVWXY0-9]? {1,2}[0-9][ABD-HJLN-UW-Z]{2}\mid GIR 0AA)\$"
)
```

Sometimes, it is necessary to check if a value is contained in the domain of valid values. For instance, the first name of a person must be a realistic one that occurs in the real world. This is not a semantic accuracy problem, because we are simple looking that the value corresponds to **a** first name, maybe it is not the right one. This problem could be solved by a constraint which checks if the provided value is contained in list of acceptable values. This constraint could be created using the facilities that our data quality management framework provides. A slightly simpler version of such a constraint type is available in the appendix section A.8.

COMPLETENESS

To improve the data quality with respect to completeness (see definition in section 4.2.2), one could think of constraints which check the absence of a value, i.e. `@NotNull`, `@NotEmpty` or `@NotBlank` of our data quality management framework. These constraints improve the data quality if they are defined as hard constraints but what happens if a value cannot be entered because it does not exist although we categorise it as a hard constraint? For instance, a registration form requires an e-mail address but the person does not have one. Then, we need additional information why the value is not available. A constraint depicted in listing 20.1 could be used to check for additional information and depending on the outcome an appropriate result is returned. Further research needs to be done, especially with respect to a closed world and open world assumption.

Listing 20.1: Validation logic (using pseudocode) which checks why a value is not available (maybe the value does not exist)

```
if value does not exist {
  return true;
else if value exists but is not available {
  return false;
else {
  return true;
}
```

CONSISTENCY

Constraints which improve the data quality with respect to consistency can be formulated because the definition of consistency given in section 4.2.3 says that this dimension is equal to semantic rules i.e. correctness. Therefore we can translate the rule into a constraint, for instance: the last remake year of a film must be greater or equal to the year of the first

release can be formulate as a cross-field constraint checking both fields of the corresponding class (i.e. a type-level constraint).

The examples mentioned above give some ideas how constraints can improve the data quality of an information system. Further research has to be done with respect to other dimensions (such as temporal data quality dimensions) and a close look at the dimensions mentioned above.

PART VI

SUMMARY

21 CONTRIBUTION

To solve the problems mentioned in the introduction (see part I) we suggest a data quality management framework for an application developer who works within the Java™ programming language environment. The basic idea is the usage of constraints to improve the quality of data. The design decisions are based on a broad survey for already existing conceptual and technical solutions with respect to constraint management and data validation which we did in the beginning of this master thesis. Our analysis has showed that we will use the JSR 349 (Bean Validation 1.1) with its reference implementation Hibernate Validator as the basis for or framework. Using this concept, we can specify the constraints using Java™ annotations for our domain model resulting in a single constraint model. Also, Bean Validation allows a developer to specify a constraint only once (i.e. in the domain model) and apply it at different points in an application for validation which avoids redundancy and unnecessary, multiple validation. Moreover, after recognising the importance of Bean Validation we compared the individual technologies with respect to Bean Validation integration which is depicted in table 14.3 and 14.4.

Within our data quality management framework, we propose an association constraint which allows a developer to specify the cardinalities of an association in a dynamic way. For instance, it is possible to specify the minimum and maximum number of students for a lecture. The JSR 349 (Bean Validation 1.1) does only allow a static specification of cardinalities which means for example that the cardinalities for every lecture would be the same because the cardinalities are specified on a per class level. Therefore, we propose several concepts and implementations to overcome this issue. Finally, our data quality management framework consists of a solution which allows the specification of the cardinalities for every object in a type-safe way using the mechanisms of Bean Validation (i.e. annotations). We call this solution a dynamic constraint because it is possible to specify the cardinalities at runtime. Moreover, we explain a solution regarding the persistence of the the cardinalities within a JPA environment.

In addition, we introduce the concept of temporal constraints which offer the possibility to specify a deadline for a constraint. This temporal dimension of a constraint makes it pos-

sible that a constraint does not have to hold right now but at a certain point in time (i.e. the deadline). To realise this concept, we have implemented a data structure which is based on the concepts of the JSR 349 (Bean Validation 1.1) and an additional feature of the reference implementation for this specification which is called Hibernate Validator. Building on this concept, we present a time-triggered validation component which is based on the job scheduling library Quartz[1]. Within this component we provide basic job definitions to trigger validation jobs using Bean Validation and/or JPA at certain points in time or on a repetitive basis.
Finally, based on the insights of LP, we propose a definition for hard and soft constraints. Afterwards, we introduce our implementation of hard and soft constraints using the JSR 349 (Bean Validation 1.1) and suggest an application method to make a quantitative statement about the quality of data.
To substantiate the basic idea mentioned in the beginning of this chapter, we analysed the definition of data quality and its multidimensional concept followed by overview of constraint types. We think that the existence of constraints can significantly improve the quality of data. Moreover, we believe that the measurement method of data quality with respect to individual dimensions correlate with the used constraints. Therefore, we provide some examples how constraints can influence the measurement of data quality with respect to individual dimensions in a positive way.

[1] See footnote 1 in chapter 18

22 CONCLUSION

Summarising, in the first stage of this master thesis, we presented the theoretical background of this work. We started with different definitions of data quality and figured out that the research community accepted that data quality is a multidimensional concept. Therefore, we explained how to discover data quality dimensions and described the most important dimensions such as accuracy, completeness and consistency. Moreover, an overview of different constraint types (e.g. a unique constraint) is given in the last chapter of part II. Next, we did a broad research within the academic community with respect to constraint specification and data validation approaches. Each publication is categorised into a tier and/or layer according to the presented context. A similar categorisation approach is taken for the analysis of the technical background where we examined mostly Java™ technologies such as GWT, JSF and JPA with respect to their data validation and constraint specification capabilities.
The second stage uses the insights that we gained in the first stage to design a data quality management framework. Therefore, we decided to use the JSR 349 (Bean Validation 1.1) technology as the basis for our framework. The technology allows a developer to specify constraints using a single constraint model (Java™ annotations) in tier/layer independent way. Furthermore, Bean Validation integrates well with other Java™ technologies such as JSF and JPA. Based on this decision we developed several features for our framework. First, we implemented an association constraint to restrict the cardinalities of an association in a dynamic way. Using the `@VariableSize` constraint in combination with our provided data structure allows a developer to specify the cardinalities of an association at runtime. Second, we introduce the concept of a temporal constraint which enables the possibility that a constraint must not hold immediately but at a certain point in time. Hence, we offer the possibility to combine an already existing constraint with a deadline using the concept of constraint composition and a special data structure provided by the data quality management framework. Closely connect to the concept of temporal constraints, our framework consists of a component which allows an application developer to specify validation jobs using a scheduler that is based on the Quartz[1] library. We provide a universal validation job which

[1] See footnote 1

can be used to validate objects that are stored in a database using a JPQL query. Finally, we introduce the concept of hard and soft constraints and show how to implement both types using Bean Validation 1.1. Furthermore, we explain an application where we can use the hard and soft constraints to make a quantitative statement about the quality of data followed by a proposal how constraints and data quality dimensions are related to each other.

23 FUTURE WORK

We conclude this thesis with a view into the feature. This chapter describes possible extension for our presented approach. We do not restrict this 'future work' chapter to a simple enumeration of additional features for our data quality management framework and hence describe conceptual ideas which need a more detailed analysis to make sure that they are reasonable enough for a integration into this framework. First, we present some technical suggestions followed by some thoughts about conceptual challenging questions.

23.1 TECHNICAL EXTENSIONS

The JSR 349 (Bean Validation 1.1), see section 10.1, does not provide a built-in uniqueness constraint. We think that one of the main reasons is that there is no precise definition of what uniqueness means within the context of Bean Validation. Is the attribute value unique among the objects currently stored in the heap? Or, is the whole object unique? Moreover, how to define the uniqueness with respect to the underlying data store? Besides those conceptual questions, a technical solution for a `PersistentUnique` constraint that can be applied to properties (attributes/fields or getter methods) of a Java™ class using JPA is only a partial solution, but seems more realistic to reach.
Also, the data structure for the association and temporal constraint could be completed to support the `@VariableSize`, `@SizeOnDeadline` and `@NotEmptyOnDeadline` constraint in combination with the map and array data structure of the Java™ programming language. Moreover, our data quality management framework uses the domain model as the place to define constraints. Furthermore, entities are validated within the data access layer (see section 12.4) and therefore no constraints are defined in the database. Although, we focus on a single constraint model it is sometimes desirable to replicate the defined constraints using the provided mechanisms of the underlying data store. For instance, data that passes the logic tier and directly interacts with the database could be validated as well if the constraints are synchronized with the data store. This extension requires a precise analysis of the underlying data stores and their capabilities for constraint definitions followed by an algorithm

which translates the validation logic of the Bean Validation constraints into the appropriate data store language. Furthermore, the opposite direction could be beneficial, too. Then, an existing system which defines the constraints in the data store could be used to extract the constraint definitions and transform them into Bean Validation constraints.
Up to now, our data quality management framework allows the declaration of constraints only at compile time which makes constraint changes in the domain model cumbersome because the application has to be stopped, changed and restarted to reflect the newly introduced constraints. A possible extension could be a component within the framework which allows the specification of constraints at runtime (i.e. constraint definition on-the-fly) by a user (or super user).
A GUI where a developer can specify constraints using for instance a UML diagram in combination with the OCL would be good for those people who like to work with a PIM and transform it into executable code. The graphical component should include a automatic transformation into executable code using the features of our data quality management framework to reflect the specified constraints of the PIM. In addition or as an alternative, a plug-in for an Integrated Development Environment (IDE) could be created which allows a developer to create, read, update and delete constraints for a domain model in an easy way.
Lastly, one should check how to integrate our data quality management framework in a typical client-server environment with a rich client that communicates with a server because our evaluation application is a typical web application with a thin client. Moreover, when testing this kind of application one has to think how to distribute the constraints between the server and client. Does the validation happen at the client-side? Does the client send the data to the server for validation? Or, are the constraints distributed/replicated between the client and server? The answers to these questions could be checked with the integration of our framework into the existing software 'FoodCASE'/'FoodCASE-Risk'. This integration would cover a feasibility study for a real application with respect to our framework, too.

23.2 CONCEPTUAL EXTENSIONS

As explained in section 19.3, it is possible to make a qualitative statement about the quality of data. We think that a more detailed exploration needs to be done within this kind of direction in which we believe that the following questions need to be answered:

- What is data quality? We know that it is a multidimensional concept but how are those dimensions defined? What dimensions are relevant?
- After the definition of data quality: How to measure data quality? We think that the quality of data can be analysed with respect to each dimension and therefore one have to come up with measurement methods for specific dimensions. For instance, looking at our data from the perspective of the completeness dimension. Is there a measurement method which calculates 'how complete' the data is? If so, can it be generalised? Be aware, that a correlation between the individual data quality dimension can exist.
- Moreover, what kind of qualitative and quantitative statements about the quality of data are possible?
- We think that the definition of constraints can improve the data quality. Looking at the measurement methods for individual data quality dimensions one can examine if the

data quality can be improved with respect to the considered dimension if an appropriate constraint is used. Consider the following questions: Is there a group and/or type of constraints which belong to the same data quality dimension? Is it possible to assign a constraint for every dimension?

- What does it mean if an object is invalid? Do we compare constraint violations of a single property across all objects of the same type or to we compare the status of an object as a whole? Is it possible to combine different perspectives while the statement about constraint violations remains reasonable?

Having defined how to measure data quality, one might want to present the results in a clear and concise way. Therefore, acquiring a concept for the presentation of constraint violations, the data validation process and the data quality would be good. Combining this concept with the idea of a constraint violation monitor that logs and analyses constraint violations would offer the possibility to create a data quality constraints profiling component. This component could be used by an administrator to analyse the reasons for constraint violations which might end in an adaptation of the specified constraints (for instance, the data quality constraints profiling component shows that some constraints are too strong for the considered context).
Up to now, our data quality management framework is developer-specific which means that that constraints are defined and validated as defined by the developer. A possible extension could be the possibility that a user can define what kind of constraints are important and how to combine them.
Finally, a performance and usability study should be constructed to evaluate the whole concept. The performance study should take into account a benchmark for large data sets as well as the user expectation when entering new data into an information system (e.g. response time for validating a web form). Testing how efficient an application developer can use the features of our data quality management framework could part of a user study. Furthermore, one should compare a specific information system which already consists of a constraint specification, data validation and data quality measurement approach with our approach that replaces the existing facilities. Ultimately, the question ‘Does the data quality increase?’ using our framework should be answered.

PART VII

APPENDIX

A SOURCE CODE

This chapter provides the references to the complete source code examples mentioned in this master thesis. The main directory which contains all the files can be accessed via the following Uniform Resource Locator (URL): `https://svn.globis.ethz.ch/repos/incubator/dqmf/` (authentication and authorisation required). The structure of this chapter is analogous to the structure of the technology/research background and the approach part of the master thesis. Furthermore, the folder structure in the code repository is similar to the chapter structure where the technology/research code can be found in the 'test' folder and the code mentioned in the approach part is located in the 'core' folder.

A.1 SRC: CROSS-TIER VALIDATION

A.1.1 SRC: BEAN VALIDATION

A.1.1.1 SRC: JSR 349: BEAN VALIDATION 1.1

SRC: HIBERNATE VALIDATOR

- **Domain model:** `https://svn.globis.ethz.ch/repos/incubator/dqmf/tests/cross_tier/bean_validation/examples/src/test/java/model/`
- **Test cases:** `https://svn.globis.ethz.ch/repos/incubator/dqmf/tests/cross_tier/bean_validation/examples/src/test/java/testsuite/SimpleBeanValidationTestCases.java`
- **Official examples:** `https://svn.globis.ethz.ch/repos/incubator/dqmf/tests/cross_tier/bean_validation/examples/src/main/java/`

A.1.1.2 SRC: JSR 303: BEAN VALIDATION 1.0

- **Domain model:** `https://svn.globis.ethz.ch/repos/incubator/dqmf/tests/cross_tier/bean_validation/examples/src/test/java/model/`
- **Test cases:** `https://svn.globis.ethz.ch/repos/incubator/dqmf/tests/cross_tier/bean_validation/examples/src/test/java/testsuite/SimpleBeanValidationTestCases.java`
- **Official examples:** `https://svn.globis.ethz.ch/repos/incubator/dqmf/tests/cross_tier/bean_validation/examples/src/main/java/`

SRC: APACHE BVAL

- **Maven depdendency file:** `https://svn.globis.ethz.ch/repos/incubator/dqmf/tests/cross_tier/bean_validation/examples/pom.xml`

A.2 SRC: LOGIC-TIER VALIDATION

A.2.1 SRC: PRESENTATION LAYER VALIDATION

A.2.1.1 SRC: JSR 314: JSF

SRC: APACHE MYFACES CORE

- **Domain model:** `https://svn.globis.ethz.ch/repos/incubator/dqmf/tests/logic_tier/presentation_layer/jsf_standard_validation/src/model/Person.java`
- **Managed Bean:** `https://svn.globis.ethz.ch/repos/incubator/dqmf/tests/logic_tier/presentation_layer/jsf_standard_validation/src/managedBean/PersonBean.java`
- **Front-end files** (`*.xhtml` and `resources/css/style.css`)**:** `https://svn.globis.ethz.ch/repos/incubator/dqmf/tests/logic_tier/presentation_layer/jsf_standard_validation/WebContent/`
- **Configuration files:** `https://svn.globis.ethz.ch/repos/incubator/dqmf/tests/logic_tier/presentation_layer/jsf_standard_validation/WebContent/WEB-INF/`

SRC: APACHE MYFACES CORE AND HIBERNATE VALIDATOR

- **Domain model:** `https://svn.globis.ethz.ch/repos/incubator/dqmf/tests/logic_tier/presentation_layer/jsf_bean_validation/src/model/Person.java`

- **Managed Bean:** https://svn.globis.ethz.ch/repos/incubator/dqmf/tests/logic_tier/presentation_layer/jsf_bean_validation/src/managedBean/PersonBean.java
- **Properties file:** https://svn.globis.ethz.ch/repos/incubator/dqmf/tests/logic_tier/presentation_layer/jsf_bean_validation/src/messages.properties
- **Front-end files** (*.xhtml and resources/css/style.css)**:** https://svn.globis.ethz.ch/repos/incubator/dqmf/tests/logic_tier/presentation_layer/jsf_bean_validation/WebContent/
- **Configuration files:** https://svn.globis.ethz.ch/repos/incubator/dqmf/tests/logic_tier/presentation_layer/jsf_bean_validation/WebContent/WEB-INF/

SRC: COMPONENT FRAMEWORKS

- **Domain model:** https://svn.globis.ethz.ch/repos/incubator/dqmf/tests/logic_tier/presentation_layer/jsf_rich_faces_bean_validation/src/model/Person.java
- **Managed Bean:** https://svn.globis.ethz.ch/repos/incubator/dqmf/tests/logic_tier/presentation_layer/jsf_rich_faces_bean_validation/src/managedBean/PersonBean.java
- **Client-side validation test:** https://svn.globis.ethz.ch/repos/incubator/dqmf/tests/logic_tier/presentation_layer/jsf_rich_faces_bean_validation/src/listener/phase/ValidationListener.java
- **Properties file:** https://svn.globis.ethz.ch/repos/incubator/dqmf/tests/logic_tier/presentation_layer/jsf_rich_faces_bean_validation/src/messages.properties
- **Front-end files** (*.xhtml and resources/css/style.css)**:** https://svn.globis.ethz.ch/repos/incubator/dqmf/tests/logic_tier/presentation_layer/jsf_rich_faces_bean_validation/WebContent/
- **Configuration files:** https://svn.globis.ethz.ch/repos/incubator/dqmf/tests/logic_tier/presentation_layer/jsf_rich_faces_bean_validation/WebContent/WEB-INF/

A.2.1.2 SRC: GWT

- **Domain model:** https://svn.globis.ethz.ch/repos/incubator/dqmf/tests/logic_tier/presentation_layer/gwt_bean_validation/src/ch/ethz/globis/shared/model/Person.java

- **Data binding** (Java™ file)**:** `https://svn.globis.ethz.ch/repos/incubator/dqmf/tests/logic_tier/presentation_layer/gwt_bean_validation/src/ch/ethz/globis/shared/model/PersonEditor.java`
- **Data binding** (XML file)**:** `https://svn.globis.ethz.ch/repos/incubator/dqmf/tests/logic_tier/presentation_layer/gwt_bean_validation/src/ch/ethz/globis/shared/model/PersonEditor.ui.xml`
- **Validation files** (constraints, groups, validators)**:** `https://svn.globis.ethz.ch/repos/incubator/dqmf/tests/logic_tier/presentation_layer/gwt_bean_validation/src/ch/ethz/globis/shared/validation/`
- **Client code:** `https://svn.globis.ethz.ch/repos/incubator/dqmf/tests/logic_tier/presentation_layer/gwt_bean_validation/src/ch/ethz/globis/client/`
- **Server code:** `https://svn.globis.ethz.ch/repos/incubator/dqmf/tests/logic_tier/presentation_layer/gwt_bean_validation/src/ch/ethz/globis/server/WelcomeServiceImpl.java`
- **GWT module:** `https://svn.globis.ethz.ch/repos/incubator/dqmf/tests/logic_tier/presentation_layer/gwt_bean_validation/src/ch/ethz/globis/GwtBeanValidation.gwt.xml`
- **Configuration file:** `https://svn.globis.ethz.ch/repos/incubator/dqmf/tests/logic_tier/presentation_layer/gwt_bean_validation/war/WEB-INF/web.xml`
- **Front-end files** (`*.html` and `style.css`)**:** `https://svn.globis.ethz.ch/repos/incubator/dqmf/tests/logic_tier/presentation_layer/gwt_bean_validation/war/`
- **Libraries for server code:** `https://svn.globis.ethz.ch/repos/incubator/dqmf/tests/logic_tier/presentation_layer/gwt_bean_validation/war/WEB-INF/lib/`

A.2.1.3 SRC: JFC: SWING

- **Domain model:** `https://svn.globis.ethz.ch/repos/incubator/dqmf/tests/logic_tier/presentation_layer/jfc_swing_validation/src/model/Person.java`

SRC: JFC SWING: ACTION LISTENER APPROACH

- **Full example:** `https://svn.globis.ethz.ch/repos/incubator/dqmf/tests/logic_tier/presentation_layer/jfc_swing_validation/src/examples/ActionListenerExample.java`

SRC: SWING FORM BUILDER

- **Full example:** `https://svn.globis.ethz.ch/repos/incubator/dqmf/tests/logic_tier/presentation_layer/jfc_swing_validation/src/examples/SwingFormBuilderExample.java`

A.2.1.4 SRC: SWT

- **Domain model:** `https://svn.globis.ethz.ch/repos/incubator/dqmf/tests/logic_tier/presentation_layer/swt_bean_validation/src/model/Person.java`
- **Full example:** `https://svn.globis.ethz.ch/repos/incubator/dqmf/tests/logic_tier/presentation_layer/swt_bean_validation/src/example/Main.java`

SRC: JFACE STANDARD VALIDATION

- **Domain model:** `https://svn.globis.ethz.ch/repos/incubator/dqmf/tests/logic_tier/presentation_layer/jface_validation/src/model/Person.java`
- **Validator:** `https://svn.globis.ethz.ch/repos/incubator/dqmf/tests/logic_tier/presentation_layer/jface_validation/src/validators/StringRequiredValidator.java`
- **Full example:** `https://svn.globis.ethz.ch/repos/incubator/dqmf/tests/logic_tier/presentation_layer/jface_validation/src/examples/JFaceStandardValidationExample.java`

SRC: JFACE BEAN VALIDATION

- **Domain model:** `https://svn.globis.ethz.ch/repos/incubator/dqmf/tests/logic_tier/presentation_layer/jface_validation/src/model/Person2.java`
- **Full example:** `https://svn.globis.ethz.ch/repos/incubator/dqmf/tests/logic_tier/presentation_layer/jface_validation/src/examples/JFaceBeanValidationExample.java`
- **Validators** (`JSR303*.java` files, `Activator.java` and `IValidatorFactoryProvider.java`)**:** `https://svn.globis.ethz.ch/repos/incubator/dqmf/tests/logic_tier/presentation_layer/jface_validation/src/validators/`

A.2.1.5 SRC: JAVAFX

SRC: FXFORM2

- **Domain model:** `https://svn.globis.ethz.ch/repos/incubator/dqmf/tests/logic_tier/presentation_layer/javafx_bean_validation/src/model/Person.java`
- **Full example:** `https://svn.globis.ethz.ch/repos/incubator/dqmf/tests/logic_tier/presentation_layer/javafx_bean_validation/src/example/JavaFXBeanValidationExample.java`

A.2.2 SRC: DATA ACCESS LAYER VALIDATION

A.2.2.1 SRC: JSR 338: JPA 2.1

SRC: ECLIPSELINK

- **Domain model** (relational DBMS)**:** `https://svn.globis.ethz.ch/repos/incubator/dqmf/tests/logic_tier/data_access_layer/eclipselink_bean_validation/src/main/java/model/rdbms/`
- **Domain model** (NoSQL DB: MongoDB)**:** `https://svn.globis.ethz.ch/repos/incubator/dqmf/tests/logic_tier/data_access_layer/eclipselink_bean_validation/src/main/java/model/mongodb/`
- **Validation group:** `https://svn.globis.ethz.ch/repos/incubator/dqmf/tests/logic_tier/data_access_layer/eclipselink_bean_validation/src/main/java/validation/groups/Special.java`
- **Example** (relational DBMS: MySQL)**:** `https://svn.globis.ethz.ch/repos/incubator/dqmf/tests/logic_tier/data_access_layer/eclipselink_bean_validation/src/main/java/example/rdbms/BeanValidationMySQL.java`
- **Example** (NoSQL DB: MongoDB)**:** `https://svn.globis.ethz.ch/repos/incubator/dqmf/tests/logic_tier/data_access_layer/eclipselink_bean_validation/src/main/java/example/nosql/BeanValidationMongoDB.java`
- **Persistence file:** `https://svn.globis.ethz.ch/repos/incubator/dqmf/tests/logic_tier/data_access_layer/eclipselink_bean_validation/src/main/resources/META-INF/persistence.xml`
- **Test suites** (relational DBMS: MySQL)**:** `https://svn.globis.ethz.ch/repos/incubator/dqmf/tests/logic_tier/data_access_layer/eclipselink_bean_validation/src/test/java/testsuites/rdbms/`
- **Test suites** (NoSQL DB: MongoDB)**:** `https://svn.globis.ethz.ch/repos/incubator/dqmf/tests/logic_tier/data_access_layer/eclipselink_bean_validation/src/test/java/testsuites/mongodb/`

SRC: HIBERNATE ORM

- **Domain model:** `https://svn.globis.ethz.ch/repos/incubator/dqmf/tests/logic_tier/data_access_layer/hibernate_orm_bean_validation/src/main/java/model/`
- **Validation group:** `https://svn.globis.ethz.ch/repos/incubator/dqmf/tests/logic_tier/data_access_layer/hibernate_orm_bean_validation/src/main/java/validation/groups/Special.java`
- **Example:** `https://svn.globis.ethz.ch/repos/incubator/dqmf/tests/logic_tier/data_access_layer/hibernate_orm_bean_validation/src/main/java/example/BeanValidationMySQL.java`
- **Persistence file:** `https://svn.globis.ethz.ch/repos/incubator/dqmf/tests/logic_tier/data_access_layer/hibernate_orm_bean_validation/src/main/resources/META-INF/persistence.xml`
- **Test suites** (MySQL)**:** `https://svn.globis.ethz.ch/repos/incubator/dqmf/tests/logic_tier/data_access_layer/hibernate_orm_bean_validation/src/test/java/testsuites/mysql/`

SRC: DATANUCLEUS

- **Domain model :** `https://svn.globis.ethz.ch/repos/incubator/dqmf/tests/logic_tier/data_access_layer/datanucleus_access_platform_bean_validation/src/main/java/model/`
- **Validation group:** `https://svn.globis.ethz.ch/repos/incubator/dqmf/tests/logic_tier/data_access_layer/datanucleus_access_platform_bean_validation/src/main/java/validation/groups/Special.java`
- **Example** (relational DBMS: MySQL)**:** `https://svn.globis.ethz.ch/repos/incubator/dqmf/tests/logic_tier/data_access_layer/datanucleus_access_platform_bean_validation/src/main/java/example/rdbms/BeanValidationMySQL.java`
- **Example** (NoSQL DB: MongoDB)**:** `https://svn.globis.ethz.ch/repos/incubator/dqmf/tests/logic_tier/data_access_layer/datanucleus_access_platform_bean_validation/src/main/java/example/nosql/BeanValidationMongoDB.java`
- **Persistence file:** `https://svn.globis.ethz.ch/repos/incubator/dqmf/tests/logic_tier/data_access_layer/datanucleus_access_platform_bean_validation/src/main/resources/META-INF/persistence.xml`

- **Test suites** (relational DBMS: MySQL)**:** `https://svn.globis.ethz.ch/repos/incubator/dqmf/tests/logic_tier/data_access_layer/datanucleus_access_platform_bean_validation/src/test/java/testsuites/rdbms/`
- **Test suites** (NoSQL DB: MongoDB)**:** `https://svn.globis.ethz.ch/repos/incubator/dqmf/tests/logic_tier/data_access_layer/datanucleus_access_platform_bean_validation/src/test/java/testsuites/mongodb/`

A.2.2.2 SRC: JSR 317: JPA 2.0

SRC: APACHE OPENJPA

- **Domain model :** `https://svn.globis.ethz.ch/repos/incubator/dqmf/tests/logic_tier/data_access_layer/apache_openjpa_bean_validation/src/main/java/model/`
- **Validation group:** `https://svn.globis.ethz.ch/repos/incubator/dqmf/tests/logic_tier/data_access_layer/apache_openjpa_bean_validation/src/main/java/validation/groups/Special.java`
- **Example:** `https://svn.globis.ethz.ch/repos/incubator/dqmf/tests/logic_tier/data_access_layer/apache_openjpa_bean_validation/src/main/java/example/BeanValidationMySQL.java`
- **Persistence file:** `https://svn.globis.ethz.ch/repos/incubator/dqmf/tests/logic_tier/data_access_layer/apache_openjpa_bean_validation/src/main/resources/META-INF/persistence.xml`
- **Test suites** (MySQL)**:** `https://svn.globis.ethz.ch/repos/incubator/dqmf/tests/logic_tier/data_access_layer/apache_openjpa_bean_validation/src/test/java/testsuites/mysql/`

SRC: BATOO JPA

- **Domain model :** `https://svn.globis.ethz.ch/repos/incubator/dqmf/tests/logic_tier/data_access_layer/batoo_jpa_bean_validation/src/main/java/model/`
- **Validation group:** `https://svn.globis.ethz.ch/repos/incubator/dqmf/tests/logic_tier/data_access_layer/batoo_jpa_bean_validation/src/main/java/validation/groups/Special.java`
- **Example:** `https://svn.globis.ethz.ch/repos/incubator/dqmf/tests/logic_tier/data_access_layer/batoo_jpa_bean_validation/src/main/java/example/BeanValidationMySQL.java`

- **Persistence file:** `https://svn.globis.ethz.ch/repos/incubator/dqmf/tests/logic_tier/data_access_layer/batoo_jpa_bean_validation/src/main/resources/META-INF/persistence.xml`
- **Test suites** (MySQL)**:** `https://svn.globis.ethz.ch/repos/incubator/dqmf/tests/logic_tier/data_access_layer/batoo_jpa_bean_validation/src/test/java/testsuites/mysql/`

A.2.2.3 SRC: NON-STANDARD JPA PROVIDERS

SRC: HIBERNATE OGM

- **Domain model :** `https://svn.globis.ethz.ch/repos/incubator/dqmf/tests/logic_tier/data_access_layer/hibernate_ogm_bean_validation/src/main/java/model/`
- **Validation group:** `https://svn.globis.ethz.ch/repos/incubator/dqmf/tests/logic_tier/data_access_layer/hibernate_ogm_bean_validation/src/main/java/validation/groups/Special.java`
- **Example** (MongoDB, Infinispan, Ehcache)**:** `https://svn.globis.ethz.ch/repos/incubator/dqmf/tests/logic_tier/data_access_layer/hibernate_ogm_bean_validation/src/main/java/example/`
- **Persistence file:** `https://svn.globis.ethz.ch/repos/incubator/dqmf/tests/logic_tier/data_access_layer/hibernate_ogm_bean_validation/src/main/resources/META-INF/persistence.xml`
- **Test suites** (MongoDB)**:** `https://svn.globis.ethz.ch/repos/incubator/dqmf/tests/logic_tier/data_access_layer/hibernate_ogm_bean_validation/src/test/java/testsuites/mongodb/`

A.3 SRC: DATA QUALITY MANAGEMENT FRAMEWORK

- **Demo application:** `https://svn.globis.ethz.ch/repos/incubator/dqmf/demo/`
- **Domain model** (lecture)**:** `https://svn.globis.ethz.ch/repos/incubator/dqmf/core/components/src/test/java/ch/ethz/globis/dqmf/model/persistent/PersistentLecture.java`
- **Domain model** (student)**:** `https://svn.globis.ethz.ch/repos/incubator/dqmf/core/components/src/test/java/ch/ethz/globis/dqmf/model/persistent/PersistentStudent.java`

- **Test suite:** `https://svn.globis.ethz.ch/repos/incubator/dqmf/core/components/src/test/java/ch/ethz/globis/dqmf/testsuites/jpa/temporal/TemporalAssociationTemporalMinTestSuite.java`

A.4 SRC: ASSOCIATION CONSTRAINT

A.4.1 SRC: DYNAMIC ASSOCIATION CONSTRAINT

- **Domain model** (student)**:** `https://svn.globis.ethz.ch/repos/incubator/dqmf/core/development/src/test/java/ch/ethz/globis/dqmf/dev/model/nonpermanent/Student.java`
- **Domain model** (lecture)**:** `https://svn.globis.ethz.ch/repos/incubator/dqmf/core/development/src/test/java/ch/ethz/globis/dqmf/dev/model/nonpermanent/Lecture.java`
- **Test suite** (development)**:** `https://svn.globis.ethz.ch/repos/incubator/dqmf/core/development/src/test/java/ch/ethz/globis/dqmf/dev/testsuites/AssociationTestSuite.java`
- **Test suite** (core components)**:** `https://svn.globis.ethz.ch/repos/incubator/dqmf/core/components/src/test/java/ch/ethz/globis/dqmf/testsuites/TestSuite.java`

A.4.1.1 SRC: TYPE-LEVEL CONSTRAINT METHODS

SRC: HAND-CRAFTED ASSOCIATION CONSTRAINT METHOD

- **Domain model:** `https://svn.globis.ethz.ch/repos/incubator/dqmf/core/development/src/test/java/ch/ethz/globis/dqmf/dev/model/nonpermanent/HandcraftedLecture.java`
- **Annotation:** `https://svn.globis.ethz.ch/repos/incubator/dqmf/core/development/src/test/java/ch/ethz/globis/dqmf/dev/constraints/annotations/associations/StudentAssociation.java`
- **Validator:** `https://svn.globis.ethz.ch/repos/incubator/dqmf/core/development/src/test/java/ch/ethz/globis/dqmf/dev/constraints/validators/StudentAssociationValidator.java`
- **Test suite:** `https://svn.globis.ethz.ch/repos/incubator/dqmf/core/development/src/test/java/ch/ethz/globis/dqmf/dev/testsuites/HandcraftedAssociationTestSuite.java`

SRC: GENERIC ASSOCIATION CONSTRAINT METHOD

- **Domain model** (lecture)**:** https://svn.globis.ethz.ch/repos/incubator/dqmf/core/development/src/test/java/ch/ethz/globis/dqmf/dev/model/nonpermanent/GenericLecture.java
- **Domain model** (professor)**:** https://svn.globis.ethz.ch/repos/incubator/dqmf/core/development/src/test/java/ch/ethz/globis/dqmf/dev/model/nonpermanent/Professor.java
- **Annotation:** https://svn.globis.ethz.ch/repos/incubator/dqmf/core/development/src/main/java/ch/ethz/globis/dqmf/dev/constraints/annotations/association/Association.java
- **Validator:** https://svn.globis.ethz.ch/repos/incubator/dqmf/core/development/src/main/java/ch/ethz/globis/dqmf/dev/constraints/validators/AssociationValidator.java
- **Enumeration:** https://svn.globis.ethz.ch/repos/incubator/dqmf/core/development/src/main/java/ch/ethz/globis/dqmf/dev/datastructure/association/AssociationRole.java
- **Data structure:** https://svn.globis.ethz.ch/repos/incubator/dqmf/core/development/src/main/java/ch/ethz/globis/dqmf/dev/datastructure/association/Associations.java
- **Test suite:** https://svn.globis.ethz.ch/repos/incubator/dqmf/core/development/src/test/java/ch/ethz/globis/dqmf/dev/testsuites/GenericAssociationTestSuite.java

SRC: INTROSPECTIVE ASSOCIATION CONSTRAINT METHOD

- **Domain model:** https://svn.globis.ethz.ch/repos/incubator/dqmf/core/development/src/test/java/ch/ethz/globis/dqmf/dev/model/nonpermanent/IntrospectiveLecture.java
- **Annotation:** https://svn.globis.ethz.ch/repos/incubator/dqmf/core/development/src/main/java/ch/ethz/globis/dqmf/dev/constraints/annotations/association/IntrospectiveAssociation.java
- **Validator:** https://svn.globis.ethz.ch/repos/incubator/dqmf/core/development/src/main/java/ch/ethz/globis/dqmf/dev/constraints/validators/IntrospectiveAssociationValidator.java
- **Test suite:** https://svn.globis.ethz.ch/repos/incubator/dqmf/core/development/src/test/java/ch/ethz/globis/dqmf/dev/testsuites/IntrospectiveAssociationTestSuite.java

A.4.1.2 SRC: ASSOCIATION COLLECTION METHOD

- **Domain model** (lecture)**:** `https://svn.globis.ethz.ch/repos/incubator/dqmf/core/components/src/test/java/ch/ethz/globis/dqmf/model/nonpermanent/Lecture.java`
- **Domain model** (student)**:** `https://svn.globis.ethz.ch/repos/incubator/dqmf/core/components/src/test/java/ch/ethz/globis/dqmf/model/nonpermanent/Student.java`
- **Annotation:** `https://svn.globis.ethz.ch/repos/incubator/dqmf/core/components/src/main/java/ch/ethz/globis/dqmf/constraints/annotations/association/VariableSize.java`
- **Validator:** `https://svn.globis.ethz.ch/repos/incubator/dqmf/core/components/src/main/java/ch/ethz/globis/dqmf/constraints/validators/VariableSizeValidator.java`
- **Interface:** `https://svn.globis.ethz.ch/repos/incubator/dqmf/core/components/src/main/java/ch/ethz/globis/dqmf/datastructure/Association.java`
- **Data structure:** `https://svn.globis.ethz.ch/repos/incubator/dqmf/core/components/src/main/java/ch/ethz/globis/dqmf/datastructure/association/AssociationCollection.java`
- **Test suite:** `https://svn.globis.ethz.ch/repos/incubator/dqmf/core/components/src/test/java/ch/ethz/globis/dqmf/testsuites/association/AssociationTestSuite.java`

A.5 SRC: TEMPORAL CONSTRAINT

A.5.1 SRC: DATA STRUCTURE

A.5.1.1 SRC: TEMPORAL INTERFACE

- **Interface:** `https://svn.globis.ethz.ch/repos/incubator/dqmf/core/components/src/main/java/ch/ethz/globis/dqmf/datastructure/Temporal.java`

A.5.1.2 SRC: PRIMITIVE TEMPORAL DATA TYPES

- **Classes:** `https://svn.globis.ethz.ch/repos/incubator/dqmf/core/components/src/main/java/ch/ethz/globis/dqmf/datastructure/temporal/`

A.5.1.3 SRC: TEMPORAL ASSOCIATION COLLECTION

- **Class:** `https://svn.globis.ethz.ch/repos/incubator/dqmf/core/components/src/main/java/ch/ethz/globis/dqmf/datastructure/temporal/TemporalAssociationCollection.java`

A.5.2 SRC: CONSTRAINTS

- **Annotation** (`@Deadline`)**:** `https://svn.globis.ethz.ch/repos/incubator/dqmf/core/components/src/main/java/ch/ethz/globis/dqmf/constraints/annotations/temporal`
- **Annotation** (`@AssertFalseOnDeadline`)**:** `https://svn.globis.ethz.ch/repos/incubator/dqmf/core/components/src/main/java/ch/ethz/globis/dqmf/constraints/annotations/temporal/AssertFalseOnDeadline.java`
- **Annotation** (`@MinOnDeadline`)**:** `https://svn.globis.ethz.ch/repos/incubator/dqmf/core/components/src/main/java/ch/ethz/globis/dqmf/constraints/annotations/temporal/MinOnDeadline.java`
- **Validator** (`@Deadline`)**:** `https://svn.globis.ethz.ch/repos/incubator/dqmf/core/components/src/main/java/ch/ethz/globis/dqmf/constraints/validators/DeadlineValidator.java`
- **Validator** (`@AssertFalseOnDeadline`)**:** `https://svn.globis.ethz.ch/repos/incubator/dqmf/core/components/src/main/java/ch/ethz/globis/dqmf/constraints/validators/AssertFalseOnDeadlineValidator.java`

A.6 SRC: TIME-TRIGGERED VALIDATION COMPONENT

A.6.1 SRC: TTVC: SCHEDULERS

- **Scheduler(JPA):** `https://svn.globis.ethz.ch/repos/incubator/dqmf/core/components/src/main/java/ch/ethz/globis/dqmf/ttvc/schedulers/JPAValidationScheduler.java`
- **Scripts:** `https://svn.globis.ethz.ch/repos/incubator/dqmf/core/components/src/main/resources/scripts/sql/`
- **Script runner utility:** `https://svn.globis.ethz.ch/repos/incubator/dqmf/core/components/src/main/java/ch/ethz/globis/dqmf/ttvc/utilities/ScriptRunner.java`
- **Properties file:** `https://svn.globis.ethz.ch/repos/incubator/dqmf/core/components/src/test/resources/quartz.properties`
- **Example:** `https://svn.globis.ethz.ch/repos/incubator/dqmf/core/components/src/test/java/ch/ethz/globis/dqmf/examples/ttvc/TTVCSchedulerExample.java`

A.6.2 SRC: TTVC: JOBS AND JOBDETAILS

A.6.2.1 SRC: TTVC: BASIC JOB

- **Basic job:** `https://svn.globis.ethz.ch/repos/incubator/dqmf/core/components/src/test/java/ch/ethz/globis/dqmf/ttvc/jobs/BasicJob.java`

A.6.2.2 SRC: TTVC: ABSTRACT VALIDATION JOB

- **Class:** `https://svn.globis.ethz.ch/repos/incubator/dqmf/core/components/src/main/java/ch/ethz/globis/dqmf/ttvc/jobs/AbstractValidationJob.java`
- **Domain model:** `https://svn.globis.ethz.ch/repos/incubator/dqmf/core/components/src/test/java/ch/ethz/globis/dqmf/model/persistent/SerializablePerson.java`
- **Example:** `https://svn.globis.ethz.ch/repos/incubator/dqmf/core/components/src/test/java/ch/ethz/globis/dqmf/examples/ttvc/TTVBasicValidationJobExample.java`

A.6.2.3 SRC: TTVC: ABSTRACT JPA VALIDATION JOB

- **Class:** `https://svn.globis.ethz.ch/repos/incubator/dqmf/core/components/src/main/java/ch/ethz/globis/dqmf/ttvc/jobs/AbstractJPAValidationJob.java`
- **Builder:** `https://svn.globis.ethz.ch/repos/incubator/dqmf/core/components/src/main/java/ch/ethz/globis/dqmf/ttvc/jobdetails/`
- **Domain model:** `https://svn.globis.ethz.ch/repos/incubator/dqmf/core/components/src/test/java/ch/ethz/globis/dqmf/model/persistent/PersistentAddress.java`
- **Example:** `https://svn.globis.ethz.ch/repos/incubator/dqmf/core/components/src/test/java/ch/ethz/globis/dqmf/examples/ttvc/TTVCBasicJPAValidationJobExample.java`

A.6.2.4 SRC: TTVC: UNIVERSAL JPA VALIDATION JOB

- **Class:** `https://svn.globis.ethz.ch/repos/incubator/dqmf/core/components/src/main/java/ch/ethz/globis/dqmf/ttvc/jobs/UniversalJPAValidationJob.java`
- **Domain model** (address)**:** `https://svn.globis.ethz.ch/repos/incubator/dqmf/core/components/src/test/java/ch/ethz/globis/dqmf/model/persistent/PersistentAddress.java`

- **Domain model** (person)**:** `https://svn.globis.ethz.ch/repos/incubator/dqmf/core/components/src/test/java/ch/ethz/globis/dqmf/model/persistent/PersistentPerson.java`
- **Example:** `https://svn.globis.ethz.ch/repos/incubator/dqmf/core/components/src/test/java/ch/ethz/globis/dqmf/examples/ttvc/TTVCUniversalJPAValidationJobExample.java`

A.6.3 SRC: TTVC: TRIGGERS

- **Example** (trigger and job detail connection)**:** `https://svn.globis.ethz.ch/repos/incubator/dqmf/core/components/src/test/java/ch/ethz/globis/dqmf/examples/ttvc/TTVCBasicTriggerExample.java`
- **Jobs:** `https://svn.globis.ethz.ch/repos/incubator/dqmf/core/components/src/test/java/ch/ethz/globis/dqmf/ttvc/jobs/`
- **Triggers:** `https://svn.globis.ethz.ch/repos/incubator/dqmf/core/components/src/main/java/ch/ethz/globis/dqmf/ttvc/triggers/`
- **Example** (daily trigger usage)**:** `https://svn.globis.ethz.ch/repos/incubator/dqmf/core/components/src/test/java/ch/ethz/globis/dqmf/examples/ttvc/TTVCDailyTriggerBuilderExample.java`
- **Example** (unrepeatable trigger usage)**:** `https://svn.globis.ethz.ch/repos/incubator/dqmf/core/components/src/test/java/ch/ethz/globis/dqmf/examples/ttvc/TTVCUnrepeatableTriggerBuilderExample.java`

A.6.4 SRC: TTVC: JOB LISTENER

- **Listener:** `https://svn.globis.ethz.ch/repos/incubator/dqmf/core/components/src/test/java/ch/ethz/globis/dqmf/ttvc/listener/jobs/BasicJobListener.java`
- **Example:** `https://svn.globis.ethz.ch/repos/incubator/dqmf/core/components/src/test/java/ch/ethz/globis/dqmf/examples/ttvc/TTVCListenerExample.java`

A.6.5 SRC: TTVC: PERSISTENT VALIDATION REPORT

- **Data structure:** `https://svn.globis.ethz.ch/repos/incubator/dqmf/core/components/src/main/java/ch/ethz/globis/dqmf/ttvc/validationreport/`
- **Example:** See example code and the Java™ class for the universal JPA validation job listed in section A.6.2.4

A.7 SRC: HARD AND SOFT CONSTRAINTS

A.7.1 SRC: SOFT CONSTRAINT IMPLEMENTATION

A.7.1.1 SRC: PAYLOAD-TRY-CATCH METHOD

- **Severity:** `https://svn.globis.ethz.ch/repos/incubator/dqmf/core/development/src/main/java/ch/ethz/globis/dqmf/dev/constraints/payloads/Severity.java`
- **Domain model:** `https://svn.globis.ethz.ch/repos/incubator/dqmf/core/development/src/test/java/ch/ethz/globis/dqmf/dev/model/persistent/PersistentPerson.java`
- **Test suite:** `https://svn.globis.ethz.ch/repos/incubator/dqmf/core/development/src/test/java/ch/ethz/globis/dqmf/dev/testsuites/TryCatchSoftConstraintJPATestSuite.java`

A.7.1.2 SRC: SOFT CONSTRAINTS VALIDATOR

- **Implementation** (EclipseLink)**:** `https://svn.globis.ethz.ch/repos/incubator/dqmf/core/development/src/test/java/ch/ethz/globis/dqmf/dev/impl/validator/`
- **Domain model:** `https://svn.globis.ethz.ch/repos/incubator/dqmf/core/development/src/test/java/ch/ethz/globis/dqmf/dev/model/persistent/PersistentPerson.java`
- **Test suite:** `https://svn.globis.ethz.ch/repos/incubator/dqmf/core/development/src/test/java/ch/ethz/globis/dqmf/dev/testsuites/SoftConstraintsValidatorJPATestSuite.java`

A.7.1.3 SRC: GROUP METHOD

- **Groups:** `https://svn.globis.ethz.ch/repos/incubator/dqmf/core/development/src/main/java/ch/ethz/globis/dqmf/dev/constraints/groups/`
- **Domain model:** `https://svn.globis.ethz.ch/repos/incubator/dqmf/core/development/src/test/java/ch/ethz/globis/dqmf/dev/model/nonpermanent/Person.java`
- **Test suite:** `https://svn.globis.ethz.ch/repos/incubator/dqmf/core/development/src/test/java/ch/ethz/globis/dqmf/dev/testsuites/SoftConstraintsWithGroupsTestSuite.java`
- **Test suite:** `https://svn.globis.ethz.ch/repos/incubator/dqmf/core/development/src/test/java/ch/ethz/globis/dqmf/dev/testsuites/SoftConstraintsWithGroupsTestSuite.java`

SRC: PENALTY IMPLEMENTATION

- **Penalty level interface:** `https://svn.globis.ethz.ch/repos/incubator/dqmf/core/components/src/main/java/ch/ethz/globis/dqmf/constraints/payloads/PenaltyLevel.java`
- **Groups:** `https://svn.globis.ethz.ch/repos/incubator/dqmf/core/components/src/main/java/ch/ethz/globis/dqmf/constraints/groups/`
- **Domain model** (non-permanent)**:** `https://svn.globis.ethz.ch/repos/incubator/dqmf/core/components/src/test/java/ch/ethz/globis/dqmf/model/nonpermanent/Person.java`
- **Domain model** (persistent)**:** `https://svn.globis.ethz.ch/repos/incubator/dqmf/core/components/src/test/java/ch/ethz/globis/dqmf/model/persistent/PersistentPerson.java`
- **Sample penalty levels** (low, medium, high)**:** `https://svn.globis.ethz.ch/repos/incubator/dqmf/core/components/src/main/java/ch/ethz/globis/dqmf/constraints/payloads/`
- **Test suite:** `https://svn.globis.ethz.ch/repos/incubator/dqmf/core/components/src/test/java/ch/ethz/globis/dqmf/testsuites/softconstraints/SoftConstraintsTestSuite.java`

SRC: APPLICATION

- **Utility classes:** `https://svn.globis.ethz.ch/repos/incubator/dqmf/core/components/src/main/java/ch/ethz/globis/dqmf/constraints/utilities/`
- **Constraint:** `https://svn.globis.ethz.ch/repos/incubator/dqmf/core/components/src/main/java/ch/ethz/globis/dqmf/constraints/annotations/membership/ContainsString.java`
- **Validator:** `https://svn.globis.ethz.ch/repos/incubator/dqmf/core/components/src/main/java/ch/ethz/globis/dqmf/constraints/validators/ContainsStringValidator.java`
- **Domain model:** `https://svn.globis.ethz.ch/repos/incubator/dqmf/core/components/src/test/java/ch/ethz/globis/dqmf/model/nonpermanent/Teacher.java`
- **Test suite:** `https://svn.globis.ethz.ch/repos/incubator/dqmf/core/components/src/test/java/ch/ethz/globis/dqmf/testsuites/softconstraints/SoftConstraintsPenaltyTestSuite.java`

The penalty level implementations and group interfaces can be found in the appendix section A.7.1.3.

A.8 SRC: CONSTRAINTS AND DATA QUALITY DIMENSIONS

- **Constraint:** `https://svn.globis.ethz.ch/repos/incubator/dqmf/core/components/src/main/java/ch/ethz/globis/dqmf/constraints/annotations/membership/ContainsString.java`
- **Validator:** `https://svn.globis.ethz.ch/repos/incubator/dqmf/core/components/src/main/java/ch/ethz/globis/dqmf/constraints/validators/ContainsStringValidator.java`
- **Domain model:** `https://svn.globis.ethz.ch/repos/incubator/dqmf/core/components/src/test/java/ch/ethz/globis/dqmf/model/nonpermanent/Teacher.java`
- **Test suite:** `https://svn.globis.ethz.ch/repos/incubator/dqmf/core/components/src/test/java/ch/ethz/globis/dqmf/testsuites/softconstraints/SoftConstraintsPenaltyTestSuite.java`

B

LIST OF ABBREVIATIONS

ADS Advantage Database Server

AJAX Asynchronous JavaScript and XML

AOP Aspect-Oriented Programming

API Application Programming Interface

ASE Adaptive Server Enterprise

AWT Abstract Window Toolkit

BSD Berkeley Software Distribution

CASE Computer-Aided Software Engineering

CDDL Common Development and Distribution Licence

CDI Context and Dependency Injection

CRUD Create, Read, Update, Delete

CSP Constraint Satisfaction Problem

CSS Cascading Style Sheets

DB Database

db4o Database for Objects

DBMS Database Management System

EE Enterprise Edition

ECA Event Condition Action

EDL Eclipse Distribution Licence

EPL Eclipse Public Licence

ETL Extract, Transform and Load

ExtVal Extensions Validator

GAC Generic Association Constraint

GPL General Public Licence

GUI Graphical User Interface

GWT Google Web Toolkit

HAC Hand-crafted Association Constraint

HQL Hibernate Query Language

HSQL Hyper Structured Query Language Database

HTML Hypertext Markup Language

HTTP Hypertext Transfer Protocol

HXTT Hongxin Technology and Trade Ltd. of Xiangtan City

IAC Introspective Association Constraint

IDE Integrated Development Environment

Java EE Java™ Platform, Enterprise Edition

JAX-RS Java™ API for RESTful Web Services

JCP Java™ Community Process

JDK Java Development Kit

JDO Java™ Data Objects

JET Java Emitter Template

JFC Java™ Foundation Classes

JML Java™ Modeling Language

JMS Java™ Message Service

JPA Java™ Persistence API

JPQL Java™ Persistence Query Language

JSF JavaServer™ Faces

JSON JavaScript Object Notation

JSR Java™ Specification Request

JTA Java Transaction API

LDAP Lightweight Directory Access Protocol

LGPL Lesser General Public Licence

LP Linear Programming

MVC Model View Controller

MDA Model-Driven Architecture

MDD Model-Driven Development

NoSQL Not only SQL

OCL Object Constraint Language

ODF Open Document Format

OGM Object/Grid Mapper

ORM Object/Relational Mapping

OS Operating System

PIM Platform-Independent Model

RAM Random Access Memory

RCP Rich Client Platform

REST Representational State Transfer

RFC Request for Comments

RIA Rich Internet Application

SE Standard Edition

SPI Service Provider Interface

SQL Structured Query Language

SRC Source Code

SWT Standard Widget Toolkit

TCK Technology Compatibility Kit

TLA Three-letter acronym

TTVC Time-Triggered Validation Component

UI User Interface

UML Unified Modeling Language

URL Uniform Resource Locator

W3C World Wide Web Consortium

XML Extensible Markup Language

C
LIST OF FIGURES

1.1 Screenshot of the Dropbox[1]registration process showing the violation errors if a user clicks on the 'Create account' button with an empty form. 3
1.2 Tier and layer overview regarding constraint definition possibilities in a typical Java™ environment[2]. 4

2.1 Visualisation of the thesis structure for part three III and four IV. Note that the chapter and section numbers are relative and not absolute references. 8

3.1 An example hierarchy of data quality dimensions adapted from [2]. . . 12
3.2 The data evolution life cycle as described in [9]. 13
3.3 The hierarchy as described in [9] puts the stages of the data evolution life cycle 3.2 into a hierarchy regarding the specificity of the data quality contribution. The hierarchy should be read as follows: upper level is more specific than the lower level. 14

4.1 Graphical representation of different real world system state mappings to information system states taken from [1]. Top row: the left mapping shows a real-life system that is properly mapped to an information system and the right mapping shows an incomplete mapping. Bottom row: The left mapping shows some ambiguity and the right one a meaningless state in the information system 16
4.2 Although there is a proper design, at operation time the user could map a real-life state to the wrong information state (this is called garbling). The user could be able to infer a corresponding real-life state based on the information system state but the inference is not correct. This theory is connected with the data quality dimension accuracy as presented in [3] 19

6.1 Running example used in [14] to analyse the transformation process of the individual tools. 30
6.2 System structure of the interceptor based approach for constraint violation as described in cite1409949. The 'Interceptor Manager' acts between the server and the client analysing the request/response messages. 31

7.1 Left: Rendering example of a validation violation with the default icons (traffic lights). Right: A form with an interdependency. It is not possible to answer the second question if 'no' is chosen. These examples can be tested on the project website of PowerForms[3]. 34

8.1 UML class diagram (adapted from [19]) with stereotypes that are translated to JSR 303 (Bean Validation 1.0) constraints. 38
8.2 Coding example that shows how to implement a constraint validation mechanism using if-then-else statements and exceptions as described in [20]. 39
8.3 Original method and a wrapper method which includes the validation logic and a call to the original method (taken from [20]). 40
8.4 Visualising the weaving process with a regular program and two aspects (taken from [21]). 40

9.1 Overview of the discussed tiers and layers with respect to possible validation technologies and the most important standards/specifications which are connected to a specific layer or tier. 46

10.1 Top part: Validation happens at different tiers and layers resulting in individual validation mechanisms for each tier and layer. Lower part: JSR 349 (Bean Validation 1.1) centralises the validation mechanism using constraint annotations within the domain model. Both pictures are taken from the reference implementation (Hibernate Validator) of the JSR 349 (Bean Validation 1.1) project website[4]. 53

11.1 HTML5 rendering example using Opera™ 12.16 and the form validation example of listing 11.1. 57

12.1 GUI example with a rendered error message using the JSF standard validator. 64
12.2 Life cycle of a HTTP request in a JSF application. 64
12.3 GUI example with a rendered error message using Hibernate Validator. 66
12.4 RichFaces rendering example of an error message. 68
12.5 GWT rendering example of a client validation violation. 74
12.6 GWT rendering example of a server validation violation. 'Name must be magic at the server' means that the entered name does not contain the word 'magic' as defined in the constraint depicted in 12.5. 74
12.7 GUI of the input form using Java™ Swing. 76
12.8 GUI of the input form using Swing Form Builder. 78
12.9 Rendered form using SWT and Hibernate Validator under Windows 7. 79

12.10JFace rendering example of a validation violation. 82
12.11JFace rendering example of a validation violation using Bean Validation. 85
12.12FXForm2 using JavaFX 2.2 to render a form with a validation violation using Hibernate Validator. 88
12.13An entity can be in three possible states: transient, persistent and detached. The flowchart shows the state transitions that can happen if the corresponding method is called. The figure was adapted from [39] 88
12.14Hibernate OGM architecture as presented on their project website[5]. . 103

15.1 Parts of the MySQL database structure which are able to store mapped objects of type `PersistentStudent` and `PersistentLecture`. . 129

16.1 UML class diagram within the university domain showing an association between lectures and students. 131
16.2 Snapshot of a possible application state showing a `HashMap` in the left part and the `AssociationDefinition` which stores the multiplicity and a reference to the corresponding association implementation in the right part. 139
16.3 UML diagram of the data structure for the association collection method. 147

17.1 UML class diagram of the `Temporal` interface. 152
17.2 UML class diagram of the primitive temporal data types which implement the `Temporal` interface to handle the temporal dimension. . . . 153
17.3 UML class diagram showing how the temporal association collection is designed. 154

18.1 UML class diagram showing the different possibilities to define a job. The `UniversalJPAValidationJob` class is a concrete example of a job implementation. 169
18.2 The first step is executed by calling the `executeFetchOperation` method. Next, it is possible to call the `getConstraintViolations` method to retrieve the set of constraint violations or the `getFetchedObjects` method to access the retrieved objects. 170
18.3 Instance of a possible `JPAValidationJobBuilder` state storing all relevant information and a list of `EntityValidationData` objects which represent the information to retrieve object from the data store using JPA features. 170
18.4 UML class diagram showing the validation report data structure which can be used to persist a validation result using JPA. 171
18.5 Screenshot after the execution of the universal JPA validation job example of the persistent validation report using MySQL as a data store. 171

19.1 'In a linear programming problem, a series of linear constraints produces a convex feasible region of possible values for those variables. In the two-variable case this region is in the shape of a convex simple polygon.'[6] 174

19.2 Visualising the concept of hard and soft constraints in the area of mechanical engineering.[7] 176
19.3 The interface `SoftConstraint` is extended by the interfaces `Info` and `Warning` which can be viewed as two intersecting sets sharing some properties. Validating both interfaces with the common ancestor is not possible with this design, because the `SoftConstraint` set does not fully cover both sets. 179
19.4 The `SoftConstraint` interface extends every special soft constraint group and therefore inherits the properties of `Info` and `Warning`. This way it is possible to validate the soft constraint subgroups with a single call to the validation API using the subinterface `SoftConstraint`. 180
19.5 UML class diagram showing three penalty level implementations: low, medium and high. 181

D
BIBLIOGRAPHY

[1] Batini, Carlo and Scannapieco, Monica. *Data Quality: Concepts, Methodologies and Techniques (Data-Centric Systems and Applications)*. Springer-Verlag New York, Inc., Secaucus, NJ, USA, 2006.

[2] Wang, Richard Y and Reddy, M.P. and Kon, Henry B. Toward quality data: An attribute-based approach. *Decision Support Systems*, 13(3-4):349–372, March 1995.

[3] Wand, Yair and Wang, Richard Y. Anchoring data quality dimensions in ontological foundations. *Communications of the ACM*, 39(11):86–95, November 1996.

[4] Madnick, Stuart E and Wang, Richard Y and Lee, Yang W and Zhu, Hongwei. Overview and Framework for Data and Information Quality Research. *Journal of Data and Information Quality*, 1(1):1–22, June 2009.

[5] Kemper, Alfons and Eickler, André. *Datenbanksysteme: Eine Einführung*. Oldenbourg, 7., akt. und erw. Auflage edition, 2009.

[6] Vassiliadis, Panos and Bouzeghoub, Mokrane and Quix, Christoph. Towards quality-oriented data warehouse usage and evolution. *Information Systems*, 25(2):89–115, April 2000.

[7] Shankaranarayanan, G and Cai, Yu. Supporting data quality management in decision-making. *Decision Support Systems*, 42(1):302–317, October 2006.

[8] Pipino, Leo L and Lee, Yang W and Wang, Richard Y. Data quality assessment. *Communications of the ACM*, 45(4):211–218, April 2002.

[9] Liu, L and Chi, LN. Evolutional Data Quality: A Theory-Specific View. In *Proceedings of the 7th International Conference on Information Quality (ICIQ-02)*, pages 292–304, 2002.

[10] Wang, Richard Y and Strong, Diane M. Beyond accuracy: what data quality means to data consumers. *Journal of Management Information Systems*, 12(4):5–33, 1996.

[11] Witt, Graham. *Writing Effective Business Rules*. Elsevier, 2012.

[12] Can Türker and Michael Gertz. Semantic integrity support in SQL:1999 and commercial (object-)relational database management systems. *The VLDB Journal*, 10(4):241–269, December 2001.

[13] Ronald G. Ross. Are Integrity Constraints Business Rules? Not! *Business Rules Journal*, 8(3), March 2007.

[14] Cabot, Jordi and Teniente, Ernest. Constraint Support in MDA Tools: A Survey. In *Proceedings of the Second European conference on Model Driven Architecture: foundations and Applications*, ECMDA-FA '06, pages 256–267, Berlin, Heidelberg, 2006. Springer-Verlag.

[15] Qianxiang Wang and Mathur, A.P. Interceptor based constraint violation detection. In *ECBS*, pages 457–464. IEEE Computer Society, 2005.

[16] Scaffidi, Christopher and Myers, Brad and Shaw, Mary. Topes: reusable abstractions for validating data. In *Proceedings of the 30th international conference on Software engineering*, ICSE '08, pages 1–10, New York, NY, USA, 2008. ACM.

[17] Brabrand, Claus and Møller, Anders and Ricky, Mikkel and Schwartzbach, Michael I. PowerForms: Declarative client-side form field validation. *World Wide Web*, 3(4):205–214, December 2000.

[18] Groenewegen, DannyM. and Visser, Eelco. Integration of data validation and user interface concerns in a DSL for web applications. *Software & Systems Modeling*, 12(1):35–52, 2013.

[19] Eban Escott, Paul Strooper, Paul King, and Ian J. Hayes. Model-driven web form validation with uml and ocl. In *Proceedings of the 11th international conference on Current Trends in Web Engineering*, ICWE '11, pages 223–235, Berlin, Heidelberg, 2012. Springer-Verlag.

[20] Froihofer, Lorenz and Glos, Gerhard and Osrael, Johannes and Goeschka, Karl M. Overview and Evaluation of Constraint Validation Approaches in Java. In *Proceedings of the 29th international conference on Software engineering*, ICSE '07, pages 313–322, Washington, DC, USA, 2007. IEEE Computer Society.

[21] Benjamin Mesing and Constantinos Constantinides and Wolfgang Lohmann. Limes: An Aspect-Oriented Constraint Checking Language. In Arend Rensink and Jos Warmer, editor, *ECMDA-FA*, volume 4066 of *Lecture Notes in Computer Science*, pages 299–315. Springer, 2006.

[22] Carmen Avila and Amritam Sarcar and Yoonsik Cheon and Cesar Yeep. Runtime Constraint Checking Approaches for OCL, A Critical Comparison. In *SEKE*, pages 393–398. Knowledge Systems Institute Graduate School, 2010.

[23] Hamie, Ali. Translating the Object Constraint Language into the Java Modelling Language. In *Proceedings of the 2004 ACM symposium on Applied computing*, SAC '04, pages 1531–1535, New York, NY, USA, 2004. ACM.

[24] Carmen Avila and Guillermo Flores and Yoonsik Cheon. A Library-Based Approach to Translating OCL Constraints to JML Assertions for Runtime Checking. In *International Conference on Software Engineering Research and Practice, July 14-17, 2008, Las Vegas*, pages 403–408, 2008.

[25] Verheecke, Bart and Van Der Straeten, Ragnhild. Specifying and implementing the operational use of constraints in object-oriented applications. In *Proceedings of the Fortieth International Conference on Tools Pacific: Objects for internet, mobile and embedded applications*, CRPIT '02, pages 23–32, Darlinghurst, Australia, Australia, 2002. Australian Computer Society, Inc.

[26] Weber, David and Leone, Stefania and Norrie, Moira C. Constraint-Based Data Quality Management Framework for Object Databases. In *ECIS*, Utrecht, The Netherlands, 2013.

[27] Jagadish, H. V. and Qian, Xiaolei. Integrity Maintenance in Object-Oriented Databases. In *Proceedings of the 18th International Conference on Very Large Data Bases*, VLDB '92, pages 469–480, San Francisco, CA, USA, 1992. Morgan Kaufmann Publishers Inc.

[28] Hussien Oakasha and Stefan Conrad and Gunter Saake. Consistency Management in Object-Oriented Databases. In *ECOOP Workshop on Object-Oriented Databases Lisbon*, pages 1–5, 1999.

[29] Medeiros, Claudia Bauzer and Pfeffer, Patrick. Object Integrity Using Rules. In *Proceedings of the European Conference on Object-Oriented Programming*, ECOOP '91, pages 219–230, London, UK, 1991. Springer-Verlag.

[30] Demuth Birgit and Hussmann Heinrich and Loecher Sten. OCL as a Specification Language for Business Rules in Database Applications. In *Proceedings of the 4th International Conference on The Unified Modeling Language, Modeling Languages, Concepts, and Tools*, ≪UML≫'01, pages 104–117, London, UK, 2001. Springer-Verlag.

[31] Demuth, Birgit and Hussmann, Heinrich. Using UML/OCL constraints for relational database design. In *Proceedings of the 2nd international conference on The unified modeling language: beyond the standard*, UML '99, pages 598–613, Berlin, Heidelberg, 1999. Springer-Verlag.

[32] Java™ Specification Request (JSR) 349: Bean Validation 1.1. `http://jcp.org/en/jsr/detail?id=349`, February 2013. [Online; accessed 23-April-2013].

[33] JavaBeans™ Specification 1.01. `http://download.oracle.com/otndocs/jcp/7224-javabeans-1.01-fr-spec-oth-JSpec/`, August 1997. [Online; accessed 13-May-2013].

[34] Java™ Specification Request (JSR) 303: Bean Validation 1.0. `http://jcp.org/en/jsr/detail?id=303`, October 2009. [Online; accessed 13-May-2013].

[35] World Wide Web Consortium (W3C). http://www.w3.org/, August 2013. [Online; accessed 01-August-2013].

[36] Hypertext Markup Language (HTML) 5. http://www.w3.org/TR/2012/CR-html5-20121217/, December 2012. [Online; accessed 01-August-2013].

[37] Java™ Specification Request (JSR) 314: JavaServer™ Faces (JSF) 2.1. http://www.jcp.org/en/jsr/detail?id=314, November 2010. [Online; accessed 14-May-2013].

[38] Java™ Specification Request (JSR) 154: Java™ Servlet 2.4. http://www.jcp.org/aboutJava/communityprocess/final/jsr154/, November 2003. [Online; accessed 15-May-2013].

[39] Röder, Daniel. *JPA mit Hibernate: Java Persistence API in der Praxis*. entwickler.press, Frankfurt am Main, 2010.

[40] Java™ Specification Request (JSR) 338: Java™ Persistence API 2.1. http://jcp.org/en/jsr/detail?id=338, May 2013. [Online; accessed 05-June-2013].

[41] Java™ Specification Request (JSR) 243: Java™ Data Objects 3.0. http://jcp.org/en/jsr/detail?id=243, August 2010. [Online; accessed 18-June-2013].

[42] Java™ Specification Request (JSR) 317: Java™ Persistence API 2.0. http://www.jcp.org/en/jsr/detail?id=317, December 2009. [Online; accessed 14-June-2013].

[43] Henrique Rocha and Marco Tulio Valente. How Annotations are Used in Java: An Empirical Study. In *SEKE*, pages 426–431. Knowledge Systems Institute Graduate School, 2011.

[44] Java SE 7 API Documentation for java.util Interface Collection<E>. http://docs.oracle.com/javase/7/docs/api/index.html, July 2011. [Online; accessed 13-August-2013].

[45] Java™ Specification Request (JSR) 341: Java™ Expression Language 3.0. http://jcp.org/en/jsr/detail?id=341, May 2013. [Online; accessed 03-October-2013].

[46] Java SE 7 API Documentation for java.util Class Date. http://docs.oracle.com/javase/7/docs/api/index.html, July 2011. [Online; accessed 13-August-2013].

[47] Java SE 7 API Documentation for java.lang Class Number. http://docs.oracle.com/javase/7/docs/api/java/lang/Number.html, July 2011. [Online; accessed 16-August-2013].

[48] Java SE 7 API Documentation for java.lang Interface CharSequence. http://docs.oracle.com/javase/7/docs/api/java/lang/CharSequence.html, July 2011. [Online; accessed 19-August-2013].

[49] V. Srikumar. Soft Constraints in Integer Linear Programs. [Online; accessed 26-September-2013], February 2013.

[50] Nebel, Bernhard and Hué, Julien and Wölfl, Stefan. Constraint Satisfaction Problems: Constraint Optimization. [Online; accessed 04-September-2013], July 2012.