

Diss. ETH No. 24471

A Constraint-Based Approach to Data Quality in Information Systems

A thesis submitted to attain the degree of
DOCTOR OF SCIENCES of ETH ZURICH
(Dr. sc. ETH Zurich)

presented by

David Weber

Dipl. Inform., University of Zurich

born April 29, 1976

citizen of Küsnacht, ZH Switzerland and Germany

accepted on the recommendation of

Prof. Dr. Moira C. Norrie, examiner
Prof. Dr. Ernest Teniente, co-examiner

2017

Sometimes you will never know the value of a moment, until it becomes a memory.

– *Theodor Seuss Geisel*

Abstract

Data volume, diversity of data, data sources and access to data have all fundamentally changed over the past years. Nowadays, the volume of data produced and used by organisations has grown in a way beyond what previously could have been imagined. Today's information systems store and manage structured, unstructured and semi-structured data, but also multimedia data such as videos, maps and images. Moreover, the number of people and devices that access data has increased greatly and it is now common for employees to access data globally in a distributed manner from different devices including laptops, tablets and mobile phones.

Due to these changes, information management has become even more challenging and complex than it has ever been. Solutions to the management of data in general, and specifically to data quality control, are challenged all the more as the importance and complexity of data quality also increases. Now then, how can good quality of data in an information system be ensured?

The reliability and usefulness of an information system is largely determined by the quality of the data that serves as input and is processed by the system. But to ensure a high level of data quality, it is necessary to precisely define data quality requirements expressed as constraints which are based on an information system's business rules. However, there is a lack of widely accepted standards, approaches, technologies and tools for measuring, managing and enhancing data quality based on constraint checking.

In our work, we acknowledge that data quality has a central role in information systems development by meeting the need for incorporating considerations on data quality into the complete data cycle encompassing not only managerial but also technical aspects, such as semantic integrity constraints. We aim to provide approaches to express various aspects of data quality as an information system output by the means of constraint checking on certain data input.

Constraint definitions tend to be distributed across the components of an information system using a variety of technology-specific representations. This causes not only more effort to define and check constraints but also additional work for maintaining constraints if business rules change which bears the risk of inconsistency within the constraint definitions in the various components.

Having indicators for data quality at hand for the different levels of data granularity such as single attributes, single instances, sets of instances, sets of entities and the overall quality of a dataset is desirable for the analysis and improvement of the data quality in an information system. The use of constraint checking methods for simplifying the evaluation of such data quality indicators appears to be applicable since (integrity) constraints and quality can both be described by assertions. Nonetheless, it is not clear how a set of constraints can be used to measure and indicate the quality of data.

Moreover, linking constraints to data quality is difficult since constraints usually

are defined upfront and, therefore, can never be exhaustive. This means that a dataset that satisfies every constraint might still be of low quality.

The goal of our work was to facilitate constraint management for software developers and take advantage of the opportunity to make use of constraint checking for indications of data quality by embedding data integrity in a continuous data quality improvement process. We therefore see data quality as an extended constraint compliance problem.

To tackle the issue of distributed constraints, we investigated alternative approaches for unified constraint management regarding the definition and checking of constraints. All of the approaches had the goal of obtaining a single, extended constraint model that avoids any inconsistencies and redundancy within the constraint specification and data validation processes. One of our approaches also follows the Model-Driven Architecture approach of using a technology-independent model that serves as a base for technology-specific code generation.

For indicating data quality based on constraint compliance, we introduced a fine-grained severity scale for constraint violations which enables less and more important constraints to be distinguished instead of traditional constraints which usually validate to either true or false. In most of our approaches, an individual constraint can be weighted with violation severity values associated, if needed, also with data quality dimensions. Based on these violation scores, we introduced metrics and measurements to provide indications of data quality for the various levels of data granularity. With an ‘error threshold’ we distinguish between traditional constraints which prohibit saving data if they are violated and a form of more relaxed constraints which allow data to be saved even if they are violated. This allows us to take different actions based on the type of violations encountered.

We further introduced data quality assessment profiles, which configure user-specific or application-specific constraints, associated with individual constraint violation severities, to enable user-specific and application-specific constraint checking with appropriate specific data quality feedback. These data quality assessment profiles can easily be adapted to different needs and are applicable for different users or applications, respectively. The need to distinguish constraint checking for different user groups was also affirmed in a large research project for food science data (TDS-Exposure) involving many different stakeholders for various countries across Europe. Moreover, we also investigated possibilities for defining constraints during the runtime of an application which was possible in some of our proposed approaches.

One large part of our work was to investigate ways of giving feedback on data quality. We studied and developed different variations for data input and data analysis feedback including adaptive data input and data quality feedback responsiveness in the case of varying constraint sets or feedback in different applications, respectively. We investigated integrated data quality management frameworks within specific applications but also independent data quality management frameworks for input and analysis feedback and monitoring of data quality including the evolution of data quality over time.

As mentioned above, low quality of a dataset cannot completely be assured by monitoring the constraint compliance of an information system since constraint compliance only serves as an objective data quality assessment. There is a need for a subjective data quality assessment where the subjective data quality indicates the perception on the quality of data of different stakeholders, such as the individuals that enter, provide

and use the data. The perceptions may vary between the individuals but can provide comparisons to the objective data quality provided by constraint compliance. Thus, we also investigated methods of how to include explicit and implicit data quality feedback from users of an information system and how to use this information in combination with other metrics, such as the constraint compliance.

Our food science data management application ‘FoodCASE’, which consists of a server logic, two clients, a web interface and a mobile application, served as a scenario testbed for our investigations.

Overall, our work provides different approaches for data quality management frameworks and multiple tools which facilitate constraint management for software developers and embed data integrity in a dynamic, continuous data quality improvement process in software engineering, based on but not limited to constraint compliance.

Zusammenfassung

Datenvolumen, -vielfalt, -quellen und -zugriff haben sich in den vergangenen Jahren grundlegend verändert. Heutzutage ist die Menge des Datenvolumens, das von Organisationen produziert und genutzt wird, weit über das hinaus gestiegen, was man sich vorstellen konnte. Die heutigen Informationssysteme speichern und verwalten strukturierte, unstrukturierte und halbstrukturierte Daten, aber auch Multimediadaten wie Videos, Karten und Bilder. Darüber hinaus ist die Anzahl der Personen und Geräte, die auf Daten zugreifen, stark gestiegen und es ist üblich, dass Mitarbeiter weltweit von verschiedenen Geräten wie Laptops, Tablets und Mobiltelefonen auf Daten zugreifen.

Aufgrund dieser Entwicklung ist Informationsmanagement noch schwieriger und komplexer geworden als je zuvor. Da die Bedeutung und Komplexität der Datenqualität ebenfalls zunimmt, sind Lösungen für Datenmanagement im Allgemeinen und Datenqualitätskontrolle im Speziellen umso mehr gefordert. Nun, wie kann gute Datenqualität in einem Informationssystem gewährleistet werden?

Die Zuverlässigkeit und Nützlichkeit eines Informationssystems werden weitgehend durch die Qualität der Daten bestimmt, die als Eingabe dienen und vom System verarbeitet werden. Um ein hohes Mass an Datenqualität zu gewährleisten, ist es notwendig, die Anforderungen an die Datenqualität genau zu definieren, die als *Constraints* (Bedingungen/Einschränkungen) ausgedrückt werden und auf den Regeln eines Informationssystems basieren. Allerdings fehlt es an weit verbreiteten Standards, Ansätzen, Technologien und Werkzeugen zur Messung, Verwaltung und Verbesserung der Datenqualität auf Grundlage der Überprüfung von Constraints.

Datenqualität spielt eine zentrale Rolle in der Entwicklung von Informationssystemen, da es notwendig ist, Datenqualitätsüberlegungen in den gesamten Datenzyklus einzubeziehen. Dies umfasst nicht nur betriebswirtschaftliche, sondern auch technische Aspekte, wie z.B. semantische Integritätsbeschränkungen (Semantic Integrity Constraints). Mit der Validierung von Daten, mittels der Überprüfung von Constraints, zielen wir darauf ab, verschiedene Aspekte der Datenqualität als Ausgabe eines Informationssystems auszudrücken.

Constraint-Definitionen neigen dazu, über die Komponenten eines Informationssystems mit einer Vielzahl von technologiespezifischen Repräsentationen verteilt zu sein. Dies führt nicht nur zu mehr Aufwand, um Constraints zu definieren und zu überprüfen, sondern auch zu zusätzlicher Arbeit für die Wartung von Constraints, wenn Geschäftsregeln ändern. Dies birgt ein Risiko für Inkonsistenz innerhalb der Constraint-Definitionen in den verschiedenen Komponenten.

Für die Analyse und Verbesserung der Datenqualität in einem Informationssystem sind Indikatoren für die Datenqualität auf den verschiedenen Ebenen der Datengranularität wünschenswert. Diese Ebenen beinhalten einzelne Attribute, einzelne Instanzen, Sätze von Instanzen, Sätze von Entitäten und die Gesamtqualität eines Datensatzes.

Die Verwendung der Überprüfung von Constraints zur Vereinfachung der Auswertung solcher Datenqualitätsindikatoren scheint naheliegend zu sein, da (Integritäts-) Constraints und Qualität beide durch Aussagen (Assertions) beschrieben werden können. Dennoch ist es nicht offensichtlich, wie Constraints verwendet werden können, um die Qualität der Daten zu messen und anzugeben.

Darüber hinaus ist die Verknüpfung von Constraints mit der Datenqualität schwierig, da Constraints in der Regel im Voraus definiert sind und daher niemals erschöpfend sein können. Dies bedeutet, dass ein Datensatz, der alle Constraints erfüllt, immer noch von geringer Qualität sein könnte.

Das Ziel unserer Arbeit war es, die Verwaltung von Constraints für Softwareentwickler zu erleichtern. Des Weiteren waren wir bestrebt, die Überprüfung von Constraints für Hinweise zur Datenqualität durch Einbettung der Datenintegrität in einen kontinuierlichen Prozess zur Verbesserung der Datenqualität zu nutzen. Wir fassen daher Datenqualität als erweiterte Problematik der *Constraint-Compliance* (Constraint-Erfüllung) auf.

Um das Problem verteilter Constraints in Angriff zu nehmen, haben wir alternative Ansätze für eine einheitliche Verwaltung von Constraints hinsichtlich der Definition und Überprüfung von Constraints untersucht. Alle Ansätze hatten das Ziel, ein einziges, erweitertes Constraint-Modell zu erhalten, das jegliche Inkonsistenzen und Redundanz innerhalb der Constraint-Definitionen und des Datenvalidierungsprozesses vermeidet. Einer unserer Ansätze folgt auch dem modellgetriebenen Architektur-Ansatz, ein technologieunabhängiges Modell zu nutzen, das als Grundlage für die technologiespezifische Codegenerierung dient.

Um Hinweise zur Datenqualität auf der Grundlage der Constraint-Compliance zu gewährleisten, haben wir eine feinkörnige Schweregrad-Skala (Severity Scale) für Constraint-Verletzungen eingeführt, die es ermöglicht, dass weniger und wichtigere Constraints statt traditioneller Constraints unterschieden werden können. Traditionelle Constraints werden in der Regel entweder als wahr (true) oder falsch (false) validiert. In den meisten unserer Ansätze kann ein individueller Constraint mit Verletzungsschwerewerten gewichtet werden, die bei Bedarf auch mit Datenqualitäts-Dimensionen assoziiert werden können. Basierend auf diesen Verletzungswerten haben wir Metriken und Messungen eingeführt, um Hinweise auf die Datenqualität für die verschiedenen Ebenen der Datengranularität zu erhalten. Mit einer Fehlerschwelle (Threshold) unterscheiden wir zwischen traditionellen Constraints, die das Speichern von Daten verhindern, wenn sie verletzt werden, und einer Form von gelockerten Constraints, die es ermöglichen, dass Daten gespeichert werden, auch wenn die Constraints verletzt werden. Dies ermöglicht es uns, verschiedene Prozesse (Actions) zu definieren, die auf der Art der Constraint-Verletzungen basieren.

Wir haben auch Datenqualitätsbewertungsprofile eingeführt, welche benutzer- oder anwendungsspezifische Constraints konfigurieren, die mit individuellen Constraints-Verletzungen assoziiert sind, um eine benutzer-, respektive anwendungsspezifische Überprüfung von Constraints mit entsprechendem spezifischen *Feedback* (Rückmeldung) zur Datenqualität zu ermöglichen. Diese Datenqualitätsbewertungsprofile lassen sich problemlos an unterschiedliche Bedürfnisse anpassen und gelten für unterschiedliche Benutzer bzw. Applikationen. Die Notwendigkeit, die Überprüfung von Constraints für verschiedene Benutzergruppen zu unterscheiden, wurde auch in einem grossen Forschungsprojekt für lebensmittelwissenschaftliche Daten (TDS-Exposure) mit vielen ver-

schiedenen *Stakeholdern* (Geschäftsinteressenten) für verschiedene Länder in ganz Europa bestätigt. Darüber hinaus haben wir auch die Möglichkeiten zur Festlegung von Constraints während der Laufzeit einer Anwendung untersucht, was in einigen unserer vorgeschlagenen Ansätze realisierbar war.

Wir befassten uns in unserer Arbeit zu einem grossen Teil damit, Möglichkeiten für Feedback zur Datenqualität zu prüfen. Folglich haben wir verschiedene Variationen für Feedback bezüglich Dateneingabe und -analyse untersucht und entwickelt. Dies umfasste auch adaptive Dateneingabe und Datenqualitäts-Feedback-Responsiveness für unterschiedliche Constraints-Sätze oder Feedback in verschiedenen Anwendungen. Wir untersuchten integrierte Datenqualitäts-Verwaltungs-Frameworks in spezifischen Anwendungen, aber auch solche, die von Anwendungen unabhängig sind. Dabei stand sowohl das Datenqualitäts-Feedback während der Dateneingabe und -analyse als auch das Datenqualitäts-Feedback für die Überwachung (Monitoring) der Datenqualität im Vordergrund, einschliesslich der Entwicklung der Datenqualität im Laufe der Zeit.

Wie oben erwähnt, kann eine geringe Qualität eines Datensatzes nicht vollständig durch die Überwachung der Constraint-Compliance eines Informationssystems sichergestellt werden, da die Constraint-Compliance nur als objektive Datenqualitätsbewertung dient. Es besteht die Notwendigkeit einer subjektiven Datenqualitätsbewertung, bei der die subjektive Datenqualität die Wahrnehmung der Datenqualität der verschiedenen Stakeholder anzeigt. Die Stakeholder sind in diesem Falle z.B. Personen, die Daten eingeben, zur Verfügung stellen und/oder benutzen. Die Wahrnehmungen können zwischen den Individuen variieren, ermöglichen jedoch Vergleiche mit der objektiven Datenqualität, die durch die Einhaltung der Constraints repräsentiert wird. Somit haben wir auch Methoden untersucht, wie man explizites und implizites Feedback zur Datenqualität von Nutzern eines Informationssystems sammeln und wie man diese Informationen in Kombination mit anderen Metriken wie der Constraint-Compliance einsetzen kann.

Unsere Applikation zur Verwaltung von wissenschaftlichen Lebensmitteldaten 'FoodCASE', die aus einer Server-Logik, zwei Clients, einem Web-Interface und einer Mobile Applikation besteht, diente als Szenario für unsere Untersuchungen.

Insgesamt bietet unsere Arbeit verschiedene Ansätze für Datenqualitäts-Verwaltungs-Frameworks und mehrere Tools, die die Verwaltung von Constraints für Softwareentwickler erleichtern und die Datenintegrität in einen dynamischen, kontinuierlichen Prozess zur Verbesserung der Datenqualität in der Software-Entwicklung integrieren, basierend, aber nicht beschränkt auf Constraint Compliance.

Acknowledgements

This work would not have been possible without the help and support of countless people and I would like to thank everyone.

First and foremost, I would like to thank Prof. Moira C. Norrie for giving me the opportunity to pursue a PhD in the GlobIS research group at ETH Zurich. She provided me with constant support throughout my doctoral studies and has also given me the freedom to follow my own ideas.

I would also like to thank Prof. Ernest Teniente who kindly agreed to be co-examiner of my work and has provided valuable feedback, which helped to improve my thesis.

Furthermore, I am grateful to the past and present members of the GlobIS group, who welcomed me in the group and supported me in the past years. I am especially grateful to Dr. Karl Presser with whom I was working on the TDS-Exposure project and who introduced me to the food science community. It was a pleasure to share the office, attend the various TDS-Exposure project meetings together and our countless discussions helped to improve our work.

My gratitude also extends to my master and bachelor students who contributed to this thesis: Michael Berli, Diana Birenbaum, Konstantina Gemenetzi, Benjamin Oser, Oliver Probst and Jakub Szymanek.

Further thanks go to my former colleagues at SIX and Netcetera, especially Frank Hähni, not only for his constant support during my employment at SIX Card Solutions, but also for his commitment to conduct a user study at SIX, and Martin Meier for the many inspiring professional discussions concerning distributed constraints.

Finally, I would like to thank my friends for always supporting and encouraging me. Especially, I am grateful to – Christian Capello, Leonhard Gallagher, Stefan Gehrig, Gian Koch, Stefania Leone, Rafael Perez, and Tilmann Zäschke – for being there for me whenever I needed an advice, a kind word or a friendly face.

Last, but by no means least, I am most grateful to my wife Simone, my parents Heide-Maria and Hubert for all their love and encouragement. Thank you, this thesis would never have been possible without you!

This thesis is dedicated to my two daughters, Anuk and Erin, who both were born during the years I was working on this thesis and who enlighten my life every day.

Table of Contents

1	Introduction	1
1.1	Case study/scenario	3
1.2	Motivation	5
1.2.1	Unified constraint management	6
1.2.2	Data quality expressed by constraints	7
1.2.3	Extended constraint checking	8
1.2.4	Feedback on data quality	9
1.3	Challenges	9
1.3.1	Unified constraint management	9
1.3.2	Data quality expressed by constraints	10
1.3.3	Extended constraint checking	11
1.3.4	Feedback on data quality	11
1.3.5	Summary and research questions	12
1.4	Contribution	12
1.4.1	Unified constraint management	13
1.4.2	Data quality expressed by constraints	14
1.4.3	Extended constraint checking	16
1.4.4	Feedback on data quality	17
1.4.5	Food science	17
1.4.6	Summary	18
1.5	Thesis overview	18
2	Background	19
2.1	Data quality	19
2.2	Data quality frameworks	22
2.3	Constraints	22
2.3.1	Basic constraints	23
2.3.2	Integrity constraints	24
2.4	General concepts	24
2.4.1	Hard-coded constraint checking	24

2.4.2	Structured Query Language	25
2.4.3	Annotation-based constraint checking	26
2.4.4	Event Condition Action paradigm	27
2.4.5	Model-Driven Architecture	27
2.4.6	Aspect-Oriented Programming	28
2.4.7	Design by Contract	29
2.4.8	Business Rules Engines	30
2.5	Unified constraint management	31
2.5.1	Constraint management functionality in databases	35
2.5.2	Bean Validation	36
2.5.3	Aspect-Oriented Programming	37
2.5.4	Business Rules Engines	38
2.5.5	Constraint translations	39
2.6	Data quality with regards to constraints	41
2.6.1	Data integrity rules for continuous data quality improvement	43
2.6.2	Data quality measurement and metrics	43
2.7	Giving feedback on data quality	49
2.8	Data quality and constraints in mobile applications	52
2.9	Conclusions	53
3	Initial approaches	55
3.1	Guidance on data quality feedback	58
3.1.1	Approach	58
3.1.2	Evaluation	62
3.1.3	Conclusions	63
3.2	Object database data quality management	64
3.2.1	Approach	65
3.2.2	Constraints framework	66
3.2.3	Conclusions	69
3.3	Data quality extended Bean Validation	69
3.3.1	Approach	70
3.3.2	Managing data quality using constraints	72
3.3.3	Use of framework	73
3.3.4	Implementation	76
3.3.5	Conclusions	78
3.4	Bean Validation OCL extension	79
3.4.1	Approach	80
3.4.2	Managing data quality using constraints	82
3.4.3	Use of framework	83

3.4.4	Implementation & evaluation	86
3.4.5	Conclusions	87
3.5	Data quality monitoring framework	88
3.5.1	Approach	88
3.5.2	Computing data quality	90
3.5.3	Data quality feedback	92
3.5.4	Implementation	94
3.5.5	Evaluation	97
3.5.6	Conclusions	99
3.6	Summary	99
4	Unified data quality assurance and analysis based on Bean Validation	101
4.1	Approach	103
4.1.1	Data quality indicators	104
4.1.2	Data quality measurement and calculation	105
4.1.3	Contracts	107
4.1.4	Error threshold	107
4.1.5	Validation groups	108
4.1.6	Constant importance weights	108
4.2	Data quality feedback	108
4.2.1	Validation and feedback configuration	111
4.3	Implementation	114
4.4	Evaluation	116
4.5	Conclusions	117
5	System-wide constraint representations with UnifiedOCL	119
5.1	Approach	121
5.2	UnifiedOCL	123
5.3	Constraint translations	126
5.4	Evaluation	128
5.5	Conclusions	130
6	Data quality management in mobile applications	133
6.1	Related work	134
6.2	Approach	135
6.2.1	Constraint injection	136
6.2.2	Adaptive user interface	138
6.2.3	Data quality feedback responsiveness	140
6.3	Implementation	141
6.3.1	Constraint injection	142

6.3.2	Adaptive user interface	144
6.3.3	Data quality feedback	145
6.4	Evaluation	147
6.5	Conclusions	149
7	Hybrid approach with subjective data quality assessment	151
7.1	Approach	153
7.2	Unified constraint management	156
7.3	Constraint checking	160
7.4	Data quality analysis	162
7.4.1	Data quality vector space model	164
7.4.2	Data quality improvement tendencies	166
7.4.3	Data quality improvement urgency score	169
7.5	Implementation	176
7.5.1	DQAnalytics Library	177
7.5.2	DQAnalytics Web module	177
7.5.3	DQAnalytics application	180
7.6	Evaluation	183
7.7	Conclusions	186
8	Conclusion & outlook	189
8.1	Final approaches summary	191
8.2	Discussion	194
8.3	Outlook	197
A	TDS-Exposure workshop 2015 – Questionnaire	199
B	TDS-Exposure workshop 2015 – Questionnaire findings	203
C	List of supported constraints with UnifiedOCL labels	207
D	List of supported built-in field constraint types of the DQAnalytics framework	211
E	Various data quality analysis visualisations of the DQAnalytics framework	213
F	User study of the DQAnalytics framework	217
G	Student contributions	223
	Acronyms	241

List of Figures

2.1	A data quality framework with 15 dimensions identified by Wang and Strong in 1996	20
2.2	Example of a distributed information system	32
2.3	Typical three-tier architecture	34
2.4	Various feedback variants	50
2.5	Twitter	50
3.1	Data quality input feedback	59
3.2	Tree visualisation of data quality analysis feedback	61
3.3	Problem table view	61
3.4	Approach overview	66
3.5	Association constraint	67
3.6	Constraints framework	68
3.7	Unified constraint model with centralised validation mechanism using Bean Validation	70
3.8	Data quality management framework features	73
3.9	UML class diagram for the university domain	74
3.10	Association collection	77
3.11	Time-triggered validation classes	78
3.12	Penalty level <code>LowPenalty</code>	78
3.13	Framework components	81
3.14	Implementation overview	86
3.15	Concept overview	90
3.17	Hierarchy graph	92
3.16	DQ analyser overview	93
3.18	Various feedback variants	94
3.19	Main components	95
4.1	Typical three-tier architecture with unified validation mechanism using Bean Validation	104

4.2	Data quality input validation	109
4.3	Data quality analysis tool	110
4.4	Constraints per weight interval	111
4.5	Percentage of constraint violations as pie chart	111
4.6	Percentage of constraint violations as bar chart	112
4.7	Contract editing and creation	112
4.8	General validation and analysis settings	113
4.9	Input validation settings	114
5.1	Approach overview	122
6.1	Scenario information system	135
6.2	Components integration	136
6.3	Constraint injection overview	137
6.4	Example constraint set	138
6.5	Adapted example constraint set	139
6.6	Adaptive interface generation	140
6.7	Different screen sizes	141
6.8	Mandatory input validation	144
6.9	Basic version: responsive data quality feedback	145
6.10	Weather version	146
6.11	Modal version	147
7.1	Data quality management framework components overview	154
7.2	Constraint repository structure [level 0-2]	158
7.3	Constraint repository structure [level 2-5]	158
7.4	Client-side constraint checking	162
7.5	Data quality vector space model	164
7.6	Data quality improvement tendencies	167
7.7	Coordinate system with angle bisections	170
7.8	Complex plane transformation	170
7.9	Shifted absolute sine for constraint compliance	172
7.10	Urgency score map for $w_i = 0$, $w_c = 1$, $w_p = 2$, $w_f = 4$	174
7.11	Data quality improvement urgency score	175
7.12	Evolution of the data quality of an instance over time	176
7.13	FoodCASEWeb GUI – Detail dialogue for a food instance with constraint violation	178
7.14	FoodCASEWeb GUI – Explicit feedback form attached to the food detail dialogue	179

7.15	DQAnalytics scoreboard – Evolution of the constraint compliance on an entity level	180
7.16	DQAnalytics – Improvement tendency chart	181
7.17	DQAnalytics – Urgency score chart	183
E.1	DQAnalytics – Vector space model chart	213
E.2	DQAnalytics – Dimension compliance chart	214
E.3	DQAnalytics – Constraint compliance analysis chart	215
E.4	DQAnalytics – List of explicit feedback	216
F.1	User study task description sheet	222

List of Tables

2.1	Selection of most widely used data quality dimensions	21
3.1	Overview of the initial approaches	56
3.2	Chosen feedback types and analysis options	98
4.1	Aggregated data quality indicators overview	105
7.1	Urgency score parameters	173
8.1	Approaches categorisation	191
8.2	Overview of the final approaches	192
B.1	Findings related to the concept of contracts	204
B.2	Findings related to input validation	205
B.3	Findings related to data quality analysis	205
B.4	Findings related to user access rights	206
B.5	Findings related to constraints	206
C.1	Bean Validation – Supported constraints	208
C.2	Hibernate Validator – Supported constraints	209
C.3	SQL (PostgreSQL) – Supported constraints	210
D.1	Built-in field constraint types	212

1

Introduction

Due to the development of the Internet, we are living in an era of massive information explosion where cloud technology and ‘Big Data’ are omnipresent. Especially for enterprises, data has become the most valuable asset and may even become the biggest trading commodity in the future. Despite the absence of a clear definition, it is widely agreed that Data Quality (DQ) is of major importance to information systems. DQ can help organisations reduce costs, operate more efficiently, and decrease risk [198]. It is especially important in the management and processing of scientific data since the methods used to capture data are constantly being revised and improved which means that the quality of data may vary over time.

DQ can indicate to what extent a concrete instance of data complies with quality rules inherent to an information system. However, DQ is a multifaceted concept and may involve many different dimensions [12, 123, 187]. DQ dimensions capture a specific facet or aspect of DQ such as accuracy, completeness or currency and contribute to a measure of the overall DQ. It is not obvious how DQ can be measured and quantified for individual information systems, but it is clear that data considered to be of poor quality in some sense can cause several problems in an information system [113, 182, 185].

It is a fundamental task to discover the quality of a dataset in most DQ improvement projects [11] where DQ is usually evaluated against certain stated requirements [86, 52, 109]. Users want to access high-quality information where quality requirements are generally user-specific. To determine the usefulness of data and user satisfaction, DQ serves as a key factor [199]. Most DQ assessment approaches are user-centred and organised in a ‘top-down’ manner where (1) requirements are identified, (2) metrics are defined to measure the DQ, and (3) the actual datasets are explored based on the defined metrics [203].

Constraint models are used in databases to specify various basic data integrity rules and ensure that only valid data is entered into a database. Other forms of constraint checking occur in other components of an information system ranging from checks encoded in software that implements the business logic to client-side form validation. While such checks guarantee certain minimal DQ requirements, they do not cater for all

dimensions of DQ such as ensuring that data is current and accurate. Further, many of these dimensions are not absolute. For example, a scientific measurement is not simply accurate or inaccurate, but rather needs to be associated with some measure of accuracy and may become outdated over time as new scientific methods evolve. It is therefore necessary to have models and methods for analysing the quality of stored data as well as preventing the input of invalid data.

For over four decades, DQ has been the target of research and development where many interesting investigations related to its dimensions, models, frameworks, measurement techniques, methodologies, and applications have been carried out, e.g. [144, 107, 189]. In the beginning, the main interest lay with how to manage datasets to find and solve DQ problems. Over time, due to its cross-disciplinary nature, involving statistics, knowledge representation, data mining, management information systems, and data integration [12], it has been approached by various communities. Consequently, the different communities, such as business analysts, solution architects, database experts and statisticians both in academia and companies discuss the applications of DQ management from different perspectives [198, 165].

Meanwhile data volume, diversity of data, data sources and access to data have all fundamentally changed [165]. In the beginning of digital information management rather small datasets of similar types were gathered from a few sources and stored on single machines. This data would be managed by a few people inside an organisation and accessed in a uniform manner. Nowadays, the volume of data produced and used by organisations has grown in a way beyond what previously could have been imagined. Also, due to the proliferation of shared/public data on the World Wide Web and the growth of the web community, today's information systems store and manage structured, unstructured and semi-structured data, but also multimedia data such as videos, maps and images. Data from various technologies such as sensor devices used for home automation, medical instrumentation, RFID readers, smart watches and cars further increases the diversity and amount of collected data. Moreover, the number of people and devices that access data has increased greatly and it is now common for employees to access data globally in a distributed manner from different devices including laptops, tablets and mobile phones. Due to these changes, information management has become even more challenging and complex than it has ever been. Solutions to the management of data in general, and specifically to DQ control, are challenged all the more as the importance and complexity of DQ also increases [165].

Now then, how can good quality of data in an information system be ensured? Most obviously, we think first of restricting the input process so that data of poor quality cannot be entered into the system. This can be done by using constraint checking at the User Interface (UI) level, within application programs or at the database level. Alongside data input validation, there exists the possibility of analysing the quality of the data already stored in the database. Users (or power users) can then be informed about any problems and take actions to deal with them. Such an analysis is related to DQ assurance or data cleaning which is concerned with the correction and improvement of data in databases as described in [66].

Thus, the reliability and usefulness of an information system is largely determined by the quality of the data that serves as input and is processed by the system. But to ensure a high level of DQ, it is necessary to precisely define DQ requirements expressed as constraints which are based on an information system's business rules. In the realm

of information systems, such constraint checking is seen as the main mechanism for controlling the validity of data at the various application levels. However, there is a lack of widely accepted standards, approaches, technologies and tools for measuring, managing and enhancing DQ based on constraint checking.

In our work, we acknowledge that DQ has a central role in information systems development by meeting the need for incorporating considerations on DQ into the complete data cycle encompassing not only managerial but also technical aspects such as semantic integrity constraints [164]. We aim to provide approaches to express various aspects of DQ as an information system output by the means of constraint checking on certain data input.

We thereby motivate to exploit the synergies across different research communities dealing in our research with those aspects of DQ which indicate the subjective adherence of a dataset to a particular set of constraints. We predominantly focus on intrinsic DQ and therefore are concerned only with the data itself and not contextual factors such as the purpose for which data is used or characteristics of the data users. Moreover, we consider information systems in their entirety rather than the database components alone.

In this work, we focus on *constraint checking* which is the act of checking data for the fulfilment of requirements expressed as conditions. Not fulfilling these conditions or constraints, respectively, results in constraint violations which can trigger various actions. We also refer to constraint checking as *data validation* or just *validation* with the same meaning. In addition there also exists *constraint maintenance* which includes repairs to data if a constraint check caused a violation. Constraint checking and constraint maintenance together can be generalised as *constraint enforcement*. *Constraint validation* in turn is seen either as the verification of a single constraint or a set of constraints against requirements such as that constraints in the set should not contradict each other.

1.1 Case study/scenario

One of the drivers of our work was the further development of the food science data management system FoodCASE¹ which already existed as a system for managing food composition data in Switzerland where the Federal Food Safety and Veterinary Office (FSVO)² is responsible for its content. Throughout our work, FoodCASE always served as a case study and testbed for the various concepts that we propose and their implementations. We will use it in the following sections to motivate our work in general and our explicit approaches. Further, we provide examples based on FoodCASE as a scenario application with scientific data specific for the food science area. Therefore, we present here a general introduction to food science studies and a summary of the FoodCASE functionality, focussing on Total Diet Study (TDS) data.

Generally, FoodCASE is used to document data for studies in the food sciences, such as foods with their associated analysed values for nutrients and contaminants as well as consumption data. The stored data is used for further assessment concerning, for example, nutrition and health risks for a certain population in a specific country. The

¹<https://www.foodcase.ch/>

²<http://www.blv.admin.ch/>

FoodCASE application consists of a database, server logic, two clients, a web interface and a mobile application.

In food composition studies, single foods such as apples, bread, chicken, and cherries are bought and analysed regarding various beneficial substances such as calcium, protein, and vitamin A to document the containing nutrients. A resulting single value, for example, would be that a hundred grams of the edible part of an apple contain five milligrams of calcium on average. Further, aggregated foods such as fruits can be composed in a calculative manner based on single foods such as apples, bananas, cherries, and pears with their appropriate single values. Moreover, based on single and/or aggregated foods, recipes can be calculated which represent dishes in a daily diet such as a fruit salad.

While food composition studies focus on nutrients, the aim of TDSs is to analyse the content of nutrients and contaminants in a population's diet, and calculate exposure assessments where the focus lies more on harmful rather than beneficial substances. A TDS consists of several process steps, including the definition of population groups and foods of interest, planning, sampling, pooling, laboratory analyses and exposure assessments. In the first step of a TDS, a previously conducted national consumption survey is used to derive a TDS food list that represents the large part of a population's diet in a specific country, for example foods such as apples, bread, and chicken. Consequently, the two datasets of food consumption and a TDS are closely related. The next planning step is to define what foods from the food list should be pooled together to reduce analysis effort (e.g. red cherries and black cherries to cherries) and what explicit products should be bought when and where (shopping list). The results of the planning are food pools (so-called samples) and a list of concrete food items (so-called subsamples). The subsamples (e.g. red cherries and black cherries) then have to be bought in shops which is called sampling. In a TDS kitchen, the subsamples are then combined to samples (pooling) which finally are analysed in a laboratory. The result of the laboratory analyses are analytical values for various substances (e.g. copper, manganese, mercury, selenium, etc.) for the different food pools. Resulting analytical values would, for example, be that one kilogram of the edible part of cherries contains 0.5056 milligram of copper (Cu) and no detectable quantity of mercury (Hg) on average. These values are then combined with the associated consumption data to assess the exposure risk.

TDS data is similar to food composition data in terms of value documentation of analysed values. However, it is different in that foods collected for a TDS are pooled before they are analysed, and more sampling information needs to be stored. The main difference in terms of data management is that, for food composition, the focus lies on the storage of analytical values and generation of new derived values whereas, in a TDS, the focus lies on value documentation for exposure assessment.

Nowadays, food and nutrition scientists need to manage an increasing amount of data regarding food composition, food consumption and TDSs. The corresponding datasets can contain information about several thousand different foods, in different versions from different studies. FoodCASE has been developed to manage these different datasets and support flexible means of linking between them as well as to generally provide support for the different processes involved in the acquisition, management and processing of food science data.

In 2014/15, Switzerland conducted the first national Swiss food consumption survey,

where FoodCASE was chosen as the storage system for the interview data. The main goal was to keep the consumption and food composition data in the same system and to enable automated linkage between the two datasets. The European project Total Diet Study Exposure (TDS-Exposure)³, conducted between 2012 and 2016, within the 7th Framework Programme for Research of the European Commission, aimed to harmonise the analysis of dietary contaminants throughout Europe, and create an EU-wide network of TDS centres. As part of the TDS-Exposure project, we first gathered appropriate requirements from the food science community and then extended FoodCASE, based on these requirements, in continuous collaboration with TDS experts from across Europe to additionally support the management of TDS data. The extended FoodCASE supports the single process steps of a TDS and provides the functionality to document all intermediate and final data.

Consequently, all data concerning food composition, food consumption and TDSs as well as additional administrative data such as user management data (e.g. user names, rights, preferences, etc.) and general food science thesauri (e.g. units, substances, analysis methods, etc.) can be managed and accessed in FoodCASE through the clients and the web interface, respectively. Additionally, a mobile application was developed to support the TDS sampling step so that shopping lists can be downloaded onto mobile devices and used in supermarkets to buy concrete food products.

In the remainder of this subsection, we present a feasibility study which was conducted as part of the TDS-Exposure project and the general architecture of FoodCASE. Further information about the FoodCASE application is provided in [148, 146].

In the TDS-Exposure project, the five countries Germany (DE), Czech Republic (CZ), Finland (FI), Iceland (IS) and Portugal (PT) conducted TDS pilot studies. Since the TDS module in FoodCASE was developed in parallel to the pilot studies, the study participants collected and managed their data first in different systems and tools such as laboratory information systems, Excel and Access. After the FoodCASE TDS extension was finalised, the datasets from these studies, as well as previously existing TDS data from UK and France (FR), was imported into the final FoodCASE application.

FoodCASE was developed using a client-server architecture to provide accessibility to the various stakeholders. The FoodCASE server is not only a database, but also contains an application server over which data is submitted and retrieved. The application server builds a single point of access to the data, so that DQ checks can be performed, business logic (such as recipe calculations) can be done in a single place and the data is protected. The application server also offers the advantage of using interfaces in different applications, including desktop, web and mobile applications, which can be served over the same business logic implementation.

1.2 Motivation

Data validation ensures that the quality of data in an information system meets a required threshold. It can detect not only errors but also anomalies or deficiencies in data. In relational databases, validation is mainly concerned with database integrity and consistency, and handled by constraint checking mechanisms. Object databases, on the other hand, provide very limited support for constraints, leaving the task of data

³<http://www.tds-exposure.eu/>

validation to the application programmer. However, even if mechanisms for checking integrity and consistency are an integral part of a database, they are not sufficient to ensure that data is clean, correct and useful.

1.2.1 Unified constraint management

Independent of the database technology, the data validation process is usually distributed across several layers of an information system and performed in different ways. For example, client-side validation might be performed using JavaScript or HTML5 form support, while Java application logic and database constraints perform other checks. Consequently, the same validation constraints are often checked in multiple layers leading to redundancy, code duplication and possible inconsistencies.

Constraints represent validation rules derived from software requirements which can change over time, especially with the adoption of agile methods. Further, changes in the set of constraints can result from minor issues such as bug fixes to major changes in the system architecture which might even involve replacing components or technologies. Keeping track of all the constraints and maintaining consistency throughout an entire information system in the face of change can be a significant challenge for software developers. The distribution of constraints makes it hard for a developer to have an overview of all constraints and that, in turn, results in increased maintenance effort. For example, while refactoring code, it is helpful if constraints are also moved, or even refactored, automatically together with attributes and methods.

To make things worse, information systems are usually not just linear architectures where only one application accesses a database. It is more likely that several applications access a shared database and that the information system exchanges data with other applications via machine-to-machine interfaces. Consequently, identical constraints might not only be checked multiple times in different layers of the information system but even by different client applications.

Consider, for example, a system such as FoodCASE for managing food science data which consists of a relational database, Java server business-logic, a Java client which connects via Remote Method Invocation (RMI) to the server Enterprise JavaBeans (EJBs) and a mobile application which connects via a *RESTful API* to a web service on the server. Assume the client is used to manage all data including thesauri and user data, while the mobile application is used to manage a subset of data related to buying food samples in grocery stores. Data is usually entered through the client and the mobile application but can also be imported directly to the database. Thus, it is necessary to validate input data at least in the client, mobile application and database but potentially also in the server business logic. For example, if it is only required to store food names in English, then there could be a constraint that the name should consist of at least three characters since there exist no foods with shorter names. This constraint could be defined in the database using Structured Query Language (SQL) as

```
CONSTRAINT CHECK (length(name::text) >= 3)
```

in the relevant table. The same constraint could be expressed in the business-logic with *Bean Validation (BV)* as an annotation on the foods name attribute:

```
@Length(min = 3)
```

Further, the client could make use of the BV, while the mobile application constraint

checking could be based on a rule engine where the rule ‘when’ statement would be defined as

```
when Food (name.length < 3)
```

Thus, initially three technology-specific constraints would have to be created from the same requirement and integrated in the appropriate components of the information system.

If, at a later point in time, the software should also be used in Germany, the requirement is changed to support German food names as well. Consequently, the constraint would have to be adapted to allow names with two characters since in German there exist food names with only two characters such as ‘Ei’ for the English ‘egg’. Each of the three technology-specific constraints (SQL, BV and *Business Rules*) would have to be adapted accordingly.

The above example illustrates that an approach where constraints could be expressed in a single format and mapped automatically to different representations in different components could be highly beneficial for maintaining consistency under change. Further, since the set of constraints can differ from one component to another, as in the example mobile application, it is necessary to keep track of the set of relevant constraints for each component.

The chosen food name length constraint in the example is a very simple constraint. Of course constraints can be much more complex and involve the validation of multiple different instances which potentially even increases the issue of distributed constraint checking. Assume, for example, a constraint which has to assert that for a pooled food (e.g. cherries) always at least twelve specific food items (e.g. red and black cherries from different stores) have to be bought. Consequently, two entities (sample and subsample) are involved where the cardinality constraint on the association between the two entities has to be checked only after all food items have been bought.

Since Model-Driven Software Development (MDSO) and with it *Model-Driven Integrity Engineering* [98] are gaining in importance, it would be convenient to be able to create the various technology-specific constraints directly from the software model.

1.2.2 Data quality expressed by constraints

Having indicators for DQ at hand for the different levels of data granularity, such as single attributes, single instances, sets of instances, sets of entities and the overall quality of a dataset, is desirable for the analysis and improvement of the DQ in an information system. The use of constraint checking methods for simplifying the evaluation of such indicators for DQ appears to be applicable since (integrity) constraints and quality can both be described by assertions. Nonetheless, it is not clear how a set of constraints can be used to measure and indicate the quality of data.

Most information systems do not allow specific measurements for DQ requirements to be defined, but rather rely only on a set of constraints which validate to either true or false. Therefore, either a user enters valid data and is able to store it in the database or the data is rejected. In many cases, this is not sufficient, as it might be appropriate to distinguish several levels of DQ or even use a continuous range from very poor DQ to very good DQ. This requires another concept for validation which supports appropriate metrics for DQ.

Consider, for example, the creation of a password for a new user in an information

system where a set of rules differ in their importance to be fulfilled. There could be two strict rules which define that passwords (1) must be at least eight characters long and (2) contain letters and numbers. These constraints have to be fulfilled to register a new user with a password although the password security might be low. If additional constraints are fulfilled, such as the password (3) should contain upper case and lower case letters, (4) should contain special characters, and (5) should be longer than twelve characters, the password security and therefore the DQ would increase. By rating the additional three constraints with a lower impact on password security or DQ, respectively, than the first two constraints, the different importance of each constraint can be represented.

Even if the opportunity for expressing DQ as constraint compliance seems to be self-evident, it is not clear how a set of constraints can be used to calculate the quality of a dataset. Measures have to be explored which serve as indicators for DQ for the different levels of data granularity.

Linking constraints to DQ is difficult since constraints usually are defined upfront and, therefore, can never be exhaustive. This means that a dataset that satisfies every constraint might still be of bad quality. Since constraint compliance only serves as an objective DQ assessment, good quality of a dataset cannot completely be assured by monitoring the constraint compliance of an information system.

Assume, for example, the constraints that a food must have a name and an English name which both should consist of at least a certain number of characters. The name is used for the national name (e.g. German: ‘Milchreis, Griessbrei/-pudding’) while the English name (e.g. ‘Pastas and rice (or other cereal) based dishes_milk rice/semolina pudding’) is used internationally for better comparison reasons in different countries. In data validation both names might fulfil the assertion of consisting of a minimum of characters but it is very hard or even impossible to verify that the translation to the English name is correct since there does not exist an exhaustive list of food names. Only a user of the system might recognise the semantic error in the English name and propose the correction to a better translation (e.g. ‘Rice/semolina pudding’).

Hence, there emerges the need for a subjective DQ assessment where the subjective DQ indicates the perception on the quality of data of different stakeholders such as the individuals that enter, provide and use the data. The perceptions may vary between the individuals but can provide comparisons to the objective DQ provided by constraint compliance.

1.2.3 Extended constraint checking

Usually, immutable sets of constraints are checked in information systems imposing the same rules to all users. This equal treatment does not necessarily represent the ‘real-world’ data requirements which should be enforced by the system. It is absolutely reasonable to apply different constraints to different groups of users where it is even possible that major differences can exist between the constraint definitions. This includes cases where different constraints are applied for different users but also the same constraints with different importance and contribution concerning DQ as mentioned above. In the TDS-Exposure project, we have experienced this with the various participants where the different institutions had diverse data validation requirements. For example, when analysing food samples in TDS laboratories, different analysis methods

such as ICP-OES⁴, ICP-MS⁵, and AAS⁶ are used to obtain the analytical values. In FoodCASE the analysis method is stored on an entity name ‘LabAnalysis’ in the attribute name ‘methodname’. For a ‘Not null’ constraint on the attribute ‘methodname’, the different institutions required five different importances ranging from the highest importance which should indicate a severe error and inhibit the data from being stored to a very low importance which should only cause an informative notice.

Thus, quality might entail different aspects of the system for different users or applications. Even for a single user, different constraints most likely vary in importance. As the end-users of a system are usually not its developers, it is crucial to enable authorised users to tune these quality aspects at any time via an intuitive and easy-to-use interface.

Further, even though applications may have similar data validation requirements, it is often difficult to reuse validation code. Slight differences between the constraint definitions result in separated constraints and potential code duplication. Most information systems do not support inheritance of constraints where similar constraints can extend each other in order to reduce redundancy.

1.2.4 Feedback on data quality

If appropriate measures and indicators for DQ have been explored it has to be examined how and when users should be given feedback about DQ based on such metrics. The issue of how to give feedback on DQ to users of an application has to be addressed by taking different types of feedback into account such as direct feedback on individual entities given during data input and feedback resulting from an analysis of the database. Both the analysis of DQ at a certain point in time and over a period of time for monitoring DQ should be supported. Therefore, feedback on DQ should be explored and viewed in a variety of textual and visual forms.

1.3 Challenges

Based on the motivations of the previous section, challenges arise due to the need for (1) unified constraint management, (2) a fine-grained severity scale for constraint violations to distinguish between less and more important constraints, (3) measures to indicate DQ for various levels of data granularity, (4) subjective DQ assessment through explicit user feedback as an extension to constraint compliance, (5) user group and application dependent constraint checking, and (6) DQ input feedback and DQ analysis feedback at a certain point in time and over a period of time.

1.3.1 Unified constraint management

One of the main challenges of this work is to explore approaches for unified constraint management which especially also handle intentionally distributed constraints since centralised constraint definition and checking is often not feasible nor desired. Thus,

⁴Inductively coupled plasma-optical emission spectrometry

⁵Inductively coupled plasma-mass spectrometry

⁶Atomic absorption spectroscopy

existing approaches such as BV, rule engines and Aspect-Oriented Programming (AOP) have to be analysed and a concept created which enables constraint definition and constraint checking mechanisms to be applied to different application layers, to different components of a distributed information system and, most probably, to various representations.

Mechanisms have to be taken into account which use the same representation for all components but also which support bi-directional translations between a platform-independent constraint model and multiple technology-specific representations. Allowing multi-translations between technology-specific representations would be especially beneficial if legacy systems want to move from one constraint representation to another or if a software developer prefers not to use a specific representation.

Adopting a constraint-based approach to DQ also requires a powerful constraint definition language such as the Object Constraint Language (OCL)⁷ to express the possibly complex rules. Existing approaches do not cover all needs of this work where a constraint definition language should also enable constraints to be defined in a detached state from a *UML* class diagram and provide means of defining common platform-dependent constraint definitions in a convenient platform-independent manner.

1.3.2 Data quality expressed by constraints

The assessment and quantification of DQ is a major challenge for organisations and for various stakeholders of an information system such as the end-users of a system, the business engineers who define the requirements and the software engineers who develop the applications. Although several conceptual frameworks for DQ have been proposed, there is still a lack of general tools and metrics to measure and control the quality of data in practice. A Data Quality Management Framework (DQMF) which has the goal of facilitating constraint management for software developers and embedding data integrity in a dynamic, continuous DQ improvement process in software engineering, faces the challenge of defining measures for indicating DQ and presenting an appropriate scoring of the DQ for data input as well as for data analysis.

First, it has to be defined what kind of score should represent the DQ on the various levels of data granularity. Second, it has to be defined how and to what extent the violation of a constraint contributes to a measure of DQ. This includes not only a general indication but also an indication on specific DQ dimensions. Therefore, a more flexible concept of constraints is needed to support a notion of less strict constraints where data can be associated with a particular indication on DQ as well as the conventional constraints used to prohibit updates that would violate a constraint. Third, a calculation concept has to be provided which aggregates the DQ scoring for the various levels.

Further, additional concepts such as validation dependent on the data source reputation and the specific data item have to be taken into account in order to represent the multiple aspects and facets of DQ. Consider, for example, the laboratory analysis of a food sample (e.g. cherries) which is analysed concerning different substances (e.g. copper and mercury). The analyses for the different substances are done in different laboratories (e.g. an independent lab and an affiliated lab) with diverging reputations

⁷<http://www.omg.org/spec/OCL/>

concerning the accuracy of their analysis. On the one hand, the analytical values from the laboratory with the better reputation could be rated with a better DQ. On the other hand, constraints for a minimum and maximum value, which define in what range an analytical value is expected, can vary dependent on the substance.

Validity and correctness are mostly evaluated by comparing data to real world values. Therefore, assessing the quality of a data item based on constraints, as proposed in this work, results in a measurement of constraint compliance which is simply seen as an indicator for DQ rather than an absolute measure. An emerging challenge is therefore to provide further concepts to extend the indicators for DQ beyond constraint compliance as an objective DQ assessment. Thus, there is the need to investigate subjective DQ assessment and how to combine the resulting information with other metrics such as the constraint compliance.

These indicators represent the objective assessment and subjective perception of DQ in numbers which are used to give feedback on the quality of data to the various stakeholders. Overall, it is challenging to combine the various needs for defining such indicators for DQ based on constraint compliance.

1.3.3 Extended constraint checking

To provide concepts and technical approaches/solutions to enable customised and easily adaptable constraint checking for different users/applications is all the more challenging with the goal of unified constraint management. Further, adding, removing and updating constraint definitions dynamically during the runtime of an application, which is desirable for maximum flexibility in defining and checking constraints, and reusing validation code for multiple constraints or constraint hierarchies, respectively, proved to be complex dependent on the technologies in use.

1.3.4 Feedback on data quality

In addition to performing constraint checking during transaction execution as supported in traditional databases, it is also necessary for an information system to provide data input validation and ways of triggering the analysis of data already stored. If, for example, a constraint changes during its lifetime, it should not only be immediately applied to new input data, but potentially also to data already stored.

For existing data that was analysed concerning DQ, the change of a constraint can render the former analysis obsolete and causes several challenges. First, such changes have to be tracked (e.g. with constraint set versions). Second, it has to be defined how a new analysis is executed, such as just notifying consumers of the out-dated analysis, automatically revalidating stored data partially or revalidating all at once. Further, serious errors have to be identified and it has to be defined what happens with data which decreased in DQ, such as displaying data with low DQ, automatically correcting data by applying certain rules, rendering data invisible or even deleting data.

Toolkits and DQMFs which extend constraint models to support DQ management are advantageously designed general enough to be used to analyse and provide feedback on DQ for any information system. Given that DQ requirements are highly dependent on the context in which the data is used in terms of the domain, the application and the

user, it is important that the measurements and computation of DQ is highly flexible and easily configurable in order to provide customised DQ feedback.

Feedback on DQ in an information system will eventually be the basis for a decision on what can be done with the data. This also depends on the recipient of the feedback who might, for example, be an information system administrator monitoring the DQ, a domain expert creating new data, or a user who reads data for further processing. Indicators on DQ as described in Sect. 1.3.2 should provide the measures and a notion of scoring DQ which then builds the basis to provide appropriate feedback on DQ in an information system. The challenge is then to provide guidance, approaches and tools for providing feedback based on the DQ information including input validation, snapshot analysis and long-term monitoring for the different end-users.

1.3.5 Summary and research questions

The aggravating challenge is to integrate all concepts for the single challenges into toolkits and a complete DQMF in order to facilitate one-stop constraint management for software developers and embed data integrity into a dynamic, continuous DQ improvement process.

Most existing work has focused on either theoretical studies alone or more practical ones covering only one requirement, such as constraint representation or constraint applications/usage.

To the best of our knowledge, there does not exist a DQMF that addresses all of the above challenges. Namely the main challenges of (1) providing a unified way of defining and checking DQ requirements by means of constraint compliance, at the same time (2) not being restricted to a specific data model combined with (3) DQ input and analysis feedback which is highly configurable with DQ assessment profiles.

In summary, DQMFs and toolkits have to be developed to tackle the challenge of embedding data integrity into a dynamic, continuous DQ improvement process for information systems and to facilitate constraint management for software developers. At the core of this thesis are the following three research questions:

- How should DQ be managed in information systems?
- To what extent can DQ be considered as a constraint problem?
- How can we integrate different types of quality controls in a single constraint model and handle them uniformly?

1.4 Contribution

The contributions of our work are mainly providing concepts and approaches concerning (1) unified constraint management for software developers, (2) expressing DQ by constraints and beyond constraint compliance, (3) extended constraint checking also for different users/applications, and (4) giving feedback on DQ during data input and data analysis for different end-users of an information system. These high-level contributions include further lower-level contributions which we will present in detail in this section.

For investigating our concepts and approaches we always developed scenario applications where we mainly focussed on object-oriented (OO) paradigms concerning the

programming languages and technologies. Alongside the fact that OO paradigms are widely used, several other factors led to this decision, such as that (1) OO paradigms are concerned with objects and data in the first place rather than with actions and logical procedures that process input data to output data, (2) OO software development offers a unifying OO modeling paradigm, and (3) OO models can be used throughout requirements analysis, design and implementation. Besides Domain-Specific Languages (DSLs), we were mainly using Java⁸ and JavaScript⁹ which are currently the most popular programming languages according to various indicators, such as the ‘TIOBE Programming Community Index’¹⁰, the ‘PYPL PopularitY of Programming Language Indicator’¹¹, and the ‘RedMonk Programming Language Rankings’¹². We also considered that information systems nowadays are heterogeneous constructs with a potential variety of technology representations.

Concerning data management, we did not focus on a specific database paradigm and therefore were using Relational Databases (RDBs), Object Databases (ODBs) and NoSQL databases in our approaches.

1.4.1 Unified constraint management

To address the issue of distributed constraint management, we investigated different alternatives for unified constraint management regarding the definition and checking of constraints. In the beginning, our vision was that constraints are defined only at one integrative location in an information system which is why we first explored such approaches. If an environment allows for such an approach, it is simpler and convenient to use but we then recognised that centralised constraint definition and checking in a single place is often not feasible nor desired, for example due to technology limitations, security or legacy reasons. Therefore, we also investigated approaches which handle intentionally distributed constraints. Nonetheless, all of the approaches had the goal of obtaining a single, extended constraint model that avoids any inconsistencies and redundancy within the constraint specification and checking processes.

Our investigations encompass both integrated approaches within specific applications and application independent approaches which can be used for any kind of application.

In this work, we propose different extended constraint models along with DQMFs to specify and manage constraints in a unified manner and identify the advantages and disadvantages of each approach. All approaches follow the goal that the constraint models are not coupled to a specific layer or technology and that constraints need to be specified only once, but can be used in different parts of an application for data validation. At the end, we achieved DQMFs which combine and extend the experience and advantages of the former investigated approaches.

While constraints are regarded as a standard approach for ensuring basic DQ in terms of data integrity and consistency, current object databases still provide very limited support for constraint definition. Thus, we explicitly also provide an approach

⁸<http://www.oracle.com/technetwork/java/index.html>

⁹<https://www.ecma-international.org/publications/standards/Ecma-262.htm>

¹⁰<http://www.tiobe.com/tiobe-index/>

¹¹<http://pypl.github.io/PYPL.html>

¹²<http://redmonk.com/sogrady/2016/07/20/language-rankings-6-16/>

that extends and complements object databases with support for constraint-based DQ management but generally any of the approaches can be used for this purpose.

In particular, we included approaches based on an Event Condition Action (ECA) mechanism in an object database, pure BV, BV extended with OCL, a new unified constraint representation based on OCL in combination with translations to and from technology-specific representations, a constraint synchronisation between two applications, and AOP.

Since one goal of our work is to bridge the gap between formal methods in software engineering and the practice of software development, one of our approaches was also to create a system that follows the Model-Driven Architecture (MDA) [100, 118] approach of using a technology-independent model that serves as a base for technology-specific code generation. We propose a powerful DSL, called *UnifiedOCL*, that is capable of denoting a data structure together with constraints of a great variety of types while also providing human readability for non-programmers. It combines structural domain information from the Unified Modeling Language (UML), constraint definitions from OCL and a set of so-called *labels* that are grouped into dictionaries and plugged into our *UnifiedOCL* grammar. Therefore, it is possible to use OCL syntax within *UnifiedOCL*. But we extended our DSL with these *labels* to overcome the limitations of OCL with regard to technology-specific representations and also to provide a convenient way of labelling attributes with appropriate constraints without the necessity of defining the potentially complex (OCL) validation each time. Consequently, *labels* either represent constraints not available in OCL (such as primary key in SQL) or simply provide a more convenient representation (such as email and credit card constraints). With this approach, it is possible to model an information system using a UML editor, enhancing the model with constraints defined in the standard OCL notation or with our label notation. This UML model can then be converted to a *UnifiedOCL* representation. Nonetheless, it is also possible to create and modify the *UnifiedOCL* representation in a text editor independent of a UML model.

Furthermore, we studied various approaches and technologies of bi-directional translations between various models and representations. This resulted in an extensible toolkit capable of efficiently translating constraints to and from *UnifiedOCL*. As a consequence, the system allows for translation from any source representation to any target representation, which we call multi-translations. Our proof-of-concept implementation supports three technology-specific representations: object-oriented language (*Java*), relational database (SQL) and business rules (*Drools*), which let us explore and cover a broad range of diverse constraints. Moreover, it can easily be extended to cater for other representations. By adhering to Model-Driven Development (MDD) principles, this approach separates the conceptual requirements from their actual representations. Consequently, the number of supported environments in terms of platforms, programming languages, frameworks and libraries is virtually infinite.

1.4.2 Data quality expressed by constraints

We provide a concept of indicating DQ based on constraint compliance where we have also incorporated the different constraint violation severity perceptions as mentioned in Sect. 1.2.2. Therefore, we have introduced a fine-grained severity scale for constraint violations which enables less and more important constraints to be distinguished instead

of traditional constraints which usually validate to either true or false. Based on these violation weights, we introduced metrics and measurements, so-called Data Quality Indicators (DQIs), to provide indications on DQ, as mentioned in Sect. 1.3.2, for the various levels of data granularity.

By introducing the concept of DQIs, where the focus is on indicating DQ rather than purely preserving integrity, we realise our vision for expressing DQ as constraint compliance and being able to deal with a wider range of DQ issues than with traditional database constraints alone. DQIs take both the importance of constraints and the severity of constraint violations into account in order to calculate an appropriate DQ score which is interpreted as a measure for constraint compliance.

A DQI provides a rating for one or multiple constraints applied to one or more instances of one or more entities where constraints are weighted with a certain importance measure. If multiple constraints and/or multiple instances are considered, we refer to these as aggregated DQIs.

In most of our approaches, an individual constraint can be weighted with violation severity values associated, if needed, also with DQ dimensions. The example below depicts a notation for the food name length constraint presented in Sect. 1.2.1.

`Food.name.Length [Consistency 0.9, Completeness 0.6]`

In this example the constraint is uniquely identified by the entity name ‘Food’, the attribute name ‘name’ and the constraint name ‘Length’. Then severity decimal values in the range between 0.0 and 1.0 (1.0 being the highest severity) are associated to certain DQ dimensions (such as ‘Consistency’ or ‘Completeness’).

Our calculations for aggregated DQIs then result in DQ scores which allow users to inspect DQ, also related to different DQ dimensions, for single and multiple object instances, single and multiple object classes and the overall DQ.

With an ‘error threshold’ (e.g. 0.8) we distinguish between classical constraints which prohibit saving data if they are violated (equal to or above threshold) and a form of more relaxed constraints which allow data to be saved even if they are violated (below threshold). This allows us to take different actions based on the type of violations encountered. Examples of these actions are consequently whether the user should be able to save their input when a violation occurs but also include the type of feedback that the user receives during data input, which will be described later. This distinction is related to the concept of hardgoals (or simply goals) and softgoals in the *Requirements Engineering* literature. In *Requirements Engineering*, operational goals can be related to hardgoals that set the basis for functional software requirements, while strategic goals can be related to softgoals, which set the basis for non-functional requirements. The satisfaction of hardgoals can be verified clearly, while softgoals are goals that do not have clear-cut criteria for their satisfaction [127, 128].

To extend the DQIs for DQ beyond constraint compliance, we investigated methods for gathering and including both explicit and implicit DQ feedback from users of an information system where explicit DQ feedback, as a subjective DQ assessment, represents the users perceived quality of the data. We further investigated how to use this information in combination with other metrics such as the constraint compliance, as an objective DQ assessment, and an individual relative weighting of the importance of entities in an information system. Consequently, we provide a concept of a DQ vector space model, measures for ‘DQ improvement tendencies’ and a ‘DQ improvement ur-

gency score' with a set of tools to analyse and monitor information systems concerning DQ based on these new metrics.

The DQ vector space model allows the rating of an instance to be represented as a combination of objective and subjective DQ assessment on the three dimensions 'constraint compliance', 'explicit feedback' and 'importance'. The interpretation of the rating in the DQ vector space model is then used to derive information about how to improve the instances in order to increase the DQ. Therefore, the measures for DQ improvement tendencies provide hints if the data, the constraints or both should be improved or if there exist requirements which cannot be defined by constraints. The DQ improvement urgency score, also based on the DQ vector space model, is additionally used to evaluate how urgent the improvement of an instance is.

1.4.3 Extended constraint checking

A further vision of our work was to enable customised and easily adaptable constraint checking for different users/applications. Therefore, we have introduced a concept of DQ assessment profiles, so-called 'contracts' or 'constraint repositories', which allow authorised users to decide on the importance of each constraint for each application as well as deactivate constraints that are not deemed relevant. As part of the TDS-Exposure project, we gathered requirements and constraints from our expert project partners for the implementation of FoodCASE. In addition, for each constraint we obtained a rating for severity if it is violated. The outcome was that the various partners from different countries, who are the users of our system, had very diverse and sometimes contradictory views on the constraint definitions, which confirms the need for our concept of contracts. For example, while some constraints were necessary for some partners, they were not for others. Moreover, even if partners agreed on the necessity of a constraint, they frequently disagreed on the severity of its violation. The example from Sect. 1.2.3 showed that some partners rated a 'Not null' constraint violation on the attribute 'methodname' on the entity 'LabAnalysis' as a severe error, while others rated the same constraint violation as of very low severity.

A contract is a collection of single DQIs, i.e. a collection of mappings between constraints, violation severities and dimensions as presented in the food name length constraint example in Sect. 1.4.2. Several contracts can be specified that are used to define customised analysis and validation profiles for different users and applications of an information system. For input validation and data analysis, different contracts can be applied but always only one contract is configured to be used for the calculation of the DQ scores at a single point in time for each of them. Contracts can also be easily shared between users. Thus, it is possible for a user to have multiple contracts, while multiple users can share the same contract. Moreover, a default contract can be specified that serves as a fallback in cases where there is no customised contract defined. If a contract does not include all constraints that are defined in the system, a default importance (e.g. 0.8) is applied.

For example, the varying importance requirements from different institutions for the `LabAnalysis.methodname.NotNull` constraint on laboratory analysis methods can be reflected with different severity decimal values (0.0 to 1.0) in different contracts. Thus, in one contract a constraint can be configured as very important (e.g. 1.0) while in another as less important (e.g. 0.5). Most probably, there exist constraints in an

information system which in any case must be configured with the highest importance, such as ‘Not null’ constraints for absolutely relevant attributes for the system to work properly (e.g. IDs). Therefore, it is also possible in our concept to prevent the overriding of default importance configurations for individual constraints.

Besides the concept of contracts, some of our frameworks mentioned in Sect. 1.4.1 also make other extended constraint checking possible, such as (1) reusing data validation in other applications, (2) providing a mechanism for constraint inheritance which enables validation code to be reused for multiple constraints or constraint hierarchies, and/or (3) defining constraints during the runtime of an application.

1.4.4 Feedback on data quality

One large part of our work was to address the issue of how to give feedback on DQ to users of an application. We therefore introduced two types of feedback to distinguish between direct feedback on individual entities given during data input, so-called Data Quality Input Feedback (DQIF), and feedback resulting from an analysis of the database, so-called Data Quality Analysis Feedback (DQAF). Our vision was to provide guidance, approaches and tools for DQIF and DQAF, where the latter includes feedback at a certain point in time (snapshot analysis) and over a period of time (long-term monitoring). The feedback should be applicable for different applications and different stakeholders such as the individuals that enter, provide and use the data but also individuals which explicitly analyse and monitor DQ.

We therefore studied, developed and evaluated different approaches for DQIF and DQ analysis toolkits for measuring and visualising DQ where one approach was also integrated into our FoodCASE system. Our approaches and toolkits serve as guidance on how to give feedback on DQ and offer a variety of charts and tables that can be used to provide users with feedback and help them identify areas where action might be required. They also provide tools to drill down on the DQ issues and identify individual problems to support the enhancement of data in order to increase the DQ of an information system.

Our approaches also include adaptive data input and DQ feedback responsiveness in the case of varying constraint sets or feedback in different applications, respectively. We investigated integrated DQMFs within specific applications but also independent DQMFs for DQIF and DQAF and monitoring of DQ including the evolution of DQ over time. Moreover, our approaches for DQIF and DQAF integrate the contributions presented above, such as unified constraint management, new indicators for DQ based on constraint compliance and beyond constraint compliance, easily adaptable feedback for different users/applications (contracts), changing constraints during the runtime and reusing validation code (constraint inheritance).

1.4.5 Food science

With the concept and implementation of our FoodCASE application for the TDS-Exposure project we also contributed to the food science community which is now able to use FoodCASE for the management of their data.

In FoodCASE, we integrated enhanced functionality for defining, validating, measuring and visualising DQ. Therefore, based on constraint definitions, FoodCASE provides

the measurements, calculations and visualisation components for giving feedback on DQ and helps users monitor constraints and DQ. The concept and implementation includes the validation and feedback for both DQIF and DQAF.

1.4.6 Summary

Our work focuses on the creation of constraint-based DQ modelling, calculation and visualisation concepts that can be applied to any type of information system and enable all of the opportunities and challenges mentioned above to be addressed. We have incorporated these concepts as the basis for creating highly flexible DQMFs for software engineers that allow the users of an information system to configure and analyse the details of every quality-related aspect according to their needs.

To evaluate our approaches, we implemented our toolkits and DQMFs either in independent scenario applications or in our existing FoodCASE food science data management system. These implementations were assessed by many participants both from academia and software industry and further evaluated by carrying out various surveys.

Overall, our work provides different approaches and multiple tools which help improve DQ as a dynamic, continuous process based on constraint compliance. We provide approaches which contribute to facilitating constraint management for software developers but also which contribute to DQ management and feedback for the end-users of an information system.

1.5 Thesis overview

The thesis is structured as follows. The background for this research is provided in Chap. 2 with a review of related work. We give an overview of the various concepts included in this work and discuss existing issues in detail.

Chap. 3 presents five initial approaches, with which we have explored alternative solutions to investigate the different requirements and goals presented in this chapter.

The initial approaches build the knowledge and decision basis for our final approaches, which consolidate and extend concepts from the initial approaches to fulfil all of our proposed requirements of a DQMF for various initial software development scenarios.

With the final approaches, presented in Chap. 4, 5, 6, and 7, we propose four alternative solutions with novel concepts for highly flexible and configurable DQMFs that can easily be adapted to the needs of different users and applications.

Finally, in Chap. 8, we outline the main contributions of the thesis, present an overview and brief assessment of our alternative final approaches, and conclude with a discussion on the criteria for deciding on an approach, before giving an outlook for future work.

2

Background

Our work is a continuation of the work presented by Presser [146]. As a contribution to the interdisciplinary field of DQ, Presser proposes a definition of DQ, a Requirement-Oriented Data Quality (RODQ) model and a conceptual RODQ framework for IT professionals and users. The RODQ model supports information system architects and users in specifying DQ requirements as well as measuring and aggregating them to a total value of DQ. The resulting total quality values allow for conclusions on the quality of considered data and for gaining an overview of the DQ. With the RODQ framework, it is possible to propose implementation concepts to derive a comprehensive DQ framework in an information system. The showcase application of Presser's work is the implementation of the first version of FoodCASE for the management of food composition data.

In this work we investigate the extent to which DQ can be considered as a constraint problem by extending and generalising existing constraint models. Our goal was to provide approaches for a DQMF that provides a **unified way** of defining and validating DQ requirements by means of constraint compliance combined with ways of giving feedback on DQ both during input of the data and during later analysis for already existing data stored in the database. Moreover, we investigated in a DQMF which is **highly configurable** with DQ profiles and is **not restricted to a specific data model**.

Therefore, we present in this chapter an overview to DQ and different types of DQMFs. We continue by providing a summary of constraints and general concepts before we delve into the issues of unified constraint management and of bringing DQ in relation to constraints. Further, we give an overview of feedback on DQ and focus on DQ and constraints in mobile applications before we conclude this chapter.

2.1 Data quality

DQ is a research area still gaining in importance. The term DQ, or Information Quality (IQ) which is often used as a synonym, is seen as one of the keys for business perform-

ance [163]. Good quality of data is the prerequisite for all actions taken based on the data. For example, data mining, which is the process of exploring structures and patterns in large datasets, intrinsically depends on data of good quality, since bad data with too many outliers might transform patterns to be irreconisable [12].

According to ISO 9000:2005, quality refers to the ‘degree to which a set of inherent characteristics fulfils a set of requirements’ [87]. In terms of DQ, we thus can state that DQ can indicate to what extent or degree a concrete instance of data complies with quality rules inherent to an information system as a set of requirements. DQ is often characterised as the data’s ‘fitness for use’ which was first coined by Juran [93]. However, DQ is a multifaceted concept for which no precise nor unique definition exists, and may involve many different dimensions [12, 123, 156, 187]. The different DQ dimensions capture a specific facet or aspect of DQ such as accuracy, completeness or currency and contribute to a measure of the overall DQ.

There exist many different definitions and prioritisations for various dimensions such as those given in [12, 2, 117, 53, 94, 39, 152, 156, 187, 182, 64] where conceptual DQ frameworks are identified that differ mostly in the types of dimensions and their categorisation. Many of them are similar in their approach in that their definitions of the dimensions tend to be descriptive and subjective [132], often using adjectives for which the semantics are overlapping or fuzzy. Probably the most acknowledged conceptual DQ framework proposed by the research community is the one from Wang and Strong [187]. They conducted a series of surveys and studies which identified, sorted and ranked DQ aspects (179 dimensions) according to their importance to consumers. The product of their work is a hierarchical framework that organises DQ dimensions into four categories – intrinsic, contextual, representational and accessibility – and fifteen DQ dimensions, as depicted in Fig. 2.1. For example, under intrinsic DQ, they group properties into the dimensions believability, accuracy, objectivity and reputation. Eppler [53] states that the majority of frameworks are built for a particular domain and only a few are general enough to be applied to any domain. For example, research efforts have already been made for defining *software process model constraints* with rules which allow the modelling of software processes using Object Management Group (OMG) standards [159]. Batini and Scannapieco [12] provide a detailed study and literature review of the field.

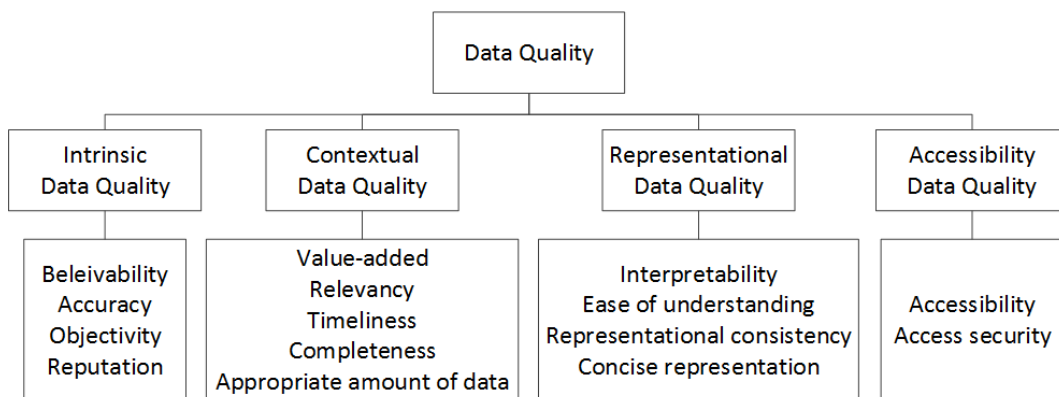


Figure 2.1: A data quality framework with 15 dimensions identified by Wang and Strong in 1996

DQ dimensions can correspond to the extension of data, which means that they

refer to data values, or to the intension of data, i.e. to the schema [12]. In this work, we are only focussing on DQ dimensions that refer to the extension of data which [12] states is most relevant to real-life applications.

A short selection of most widely used DQ dimensions with its definitions is given in Tab. 2.1.

DQ Dimension	Definition
correctness/accuracy	the data correctly represents the true value
completeness	the entity has values for all expected attributes
consistency	the data is free from contradiction
currency	the data is up-to-date
availability	the data is readable and interpretable by users in the right point in time

Table 2.1: Selection of most widely used data quality dimensions

Dimensions can be rated as more or less important, depending on the data requirements and the domain. For example, one application could require the underlying data mainly to be correct while it would not require all data to be complete. Another application might strive for completeness while accepting a reduction in consistency [123]. Moreover, it is possible that the individual DQ dimensions contradict each other, for example, the completeness dimension and the consistency dimension. According to [7], it is not trivial to rate the completeness and consistency dimensions for decision making. An increase in completeness, that is, having more data, usually results in a decrease in consistency. Comparable to that, an increase in the currency dimension, that is, having more current data, can result in a decrease in correctness.

Wand and Wang [182] state that there is no consistent view of the different DQ dimensions and the quality of data is subjective because it could depend on the application and also the design of the information system which generates data. They identified 118 DQ attributes and recognised that, in the end, it is always the data consumer who has to decide whether the data is fit for a particular use. Further, Pipino et al. [144] and Vassiliadis et al. [179] state that DQ must be treated both subjectively and objectively. They agree that DQ is subjective since it depends on both the user and information system implementation, but that there needs to be an objective way of measuring DQ so that a user can compare the outcome with their expectations. We consider this also for appropriate measurements and metrics on DQ in Sect. 2.6.2. In [169], the definition of DQ is connected with the environment of the decision-makers since 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’. In recent work related to the management of food science data [147], we presented a study of DQ requirements to measure and control the quality of empirical data in practice and found that DQ requirements can be understood differently by various groups of information system users or stakeholders.

Two classifications of the types of errors in data can be distinguished, namely syntactic and semantic errors [59]. Syntactic errors are values in the database that violate the domain constraints. For example, a last name containing numbers (e.g. Smi7h) or

the value 300 for an age attribute of a person object, instead of a value in the range of [0,130]. In contrast, semantic errors exist, if the real-world objects are in conflict with their representations in the database. For example, if a ZIP code of an entity, representing customer addresses, is inconsistent with the corresponding city. According to [59], it is usually much easier to resolve syntactic errors than semantic errors.

Syntactic and semantic errors should be avoided or corrected to increase the DQ of an information system. Moreover, the data should satisfy the requirements of the most relevant DQ dimensions.

An overview of DQ dimension definitions is given for example in [90, 102, 107], while overviews of the past decades of research on DQ for traditional information systems are given in [198, 164, 16, 113]. Overviews and evaluations of the most elaborated and cited taxonomies of DQ dimensions are provided in [61, 112, 149, 144].

2.2 Data quality frameworks

There exist various types of DQ frameworks. First, there exist conceptual frameworks, such as the one from Wang and Strong [187], which define and categorise various aspects of DQ, so-called dimensions as explained above. Second, there exist non-abstract industry-adopted frameworks, such as the ‘GS1 Data Quality Framework’¹, the ‘International Monetary Fund Data Quality Assessment Framework’² and the ‘Australian Bureau of Statistics Data Quality Framework’³, which are intended mainly to educate businesses about the meaning and importance of DQ, and offer a best practice guide to controlling and improving DQ.

A third type is the implementation of DQ requirements definition, validation and violations feedback consolidated in a concrete DQMF that can be used by software developers. Approaches have been proposed to overcome the distribution of constraints and consequent redundancies as well as facilitating constraint management, which are summarised in [191] and the subsequent Sect. 2.5.1 but do not relate the constraint violations to DQ. Liu and Kochut [108] proposed a process constraint modelling framework for web services and workflow applications. It allows constraints to be defined using the Process Constraint Language (PCL) and Process Constraint Ontology (ProContO), which can be validated during the design phase as well as at runtime. Sneed et al. [171] discuss the need for tool support for assessing the DQ in relational databases for both schema and instance level integrity constraints. They also provide a case study of how such a tool should work.

2.3 Constraints

In the realm of information systems, business rules express DQ requirements, while constraints are seen as the main mechanism for controlling DQ at the database level. A concise overview of the history of constraints and business rules is presented in [196]. Nonetheless, we provide here a summary of the most relevant constraints.

¹<http://www.gs1.org/>

²<http://www.imf.org/external/>

³<https://www.nss.gov.au/dataquality/aboutqualityframework.jsp>

If we want to describe in a data model, for example, that the salary of a manager always has to be higher than the salary of his or her subordinates, we have to express it in additional restrictions on the values of the data and/or in the structure. Such logical restrictions on data are called *constraints*. In the literature, a constraint is often described as a property for a set or a relation which is either true or false. In a data model, constraints are required for semantic and integrity reasons. On the one hand, they permit schemas to more accurately reflect the real-world situation and, on the other hand, they permit the Database Management System (DBMS) to restrict the possible database states that can be generated from a given schema to those that meet the constraints [175].

2.3.1 Basic constraints

We can identify two basic types of constraints. The *inherent constraints* which are an integral part of the structures of a data model such as the tree structured relationships in a hierarchical database and the *explicit constraints* which provide a flexible mechanism for augmenting the structure specification of databases. A third type, the *implicit constraints* are restrictions derived from either inherent or explicit constraints. For example, in a hierarchical database each segment has at most one parent segment. It follows from this that each segment has only one ancestor segment of any type. The first constraint is inherent. The second can be considered implicit. A data model that is more restrictive in terms of structure incorporates more inherent constraints and therefore fewer explicit constraints have to be defined in order to represent all restrictions. Explicit constraints are defined separately from the database structure with a constraint specification facility. The facility consists of a constraint definition language, a verification mechanism and enforcement algorithms. It ensures that the set of constraints is consistent [175].

There exist two ways to specify explicit constraints. The *static specification* defines which database states are valid whereas the *dynamic specification* defines which state transitions are allowed. Usually, both specification types are necessary to specify all constraints of a data model.

The most ordinary constraints are the *value constraints* which restrict data to a certain range or domain. Much more difficult to enforce are the *aggregate constraints* which reflect constraints on aggregated values such as the total of all salaries cannot exceed a certain amount. Constraints that specify restrictions on the mappings between attributes and/or between entity types are known as *cardinality constraints*. We can specify a minimum cardinality and a maximum cardinality for each mapping of a binary relation. Further, there exist special cardinality constraints such as *existency constraints* or *dependency constraints* and *functional dependencies* with its different mappings depending on the minimum and maximum cardinalities [175].

In entity relationship models, several *classification constraints* (e.g. disjoint, cover) and *constraints between collections and sub collections* (e.g. equal, total) have also been proposed [134].

2.3.2 Integrity constraints

The most commonly used constraint classification is the one of *integrity constraints* frequently also called *consistency constraints*. They define rules which should guarantee the integrity of a database. Integrity constraints are supported by most database systems. In Relational Database Management Systems (RDBMSs) and Object Oriented Database Management Systems (OODBMSs), some integrity constraints can be represented with the Data Definition Language (DDL) at schema definition time (considered as inherent constraints) whereas other integrity constraints (considered as explicit constraints) are expressed and enforced by verification mechanisms such as check conditions and assertions in RDBMSs or specific methods in OODBMSs [13].

In RDBMSs the definition of integrity constraints is often supported while OODBMSs still provide more limited functionality for constraint definition. Both database management system types lack functionality for defining and checking more complex constraints, which do not have a clear-cut criteria for their satisfaction.

The terms *integrity constraints*, *system rules* and *data rules* can be synonymously used and define constraints on a (persistent) data store or data which is included in a transaction. A specification of conditions on database records which must be fulfilled to represent the real world in a correct way, are referred to as *semantic integrity constraints* [176]. According to [161], integrity constraints are solely concerned with the integrity of data, whereas business rules consider the decisions of people. The concept of (semantic) integrity constraints is not restricted to database records, even if they are often mentioned in the context of databases.

2.4 General concepts

In this section, we describe a selection of general concepts which are the basis for defining and/or checking constraints. These concepts are presented without special regard to unified constraint management which is the main topic in the subsequent section.

There exist multiple ways of converting data requirements to data constraints, and how these constraints can be checked in a specific software implementation. Here, we present different approaches, and discuss their advantages and disadvantages. Consider a **Person** entity with two string attributes **name** and **email**, where the organisation owning the data defined two requirements on the data, namely:

1. **Person.name**: Must not be null and must have a minimum length of 2 characters.
2. **Person.email**: Must not be null and must be a valid e-mail address.

We will use this example in the following subsections where it is used to exemplify how these requirements are represented as constraints in specific approaches.

2.4.1 Hard-coded constraint checking

Possibly the most obvious approach to defining and checking constraints is to add if-else conditional blocks in software code whenever objects of a certain entity are manipulated. The manipulation is interrupted and an exception is raised as soon as the data requirements are not fulfilled. This solution is simple to implement, but it suffers from

diverse drawbacks: (1) The code base is expanded with code that is not essential to the business logic; (2) The constraint checking code is often redundant and distributed over the whole code base which is error-prone and time consuming in terms of constraint maintenance; (3) The constraint checking code has to be implemented in multiple programming languages in the case of multi-platform applications, such as a web-application. Nonetheless, an advantage is that constraints can be defined natively in the programming language of the application without any additional tools and this approach is simple to understand.

2.4.2 Structured Query Language

The declarative language SQL is a standard manner to define and manipulate RDBs. According to the DB-Engines ranking⁴, RDBs are still the most popular databases. Thus, it is of high importance to address the issues with constraints in these systems.

Integrity constraints in RDBMSs may be expressed through SQL as:

- *Special keywords*: For example the `NOT NULL` keyword or the `PRIMARY KEY` keyword for a column.
- *Default values*: A certain column default value in a table.
- *Column check constraints*: An arbitrary constraint defined for a column.
- *Table check constraints*: An arbitrary constraint involving multiple columns defined for a table.

The person constraints example from above could be represented in SQL as shown in List. 2.1.

```
create table person (  
  name character varying NOT NULL,  
  email character varying NOT NULL,  
  
  CONSTRAINT chk_name CHECK (length(name::text) >= 2),  
  CONSTRAINT chk_email CHECK (email::text ~  
    '^\\w+(\\.\\w+)*+@\\w+(\\.\\w+)+$'::text)  
)
```

Listing 2.1: SQL example

Constraint checking based on SQL is convenient if many database entries have to be accessed and in the case of simple constraint definitions. For example, if an aggregation over multiple entries has to be calculated or a `UNIQUE` constraint has to be checked which ensures the uniqueness of a value over all data entries. However, the constraint checking cannot be used in the business logic or in a client since it is restricted to the database.

The various approaches provide different possibilities for defining certain constraints. In SQL, for example, there exists no specific representation for an email constraint which is why such a constraint has to be expressed with an appropriate pattern match.

⁴<http://db-engines.com/en/ranking>

2.4.3 Annotation-based constraint checking

Constraint checking methods based on annotations are common for many programming languages which have integrated constraint checking capabilities, or they are at least extensible to provide these features. BV⁵, the Apache Struts Framework⁶, OVal⁷ and the Java Modeling Language (JML)⁸ are examples for such annotation-based constraint checking approaches for the Java programming language.

In these approaches, developers specify constraints by annotating Beans with Java annotations. List. 2.2 shows the BV annotations for the `Person`-Entity from the example above. In BV, there exists a specific notation `@Email` for an email constraint.

```
public class Person {
    @NotNull @Size(min=2)
    private String name;
    @NotNull @Email
    private String email;
}
```

Listing 2.2: BV example

In the subsequent Sect. 2.5.2, we will explicitly focus on BV as an approach for unified constraint management. It follows the Design by Contract (DbC) paradigm supporting pre- and postconditions, and invariants.

The validation mechanism introduced in *HTML5* is another example of constraint checking based on annotations where the validation is available in the *form*-context for *input*-elements.

The appropriate markup for an input form which ensures the requirements from the example above is shown in List. 2.3.

```
<form method="POST">
  <input type="text" required pattern=".{2,}" name="name">
  <input type="email" required name="email">
  <input type="submit" value="submit">
</form>
```

Listing 2.3: HTML5 form validation

As presented in the listing, the `pattern` validator expresses the *minlength* constraint and email addresses can be validated by defining the input type to `email` in combination with the `required` attribute. The supported validators for HTML5 forms can be found in the appropriate HTML5 documentation⁹.

Specifying constraints with annotations has the advantage of being a well-known programming language construct already in widespread use [158]. Moreover, annotations are close to the source code, which can be beneficial compared to approaches that define constraints in documentation, a separate configuration file or in the database. In

⁵<http://beanvalidation.org/>

⁶<https://struts.apache.org/docs/validation.html>

⁷<http://oval.sourceforge.net/>

⁸<http://www.eecs.ucf.edu/~leavens/JML//index.shtml>

⁹<https://www.w3.org/TR/html5/forms.html#forms>

the case of BV, it is very convenient to define annotations in the already existing object model and to easily write custom constraints in the same language as the programming language (Java). Further, the constraint violation messages in BV are very detailed which usually is not the case in SQL.

A disadvantage of constraint checking based on annotations is that the business logic code is expanded with annotations which can reduce the readability. Furthermore, and more important, a flexible constraint checking is usually not supported with these methods which means that a constraint violation results in an exception and consequently the data is rejected. Note that the same issue exists in SQL. A further drawback of the annotation-based approaches is that constraint parameters are static which often causes the implementation of custom constraints. Moreover, constraints cannot be changed during runtime, as the application has to be recompiled.

2.4.4 Event Condition Action paradigm

Active databases have been proposed as a means of realising mechanisms to enforce constraints. The work of Eswaran and Chamberlain on System R [56] is one of the earliest notions of an active database. In active database systems and other event driven architectures, the paradigm of ECA is used. *Events* occur in a system, such as the commit request of a database transaction. A mechanism in the system monitors the events that occur and triggers a registered *action* if a particular *condition* is fulfilled. In the ECA paradigm, it is usually possible to configure the coupling mode between the triggering event and the condition evaluation as well as between the condition evaluation and the execution of the action. Mechanisms such as assertions and triggers can be generalised by ECA rules [31].

In the subsequent Sect. 2.5.1, we will explicitly focus on enhancing object databases with constraint functionality where it is appropriate to base this on the ECA paradigm.

2.4.5 Model-Driven Architecture

MDA [100, 118] is an approach proposed by the OMG¹⁰ for dealing with the problem of constantly changing technologies and business requirements during the software development life cycle. In order for the logic of the system to be completely independent of the underlying platform, a system is first modelled using a Platform-Independent Model (PIM), e.g. UML or Meta-Object Facility (MOF), and then translated into a Platform-Specific Model (PSM). In other words, the logic of the system should be completely independent of the underlying platform. The heart of this approach consists of widely used models that reflect details of either a platform independent system specification or technology-specific representations. MDA is generalised by the concept of Model-Driven Engineering (MDE) [98] which includes also the phases of *process* and *analysis* that result in a software development methodology.

These model-centric methodologies for software development became significant in academia and industry and within them UML and OCL [140, 190] play an important role. OCL is part of UML which is the standard for object-oriented modelling [72]. OCL, also standardised by the OMG, is a formal constraint modelling language that

¹⁰<http://www.omg.org/mda/>

complements UML in the field of constraint specifications. It provides a convenient platform independent way of defining constraints which consequently leads to its limitation that it cannot be directly used to perform assertions during the runtime of an application since assertions should be defined differently depending on the selected language/platform of the execution environment. Further, with OCL, it is not possible to explicitly express all types of constraints which can be defined by technology-specific representations. For example, there does not exist a dedicated keyword to express the *primary key* constraint, one of the most common constraints in the relational model. Moreover, validating patterns such as an email address require specifying regular expressions or a pattern matching for the type *string* which is not supported by the latest specification of the OCL standard library¹¹. In OCL, there exists also no way to enforce a particular precision of numbers (type *real*), e.g. by defining the number of decimal places nor does OCL have a dedicated type to store time or date related values. For instance, in Java, we can enforce an email pattern by using the `@Email` annotation, the precision of numbers using the `@Digits` annotation and time related constraints with the annotations `@Past` or `@Future`. Additionally, OCL is not powerful enough to allow for the validation of numbers that require complex algorithms computing checksums such as the *Luhn algorithm* for a credit card number. One could use a less unambiguous representation of the *primary key* constraint by combining the *not null* and *unique* constraints but, in any case, the representation will be less convenient for developers and potentially also more error-prone.

The person constraints example from above could be represented in OCL as shown in List. 2.4.

```
context Person
  invariant Name:
    self.name <> null
    and self.name.size() > 1
  invariant Email:
    self.email <> null
    self.email.matches('^ [a-z0-9_+-.]@[a-z0-9-\.]+\.[a-z]{2,4}$')
}
```

Listing 2.4: OCL example

There exist other constraint languages such as *ARL* which is an extension of *ALICE* [178]. With *ARL* it is possible to specify active database rules with an object-oriented view of data. Rules can be part of different rule sets which makes it possible that different users can have different sets of rules for different situations. An overview of different rule languages and the features that they support can be found in [178]. In spite of these limitations, OCL is the most widely accepted constraint language also due to its standardisation and multiple extensions.

2.4.6 Aspect-Oriented Programming

AOP is a powerful programming paradigm that enables to enhance the modularity of a program. Functionality which does not concern the main logic of a program, so-called

¹¹<http://www.omg.org/spec/OCL/2.4/PDF/>

cross-cutting concerns such as security, logging, persistence, debugging, tracing, distribution and performance monitoring, can be separated from the rest of the system [89]. This is especially helpful in large applications, for example, to reduce code duplication. AOP allows to add additional behaviour, defined in additional pieces of code (so-called *advices*), to existing code without modifying the code itself. *Pointcuts* specify where advices can be injected into the code, such as before or after the call of a method with a certain name. The combination of a pointcut and an advice is called *aspect*. The cross-cutting concern, in our case the constraint checking logic, is defined in such aspects. *Join points* specify when aspects should be executed in the control flow of a program, such as before or after a method execution. A weaving process injects the aspects into the application code at the specific pointcuts.

AOP implementations in Java are, for example, AspectJ¹² and Spring AOP¹³.

List. 2.5 shows an AspectJ aspect definition.

```
aspect Person {
    pointcut callSetMethod() : execution(* Person.set*(..));
    void around() : callSetFieldMethod() {
        // validation code
    }
}
```

Listing 2.5: AspectJ pointcut example

Before each call of a method with the prefix `set` in its name which is defined on the `Person` entity, the method `callSetFieldMethod` is executed. Considering the `Person-Entity` example from above, the `callSetFieldMethod` is executed before the methods `setName(String name)` and `setEmail(String email)` are called. The relevant constraints can be validated since the validation code is able to access the parameters provided to the method. The execution of the method is continued if the arguments do not violate any constraints. If the arguments do not fulfil a constraint, an appropriate action can be executed, such as correcting the data or throwing an exception.

2.4.7 Design by Contract

Many constraint languages/frameworks, such as OCL and BV, support the declaration of invariants and/or preconditions and postconditions on operations. This is used in the software engineering concept of DbC where encapsulated logic is associated with, and relies on, other logic. It was introduced by Bertrand Meyer and states that the single program modules define formal contracts which assert that they do not accept and do not provide invalid data [120].

‘DbC is a methodology to guide analysis, design, documentation, testing and other tasks of software development. The application of DbC to software design requires the developer to build software systems based on precisely defined contracts between cooperating components. Contracts, similar to real-life contracts, define the relationship

¹²<http://projects.eclipse.org/projects/tools.aspectj>

¹³<http://docs.spring.io/spring/docs/current/spring-framework-reference/html/aop.html>

between two cooperating components – the client and the supplier – by expressing each party’s expectations and guarantees’ [9].

According to the definition above, DbC can be used for constraint checking similar to described in the previous Sect. 2.4.6 on AOP. It is checked whether the input received by the setter method fulfils the contract which is in our case the constraint.

Some less widely used programming languages, such as D¹⁴, Eiffel¹⁵, Clojure¹⁶ and Ada¹⁷, natively support DbC at least partly. Since version 4, the .NET framework¹⁸ contains the class `System.Diagnostics.Contracts` which can be used to implement DbC.

Java is a programming language which does not natively support DbC. BV is a DbC extension for Java that supports the pre- and postcondition concepts of DbC. Thus, before and after a method is called which manipulates a data instance, the contracts, or in our case the constraints, are checked. Since BV version 1.1, there exists the possibility to define pre- and postconditions directly in the method headers through annotations which was previously already available in the Oval framework and in JML for Java. Other implementations, such as GContracts¹⁹ for Groovy use Application Programming Interfaces (APIs) for compiler extensions in order to add language elements.

2.4.8 Business Rules Engines

Business rules and rule engines provide another way of defining and validating constraints.

A Business Rules Engine (BRE) is a software system that executes rules in any form to validate data and enables execution separated from the application code. Given data can be validated based on one or more sets of rules. Examples of such BREs are the Java-based Drools²⁰ and the JavaScript-based Nools²¹. Drools is a solution for a Business Rules Management System (BRMS) which provides a BRE and a rules management application. Nools can be used directly in browsers or with Node.js²² and supports JavaScript development with a simple BRE.

The basis for many BREs, such as the one used in Drools, is the *Rete algorithm* which was designed by Forgy [62]. It is a pattern matching algorithm used to determine which rule in a set of system rules should fire based on its data store. For BREs, it is a de facto standard although newer algorithms, based on Rete, such as the *PHREAK* algorithm for Drools, might be used.

Business rules are used to define or constrain complex aspects of business in a convenient way. They enable to define actions which are executed dependent on the fulfilment of a condition at certain events during the runtime of an application. Business rules can be used to express integrity constraints for data validation or conditions of operations, such as preconditions.

¹⁴<https://dlang.org/spec/contracts.html>

¹⁵<https://www.eiffel.com/values/design-by-contract/>

¹⁶<https://clojure.github.io/core.contracts/>

¹⁷<http://www.ada-auth.org/standards/12rat/html/Rat12-1-3-1.html>

¹⁸[https://msdn.microsoft.com/en-us/library/dd264808\(v=vs.110\).aspx](https://msdn.microsoft.com/en-us/library/dd264808(v=vs.110).aspx)

¹⁹<https://github.com/andreteingress/gcontracts/wiki>

²⁰<http://www.drools.org/>

²¹<https://github.com/C2F0/nools>

²²<https://nodejs.org/en/>

List. 2.6 shows how constraints can be defined with the BRMS Drools based on the example already used in the previous subsections.

```
rule "Person_name"
  when
    $a : Person (! (name != null && name.trim().length() > 1))
  then
    <actions>
end;

rule "Person_email"
  when
    $a : Person (! (email != null && validateEmail(email) == true))
  then
    <actions>
end;
}
```

Listing 2.6: Drools example

In Drools, business rules consist of a *condition* and a *consequence*. The condition is the constraint which reduces the whole domain of objects, reachable by the BRE, into a subset. In case of object validations, this is the subset of those objects that do not fulfil constraints. The objects of the subset can then be matched with the corresponding consequence. The consequence is the action that is executed in the case that the condition is evaluated to true, for example, writing a log entry.

The keyword **when** precedes the condition while the keyword **then** precedes the consequence. The consequence part is denoted in List. 2.6 with `<actions>` where any actions could be defined that execute when objects fulfil the condition of particular rules. In Drools, the body of the condition expresses the constraints which can be defined in Drools or Java language expressions syntax.

2.5 Unified constraint management

In software engineering, often data is stored in an information system and used by multiple applications. Constraints are scattered throughout the applications which makes it hard to maintain them consistently. This is known as Application-Oriented Integrity Maintenance (AOIM) [200, 43].

In almost any software design process, integrity constraints are very important and violating them leads to erroneous system behaviour. If the enforcement of constraints is mostly the responsibility of the application programmer, several problems can occur. Application programs contain code for consistency checks and exception handling which expands the code. Same or similar constraints may be checked multiple times which increases code redundancy. Changing constraints may lead to rewriting several parts of the software possibly also in different applications. Programmers might not be able to provide efficient code for constraint checking because they lack knowledge about the techniques used for implementing the database or knowledge base. Therefore, the centralisation and automation of consistency enforcement would simplify the process of

software engineering [49].

Let us examine in more detail the case of an information system with a web interface and a mobile application on the presentation tier, a logic tier and a database on the data tier. In this scenario, shown in Fig. 2.2, the web interface is implemented in HTML5 and JavaScript, the mobile application in mobile platform languages, and the business logic in Java.

To reach high DQ, data should only be stored in the database if the data requirements are fulfilled. Usually, this is done by ensuring that no constraints are violated by the data input, or that the data input is rejected.

The web interface provides HTML forms which allow the user to enter data into the information system. The *POST*-data is transmitted to the business logic over the HTTP protocol where it is received and further processed. Based on constraint definitions, the user input data has to be validated to check if it fulfils the data requirements. The constraint definitions and the constraint checking logic must be implemented twice in the case of a web application, once on the logic tier, and once on the presentation tier in HTML and JavaScript.

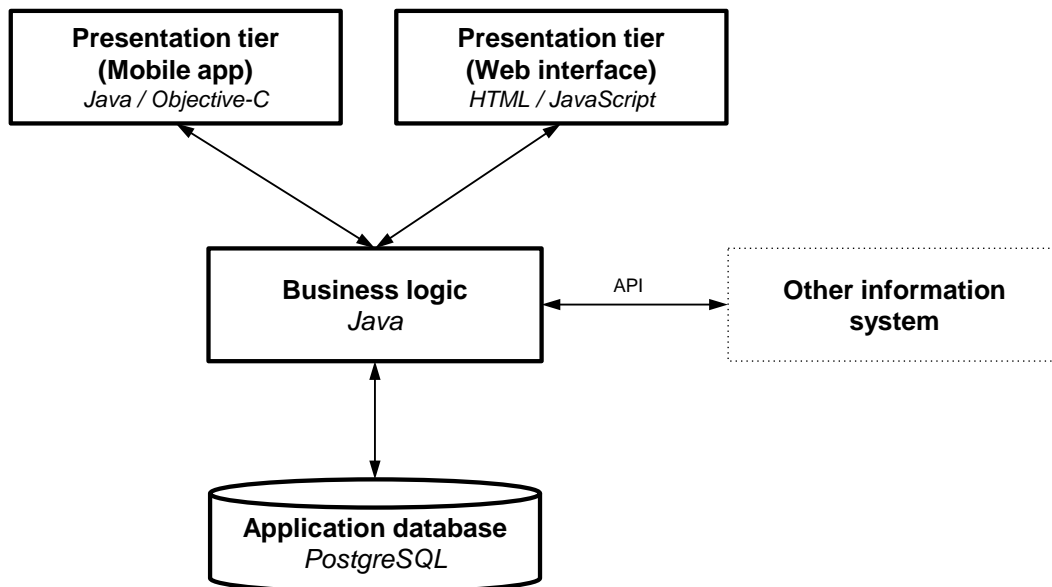


Figure 2.2: Example of a distributed information system

Alternatively, all validation code could be placed in the business logic while no validation would be done in the web interface. However, this is not recommended due to the following reasons: (1) For each validation, the input data is transmitted to the business logic which causes a complete HTTP transmission cycle, even in the case of very simple validations, such as a check for a correct format of a value; (2) After having submitted a form, a user has to wait for the success or error validation result of the business logic. It is desirable for users to receive instantaneous feedback, if possible even while they are still typing which requires constraint checking in the web interface. Another alternative would be to restrict constraint checking to the web interface. Due to security reasons, this solution has to be rejected since HTML and JavaScript validators can easily be bypassed.

In summary, constraint checking cannot happen in only one tier or component. Thus, the challenge is to ensure that the constraint checking logic is consistent while

the constraint definition and validation code is distributed among all components which interact with the data.

What typically happens in practice, for performing constraint checking in the web interface as well as in the server application, is that everything is implemented twice, once in HTML and/or JavaScript and once in Java. Consequently, this means that a change to a constraint definition usually requires that both a back-end/middleware developer for the server-side and a front-end developer with HTML and JavaScript know-how update the validation code (or one developer with both skills). This manual approach quickly leads to a serious issue once an environment is considered where multiple applications or other interacting information systems are transferring data with the logic tier.

It can be argued that constraint maintenance is simple since there exist proven constraint checking mechanisms for many programming languages, for example BV for Java. This might be the case for small information systems with a monolithic architecture. However, the majority of information systems are built with a distributed architecture that includes many (sub-)systems which are interacting together. Further, they are often implemented with different programming languages and potentially deployed on different platforms. An information system which consists of both a web interface and a mobile application for data input and output, as well as a business logic component, is an example for such a heterogeneous and distributed system. Usually, the business logic component is implemented with different programming languages and deployed on a different platform than the web interface and the mobile application.

In summary, constraint checking code is often distributed among the different components of an information system which results in an error-prone and recurrent task of defining and checking constraints during the implementation and maintenance of an information system. Thus, it is easy to miss a location where constraint checking happens in the code. Usually, data validation takes place at different tiers of an information system. For example, in the logic tier to prevent persisting incorrect data to the database, and in the presentation tier where users receive instant notification in the case of incorrect input data.

Even if only one information system or application, respectively, is accessing data, the appropriate constraints are usually distributed among the various layers and components. To examine the different approaches to constraint specification and data validation in more detail, consider a typical three-tier architecture consisting of presentation, logic and data tiers as shown in Fig. 2.3. The logic tier is a bundling of presentation, business and data access layers.

As already indicated, database constraints support data validation in the data tier. Relational and object-relational databases specify constraints in SQL as discussed in detail by [176]. Although object databases usually leave it to the programmer to handle much of the data validation beyond basic typing, several approaches have been proposed in research such as the use of a constraints catalogue to validate objects at well-defined points [135]. Data validation can be performed in the presentation tier on the client-side using form field validation supported directly in HTML5 or client-side code. PowerForms [19] is a proposal for a declarative and domain-specific language for client-side form field validation that allows constraints over several HTML elements to be declared using regular expressions. There the constraints are defined in a *power form document*, and are then translated to JavaScript and HTML. This approach does not cover the

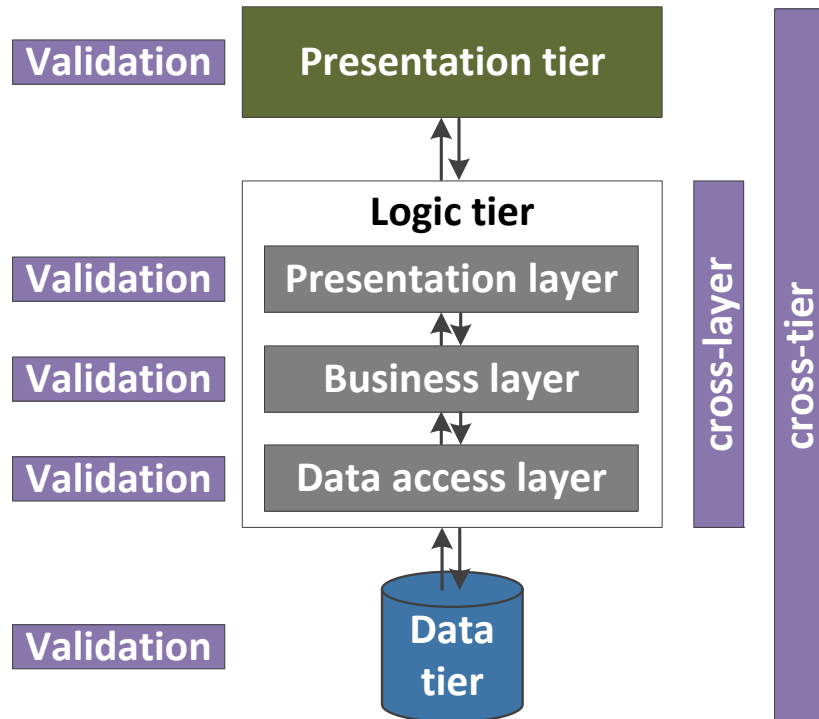


Figure 2.3: Typical three-tier architecture

server-side of the web application, but it generates JavaScript and allows non-JavaScript programmers to create the validation code.

A lot of research has addressed ways in which data validation code within the logic tier can be supported, often through automatic code generation. In the case of presentation layer validation, the layer generates the front-end that is shipped to the presentation tier including data validation components. For example, web forms can be generated automatically from UML and OCL [54].

A survey of approaches for performing validation in the business layer is presented in [65]. There, the different validation techniques within the Java programming language environment range from hand-crafted constraints using if-then-else statements and exception mechanisms over code instrumentation and compiler-based approaches to explicit constraint classes and interceptor mechanisms. Different approaches for constraint checking using OCL are compared in [4] where they consider hand-crafted if-then-else statements and Java assertions as well as approaches based on DbC. Additionally, in [180], an approach is described that translates OCL constraints on UML diagrams into code using object-oriented mechanisms to represent constraints.

Cross-layer validation is not coupled to a specific layer within the logic tier. An example is presented in [70] where they use a sub-language of WebDSL [79, 69] to handle data validation rules. WebDSL is a domain-specific language for creating web applications translated to a running Java web application relying on a customised Java framework. Since the focus is on web applications, it is not directly applicable to desktop applications. A system may also use cross-tier validation that allows data to be validated in different tiers using the same concept. Researchers have proposed many different solutions for this including tools to transform integrity constraints into running code [22]. Moreover, Wang and Mathur [183] present a system that 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. A sophisticated concept for validating input data using the idea of an abstraction of a data category that can be used for data validation is proposed in [167]. The abstraction is transformed into executable code and a similarity function checks the actual value against the defined constraint.

In previous publications, we have already presented different approaches for tackling the issue of distributed constraints and for providing unified constraint management. In [191], for example, a framework for data access layer validation was presented where constraints are defined in a configuration file using a special constraint definition language. The validation process follows the ECA paradigm where an application can register for certain database events, e.g. a commit. Further, it is possible to define constraints which do not have a clear-cut criteria for their satisfaction and assign custom validations. Another approach presented in [193] supports bi-directional translations between an OCL-like platform-independent constraint model and multiple technology-specific representations, allowing for multi-translations between technology-specific representations. There, our OCL extension, called *UnifiedOCL*, enables constraints to be defined in a detached state from the UML class diagram and, by using pluggable label dictionaries, we provide a means of defining common platform-dependent constraint definitions in a convenient platform-independent manner.

2.5.1 Constraint management functionality in databases

A lot of research has been conducted in the field of semantic data models, for example early efforts include [133, 142, 74] where data models with semantically rich constructs and forms of constraints were proposed. These models are targeted at capturing the semantics of an application domain in a more natural way. For example, in [133], an object-oriented model is proposed that integrates features of extended Entity-Relationship-Models (ERMs) such as rich classification structures over collections of objects and relationships. Several approaches of embedding constraints into object databases have been introduced. In [130] they provide simple error check methods for individual classes as well as the organisation of integrity constraints into duplicated classes. In [50, 18], they propose constraint checking in basic update operations. A more advanced approach is DICE [57] where every integrity constraint type is implemented by a parametrised integrity check method and an exception handling method. Nonetheless, such an approach has not been applied in practice and is not integrated in current object databases.

In fact, current databases generally lack sophisticated constraint support. RDBMSs typically provide support for integrity constraints, such as key and domain constraints. Inherent integrity constraints are represented with the DDL at schema definition time and explicit constraints are expressed and enforced by verification mechanisms such as check conditions and assertions. For simple validations these checks can be easily defined but quickly become complex for more advanced constraints. Stored procedures and user-defined functions are relatively simple mechanisms for extending capabilities of databases. Further, triggers can be used to automatically execute procedural statements when a database is modified but cannot trigger on select-queries. Moreover, these verification mechanisms are limited to the schema definition time and inherently to validation in the database layer.

Object databases provide even less constraint support. For example, the commercial product Versant²³ only provides unique field constraints and explicit constraints are usually specified by means of (callback-)methods [13]. For constraint enforcement, typically active database techniques such as the ECA paradigm are used [141]. The checking of integrity constraints in object databases is a fundamental problem in database design [154, 63] and the functionality to declare, enforce and maintain integrity constraints in existing object databases is still very limited [200, 63, 49]. Furthermore, both relational and object database management systems lack functionality for defining more complex constraints or relaxed validations of DQ controls, which, for example, only indicate possible DQ deficiency.

OODBMSs especially do not have adequate support for constraints defined in class composition and inheritance hierarchies [5, 42, 63]. Zaqaibeh et al. [201] present an Assertion Model of Integrity Constraints (AMIC) which especially can represent constraints and complex relationships among attributes and classes that are derived from such composition and inheritance hierarchies. Their approach considers both the validation of a constraint set given by a software designer during compile-time and the enforcement of data integrity during the runtime for data given by software end-users.

Other approaches have been proposed to overcome the distribution of constraints and consequent redundancies as well as facilitating constraint management. So-called Centralised Integrity Maintenance (CIM) approaches [200, 43, 49, 177] introduce a dedicated component next to the database that centralises the management of integrity constraints and is responsible for constraint enforcement. Defining and managing constraints separately from the object database allows the definition of complex and sophisticated constraint types which may go beyond the constraint support proposed for object databases, for example [57].

2.5.2 Bean Validation

To decide on a unified constraint management approach, we also analysed programming language features, libraries, frameworks and language specifications. The most suitable technology for the Java programming language turned out to be BV, which is the most common representative of cross-tier validation. As a cross-tier validation concept, it naturally also supports cross-layer validation. BV as specified in the recent Java Specification Request (JSR) 349 (BV 1.1) allows constraints to be defined in a uniform way. Thus, BV constraints are layer/tier independent and can be used for validation at different places within a three-tier architecture, thereby avoiding redundancy, as there is no need to replicate constraints in, for example, the database and on the client-side. BV centralises the definition of validation logic by introducing a metadata model and API to validate JavaBeans. The idea is to enrich JavaBeans with annotations to express constraints within the domain model. The metadata model is not restricted to a specific layer and can be used with other technologies. Furthermore, BV introduces constraints for method arguments and return values, but does not intend to provide a full ‘Programming by Contract’ framework. In BV, developers usually annotate Beans with Java constraint annotations, but if a developer really wants to separate the constraint definitions in a configuration file, BV also offers the opportunity

²³<http://www.actian.com/products/operational-databases/versant/#VOD>

to create a corresponding Extensible Markup Language (XML) file. Analysing the integration convenience of BV, we evaluated several Java Graphical User Interface (GUI) technologies including JavaServer Faces (JSF)²⁴, Google Web Toolkit (GWT)²⁵, Java Foundation Classes Swing²⁶, Standard Widget Toolkit (SWT)²⁷, JavaFX²⁸ and the Eclipse Rich Client Platform (RCP)²⁹. For JSF (since version 2.0) and GWT (since version 2.5), BV is fully integrated. BV is only partially integrated for Swing, SWT, JFace, RCP and JavaFX since some manual steps have to be taken to completely integrate it, such as handling the validation process with action/event listeners, which in turn invoke the BV. Nonetheless, there exist certain libraries, such as Swing Form Builder³⁰, JFace standard validation³¹, JFace BV with JFace Data Binding³² and FXForm2³³, which integrate BV for the appropriate technology or which can be used to overcome the issue of non-existing data validation mechanisms. BV also easily integrates with JSR 338 (Java Persistence API (JPA) 2.1), which defines a standard for managing the mapping between Java objects and relational databases. The standard explicitly states that its purpose is only the mapping from objects to relational databases, but some implementations of the standard, such as Hibernate OGM, also support NoSQL databases. Since BV is a cross-tier validation approach, it satisfies our requirement for using a single constraint model and managing constraints in a single place. Further, since it is based on Java annotations, which are already familiar to developers and integrated into various GUI technologies, it meets our additional requirement for ease of use. Given the trend observed for increased integration of BV into various Java technologies, we believe that the case for using it will become even stronger in the future.

At the time of writing, the new JSR 380³⁴ for BV 2.0 is in the review process. The main focus of this specification lies in supporting and leveraging Java 8, but other issues such as the support for more *customised payloads* of constraint violations might also be addressed.

2.5.3 Aspect-Oriented Programming

A variant of OCL is proposed in [68] which enables to express constraints that specify system properties such as computation precision or timing. The author recognised that slight variations of the same constraints are used at different locations in the system. The work proposes an approach based on AOP to define constraints in a modular way. This AOP approach provides the possibility for defining and managing constraints in a single place which reduces redundancy in terms of constraint definition.

An example for an aspect-oriented constraint checking language is Limes [119]. Limes allows to check constraints, defined on a UML-model level, after the implementa-

²⁴<http://www.oracle.com/technetwork/java/javase/javaserverfaces-139869.html>

²⁵<http://www.gwtproject.org/>

²⁶<http://docs.oracle.com/javase/tutorial/uiswing/start/about.html>

²⁷<https://www.eclipse.org/swt/>

²⁸<http://docs.oracle.com/javafx>

²⁹https://wiki.eclipse.org/Rich_Client_Platform

³⁰<https://github.com/aeremenok/swing-formbuilder>

³¹<http://wiki.eclipse.org/JFace>

³²http://wiki.eclipse.org/JFace_Data_Binding

³³<http://dooapp.github.io/FXForm2/>

³⁴<https://jcp.org/en/jsr/detail?id=380>

tion. The validation procedures and the specification of when and where the constraint checking takes place is defined with AspectJ based on AOP. Thus, for the constraint checking there is no need for modifying the implementation.

If the validation logic aspects are organised in a separated constraint definition file, the AOP approach allows constraints to be added and modified during runtime, so that the constraint modifications are immediately effective without the need for recompiling the application.

In [24] an aspect-oriented approach is proposed which deals with information system development, while avoiding information restatement, code duplication and problems related to cross-cutting concerns. In their work, they divide system concerns, describe them separately and let them compose together through an aspect weaver. Their aspect-oriented framework provides automatic integration of business rules and dynamic construction of context-aware UIs. Further approaches based on AOP, such as [168, 26, 25, 96, 97], focus on generating UIs and are therefore described later in Sect. 6.1 where we present related work on adaptive UIs in mobile applications.

2.5.4 Business Rules Engines

BRMSs/BREs, such as Drools and Nools (see also Sect. 2.4.8), provide comprehensive functionality for resolving business rules in information systems. Rules can be modified without changing the software code base since their definition is decoupled from the application which is consuming them. With BRMSs it is possible to define and check constraints, although they are not primarily optimised for data input validation with resulting validation feedback or general capabilities related to DQ. Thus, it must be carefully weighed if the overhead for integrating and managing them is worth the provided functionality.

There exist many different approaches which use BREs or ECA rules for maintaining and checking integrity constraints, for example [85] which proposes a rule-based approach for the runtime monitoring of instance-spanning constraints, [6] which presents a transformation based approach to automate the process of transforming business rules to OCL, and [84] which presents a model for maintaining the consistency of mobile databases with a rule-based mechanism based on ECA rules. Further work is even concerned with the assessment of DQ with rule-based approaches. For example: [116] proposes the use of logic rules to assess the quality of measures in a data warehouse. [111] proposes a business rules approach to building a data validation engine that transforms declarative DQ rules into code that objectively measures and reports levels of DQ based on user expectations. [110] describes a framework for defining DQ and business rules that qualify data values within their context, as well as the mechanism for using a rule-based system for measuring conformity to these business rules.

In summary, we can identify the following advantages in using a BRE for the work with constraints: (1) Validation based on a BRE separates the definition of constraints/rules and the logic for checking them from the rest of the information system; (2) All constraints/rules can be managed in one place; (3) The definition of rules is easy to understand.

2.5.5 Constraint translations

During the last years, MDA techniques have been comprehensively developed and consequently translation mechanisms for modelling languages have also gained a lot of attention. Demuth et al. [41] propose Constraint-Driven Modelling (CDM) as a generic approach that guides the construction of new models while conserving consistency with the related models and eliminating issues arising from re-transformation, uncertainties, and bi-directionality. In their work, transformations are used to generate constraints on a target model rather than the target model itself but the resulting constraint representation cannot be used for validation during the runtime of an application.

Since OCL is unable to express some forms of constraints frequently used by developers and cannot be validated directly at runtime, and constraint definitions are usually distributed in various components, an approach is needed which enables developers to specify their constraints in a unified way and automatically map these to technology-specific representations which are validated during runtime.

Regarding constraint transformations there exist a few approaches which translate modelling languages such as OCL to another technology-specific language in which assertions during runtime can be performed. In the remainder of this section we discuss the works reported in literature that provide support for translating OCL to and from other constraint representations. Additionally, we discuss an implementation which translates BV annotations to a JavaScript Object Notation (JSON) representation which we used for one of our approaches.

Most of the approaches provide translations from OCL to mainly SQL such as [138, 136, 48, 76]. Obrenovic et al. [136] for example present an approach for the transformation of check constraints specified at the platform independent level into the relational model, and further transformation into executable SQL/DDDL code. Therefore, they also developed a DSL and embedded it into a tool for platform independent design and automated prototyping on information systems. There exist also various translation approaches to Java [194, 44] and specifically also to JML [104] such as [73, 75, 3]. There are also AOP approaches such as [99, 28] where design constraints defined in OCL are translated into aspects that, when applied to a particular implementation, check the constraints at runtime. A comparison of different approaches for constraint checking using OCL is provided in [4] where they consider several approaches including direct translation to implementation languages, use of executable assertion languages, and use of AOP languages. As a state-of-the-art tool for model-driven software development, AndroMDA³⁵ is a code generation framework that follows the MDA paradigm. It takes a UML model from a UML tool and generates classes and deployable components (*J2EE* or other) specific for an application architecture.

Very few approaches target translations from a technology-specific representation. An example is the work of Cosentino and Martínez [30] which discusses a reverse engineering approach focused on extracting integrity constraints from relational databases and mapping them into OCL expressions. They also provide a working tool capable of inferring those OCL constraints from SQL schema elements (column constraints, triggers). The authors introduce a mapping from an SQL schema into a UML class diagram since OCL constraints are not *context-free* and therefore cannot exist in a detached state from the UML class diagram.

³⁵<http://andromda.sourceforge.net/>

Most related to our work is that of Moiseev et al. in [126, 125] and Baresi and Young [10] concerning translations from OCL to any assertion, and the work of Shimba et al. [170] which is one of the very few works concerning bi-directional translations between OCL and JML.

Moiseev et al. [126, 125] used the same MDA approach as we do by building an Abstract Syntax Tree (AST) but they were focussing on similarities between particular languages (e.g. imperative ones), and they applied sequences of hierarchical transformations for concepts derived from language similarities. Thus, their transformations are more fine-grained, but only consider constructs in isolation. Our approach clearly defines just one intermediate form, but is powerful enough to capture all the concepts. Further, their work is more related to OCL construct transformations in depth, while we are not focussing on the transformations itself but on constraints and their transformations as part of a whole framework. A limitation of their solution is the fact that they only consider unidirectional transformations. Thus, they can only transform OCL to other languages while our solution is bi-directional.

Baresi and Young [10] present a prototype of a tool for translating OCL assertions to implementation assertions which is also limited to unidirectional transformations. Their translations are accomplished by applying externally defined rules, which can vary according to the target notation and users' needs. Unfortunately, they do not describe these rules in detail and their work in this area seems to be discontinued.

Shimba et al. [170] present a concrete method of implementing the translation from OCL to JML as well as the reverse translation using Eclipse Modelling Tools³⁶. Like ours, their implementation is based on *Eclipse*³⁷ and *Xtext*³⁸. However, although they mention Query/View/Transformation (QVT) Operational³⁹ and the ATL Transformation Language (ATL)⁴⁰, they do not explicitly state if they use it for writing translation rules. We used QVT for only one case, using *Xtend*⁴¹ for all others. With their bi-directional translation between OCL and JML, the authors support development by Round Trip Engineering at the design level but do not support multi-translations among various specification languages.

Allowing multi-translations between technology-specific representations is especially beneficial if legacy systems want to move from one constraint representation to another or if a software developer prefers not to use a specific representation.

An implementation of a specific constraint translation is *Valdr*⁴², which provides client-side constraint checking in combination with a constraint transformation mechanism as a JavaScript application. Constraints are homogeneously defined in a JSON file which is consumed by an AngularJS web client plugin. *Valdr* further takes care of the data input validation and of notifying the user with the validation feedback. The constraint checking and the notifications are decoupled from the web interface in most cases. The forms which need constraint checking have to be annotated with a specific Cascading Style Sheets (CSS) class to make use of the client-side validation

³⁶<https://www.eclipse.org/downloads/packages/eclipse-modeling-tools/junosr1>

³⁷<https://eclipse.org/>

³⁸<http://www.eclipse.org/Xtext/>

³⁹<https://projects.eclipse.org/projects/modeling.mmt.qvt-oml>

⁴⁰<https://wiki.eclipse.org/ATL>

⁴¹<https://eclipse.org/xtend/>

⁴²<https://github.com/netceteragroup/valdr>

functionality provided by Valdr. Besides, a mechanism for the transformation of BV annotations to the appropriate JSON representation is provided. Thus, it is possible to modify the constraints on the server-side and generate the corresponding JSON file by running the transformation process. Consequently, the constraint modifications take effect immediately on the client-side.

[1] proposes an algebraic approach with constraint modifications and transformations for data schema transformation. They suggest to formalise the propagation and introduction of constraints by using enhanced data refinement laws during schema transformation. This allows for preserving or modifying constraints during the modification of the data schema, as an example of data schema mapping from XML to SQL.

2.6 Data quality with regards to constraints

After having given an overview of DQ, DQMFs, constraints, general concepts and unified constraint management we will now provide the background of bringing DQ in relation to constraints.

It is not obvious how DQ can be measured and quantified for individual information systems, but it is clear that data considered to be of poor quality in some sense can cause several problems in an information system, see for example [113, 182, 185].

Although the research community has been concerned with several conceptual/descriptive frameworks for DQ, there does not exist a DQMF implementation which is based on a model that integrates all types of constraints.

Approaches on how to assess DQ and identify requirements is comprehensively analysed by [11] where three core aspects of such approaches are identified, namely ‘data and process analysis’, ‘DQ requirements analysis’ and ‘DQ analysis’. In our work, we assume that the DQ requirements are already identified and expressed by constraints in order to focus on the DQ analysis where datasets are analysed regarding their ‘constraint compliance’.

As described above, integrity constraints are commonly defined for relational databases in the database schema. At any point in time, these constraints must be fulfilled for all data which is stored in the database [12]. The main focus of integrity constraints is the accuracy dimension. However, their usage is limited to the validation of relatively simple accuracy checks, such as a length check of a ZIP code. Usually, it is not possible with integrity constraints to encompass all requirements for accuracy, such as a correct non-standard ZIP code format [186].

The use of constraint checking methods for simplifying the evaluation of DQ appears to be applicable since (integrity) constraints and quality can both be described by assertions. Similarities and differences between integrity and quality are described in [33]. Nonetheless, linking constraints to DQ is difficult since constraints usually are defined upfront and cannot capture all DQ issues, such as semantic errors. Therefore, constraints can never be exhaustive for indicating the quality of data, which means that a dataset that satisfies every constraint might still be of bad quality. As an example, suppose that a constraint states that the grades for an exam should be whole numbers between 0 and 20, the satisfaction of the constraint does not ensure that the grades entered in the system are a correct representation of the grades noted on the exam scripts. Likewise, objects that are subject to no constraints are not linked to DQ or

would receive a default DQ score. There is no certainty that an item that is not subject to any formal constraint is always a valid data item. Validity and correctness are mostly evaluated by comparing data to real world values. Therefore, assessing the quality of an item based on constraints, as proposed in this work, results in a measurement of ‘constraint compliance’ which is simply seen as an indicator for DQ rather than an absolute measure.

An approach for improving DQ of relational databases based on Conditional Functional Dependencies (CFDs) is proposed in [60, 58]. CFDs are a class of integrity constraints that are intended to capture DQ issues or conditions, and be used to directly support data cleaning processes [163]. Their work is focusing on checking and enforcing integrity constraints in a DBMS and not the application logic. Decker [33] describes an approach which expresses quality properties as constraints and monitors them with inconsistency-tolerant integrity checking methods. He also argues for two benefits: (1) Using the expressive power of the syntax of semantic integrity constraints also for describing arbitrarily general quality properties of data; (2) Making use of established integrity checking methods in order to efficiently check data also for quality. How integrity constraints can be used to model, measure, monitor and repair the quality of information stored in databases is presented in [37, 35, 36, 38, 34], where they focus on measuring quality by measuring inconsistency. They provide several inconsistency metrics by not only counting constraint violations but also assigning and aggregating specific application-dependent weights to different constraint violations. The work of Decker as mentioned above is most closely related to our work, but it focusses on constraints in relational databases. Further, it handles neither unified constraint management nor customisable constraint checking as investigated in our work.

As proposed by Presser [146], constraints in relation to DQ can be classified generally into hard constraints, soft constraints and other indicators. Hard constraints are necessary conditions that data has to fulfil to be valid. If a hard constraint is violated, DQ is deficient. Soft constraints are not just valid or invalid, but rather increase or decrease DQ. A soft constraint in a food composition database could be defined, for example, in the dimension completeness for a food and its nutrient values. It could be requested that a food should have at least four nutrient values such as carbohydrate, energy, fat and protein. Another example would be that a measurement method or information to samples should be given for each nutrient. But it also could be possible that there is no such detailed information available. In such a case, the soft constraint is violated and the DQ lowered but not considered deficient. In the definition of Presser, DQ indicators also only increase or decrease DQ like soft constraints but they are DQ requirements that can, but do not have to, have an impact on the DQ. For example, a typical DQ indicator would be the age of data as it might indicate that the methods used to collect the data are outdated [146].

Usually a data model is defined in terms of the structures, constraints and operations and the notion of (hard) constraints in data models is well known. By also taking soft constraints and other DQ indicators into account for our constraint model, and by refining and expanding them, we go beyond the commonly known (hard) constraints.

2.6.1 Data integrity rules for continuous data quality improvement

Linking quality to a continuous improvement process is described in Total Quality Management (TQM) [93, 40] which is a practical approach for improving quality. Madnick and Wang [114] introduced an adaptation of TQM principles for the context of DQ. Their Total Data Quality Management (TDQM) cycle consists of defining, measuring, analysing, and improving DQ through multiple, continuous improvement cycles. Thus, (1) a definition of DQ is acquired where, in a typical DQ improvement project, a subset of DQ dimensions are chosen, (2) DQ is measured along the chosen dimensions using DQ metrics which are measurable at a reasonable cost and useful for improvement activities, (3) the measures are analysed and it is decided whether and how to improve the quality of data, and (4) improvement actions are taken to change data values directly and/or change the processes that generate the data. The result of the improvement activities are refinements to the definition and metrics for DQ which are then used for a next TDQM cycle. Measures of DQ during *phase 3*, such as a percentage of values outside the range are used to investigate problematic values and to determine root causes of out-of-specification values. The improvement is either to adapt inappropriate quality definitions or apply corrective actions to the data and/or processes. Changing processes is important since incorrect processes otherwise continue to produce poor quality data.

In [106] the authors propose an approach to reflect the dynamic and global context of business process changes. Their *action research* merges data integrity theory with management theories about quality improvement using a DQ lens. They go beyond the conventional practice for the application of data integrity as a one-time static process applied when data enters the database. Instead they propose that the application of data integrity be viewed as a dynamic, continuous process, embedded in an overall DQ improvement process. They argue that constraint-based rules in databases help ensure that data represents the real-world states, but that a real-world state is dynamic, not static, changing over time and therefore organisations are in need of a process that guides the mapping of changing real-world states into redefined data integrity rules. Thus, they propose an action research cycle for DQ improvement which extends the TDQM cycle by the phases (1) Identifying Inquiry, (2) Diagnosing Problems, (3) Planning Solution, (4) Implementing Solution, and (5) Reflecting and Learning. Further they present how their cycle was applied to process-embedded data integrity in a global manufacturing company for different data integrity constraint categories.

2.6.2 Data quality measurement and metrics

If DQ is seen as a constraint compliance problem, which means that measuring and quantifying DQ in an information system is based on the degree to which data complies with certain constraints, the set of requirements can be expressed by a set of constraints. However, specific measurements for DQ are not supported in most information systems, which tend to rely on a set of constraints that only validate to either true or false. Thus, either a user enters valid data and is able to store it in the database or the data is rejected. In many cases, this is not sufficient, as it might be appropriate to distinguish several levels of DQ or even use a continuous range from very good DQ to very poor DQ. This requires another concept for validation as well as metrics for DQ.

We therefore chose to investigate how constraint models could be extended to sup-

port DQ management. Our goal was to design a general framework that could be used to analyse and provide feedback on DQ for any information system. Given that DQ requirements are highly dependent on the context in which the data is used in terms of the domain, the application and the user, it is important that the computation of DQ is highly flexible and easily configurable. In particular, the set of DQ dimensions considered should not be fixed.

In [143] it is emphasised that the process of measuring DQ must confront two major questions: (1) Exactly what should be measured? and (2) What is the metric that will be used to measure the variable? The author also states that, often, it is much more difficult answering the first question than the second.

Organisations are in need to quantify and improve the quality of their data since their decision making is based on their data. Due to the diversity of DQ definitions, many approaches exist on how the quality of data in an information system can be assessed, scored and quantified.

A summary of methodologies for assessing and improving DQ is provided in [11] where a sequence of the following three common DQ methodology phases is presented:

1. *State reconstruction*: Collect context information about the processes and services in the organisation, data collections and related procedures, DQ issues and associated costs. This phase can be skipped if documentation exists from previous analyses.
2. *Assessment/measurement*: Define relevant DQ dimensions and measure the quality of a data collection along these dimensions.
3. *Improvement*: Select steps, strategies and techniques to correct the issues collected in *phase 1* and consequently improve the DQ.

In our work, we mainly focus on the second and third phases. The DQ issues are represented by constraint violations. For the assessment of DQ in the second phase, we provide metrics for indicating DQ based on constraint compliance with configurable DQ dimensions associated. The third phase is addressed by including features that support DQ improvement processes.

A simple method to determine the accuracy of a data point is to calculate the distance between the value of a data point and its real value. In the case of empirical sciences such as physics, chemistry, biology, environmental science and food science, real values are often not known and are only approximately measurable. For example, if one considers the Vitamin C value of an apple, a food scientist can only measure this nutrient using the most current analysis method. As methods improve over time, the measured value should hopefully come closer to the real value, but the real value remains unknown. Whether the apples currently in a specific fruit bowl have the same value is also unknown.

With empirical data, the source of the data as well as the methods used to measure and process values, all contribute to a scientist's assessment of the reliability of that data. This means that the life cycle of empirical data plays an important role in determining DQ. This was also recognised in the work of [53] which proposes an information quality framework where the dimensions of information quality are categorised according to four levels and four phases. The four levels describe the relation of data to the target community, the information product, information process and infrastructure.

The four phases represent the information life cycle from a user's point of view, namely finding information, understanding and evaluating it, adapting it to the user's context and applying it in the correct manner. In case of empirical data, it is important to determine what the appropriate phases of its life cycle are and what criteria are important in each of these phases to produce a measure of DQ. Based on these criteria, a DQ framework could be designed to manage the quality of empirical data as part of a scientific information system by storing the relevant metadata and using it to derive a measure of the reliability of data values.

The use of metadata to evaluate DQ has been proposed by other researchers, for example [121] and [162]. Specifically, Rothenberg argues that information producers should perform verification, validation, and certification of their data and then provide DQ metadata along with the datasets. Also, [131] presented a mediation framework for the querying of data in molecular biology where data is selected from different data sources based on DQ information stored as metadata. The authors based their approach on the DQ framework of Wang and Strong [187], defining scores for each of the DQ dimensions and normalising them to build a weighted sum. The DQ information for the different sources is stored as metadata and the mediator performs a quality-driven source selection when executing queries, returning only data of higher quality to users.

Caballero et al. [21], with their Data Quality Measurement Information Model (DQ-MIM) approach, provide ideas on how to assess DQ by means of the reliability of the data sources and on how different DQ dimension measures can be combined. They further present a concept on how data, provided in an DQ XML file, can be validated for different roles of stakeholders, such as 'Information Supplier', 'Information Manufacturer', 'Information Consumer', and 'Information Manager' as identified by [184]. They base their work on the idea of storing data together with metadata, and on recommendations, that measures about DQ should be stored with the data model, to allow for reusing the measures in other measurement processes [139]. For each measurement process, different values of metadata can be required by different roles. Both data and its corresponding metadata are brought together for each role which is related to the concept of *tagging*. Tagging data with *quality indicators*, which are objective characteristics of the data and its manufacturing process, is proposed in [185] where the user may assess the data's quality for the intended application based on these indicators. Caballero et al. [21] store the DQ metadata as a measurement plan for various stakeholders in a DQMIM XML file which together with the data (DQ XML as mentioned above) can be processed for appropriate measurement results. Although they provide a concept, which is related to our contracts approach, their work is limited to processing XML files and no tool (case study) has so far been implemented for automating the measurement of appropriate data in an application.

The impact of DQ information (such as DQ metadata or tags) on decision making processes and decision outcomes is described in recent work such as [124, 150] where it is also indicated that providing DQ metadata along with the actual data is important so that decision makers can gauge the appropriateness of the DQ level for the task at hand [27]. Moges et al. [124] also provide an overview of the different DQ metadata formats explored in the literature such as 'Ordinal', 'Interval', 'Probability', 'Range' and 'Graphical'.

Research on tools and metrics to support DQ management in practice tends to be limited to specific dimensions or domains. For example, specific metrics have been pro-

posed for the DQ dimensions of timeliness [101, 80, 8] and accuracy [101, 80]. Pipino et al. [144] on the other hand presented three general ways, so-called *functional forms*, of deriving measures of DQ based on simple ratio, the minimum or maximum operation and the weighted average. The simple ratio measures the ratio of current outcomes to total outcomes. For instance, if a column of a table should contain at least one occurrence of all 50 US states but it only contains 43 states, then we have population incompleteness and a ratio of 43/50. Minimum or maximum can be used to aggregate multiple DQ dimensions by simply selecting the minimum or maximum value, respectively, from the normalised DQ values of the individual DQ dimensions. Alternatively, one could use a weighted average to allow that certain dimensions are given more importance than others in determining DQ. Further approaches for the measurement of DQ are presented, for example, in [77, 21, 145].

In [145] the authors provide an overview of measurements including the different classes of scales, such as nominal, ordinal, interval, and ratio. They state that a lack of attention to the scale type can lead to improper interpretation and application of measurement results, especially, if different dimensions are combined to develop an overall index of DQ in an organisation. Thus, it is indispensable that DQ is precisely defined and the correct scales are applied for the appropriate DQ dimensions. In their work, they present exact definitions of several DQ dimensions (completeness, correctness, system currency, storage time, and volatility) and show that these dimensions can be measured using the ratio scale. Further, they propose DQ measures for each DQ dimension and generalise the results to any DQ dimension for which certain prerequisites hold.

Due to a lack of common DQ measurement terminology, Caballero et al. [21] provide an overview of DQ measurement terms found in the literature, focussing on the most representative proposals [105, 109, 157, 132, 51]. Their DQMIM approach provides a standardisation of the referred terms by following ISO/IEC 15939 [88] as a basis. Further, they analyse DQ measurement terms by categorising them in several question topics important for DQ assessment, such as why, what, where, who, whose, how, how much, how many, and when. In the ‘When to measure?’ question, for example, static and dynamic ways of performing a measurement as proposed by [109], can be distinguished. Thus, different DQ snapshots (static measurements) could be taken at different points of the data life cycle (dynamic measurement) for tracking the different DQ values through the information system. In their work, they state that measurements should have a clearly defined purpose and propose a measurement plan based on the information needs.

Two different perspectives on the measurement of quality can be distinguished [78, 93, 174]: ‘Quality of Design’ and ‘Quality of Conformance’. As defined by Heinrich et al. [77], ‘Quality of Design denotes the degree of correspondence between the users’ requirements and the specification of the information system (e.g. specified by means of data schemata). In contrast, Quality of Conformance represents the degree of correspondence between the specification and the existing realisation in information systems (e.g. data schemata vs. set of stored customer data).’ Similar to our work, they focus in their work on ‘Quality of Conformance’, where they analysed how DQ can be quantified with respect to particular dimensions. As a basis for their analysis of metrics in the literature and their design of new metrics for the dimensions of correctness and timeliness, they state the following requirements for designing adequate metrics:

- R1 *Normalisation*: To assure that the values of the metrics are comparable, an adequate normalisation is necessary.
- R2 *Inverval scale*: Metrics are required to be interval scaled to support both the monitoring of how DQ level changes over time and the economic evaluation of metrics.
- R3 *Interpretability*: DQ metrics have to be comprehensible to provide measurements which are easy to interpret by business users.
- R4 *Aggregation*: Metrics shall allow a flexible application where it must be possible to quantify DQ on the level of attribute values, tuples, relations and the whole database.
- *Interpretation consistency*: On each level the values must have consistent semantic interpretation.
 - *Aggregation consistency*: On a given level to the next higher level, metrics must allow aggregation of values.
- R5 *Adaptivity*: The metrics should be adaptable to the context of a particular application.
- R6 *Feasibility*: Metrics should be based on input parameters that are determinable and the measurement procedure should be accomplished at a high level of automation.

In summary, there is still not a universal set of DQ dimensions valid for any context nor an exhaustive set of measures for these dimensions [23]. Further, little has been done on addressing the general requirements of DQ in scientific information systems with a view to developing a tool to support DQ management in practice. As a first step, it is important to gain an in-depth understanding of what DQ means to the various stakeholders in a scientific information system. Further, it is necessary to investigate not only how work practices can affect DQ, but also how a tool to manage DQ could have a positive influence on work practices and support the planning of future work.

In [144], the authors raise the question ‘Is there a single aggregate DQ measure?’. They argue that, in practice, companies would want to develop an *index of DQ* as a single aggregate measure of their DQ. ‘A single-valued, aggregate DQ measure would be subject to all the deficiencies associated with widely used indexes like the Dow Jones Industrial Average and the Consumer Price Index.’ [144] With our approaches on DQ metrics and measurements we also followed this goal of a single aggregate DQ measure as an index of DQ.

Objective and subjective data quality assessment

Most DQ measures aim to solve specific problems and are therefore developed on an ad hoc basis [82, 103]. But to provide a complete DQ assessment, both the objective measurements based on the dataset itself, and the subjective perceptions of the individuals involved with the data have to be taken into account [144].

The experiences and needs of the various stakeholders are reflect by subjective DQ assessments [8, 184] where questionnaires can be used to measure the perceptions of stakeholders concerning the various DQ dimensions. One example that has been used

in practice is the Information Quality Assessment (IQA) instrument [105] or a subset of the IQA, e.g. statements such as ‘The information is incomplete’, ‘The information is believable’, and ‘The information is correct’ [143]. Objective DQ assessments are either task-independent or task-dependent. Metrics for task-independent DQ assessments can be applied to any dataset regardless of the context and task while metrics for task-dependent DQ assessments, such as business rules, regulations, and constraints, cannot be established without knowledge of the specific application contexts. Examples of such objective DQ metrics are the three functional forms presented above [144].

Pipino et al. [144] and Lee et al. [105] describe principles that can help organisations develop usable DQ metrics where they present subjective and objective DQ assessments and an approach that combines both assessments and illustrate how it has been used in practice. They argue that the following three steps, presented in [144], are needed to use the subjective and objective metrics to improve organisational DQ:

1. ‘Performing subjective and objective DQ assessments’
2. ‘Comparing the results of the assessments, identifying discrepancies, and determining root causes of discrepancies’
3. ‘Determining and taking necessary actions for improvement’

They propose to compare the subjective and objective assessments of a specific DQ dimension during the analysis. The outcome of the analysis is a categorisation into one of the four combinations of objective and subjective DQ assessments: low/low, low/high, high/low and high/high. The goal is to achieve a DQ state of the combination high/high. For any other combination, the company must examine root causes and perform corrective actions, which differ for each combination.

A variant of the approach above is to assess (1) subjective perceptions of information quality, that is, the perceived quality of information along some specific quality dimensions, and (2) objective quantitative measures of the same dimensions, where measurable [143]. It is stated that ‘these subjective measures can provide comparisons across stakeholders and, as a consequence, provide valuable information regarding the disconnects among the different stakeholders with respect to specific data’.

One of the approaches for a DQMF presented in this work includes, besides objective DQ assessment, two concepts for subjective DQ assessment which are established in research, but are not extensively explored in the context of DQ: ‘Implicit user feedback’ and ‘explicit user feedback’. Implicit feedback is information collected from user behaviour, such as click-through data on the web. Typically, implicit feedback can be gathered at much lower cost, but the interpretation of implicit feedback is much more complex than the interpretation of explicit feedback [91]. Explicit feedback is information that users provide deliberately, for example by filling in a feedback form.

In our work, we infer implicit feedback from user click-data which enables us to rate the user’s interest in the diverse data instances. This feedback is used to determine the importance of an instance or entity related to the other instances and entities contained in the information system.

In e-commerce, explicit feedback has grown strongly, where the influence on potential new customers based on online hotel reviews is the prime example [181]. We investigate the usefulness of explicit feedback related to the assessment of DQ. Our aim is to use user feedback on the quality of data to improve the analysis of quality issues

and to enhance the DQ in an information system. Therefore, we provide an implementation of a mechanism to collect explicit feedback during data input as well as to enable the assessment of the received feedback for improving DQ.

2.7 Giving feedback on data quality

How can good quality of data in an information system be ensured? Most obviously, we think first of restricting the input process so that data of poor quality cannot be entered into the system. This can be done at the UI level, within application programs or at the database level using constraints. Alongside data input validation, there exists the possibility of analysing the quality of the data already stored in the database. Users (or power users) can then be informed about any problems and take actions to deal with them. Such an analysis is related to DQ assurance or data cleaning which is concerned with the correction and improvement of data in databases as described in [66].

For both types of validation, referred to as DQIF and DQAF, respectively, it is necessary to give feedback to the user to enable the user to correct the data. We will look at the requirements of each of these in turn before going on to discuss what has been done so far on DQ in the domain of food sciences.

Data Quality Input Feedback There exist many possibilities for UI data input validation feedback. We give an overview of the most common techniques and distinguish three aspects of the feedback, namely (1) where, (2) how and (3) when.

1. In a UI, usually there exists an input area and many input components such as form fields. Feedback can be given close to the input data (e.g. above or to the right of a field or as a tool tip) or outside of the input area (above or below).
2. How to give feedback also depends on where the feedback is given. Textual feedback or feedback with symbols (e.g. a stop sign) can be given anywhere. The feedback can be supported by colouring the various components such as messages, field labels or the field itself. Usually green, yellow and red are used to show the different states from valid to invalid data. When colouring a field, there exist variations such as a glow effect or change to the line or background colour.
3. Validation and feedback can take place either when a single data item is entered or after all data in an input area has been entered. A validation time could, for example, be defined as when the focus changes from one field to another or when pressing a specific button. It is also possible to check the input as the user is entering it in the case of an input text field.

If the data submitted by the user is valid, it is good practice to let the user know that everything went as planned. If it is invalid, the user should be informed (1) that an error has occurred, (2) where the error occurred and (3) how the error can be repaired. Thus, feedback not only involves highlighting where an error occurred but also providing information about what is missing or needs to be changed in order for the data to be valid or of higher quality.

Figure 2.4a shows an example of validation feedback outside of the input area supported by a warning symbol and a surrounding red line for the message area. Fig. 2.4b and Fig. 2.5 show examples of feedback next to the input data, in both cases with

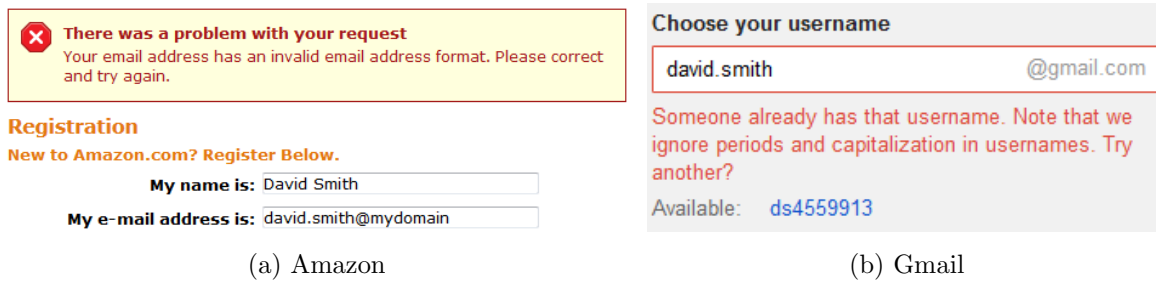


Figure 2.4: Various feedback variants

examples of correct input. In Fig. 2.4b, the fields are also highlighted with a red line colour, whereas in Fig. 2.5 a small bar shows the quality of the password field as the data is entered.



Figure 2.5: Twitter

Researchers have proposed a number of ways of making it easier for developers to specify validation conditions as well as feedback to be given. PowerForms [19] is a proposal for a declarative and domain-specific language for client-side form field

validation. 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. On submit, validation violations are displayed within a JavaScript alert box. [17] state that to facilitate the ease of use of UIs, users should be guided in ways such as highlighting and describing invalid input, and showing/hiding or enabling/disabling particular UI widgets. They present a formal model and prototype (Cepheus) for enabling the specification of user input evaluation rules and interface responses by domain experts. [70] present a sublanguage of WebDSL to handle data validation rules. 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.

Data Quality Analysis Feedback In contrast to DQIF, the user gets feedback about multiple or all entities in a database at the same time in DQAF. Therefore, the feedback should include ways of identifying datasets where a correction might be required. Many different visualisations based for example on charts and tables can be used to give the appropriate feedback. It is also important to provide a means of navigating from overviews to individual data entities and their associated violations in order to identify problems.

Data cleaning tasks such as the removal of duplicate records, data standardisation and data profiling are described in [66]. The authors provide a guide offering practical advice on options available for building or choosing a data cleaning solution. Such solutions also include the automated correction of invalid data once the defects have been determined. The prime example used in data cleaning is the domain of address data where tools such as SQL Power DQguru⁴³, OpenRefine⁴⁴ and Wrangler [95] have been developed.

⁴³<http://www.sqlpower.ca/page/dqguru>

⁴⁴<http://openrefine.org>

Methods for error detection that go beyond integrity analysis are reviewed and presented in [115]. The applicable methods include: statistical outlier detection, pattern matching, clustering and data mining techniques.

Studies on data feedback already exist in domains other than food science. For example, the study of [20] in the healthcare domain illustrates the diversity of hospital-based efforts for data feedback and highlights successful strategies and common pitfalls in designing and implementing data feedback to support performance improvement. They came up with seven key themes such as ‘The source and timeliness of data are critical to perceived validity’.

The initial DQ evaluation procedures developed by the US Department of Agriculture (USDA)⁴⁵ were manual processes to assess the quality of analytical data for iron, selenium and carotenoids in foods. In the course of a redesign of the software system used at the USDA, these procedures were taken a step further and a generic system was developed [81]. The five original evaluation categories ‘Sampling Plan’, ‘Number of Samples’, ‘Sample Handling’, ‘Analytical Method’ and ‘Analytical Quality Control’ were maintained, but the quality assessment questions were made more objective. According to the answers to these questions, a single numeric Quality Index (QIX) is assigned to a nutritional component. At aggregation, a Confidence Code (ConfC) is assigned to the combined value, which is calculated as the weighted mean of the individual values from the different sources of data. The ConfC is derived from the QIXs by summing up the adjusted ratings of the individual values.

The European Food Information Resource (EuroFIR)⁴⁶ developed a QIX [166] similar to that of USDA. In fact, they adopted the five categories from the USDA QIX and additionally added ‘Food Description’ and ‘Component Identification’. For all of the seven categories, a set of questions is defined. There are 34 questions in total, which, all except one, can be answered with ‘Yes’, ‘No’ or ‘Not Applicable’.

Although several conceptual frameworks for DQ have been proposed, there is still a lack of general tools and metrics to measure and control the quality of data in practice. As a first step in this direction, Presser [147, 146] carried out a detailed study of DQ requirements for the FoodCASE information system to manage food composition data. As users, system designers and developers as well as food compilers and project managers were included. In addition to determining which dimensions of DQ specified in existing conceptual frameworks users consider important in assessing the reliability of data, the users were also asked to assess the importance of various criteria related specifically to empirical data.

Based on the results of this analysis, Presser [146] integrated enhanced functionality for defining, validating, measuring and visualising DQ into a first version of FoodCASE. Based on constraint definitions, this version of FoodCASE already provided measurements, calculations and visualisation components for visualising DQ and helped users monitor constraints and DQ. The concept and implementation includes the validation and feedback for both DQIF and DQAF.

In addition to performing constraint checking during transaction execution as supported in traditional databases, in this former version of FoodCASE it is also possible to trigger DQ analysis which will use constraints to measure the quality of data. Its

⁴⁵<https://www.usda.gov/wps/portal/usda/usdahome>

⁴⁶<http://www.eurofir.org/>

integrated DQ framework is general and application developers can define their own DQ measurements and configure customised DQ feedback. Presser [146] identified 156 DQ requirements specific to the food science domain which are validated in FoodCASE and to which the end user gets DQ feedback.

2.8 Data quality and constraints in mobile applications

As a result of rapid developments in portable computing technologies and wireless communication, mobile development is gaining in importance [45]. Nonetheless, only little research exists which investigates issues in mobile applications related to DQ. Thus, there is the need for concepts concerning DQ especially for mobile solutions. Mobile applications have small screens, reduced performance capacities, limited battery life and they are always with a user. These features characterise mobile applications which can open a new perspective for investigations.

Data quality In [47], a novel framework is presented for providing *Quality of Information* evaluation in the context of image applications in constrained networks. The approach proposes the mapping of data attributes related to DQ dimensions, such as timeliness, accuracy and precision, to a measure of quality of information. The data, that is, an image, is separated from information which is contained in this data, for example, the number of people on the image. The approach suggests that the ability to extract useful and meaningful information from data implies high DQ.

For DQ research, one of the most popular application domains is medicine and medical records. The importance of medicine and human's health demand for very high quality of the data. The DQ of paper care data against mobile care data for medical records is compared by De La Harpe [32] from a semiotic point of view. Several parameters, which are important for medical DQ, were evaluated and De La Harpe concludes that mobile health solutions are promising to result in better quality care data. However, the human involvement as well as all features of DQ must be considered.

In desktop and mobile applications it is in equal measure important to track DQ and inform users about potential DQ improvements. However, since mobile applications possess special characteristics, as mentioned above, different ways of examining a DQMF are needed. Possibilities for participatory sensing using mobile devices were investigated in [155]. The authors developed a framework for work evaluation which does not correspond entirely to the traditional DQ attributes. The framework provides interesting features for giving feedback to users: Prompts, adaptive interfaces, positive reinforcement and social validation. Besides using the characteristics of the mobile devices for these features, also the social behaviour of mobile users was exploited. The aim was to encourage users to be more active and interact with other people.

In [188] the quality of spatial data in Geographic Information System (GIS) applications is discussed. DQ plays a very important role in these applications since the accuracy of the presented information and the user experience fully depend on the DQ. Thus, it is inevitable for GIS applications to support elaborate data checks to guarantee a constant high DQ. The work provides visualisations for DQ assessment in an application with the aim to support users to understand the acquired DQ in an easy

and convenient manner. These visualisations should also be understandable for persons who are not specialised in GIS applications.

Constraints In [46] a framework is proposed which caches relevant data items that are needed during integrity constraint checking in mobile databases. Choi et al. [29] propose a synchronisation algorithm between a server-side database and a mobile database which is based on message digest. Based on the work of [201], which focuses on constraint checking in OODBMSs, an extended approach is presented in [202] which is designed to enforce and maintain integrity constraints also in mobile databases. In addition to their adapted assertion constraint model they specify an Object Assertion Language for Integrity Constraints (OALIC) that is proposed to handle classes and their attributes and constraints. The OALIC grammar facilitates the usage of their model and is designed to simplify constraints to any object data model. In their work, constraints are translated into rules and relationships, regardless of whether the constraints are derived from composition or inheritance hierarchies.

2.9 Conclusions

In this chapter, we have presented an overview to DQ and different types of DQMFs. We provided a summary of constraints with related general concepts and explained the issues concerning unified constraint management as well as associating constraints with DQ. Moreover, after having given an overview of feedback on DQ, we specifically explained DQ and constraints in the context of mobile applications.

Most existing work concerning DQ and constraint management has focused on either just theoretical studies or more practical ones covering only one requirement, such as the constraint representation or the constraint applications/usage.

We have shown that only very few existing work focuses on expressing DQ by means of constraints. Although several conceptual frameworks for DQ have been proposed, there is still a lack of general tools and metrics to measure and control the quality of data in practice. None of the existing work presents a general framework that could be used to analyse and provide feedback on DQ for any information system.

Moreover, we have shown that there is a need for DQMFs which manage constraints in a unified way in information systems that are implemented in a distributed manner. Such DQMFs must provide measures and concepts to indicate DQ based on constraint compliance although it is not obvious how DQ can be measured and quantified for individual information systems. Further, it is advisable to include measures and concepts that combine objective DQ assessment, such as constraint compliance, with subjective DQ assessment since it can be used to better understand quality issues and to improve DQ.

To the best of our knowledge, there is no DQMF that provides a unified way of defining and validating DQ requirements by means of constraint compliance and subjective DQ assessment combined with extended constraint checking, DQIF and DQAF as presented in this work.

3

Initial approaches

In this chapter the concepts, implementation and evaluation of the initial approaches for DQMFs are presented. With the initial approaches we investigated the different requirements and goals presented in Chap. 1 by exploring alternative solutions to fulfilling them. The specific aim of these approaches was to explore various ways of (1) defining and checking simple and complex constraints, (2) measures of constraint compliance, (3) bringing constraint compliance in relation to DQ, and (4) giving DQIF and DQAF. In the beginning we focussed on unified constraint management as well as constraint-based DQ measurement and feedback with the objective to build a knowledge base for the final decision on an appropriate approach. All initial approaches, except for the one presented in Sect. 3.1 (*Guidance on data quality feedback*), already satisfy unified constraint management in different ways and base objective DQ assessment on constraint compliance. After having explored the various initial approaches, we recognised that there was not a decision to take for one single combined approach but for a variation of approaches for different problems and complexities. Thus, as we will see in the subsequent chapters, these initial approaches build the knowledge and decision basis for our final approaches, which consolidate and extend concepts from the initial approaches to fulfil all our proposed requirements of a DQMF for various initial software development scenarios.

We first give an overview of all initial approaches before explaining each approach in detail in the subsequent sections. For all approaches, we summarise their main features and distinguish between DQMFs which follow either an *integrated* approach and as such are dependent on the specific implementation, or a *separated* approach which facilitates applying it to any information system. In Tab. 3.1 all initial approaches are listed with their approach name (Approach), a brief description of the main topic (Topic), their main features (Features), their categorisation of dependency (Dep.) where ‘Int.’ indicates an *integrated* approach and ‘Sep.’ a *separated* approach, and a reference to the section (Sect.) where the approach is described in detail.

In Sect. 3.1, **Guidance on data quality feedback**, we provide general guidance on giving feedback on DQ in an information system based on the concepts and imple-

Approach	Topic	Features	Dep.	Sect.
<i>Guidance on data quality feedback</i>	Feedback on DQ	DQIF and DQAF; DQ requirements as Hard Constraints (HCs) and Soft Constraints (SCs); initial measures; first version of the FoodCASE DQMFs	Int.	3.1
<i>Object database data quality management</i>	ODB constraints extension	unified constraint management in a separated repository defined in own definition language; ECA based; HCs/SCs; extensible for new constraint types	Sep.	3.2
<i>Data quality extended Bean Validation</i>	Several extensions for BV	unified constraint management based on BV; new extended HCs/SCs; new measures; support of runtime configuration; complex constraints (cardinality, temporal); time-triggered validation	Int.	3.3
<i>Bean Validation OCL extension</i>	OCL extension for BV	unified constraint management based on BV; extended HCs/SCs; definition of constraints with OCL for BV; limited runtime constraint definition	Int.	3.4
<i>Data quality monitoring framework</i>	Independent DQ monitoring	DQ monitoring; new fine grained severity scale; general DQIs; new measures; limited constraint inheritance; association of individual constraints with DQ dimensions; new DQAF	Sep.	3.5

Table 3.1: Overview of the initial approaches

mentation of the first integrated DQMF in FoodCASE developed by Presser [146]. The framework introduced in this approach allows users to configure both the DQ metrics and how these metrics are visualised, already including initial measures for DQ requirements as HCs and SCs, and initial proposals for DQIF and DQAF. Note however, that this approach was not concerned about unified constraint management, extended constraint checking and extended DQIs. We evaluated both the DQMF and how feedback on DQ is presented to the users with experts in the food sciences and present their feedback as a foundation for the approaches developed in this thesis.

In Sect. 3.2, **Object database data quality management**, we propose an approach to enhance object databases with constraint functionality in a unified way where we used a component external to the database, for a centralised management of integrity constraints. This approach complements object databases with support for constraint

definition, management and validation. We introduce a framework where DQ controls are specified by means of constraints which are defined and managed within the framework and checked against the database based on an ECA paradigm. This approach provides flexible and customisable constraint checking, where constraints can either be used to ensure DQ in a strict and traditional way with constraint violations resulting in exceptions, or in a relaxed way where constraint violation will generate user warnings and indicate possible deficiencies in DQ.

In Sect. 3.3, **Data quality extended Bean Validation**, we present a DQMF that uses a single constraint model to avoid inconsistencies and redundancy within the constraint specification and validation processes. Further, we discuss how classic constraint models need to be extended to support the various forms of data validation required for the management of DQ and show how this has been realised in the context of the Java programming language environment by building on BV. Specifically, we describe how we have extended BV to support runtime configuration of cardinality constraints over associations as well as temporal constraints together with a time-triggered validation component for the scheduling of validation jobs. We also introduce a new concept of SCs in the context of BV to support more flexible forms of DQ management that, instead of prohibiting values which violate constraints, marks them as being of low quality. In contrast to the former introduced SCs, the new concept includes a distinction of different groups and constraint violation severities for SCs.

Further, we propose in Sect. 3.4, **Bean Validation OCL extension**, another approach that enables Java developers to ensure that certain integrity guarantees of an information system can be met through a single, extended constraint model. This approach also rests on the new concept of SCs, introduced in Sect. 3.3, as a basis for measuring DQ alongside the traditional HCs that prohibit invalid data. We describe how the approach can be realised by extending BV to support the definition of constraints with OCL. Further, we show how it is possible to dynamically update constraint definitions during application runtime to support more flexible and powerful ways of managing DQ requirements.

In Sect. 3.5, **Data quality monitoring framework**, we explain how we have extended classic constraint models to provide a general, independent DQ monitoring framework for analysing and giving feedback on DQ in information systems. The basic idea was to extend the concept of constraints to a concept of general DQIs by associating individual constraints with DQ dimensions and assigning violation penalties to them. With these violation penalties, we provide a customisable, fine grained distinction of different severities associated with DQ in the case that constraints are violated. We also used this concept of fine grained severities for our final approaches, which can be seen as an enhancement of the concept of HCs and SCs. In this approach, our model has been realised as a complete DQMF together with a data analysis tool that allows users to inspect different dimensions of DQ for single and multiple object instances, single and multiple object classes and the overall DQ independent of the application and data model. We further propose a means of measuring and calculating DQ for the various levels of DQ feedback taking constraint inheritance into account. We present our DQ analysis tool which provides different types of textual and visual feedback and report on an initial user study.

3.1 Guidance on data quality feedback

So far we have shown that DQ is of major importance to information systems and may vary over time, especially also in the management and processing of scientific data. In previous work [147], we introduced a DQ requirements framework and have shown the importance of the different DQ dimensions for various user groups in the food science domain. With the concepts presented in this section, we address the issue of how to give feedback on DQ to users of an application.

We examine here how and when users should be given feedback about DQ based on a concept for validation which distinguishes several levels of DQ and metrics for DQ which were introduced by Presser [146]. Presser's approach was implemented as a DQ toolkit for measuring and visualising DQ integrated into FoodCASE. The toolkit offers a variety of charts and tables that can be used to provide users with feedback on DQ and help them identify areas where action might be required. It also provides tools to drill down on the DQ issues and identify individual problems.

To evaluate the approach concerning the proposed means of providing feedback on DQ, such as DQIF and DQAF, we carried out a survey involving several food science data experts across Europe.

The work presented in the remainder of this section was previously published in [192]. Therefore, we present here only the main features of the DQ toolkit along with a summary of the results of our survey. We focus on the most interesting insights and conclusions concerning how DQ requirements should be defined, who should define them and how feedback should be given to users. For the interested reader, we refer to [192].

3.1.1 Approach

The DQ analysis toolkit is based on the concept of a RODQ model developed by [146]. A RODQ model is a conceptual model that describes the DQ requirements of an information system independently of its implementation, as described in Chap. 2 on page 19. In this respect, it complements other established modelling languages such as ERM and UML. Furthermore, the RODQ model describes how to assess the DQ.

The central element of an RODQ model is the DQ requirements where [146] distinguishes three types of DQ requirements:

- A HC is a DQ requirement which absolutely must be fulfilled. If an HC is violated, the data is invalid and cannot be used. Therefore, the system should enforce checking HCs and not allow input data to be stored unless all HCs are satisfied.
- A SC is a DQ requirement which is highly recommended to be fulfilled. If a SC is not satisfied, DQ decreases. However, it might not always be possible to adhere to all SCs. Hence, the system cannot enforce checking them.
- An indicator is a property that can be used to estimate DQ, rather than a constraint that can be clearly satisfied or not. A typical example is the age of the data. If the data is old, it is likely to be outdated, but it is still possible that it is correct.

As already introduced, we distinguish between two types of validation and DQ feedback for the user, namely DQIF and DQAF. Both make use of the DQ requirements

defined in the information system. DQIF is only concerned with the DQ requirements of the input data entity which can include relations to other entities whereas DQAF analyses all DQ requirements in the information system. For each type, we will present an example from the FoodCASE system and describe briefly how feedback is given to the user. For the sake of brevity, we will only give an overview of the main features here. A detailed description of the toolkit can be found in [122].

The screenshot displays a web-based form for entering food details. At the top, there's a 'File' menu with 'Save', 'Save & close', and 'Close' options. The main form area is titled 'Food' and contains several input fields: 'Food id' (293), 'Name' (Beeren Crunch (bio-familia AG)), 'English name' (redacted), 'Scientific name', and 'Version' (head [1] (public)). A 'Public' checkbox is checked, and there's a 'Marked for recomputation' checkbox. To the right, there's a small image of a 'Beeren Crunch' product. Below the main form, there are several tabs: 'Own Categories', 'LanguagL', 'FoodEx2', 'Markers', 'Other Properties', 'Remarks', 'Values', and 'Aggregation'. The 'Food Info' tab is active, showing fields for 'Brand name', 'Commercial name', 'Generic name', 'EuroFIR classification', 'Retention factor classification', and 'Eurocode2 classification'. At the bottom, a 'Data quality evaluation' panel shows three messages: a red message for 'English name: Mandatory field empty', an orange message for 'Retention factor classification: Without a classification, all retention factors are assumed to be 1.', and another orange message for 'If this food is used for recipe calculation, then density can be required.'.

Figure 3.1: Data quality input feedback

change is triggered for warnings.

The example shows a food entity with the violation of one HC and two SCs. The appropriate DQ requirements are: (1) Attribute 'English name' is mandatory and cannot be empty (HC); (2) The attribute 'Retention factor classification' is not mandatory but recommended. It is possible to leave it empty but consequently all retention factors are assumed to have the value one (SC); (3) The attribute 'Density' (not visible in the screenshot since it is on a subtab of the input area) is not mandatory but can be required if the food is used for a recipe calculation (SC). On the upper right of the DQ evaluation panel, a shortcut button provides quick access to the data record in the DQ analysis toolkit.

Concerning the *where*, *how* and *when* of giving feedback introduced in Sect. 2.7, it

Fig. 3.1 shows an example of the DQIF for a food entity. On every screen where data can be edited, there exists a DQ evaluation panel at the bottom. This panel lists all problems of the current data record. Problems are divided into two categories: errors and warnings. Errors correspond to the violation of HCs and are displayed on the panel in red: If any are present, the data cannot be saved. Warnings corresponding to the violation of SCs and presence of low quality indicators are displayed on the panel in orange. No additional background colour

was decided to give all textual feedback in a single location below the input area with the colouring indicating the severity of the problems. Only the evaluation panel is used for textual feedback since the error and warning messages are often rather long and it would be difficult to show an abbreviation close to the appropriate field with such a high density of information and UI components as in FoodCASE. It is also indicated where HCs have been violated by changing the background colour of appropriate fields. This is only done for HCs to keep the colouring within the input area to a minimum. The validation is triggered when the focus is moved from a field and consequently feedback is given immediately on data input for single components in the input area. The input is not checked as the user is entering data in order to reduce the client server communication for performance reasons. The DQ evaluation panel also shows hints for how problems can be repaired if this information has been provided with the appropriate DQ requirement.

For the DQAF, the DQ analysis toolkit first runs a DQ assessment where all relevant data is fetched from the database, all DQ requirements checked, the presentation of the DQ prepared and the results stored in the database for later analysis. Thus, it can be considered as a snapshot of the current DQ in the system. A DQ assessment can be triggered manually by the system administrator or automatically by a timer. Upon completion, a variety of different charts and tables support the user in judging where action is needed.

In the toolkit, every DQ requirement is assigned a type (HC/SC/indicator) which determines how the associated DQ measures will be rendered. The type of the DQ requirement will also be taken into account by the users when specifying the weights (importance) of the DQ requirements. The DQ analysis toolkit provides a total of eleven different views which can be divided into two categories: (1) *DQ views* provide an overview of the DQ by visualising the DQ of either a selected tree node and its direct children or the entire DQ tree; (2) *Problem views* on the other hand provide the possibility to drill down on the DQ issues and identify individual problems.

The toolkit provides flexible means of allowing users to configure their own analysis tree definitions and analysis runs. It is beyond the scope of this section to describe this in detail, but we show an example of one of the many possible visualisations in Fig. 3.2 where the feedback is shown as a tree. For a food entity, there exists a DQ requirement specifying that a value for the attribute energy should be provided since various calculations depend on it. This is defined as a SC since it is not something that the system can enforce to be checked at all points in time. The example shows a custom tree view for aggregated food entities to analyse mandatory components in detail. The DQ tree view shows the entire DQ tree with each node labelled with the mean DQ. Nodes (and edges) can be selected by clicking on them. A table will appear which shows the statistical properties of the selected node. To switch to another view, the buttons at the top or the context menu can be used. The lowest level can be collapsed/uncollapsed using the context menu or keyboard shortcuts. All nodes are rendered as progress bars indicating the DQ of the criteria they represent. The DQ requirement ‘For every food energy must be provided’ is displayed as a subnode of ‘Mandatory Components’ and ‘Aggregated Food Data’. By default, all the DQ values in the DQ analysis toolkit are displayed as percentages. In this example, it can be seen that only 99.6% of the data fulfil this DQ requirement.

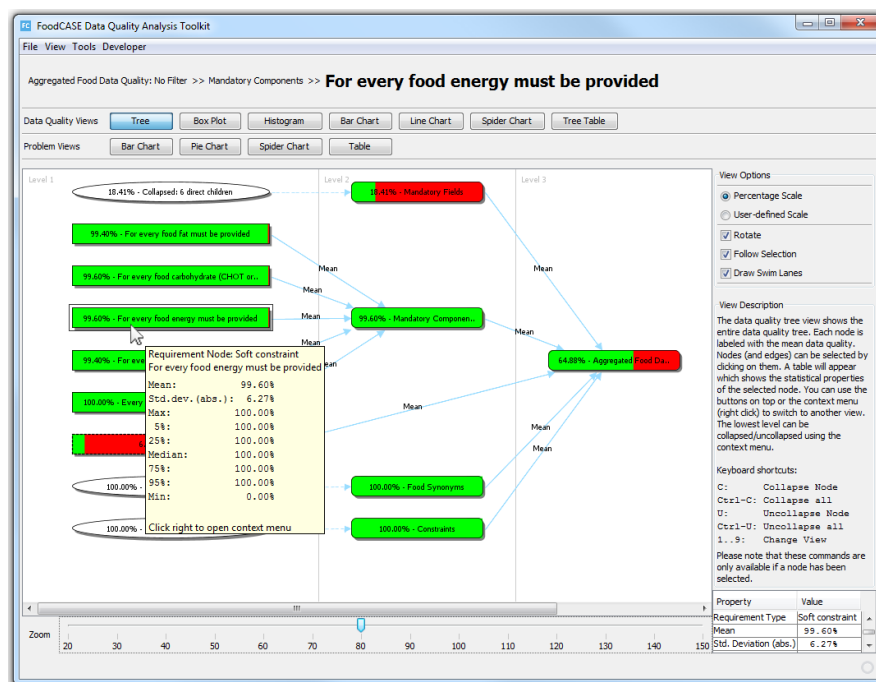


Figure 3.2: Tree visualisation of data quality analysis feedback

FoodCASE entity editor screen and the problem can be fixed. It was chosen to visualise the DQAF with the most common charts such as Box Plot, Histogram, Bar Chart, Line Chart, Spider Chart, Pie Chart, Table and Tree Table since with this choice of visualisations it is possible to analyse the various facets of DQ very well and these types are already familiar to users. The Tree serves as main DQ view since it gives the best overview and navigation options.

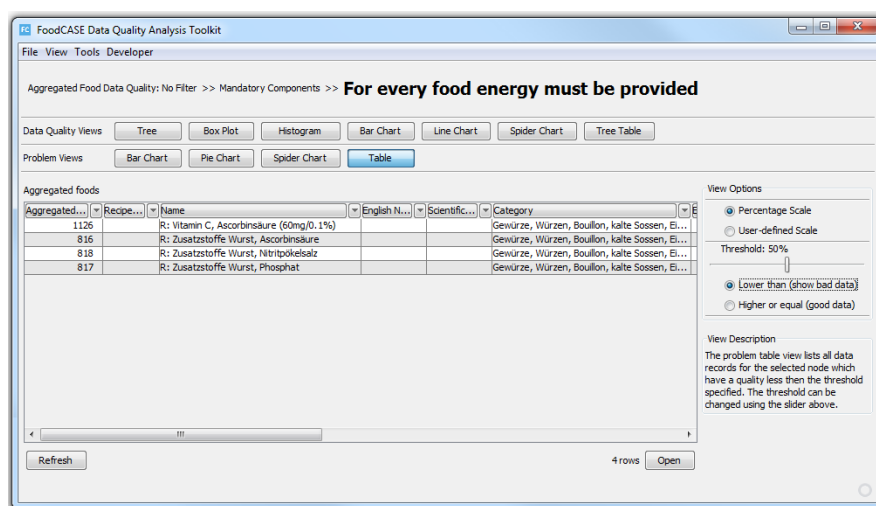


Figure 3.3: Problem table view

following, we will refer to these entities as the DQ entities. Each DQ requirement has to map every data record in the DQ entity to a DQ value between 0.0 (worst) and 1.0 (best). If a DQ requirement is not applicable to a certain data record, null may be returned. For example, if a database table contains data about customers, a

From this overview, it is possible to drill down on the DQ issues and identify individual problems in the data. The user can switch to the problem table view which is depicted in Fig. 3.3. It shows that four aggregated foods have been identified where the component energy is missing. By double-clicking on a row, the data record is opened in the appropriate

According to [146], a DQ requirement is always associated with a DQ object, where a DQ object corresponds to a real world object on which the DQ check should be performed. In the DQ analysis toolkit, the DQ objects are the database entities corresponding to the business concepts of food science data. In the

DQ requirement could be defined as ‘For every person, last name and first name must be provided’. Now if there is a customer for whom only the last name but not the first name is known, the DQ requirement is only partially fulfilled. So the DQ could be defined to be 0.7 (70%) because usually the last name is more important than the first name. If a customer is not a person but a company, the DQ requirement is not applicable, so `null` should be returned.

Once all DQ requirements are gathered, similar DQ requirements can be grouped together. This process can be repeated recursively ending up with a DQ tree. The root node will then be a single number representing the overall DQ. Since the grouping of the DQ requirements may be a matter of individual taste, it is possible to define different DQ analysis trees using the same DQ requirements. Similarly, the importance of the DQ requirement may be controversial. Because of this, the weight of each DQ requirement can be specified in each tree definition independently.

3.1.2 Evaluation

To evaluate how the DQ feedback of our information system is perceived by users and how it can be improved, we presented our information system *FoodCASE* with the integrated DQ analysis toolkit to food science experts and subsequently conducted a survey.

As part of an annual general assembly meeting of the European project TDS-Exposure, we organised a workshop with several food science data experts from different countries in Europe. The participants had a variety of roles including food compiler, head of unit, project coordinator, exposure assessment researcher, food database manager, research assistant, scientific manager/assistant, post doctoral researcher and PhD student. We used half a day to present the information system with the different possibilities for managing scientific food data explained by using screenshots of the various GUIs and showing how to enter data for various use cases. After the presentation of the information system and a subsequent extensive discussion in the plenum, we provided a questionnaire which the participants had to fill out. A total of 16 participants completed the questionnaire, which, although a relatively small number, included food science experts from most of the partners involved in the TDS-Exposure project.

The questionnaire included 49 questions on various topics such as ‘DQ in general’, ‘Who should be able to define DQ requirements’, ‘How should DQ requirements be defined’, ‘How should DQ feedback be given to users’, ‘Distinction between HCs and SCs’ and ‘Personal information and user skills’.

Based on the survey results, we present here only a short list of the most important key findings for DQ feedback in the domain of food science data. For the interested reader, we refer to [192].

1. An information system should clearly offer the functionality to analyse and improve DQ
2. Access to data should be controlled and restricted. At the same time, providing users with the possibility to recall the change history of data is clearly important where changes should be traceable to users.

3. It should be possible for the institution using the software to define DQ requirements, but this should be done by a non-IT person and limited to ‘power users’
4. DQ requirements should be managed in one place and in a unified way
5. It should be possible to rate the relevancy of each DQ requirement because not all requirements have the same importance
6. It is clearly not preferable to provide the possibility to deactivate all DQ requirements
7. A summary such as ‘5 of 9 DQ requirements are satisfied’ would motivate users to increase DQ
8. A panel at the bottom of a GUI is a good solution for the DQ feedback but, for a lot of users, it would be preferable if DQ feedback was displayed close to the corresponding field(s)
9. An accentuation such as highlighting or colouring the corresponding field(s) and the usage of colours to indicate different types of DQ requirements is considered helpful
10. The distinction between HCs and SCs is useful and both types should exist
11. The number of HCs should be as small as possible with most users preferring to have more SCs than HCs

3.1.3 Conclusions

Presser [146] has designed and implemented a general DQ analysis toolkit which allows DQ in information systems to be measured and visualised in a customised way. To evaluate the toolkit, it was integrated into the FoodCASE system for managing food composition data and solicited feedback from experts in the domain.

The toolkit allows users in a particular domain to define their own DQ requirement and configure the metrics for measuring DQ as well as how results of DQ analysis are visualised. In the case of the food science domain, this means that the system is not limited to the quality measures as defined by the USDA and EuroFIR and can easily be extended to meet the requirements of a particular institution or group of users. For example, Presser has identified and defined 156 DQ requirements for the food science domain in total which the toolkit is able to validate and for which DQ feedback can be provided. Furthermore, it is possible to define whether the current DQ should be analysed or a historical snapshot is used. Once it is recognised that there are problems in a certain area of the data, the toolkit allows the user to quantify the extent of the problem and drill down on them.

Based on feedback from experts in the food sciences, we presented a list of key findings indicating the needs for users to get DQ feedback. Based on these findings, we conclude with a summary of the features that the toolkit already supports followed by proposals for future extensions and enhancements which we took into account for our further approaches described in the subsequent sections.

The information system FoodCASE already provided the following: (1) The functionality to analyse and improve DQ; (2) The access to data is controlled and restricted; (3) It is possible to recall the change history and each change of data is traceable to a given user; (4) DQ requirements can be defined by certain power users; (5) It is possible to rate the relevancy of each DQ requirement by defining custom DQ feedback trees with custom weights of certain DQ requirements; (6) The definition of DQ requirements as HCs, SCs or indicators is possible and the user is free to decide how to categorise individual DQ requirements; (7) Feedback distinction between different types of DQ requirements is visualised in a DQ evaluation panel at the bottom of each screen.

Based on the evaluation outcome, we planned to: (1) Define and manage the DQ requirements in one place in a unified way, by changing the DQ requirement definitions to annotations in the entities and performing the validation by the means of BV; (2) Provide a DQ summary in each screen such as ‘5 of 9 DQ requirements are satisfied’ in order to motivate users to increase DQ; (3) Keep the evaluation panel for detailed feedback but add short DQ feedback close to the appropriate fields, if need be by an additional accompanied symbol such as a light bulb with attached tooltip.

Additionally we planned to: (1) Make the distinction between HCs, SCs and indicators transparent and only distinguish between different actions that are triggered by violations; (2) Render the validation and measurement of DQ with different metrics dependent on profiles for diverse applications; (3) Provide the possibility to let the DQ metrics undergo a DQ evolution since DQ evaluation can change over time. (4) Conduct a user study and survey for the adapted information system within the TDS-Exposure project; (4) Gather more detailed feedback about the DQAF for further improvement.

3.2 Object database data quality management

In this section, we present a constraint-based approach to DQ that extends and complements object databases with support for constraint-based DQ management. With this approach, information systems are designed by means of a semantically rich data model, from which constraints are extracted and imported into our separated constraints framework. Users are free to configure additional constraint definitions to those extracted. The constraints are managed within the framework and validated against the database using an ECA paradigm.

While constraint violations are typically prevented by an action that, for example, aborts a transaction or throws an error message, in this approach, such actions may be configured to perform arbitrary functionality. For instance, there may be cases where constraint violations should be handled in a more relaxed way such as simply informing the user of possible DQ deficiencies.

This approach avoids the distribution of constraint definitions, such as basic checks for type safety of an input field in a GUI or complex business rules, across the information system and that DQ checks are performed in different parts of the system. Thus, by using this approach, software engineers are no longer forced to implement DQ controls in the application code which facilitates keeping an overview of all DQ controls and managing them efficiently. Consequently, it can be avoided that identical constraints are defined and checked multiple times in different layers of an information system, or even in different client applications, which results in reduced maintenance effort and

increased performance.

We implemented the approach in a framework that is extensible and provides developers the freedom to define new custom constraint types. Moreover, the fact that constraints are managed in a separate component, independent of the application code, allows for the reuse of constraint definitions across applications. This further reduces redundancies and increases development efficiency.

The work presented in the remainder of this section was previously published in [191]. Therefore, we present here only the main features of the approach and framework along with a discussion and summary. For the interested reader, we refer to [191].

3.2.1 Approach

It is a complex undertaking to ensure DQ in large applications. Our approach, presented in this subsection, supports a software developer in defining, checking and maintaining constraints in information systems in order to enhance and control the DQ. The goal of this approach is to provide the uniform handling of different constraint types. Therefore, we separate the constraint definitions and data validations of a given application domain from its database, business logic and graphical user interface and propose a framework that manages and validates constraints in a clearly represented and well organised manner. Our focus lies on managing constraints for semantically rich data models in order to complement object databases with an extended DQ management facility. This approach distinguishes two types of constraints, based on the constraint types described in the antecedent section *Guidance on data quality feedback*, in Sect. 3.1.1: HCs and SCs. HCs such as integrity constraints are necessary conditions that data has to fulfil to be valid. If a HC is violated, DQ is deficient. For example, a HC could specify that an attribute value for email address has to contain an @ in order to be valid. To offer more flexibility, SCs are constraints that are validated in a more relaxed way compared to HCs. They could also be seen as recommendations since they are not just valid or invalid, but rather increase or decrease DQ. For example, an attribute description could be recommended to be longer than 30 characters to increase DQ. Another example is the age of data, which could indicate that data is outdated. Violations of SCs do not lead to errors, but the user is instead informed of possible DQ deficiencies. For the sake of simplicity, we only distinguished here between HCs and SCs by including the indicators, described in Sect. 3.1.1, in the category of SCs.

As shown in the top part of Fig. 3.4, the design of an application is divided into the design of the structural data model and DQ controls represented by means of constraints. In this approach, DQ controls consist of inherent HCs that are extracted from the application model into constraint definitions and additional explicit constraints may be defined by the developer. Such explicit constraints can either be HCs or SCs. As shown in Fig. 3.4 on the right side, these constraint definitions are taken as input to the constraints framework which manages and validates the defined constraints while the applications are running. As indicated in Fig. 3.4, this approach allows the definition of constraints for various applications that access the same database and it is possible to reuse constraint definitions from one application in another. For example, a course management application running on top of a university database could reuse constraints defined in a study management application running on top of the same database. The constraints framework interacts with the database based on an ECA paradigm, where

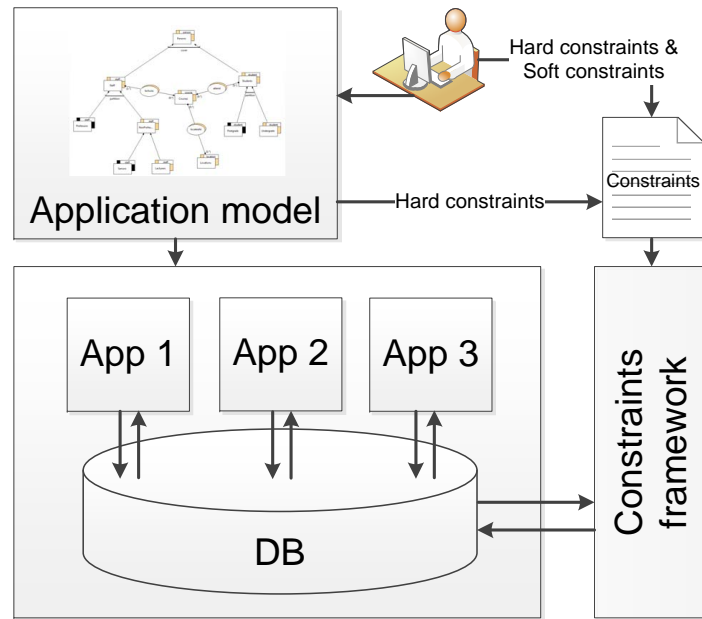


Figure 3.4: Approach overview

constraint checks are triggered by database events. In the case of a violation of a HC, the framework triggers an exception in the application, while a warning is thrown in the case of a SC violation.

3.2.2 Constraints framework

In this subsection, we present the constraints framework which realises this approach and is used to define, validate and manage constraints.

The constraints for an application are defined with a constraint definition language that is based on the DDL proposed in [197]. Our approach supports the definition of simple constraints such as consistency constraints for single attributes, but it is also possible to define constraints that involve more than one object. We distinguish the definition of HCs and SCs with the appropriate key words `hardconstraint` and `softconstraint`.

The constraint definitions can have different structures depending on the constraint type but are always of the form of

```
type name details [events] validator {'message'};
```

The type defines if the constraint is validated as a HC or SC. Each constraint has a name which identifies the kind of constraint and details which provide information about the individual attributes and conditions. The details are followed by a list of database events upon which the validation should take place. The validator defines the validation type. This can be the standard validator or a custom one. The standard validator offers simple attribute validations, while the custom validators usually offer more complex validations. The message is optional, and, if present, it will be used for the exception or text message, respectively. Otherwise, a standard text will be used. Below, we give an excerpt of a constraint definition file with four constraint definitions.

```

hardconstraint length Address.phone = 13 [Creating, Updating] Standard;
softconstraint length Course.description >=30 [Creating, Updating] Standard;
hardconstraint association attends from Student (5:*) to Course (0:*)
    [Committing] Custom;
hardconstraint cover (Staff and Student) Person [Committing] Custom 'Cover
violated';

```

Listing 3.1: Constraint examples

As shown in the example in List. 3.1, the length constraint could be defined as a HC or SC. The first constraint definition defines that the attribute `phone` of the class `Address` must have the length of 13. This will be checked upon the database events *Creating* and *Updating*, thus when an `Address` object is created or updated and the object is to be stored in the database. The validation will be done with the standard validator and, if a violation occurs, an error with the default message for the length constraint will be thrown. The length constraints are instances of two different constraint types with the same constraint name but different validations. The default message is of the form ‘Constraint violated:’ with an output of the constraint details attached as a string. The second line also defines a length constraint specifying that the attribute `description` of the class `Course` should be at least 30 characters long. If a violation occurs only a message will be thrown as a hint that the DQ could be increased by adding more details to the description. On the third and fourth line, more complex constraints are defined: an association with cardinality constraints and a cover constraint specifying that every person must be either staff or student (or both). Both definitions are validated with a custom validator upon the *Committing* event. The cover constraint defines a custom message which is passed to the application upon constraint violation.

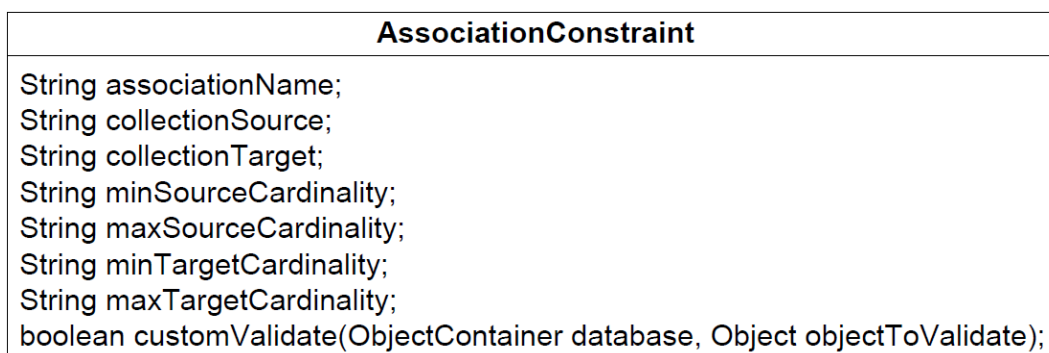


Figure 3.5: Association constraint

The constraint types are represented by Java classes defining particular attributes that hold the detailed information. As an example, the UML diagram of an association constraint is depicted in Fig. 3.5. It shows the attributes that are required to store the detailed information of the association constraint and the `customValidate` method.

Fig. 3.6 gives an overview of the constraints framework and its different components. In the left upper corner, the input file is depicted that contains all constraint definitions. The `ConstraintsGenerator` is responsible for parsing the input file and generating the appropriate constraint definition objects and validation configuration. The constraint

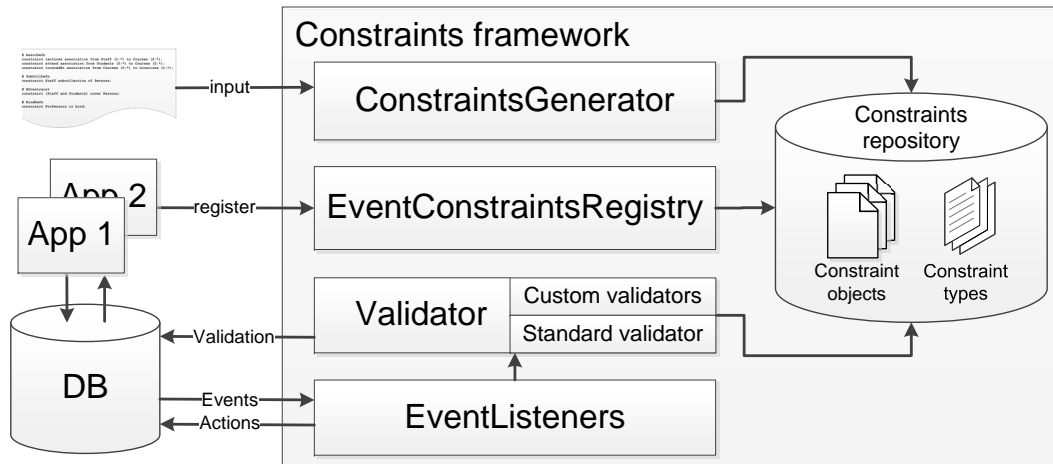


Figure 3.6: Constraints framework

definition objects and the validation configuration are stored in the constraints repository. The `EventConstraintsRegistry` is responsible for linking the constraints from the repository to the appropriate `EventListeners` depending on the database events that were configured with the constraints in the input file. The `EventConstraintsRegistry` also adds the `EventListeners` as listeners to the adequate db4o database events. For example, an association that is defined with the *Creating* event and custom validation would cause the framework to associate the ‘CreatingListener’ and the concrete association constraint to it. Afterwards, the ‘CreatingListener’ would be added as a listener to the *Creating* event. Upon notification, the ‘CreatingListener’ would trigger the custom validation.

The method `EventConstraintRegistry.registerForValidation(DB)` is invoked by applications to register themselves and their database, DB, with the constraints framework.

The framework works with an ECA mechanism. If a database event occurs due to database operations triggered by the applications, the applicable `EventListener` is notified. The `EventListener` holds a list of constraints that were configured to be validated with the according event. Upon a triggered event, the suitable `EventListener` passes the constraint list with the associated objects for validation to the `Validator`. The `Validator` decides which validation method has to be invoked for each constraint based on the configured validation type in the constraint and consequently invokes the validation of the constraint with the objects received from the database event. As a standard validator, we use the validator from the validation framework *OVal*¹. *OVal* is a pragmatic and extensible general purpose validation framework for Java where objects can easily be validated on demand. *OVal* supports, but is not limited to, the validation of *JavaBeans* and can therefore also be considered as a BV framework. To validate objects, it extracts a validation configuration from the constraints repository. As shown in Fig. 3.6, the validations can imply the participating objects received from the db4o event but validations can also perform extensive checks against the database. Depending on the validation result, an exception or a message is thrown, and the transaction may be aborted. HCs will rather raise an exception while SCs will alert with a message. The application then has to handle the validation result of the framework.

¹<http://oval.sourceforge.net/>

3.2.3 Conclusions

In this section, we have presented an approach and framework of a constraint-based approach to DQ management for object databases, in particular proposing a solution for the object database db4o. The framework, however, could also be used for other object databases and we also planned to support the relational model to provide advanced support for data validation based on constraints. The approach therefore is designed to offer a separated framework rather than as an integral part of a database.

The framework provides support for basic constraint checks as offered by *OVal* as well as rich classification constraints. We planned to extend the framework with constraint types that allow the definition of complex DQ controls that cover DQ dimensions such as the ones defined in [187, 182]. Such an extension could also identify for which dimensions constraints can be implemented and for which dimensions it would be difficult or even impossible. For example, we can think of defining constraints for dimensions such as timeliness and completeness but it seems to be difficult to define constraints for other dimensions like reliability. Since the implementation of a comprehensive set of constraints types covering all these DQ dimensions is still work in progress, we do not have a complete formal definition of the constraints language. Our definition language is based on the *OMS DDL* presented in [197] and the complete formal definition would be a goal of future work.

Furthermore, while the implementation supports the definition and checking of HCs and SCs, we planned to generalise this approach so that arbitrary actions can be performed upon a constraint violation.

3.3 Data quality extended Bean Validation

We now describe in this section a first integrated DQMF approach which addresses the problems of distributed constraint definitions and data validation, as introduced in Sect. 1.2.1.

We present a DQMF that allows constraints to be specified and managed in a single place. The constraint model is not coupled to a specific layer or technology. Although a constraint needs to be specified only once, it can be used in different parts of an application, providing the possibility that data is validated only once. The framework also makes it possible to reuse data validation in other applications.

Reviewing the options of a unified constraint management approach explicitly for the Java programming language revealed BV as most promising approach to base on. Therefore, the solution, proposed in this section, has been implemented in the Java programming language environment by building on BV. Although BV has the necessary support for cross-tier validation and already supports a rich set of constraints, it does not support various features essential for more extensive DQ management which are needed to fulfil the requirements of a DQMF. For example, it is not possible to control the time when data is validated to allow temporary inconsistencies. Nor can constraints be updated dynamically which is required, for example, to allow users to specify cardinality constraints over associations at runtime. Further, constraints in BV are considered as all-or-nothing in the sense that data which violates constraints is prohibited. To achieve that data is not prohibited but instead will be associated with a particular quality indicator, we have enhanced the already introduced concept of SCs further to associate

different levels of penalties with data in the case of BV constraint violations in order to calculate the appropriate DQI.

3.3.1 Approach

Our aim for this integrated approach was to provide developers with a unified constraint framework that will avoid inconsistencies and redundancies. Specifically, it should meet the following requirements: (1) The possibility to specify all constraints for a data model using a single constraint model; (2) Constraints need only be defined once within a system to avoid redundancy; (3) Validation within a system is performed based on the defined constraints only; (4) Single validation of any constraint to avoid redundancy in system validation processes; (5) Constraints can be specified and changed dynamically during runtime; (6) The use of constraints to express and measure different levels of DQ; (7) Control over when constraints are validated; (8) Control over the validation processes and what happens when a constraint is violated.

Usually, data validation happens in different tiers and layers as depicted in Fig. 2.3 on page 34, resulting in separate validation mechanisms for each tier and layer. This approach centralises the validation mechanism using constraint annotations within the domain model based on BV as shown in Fig. 3.7.

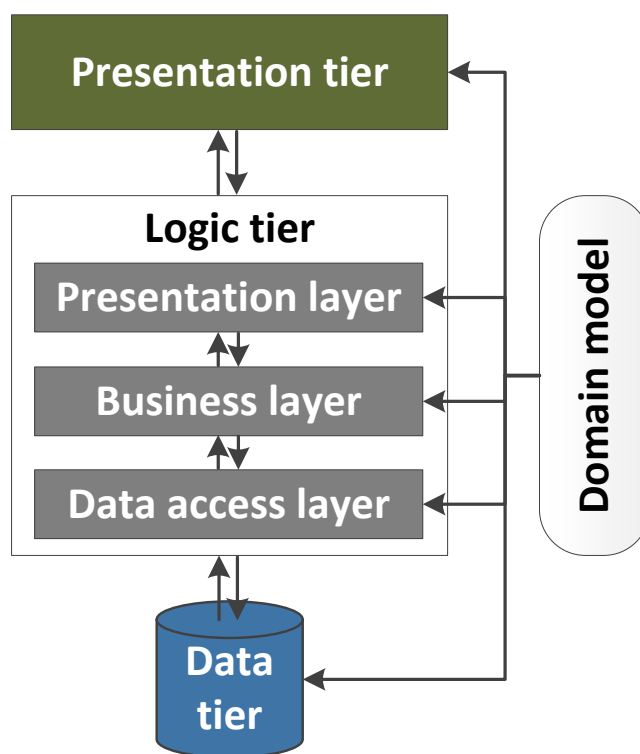


Figure 3.7: Unified constraint model with centralised validation mechanism using Bean Validation

We note that, since it can sometimes be necessary in a multi-tier architecture to re-validate data in a specific layer at a given time, we give developers the freedom to invoke the validation process manually at any time which is also generally supported by BV.

Using BV, requirements 1 to 4 are already met in terms of the architectural requirement. Further, any technology that integrates BV will be able to make use of the

DQMF, presented in this section. However, BV does not satisfy requirements 5 to 8 and it was therefore necessary to extend BV and introduce new concepts to support these requirements. In the remainder of this subsection, we will examine each of these requirements in more detail along with the concepts that were introduced to support them.

Association constraint As an example of a constraint that needs to be specified dynamically, we take the example of a cardinality constraint over an association. Consider the case of a university where the number of students attending a course cannot be greater than the room size. Then, each course instance must have its own limit for the maximum number of students. Although, BV provides a way to constrain the cardinalities of an association at compile time, it does not allow constraints to be configured at runtime per object instance since Java annotations only allow the use of constants. This means that it would not be possible to specify the actual cardinalities at runtime.

We therefore introduced our own association constraint with a special data structure which allows the cardinalities of an association to be specified for individual object instances at runtime which fulfils requirement 5 for cardinality constraints. This results in a more flexible solution which is statically type safe and reusable.

Temporal constraint BV also does not support deferred constraint checking which is necessary in certain situations. For example, consider a system that handles registrations for events. People can register for events within a certain date range and there exists a deadline for enrolment. It might not make sense to arrange some events if there are too few people enrolled. Therefore, for each event, we could specify the minimum number of enrolments as a constraint. However, the validation of this constraint needs to be deferred to the deadline date, so that only the events which have too few people registered and are past the deadline will result in a violation.

We therefore introduced the concept of a temporal constraint which is a constraint coupled with a deadline. The constraint may be violated before the deadline, but has to hold afterwards. In other words, the temporal constraint can only be invalid at a future point in time specified by the deadline, and therefore the validation process should only be invoked at that time.

While developers can create a temporal constraint by combining any form of constraint with a deadline, for ease of use, we provide implementations for several forms of commonly used temporal constraints in the framework.

Time-triggered validation component Closely connected to temporal constraints, we offer a time-triggered validation component *TTVC* to provide a developer with the possibility to define and schedule validation jobs at certain points in time or within a specified interval. A job can be defined as a unit of work and a trigger within the framework is a mechanism by which jobs are scheduled. Many triggers can point to the same job, but a single trigger can only point to one job. The scheduler is the central subcomponent which manages all relevant information to start jobs with individual triggers. A job listener observes the state of a job and is called if certain events occur. The framework provides several predefined components which enable developers to create their own validation jobs that can use the functionality of BV and JPA and it consists of a universal validation job which can access objects in a data store. It also provides the triggers for one-time and repeatable jobs.

TTVC can be used to trigger the validation of objects with any constraints but it is especially useful to validate objects which contain temporal constraints since they have to be valid after a certain deadline. By providing *temporal constraints* and the *TTVC* we fulfil requirement 7.

Flexible data validation Another limitation of BV for DQ management is that it does not allow for cases where data validation requires something more flexible than a strict true/false outcome. Assume, for example, that persons with a name and email address are stored in an information system. The name is a mandatory attribute which can be validated with a ‘not null’ constraint. Persons can exist which do not have an email address and therefore it is not possible to define the email attribute as mandatory. Nevertheless, we might want to associate such instances as having a lower DQ with respect to completeness in cases where the person is known to have an email address but it has not been supplied.

To provide developers with control over the validation process and allow them to associate DQIs with data items, we used the notions of HCs and SCs as already introduced in earlier approaches and adapted their definition for this approach.

The notions presented below are related to the concept of hardgoals and softgoals in the *Requirements Engineering* literature, as described in Sect. 1.4.2.

A HC is a restriction on a Bean instance, on the value of a field or the value of a JavaBean property, e.g. a getter method, that can be validated against a given value with two possible outcomes of the validation. The outcome is `True` if the given value fulfils the HC and `False` otherwise. In summary:

$$\text{validate}(HC, value) = \begin{cases} True, & \text{if } value \text{ fulfils } HC \\ False, & \text{otherwise} \end{cases}$$

This is equivalent to the BV definition of constraints and hence every declared constraint is by default a HC.

In contrast to HCs, a SC can associate a penalty with a violation rather than prohibiting a violation. Formally, a SC is a restriction on a Bean instance, the value of a field or the value of a JavaBean property, e.g. a getter method, that can be validated against a given value. A value can violate a SC resulting in a penalty, in which case the validation process itself is still valid. In summary:

$$\text{validate}(SC, value) = \begin{cases} True, & \text{if } value \text{ fulfils } SC \\ P, & SC \text{ is violated, penalty } P \text{ added} \end{cases}$$

With our new approach on HCs and SCs, requirement 6 is fulfilled. In the subsequent subsection, we will discuss how these constraints can be used to measure DQ. The combined usage of the features presented in this subsection, this approach fulfils requirement 8.

Having described the additional features supported in the framework over BV, an overview of the main components is given in Fig. 3.8.

3.3.2 Managing data quality using constraints

We now list the components available in the framework and show how data can be validated in a flexible way using a single constraint model.

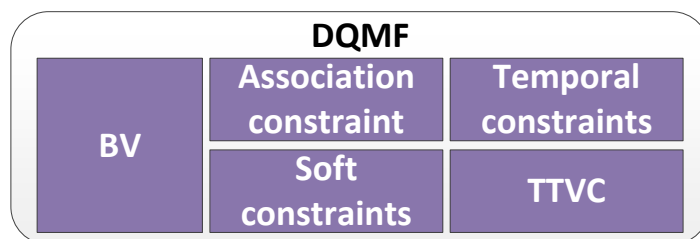


Figure 3.8: Data quality management framework features

The BV standard specifies a set of the most common constraints such as `@NotNull` and `@Max` which can be used as Java annotations in JavaBeans. The concrete implementation for BV that we used, Hibernate Validator 5, additionally provides specific built-in constraints such as `@Email` and `@CreditCardNumber` for checking data value formats. Therefore, the framework supports all constraints from the standard plus the specific ones from Hibernate Validator. A detailed list of these constraints is available in the Hibernate Validator documentation².

In our framework, presented in this section, all constraints are defined using Java annotations and validated using the concept of BV where it provides the following additional constraint support: (1) The annotation `@VariableSize` for the association constraint; (2) The annotation `@Deadline` which can be combined with other constraints to give temporal constraints; (3) A set of primitive temporal data types such as `TemporalBoolean` which can be used for temporal constraints; (4) A set of temporal constraints for the most common existing constraints such as `@AssertTrueOnDeadline`; (5) A time-triggered validation component (*TTVC*); (6) A mechanism to define and validate HCs and SCs by adding appropriate `Group` and `Payload` elements to Bean annotations.

Details of the full set of temporal data types and temporal constraints are described in [151].

The concept of HCs and SCs described in Sect. 3.3.1 enables a developer to validate data and enhance DQ in a more flexible way. Due to their enhanced definition in this approach, SCs can be used to distinguish different severity levels of constraint violations by configuring them into different SC groups such as `Info` and `Warning`. Also, it is possible to add individual penalty scores to SCs such as `LowPenalty`, `MediumPenalty` and `HighPenalty` which makes it possible to set the importance of the fulfilment of a constraint in a more fine grained way.

In contrast to HCs, SCs can be considered as ‘nice to be fulfilled’ rather than ‘need to be fulfilled’. The penalty score can be used to assess how important it is that a SC should be fulfilled. With respect to SCs, data is therefore considered to be of ‘high’ quality if the sum of all penalty scores is low. Therefore, SCs can be considered as indicators for DQ.

3.3.3 Use of framework

Having described the types of constraints and mechanisms supported in the framework, we give examples for all the components to show how a developer would use it and to show that the aims defined in Sect. 3.3.1 can be achieved with the framework using BV.

²<http://hibernate.org/validator/>

An association constraint is specified by declaring a field with the type of our custom data structure `AssociationCollection<T>` within the class where the association is required. Our custom annotation `@VariableSize` represents the association constraint and can be used directly at the corresponding field or an appropriate getter method which accesses the field. The annotation does not take any additional arguments and can only be used on data types conforming to our interface `Association`.

Assume the classes `Course` and `Student` in a university system and the association ‘attended by’ between courses and students as depicted in Fig. 3.9. We add a collection of students to the course class and annotate it with our `@VariableSize` annotation as depicted in List. 3.2.

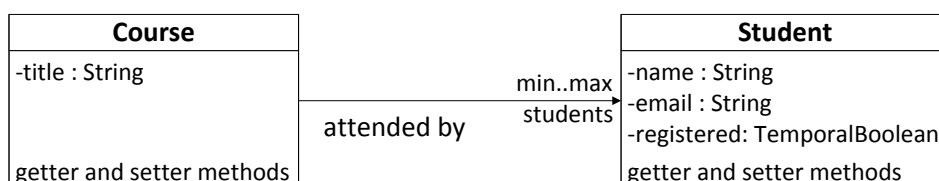


Figure 3.9: UML class diagram for the university domain

```

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

Listing 3.2: Code fragment showing the usage of the `@VariableSize` annotation.

At runtime, an application can create `Course` instances and set the minimum and maximum cardinalities of the association by using for example a constructor as depicted in List. 3.3. To set the cardinalities after the instantiation appropriate setter methods could be used.

```

public Course(String title, int min, int max) {
    this.title = title;
    students.setMinimum(min);
    students.setMaximum(max);
}
  
```

Listing 3.3: Code fragment showing the usage of the association constraint.

We use `@AssertTrueOnDeadline` to explain the use of temporal constraints. In List. 3.4, our temporal constraint annotation `@AssertTrueOnDeadline` is given for the attribute `registered` in class `Student`.

```

@AssertTrueOnDeadline
private TemporalBoolean registered;

public void setRegistered(TemporalBoolean registered) {
    this.registered = registered;
}
  
```

Listing 3.4: Usage of the temporal constraint annotation `@AssertTrueOnDeadline`.

The Boolean constraint and its deadline can be applied at runtime, for example, as depicted in List. 3.5 where the `deadline` parameter is a Java `Date`. `TemporalBoolean` is one of the primitive temporal data types which we provide in the framework.

```
student.setRegistered(new TemporalBoolean(true, deadline));
```

Listing 3.5: Example for setting a deadline.

To use our *TTVC*, either our abstract job classes can be used to create own jobs to make use of the additional BV and JPA functionality, or, our concrete `UniversalJPAValidationJob` can be used.

The following example shows how to set up a job to validate a persistent JPA entity class with a daily trigger based on BV. For the class `Student` depicted in List. 3.6, assume there is a JPA entity (`@Entity`) persisted in a table `STUDENT` that defines a HC `@NotEmpty` on the field `name`.

```
@Entity
@Table(name = "STUDENT")
public class Student {
    ...
    @NotEmpty(message = "name is mandatory")
    private String name;
    ...
}
```

Listing 3.6: Example of a Student class.

We set up an entity validation data builder (`EntityValidationData`) with `Student.class` and use it to build a job with our `JPAValidationJobBuilder` as depicted in List. 3.7. We create a job `JobDetail` with the job-id `job-01`, entity id `id-01`, a job description ‘Validate Student’ and the validation job class `UniversalJPAValidationJob.class`.

```
// Set up the entity validation data builder with the entity
EntityValidationData.Builder entityIdbuilder = new
    EntityValidationData.Builder(Student.class);

// Build a job based on the entity class
JobDetail universalJob = new JPAValidationJobBuilder(
    "job-01", entityIdbuilder
    .withEntityId("id-01")
    .withJobDescription("Validate Student")
    .withValidationJobClass(UniversalJPAValidationJob.class)
    .build());
```

Listing 3.7: Example of building a job with `JPAValidationJobBuilder`.

A trigger is then created with our `DailyTriggerBuilder` and the job scheduled with the trigger by using our `JPAValidationScheduler` as depicted in List. 3.8 where `db_name` defines the data store.

```

// Building trigger that fires daily at 18:30 o'clock
Trigger dailyTrigger = new DailyTriggerBuilder("t1", 18, 30)
    .withTriggerGroup("ttvc-tests-triggers").build();

// Set the data store
JPValidationScheduler jpaValidationScheduler = new
    JPValidationScheduler("db_name");

// Schedule the job with the trigger
Scheduler scheduler = jpaValidationScheduler.getScheduler();
scheduler.start();
scheduler.scheduleJob(universalJob, dailyTrigger);

```

Listing 3.8: Example of scheduling a job with a trigger.

As explained in Sect. 3.3.1, every constraint declared without further configuration using BV is automatically a HC. To implement SCs using BV, we use the `Group` and `Payload` elements of the BV standard.

An example showing the use of HCs and SCs is given in List. 3.9 where a class, e.g. `Student`, has a HC (without group and payload element) for the field `name` and a SC (with group `Info` and payload `LowPenalty`) for the field `email`. Both constraints assert that the appropriate field should not be null, but in the case of the SC a null value would be allowed but given a penalty.

```

@NotNull(message = "name is mandatory")
private String name;

@NotNull(message = "not mandatory",
    payload = LowPenalty.class, groups = Info.class)
private String email;

```

Listing 3.9: Definition of a HC and a SC.

3.3.4 Implementation

We now briefly present the concrete technologies used for the implementation of our framework, presented in this section, and details of selected components. A detailed description of the implementation can be found in [151].

For BV, we used Hibernate Validator 5 which is the reference implementation for BV 1.1. It is the only implementation so far that supports BV 1.1 (instead of 1.0) and it provides features which we use for the implementation of temporal constraints.

We also used the reference implementation EclipseLink for JPA 2.1 to provide the complete list of features of the DQMF in combination with a persistence tier. It is needed to persist the data structure of our association constraint and the specified deadline for objects that contain a temporal constraint.

Our constraints are implemented as Java annotations where classes, methods and variables may be annotated. For each annotation such as `@VariableSize`, we implemented an appropriate validator and associated it to the an-

notation as depicted in List. 3.10. The listing shows that the `@VariableSize` annotation is associated with our `VariableSizeValidator`.

```
@Constraint(validatedBy = { VariableSizeValidator.class })
public @interface VariableSize {
    ...
}
```

Listing 3.10: Implementation of the `@VariableSize` annotation.

This approach solves the setting of cardinalities for an association per object instance with the help of our custom data structure `AssociationCollection<T>`. Together with our `Association` interface implementation, the association collection can control the cardinalities of an association with the appropriate getter and setter methods as depicted in Fig. 3.10.

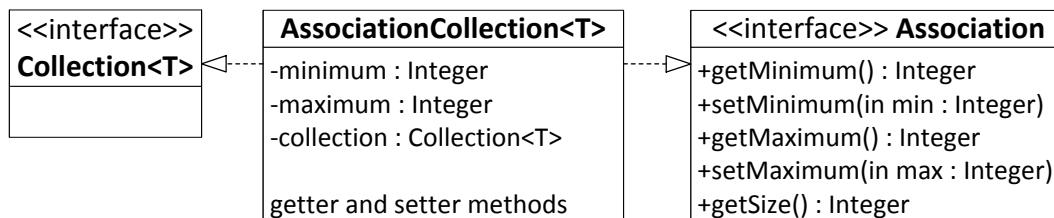


Figure 3.10: Association collection

For the temporal constraints, we made use of constraint composition which is a feature of BV and allows a developer to create a new constraint based on existing ones. We combined our deadline constraint (`@Deadline`) with several existing constraints to get a set of temporal constraints. For example, we combined `@Deadline` with the built-in `@AssertTrue` which resulted in the new annotation `@AssertTrueOnDeadline` for the combined constraint.

The `@Deadline` annotation does not take any additional arguments and is associated with our `DeadlineValidator`. Since a usage of `@Deadline` on its own does not make sense, the scope of the annotation usage is restricted in that the annotation can only be used in composition with other annotations. Moreover, the `@Deadline` annotation and any composition including it can only be used on our declared type `Temporal`. The `Temporal` interface provides the methods for getting and setting a deadline as a Java date.

Our implementation of the *TTVC* is based on BV 1.1, JPA 2.1 and the open source job scheduling library Quartz³. Our jobs extend the Quartz Job, while our `JPAValidationScheduler` consists of a Quartz scheduler.

We provide the abstract jobs `AbstractValidationJob` for validating objects using BV and `AbstractJPAValidationJob` to access and validate objects in a data store using JPA. Those abstract classes can be used by developers to create their own Quartz job details and make usage of our additional BV and JPA functionality. Our `UniversalJPAValidationJob` extends the `AbstractJPAValidationJob` and implements the concrete method `execute` which gets all necessary objects from the database. A simplified UML diagram of the job classes is depicted in Fig. 3.11.

³<http://quartz-scheduler.org>

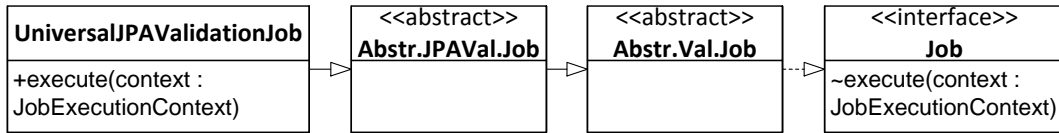
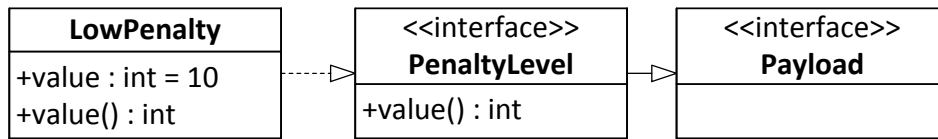


Figure 3.11: Time-triggered validation classes

Triggers are created by our builder classes `UnrepeatableTriggerBuilder` and `DailyTriggerBuilder` which use the common builder pattern.

For the grouping of constraints, we created the group `SoftConstraint` and its two subsets `Info` and `Warning`. Using this group design, we are able to validate every subgroup of the `SoftConstraint` group which we consider as SCs. If no group is specified, the BV implementation considers the constraint to be part of the `Default` group and it is treated as a HC.

To realise the penalty scores, we created an interface `PenaltyLevel` that extends the standard `Payload` interface and contains a `value` method to return the corresponding penalty score. Additionally, we implemented three penalty classes such as `LowPenalty` (depicted in Fig. 3.12) which implement our `PenaltyLevel` interface. Therefore, we are able to get the individual penalty scores of each penalty level to make a quantitative analysis about the fulfilment of the SCs.

Figure 3.12: Penalty level `LowPenalty`

The framework is also extensible since it is possible to extend the data structure for temporal constraints, to create customised temporal constraints and to add additional groups and penalty levels for SCs.

We experienced during our research that the BV integration process for the presentation and data access layer (e.g. how to enable BV) does not cause much overhead. Moreover, BV is a standardised concept which runs in any Java *SE* and *EE* environment which also supports our decision for BV 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.

3.3.5 Conclusions

In this section, we have presented a framework for Java developers to support the management of DQ in information systems. The framework builds on BV which supports a classic constraint model, but was then extended with additional forms of constraints and functionality to provide more control over when and how validation is done and what happens when constraints are violated. In particular, we introduced a runtime configuration of cardinality constraints over associations, temporal constraints that allow validation to be deferred until a deadline and SCs that can be used to associate different levels of DQ with values.

The resulting framework therefore achieves the main goals that we defined in this section. Nevertheless, there remained several extensions which we addressed with further approaches or which still have to be addresses in the future, respectively. One major direction of research is to investigate in more detail what measures of constraints fulfilment would be most appropriate and how information on overall and detailed DQ based on these should be presented.

We also planned to integrate this approach into our FoodCASE information system. This enables us to evaluate the approach and framework in terms of DQ requirements in an existing scientific information system, as well as application developer support. We note that this involves integrating the framework into a typical client-server environment.

3.4 Bean Validation OCL extension

The approach presented in this section is related to the approach presented in the antecedent Sect. 3.3, *Data quality extended Bean Validation*. Not only since we aimed to address the same issues and fulfil the same requirements for unified constraint management and indicating DQ with it, but also since it is as well based on BV. In the *Data quality extended Bean Validation* approach, we experienced the difficulty of defining (1) complex constraints with standard BV constraint definitions and (2) arbitrary constraints at runtime with BV.

BV provides constraint definitions with predefined annotations for basic constraints. For complex constraints it is necessary to define customised annotations and appropriate Java validators. Our decision to adopt a constraint-based approach to DQ requires that also complex constraints can be defined in a way that is convenient as possible. Therefore, we saw the need to explore how BV could be extended to define constraints with a powerful constraint definition language such as OCL for the expression of complex rules, which adds additional convenience for software developers.

Besides the need for a more flexible concept of constraints to support our notion of SCs, for maximum flexibility, a DQMF should provide the possibility to add, remove and update constraints dynamically at runtime. With standard BV it is neither possible to associate data with a particular quality indicator nor to manage constraints during runtime.

Since BV does not support all of the features essential for more extensive DQ management outlined above, we extended the BV approach to allow constraints to be specified using OCL also at runtime and introduced our concept of SCs which associate penalties with data in the case of violation in order to calculate an appropriate DQ indicator.

The framework which we present, as an implementation of this approach, uses the object validation framework *OVal* which we already used for the approach presented in Sect. 3.2, *Object database data quality management*, and was introduced in Sect. 3.2.2. It provides programming by contract, a limited constraint definition during runtime and a scripting extension API.

The solution we present in this section, is designed to support developers working in the Java programming language environment and using Java Data Objects (JDO)⁴ for persistence based on relational or object technologies.

⁴<http://www.oracle.com/technetwork/java/index-jsp-135919.html>

3.4.1 Approach

Our aim for this integrated approach was to provide developers with a unified constraint framework with which it is possible to avoid inconsistencies and redundancies. The framework must support the definition and validation of complex rules with OCL not limited to compile time and supply the basis for measuring and calculating DQ. Specifically, it should meet the six (1-6) requirements presented in Sect. 3.3.1 and additionally (7) The ability to define constraints in OCL.

We note that the statements which base on BV in the approach *Data quality extended Bean Validation*, presented in Sect. 3.3.1, are also valid for this approach such as that this approach centralises the validation mechanism using constraint annotations within the domain model based on BV as shown in Fig. 3.7.

Further, using BV, requirements 1 to 4 are already met in terms of the architectural requirement and any technology that integrates BV will be able to make use of the DQMF, presented in this section. However, BV does not satisfy requirements 5 to 7 and it was therefore necessary to extend BV and introduce new concepts to support these requirements. In the remainder of this section, we will examine each of these requirements in more detail along with the concepts that were introduced to support them.

To enable all our requirements to be met, we decided to base the framework on *OVal* which provides more suitable features for this approach than the reference implementation of BV 1.1 (JSR 349), Hibernate Validator. Although it does not implement all requirements of the BV standard, it is possible to configure *OVal* to interpret BV (JSR 303)⁵ and JavaBean-JPA annotations. Moreover, on the roadmap for *OVal* is support for inherited constraints, the complete implementation of the BV standard and a tighter integration with web application frameworks such as JSF, Struts, etc. In *OVal* three different configurers exist which enable developers to define constraints with annotations, in an XML file or as Plain Old Java Objects (POJOs).

It would have been possible to use Hibernate Validator to express and measure different levels of DQ (requirement 6), for example by using the *groups* and *payload* attributes (as in the approach *Data quality extended Bean Validation*, presented in Sect. 3.3.1), but there is no support for external constraint languages such as OCL (besides the XML descriptors) although it should be possible to implement an OCL parser based on the programmatic API (requirement 7). Further, Hibernate Validator does not provide a mechanism for specifying constraints during runtime (requirement 5).

Our extensions, which we explain in more detail below, are depicted in the upper part of Fig. 3.13 which provides an overview of the main framework components. The extensions for requirements 5 and 7 are depicted in the component ‘OCL- & runtime support for JDO databases’ while the extensions for requirement 6 is depicted in the component ‘Data quality indicator extension’.

OCL extension Extending BV with the ability to define constraints with OCL could be done using either an implementation of the OCL OMG standard such as Eclipse OCL or Dresden OCL as an interpreter for OCL expressions in the constraint definitions or by writing our own OCL interpreter and parser. We opted to use an existing

⁵<https://jcp.org/en/jsr/detail?id=303>

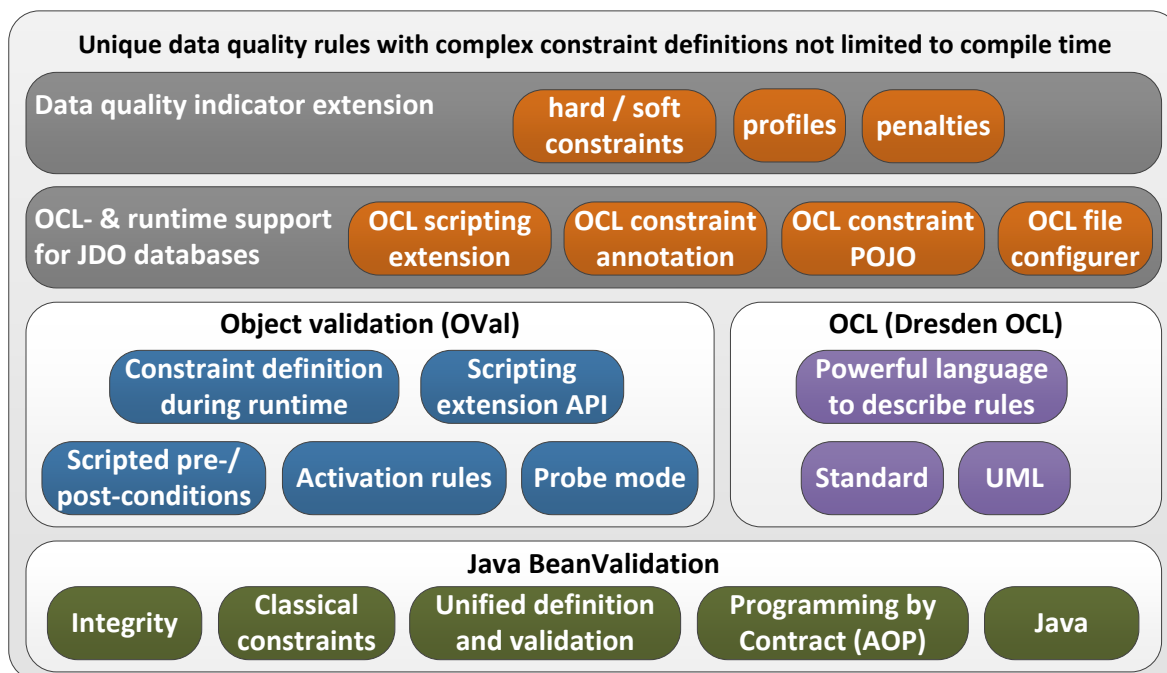


Figure 3.13: Framework components

implementation to reduce the implementation and maintenance effort, but wanted to use one which is not dependent on a particular meta model such as UML or *Ecore* of the Eclipse Modeling Framework⁶. We therefore chose to extend BV using Dresden OCL⁷ which provides a set of tools to parse and evaluate OCL constraints on various models such as UML and *Ecore* but additionally on a simplified model in Java. While *OVal* takes care of the validation process, Dresden OCL serves as a flexible and extensible standalone interpreter for OCL expressions, enabling an application programmer to define constraints in OCL based only on the simplified model in Java rather than requiring them to use a meta model.

Although *OVal* already supports many expression languages, there is no built-in support for OCL. Fortunately, it is possible to implement support for new expression languages and register them for assertions as well as pre- and post-conditions. The framework, presented in this section, therefore provides its own constraint annotation (`@OclConstraint`), an OCL constraint check extension to the *OVal* annotation check, an *OVal* expression language extension implementation of OCL, an additional *OVal* configurer implementation for OCL and an extension to the JDO persistence manager (constraint manager) and persistence manager factory (constraint manager factory). Our constraint manager includes an extension to the *OVal* validator to support our OCL constraint checks.

In summary, we provide an extension to BV based on *OVal* and Dresden OCL with which it is possible to define constraints in OCL either by using our annotations, an OCL configuration file, constraints as POJOs or any combination of these and thus fulfils requirement 7.

⁶<http://eclipse.org/modeling/emf/>

⁷<https://github.com/dresden-ocl>

Dynamic constraint definition Whether method is used to define constraints in *OVal*, the framework always generates constraint POJOs behind the scenes. It is also possible to directly define constraints as POJOs and add, remove or update them during runtime. Our OCL constraint check extension to the *OVal* annotation check, as described above, can be used as such a POJO. By using the *OVal* POJO configurer, our OCL constraints can therefore also be specified and changed dynamically during runtime which fulfils requirement 5.

DQ indicators In Sect. 3.3.1 on page 72 we have presented the notions of HCs and SCs adapted to the *Data quality extended Bean Validation* approach. For the approach presented in this section we base on the same definitions.

We will discuss how these constraints can be used to measure DQ in the subsequent subsection.

For our own constraint annotation `@OclConstraint`, it is possible to define profiles and a severity (given as an integer) together with an OCL expression as designed in the *OVal* framework. An application developer can make use of these profiles to distinguish not only between the two major levels of DQ, HCs and SCs, but also between additional levels of DQ within the SCs. The severity can be used as the penalty for SCs which is why they can be used as synonyms. Assigning constraints a profile and a penalty allows fine-grained control over constraint evaluation and provides the basis for measuring different levels of DQ which fulfils requirement 6.

Our framework is based on the JDO-API and therefore completely independent of the underlying database apart from the fact that it must support JDO. We support the coupling modes *immediate* and *deferred* but not yet *separate* of the ECA paradigm.

3.4.2 Managing data quality using constraints

We now describe the components available in our framework and show how data can be validated in a flexible way using a single constraint model.

The BV standard specifies a set of the most common constraints such as `@NotNull` and `@Max` which can be used as Java annotations. The concrete implementation for BV that we use, *OVal*, additionally provides specific built-in constraints such as `@Email` for checking data value formats. Therefore, the framework supports all constraints from the standard plus the specific ones from *OVal*. A detailed list of these constraints is available in the *OVal* documentation⁸.

In the framework, all constraints can be defined using Java annotations, OCL configuration files, XML configuration files or POJOs and are validated using the concept of BV with our framework providing the following additional constraint support: (1) The annotations `@Unique` and `@PrimaryKey` specify unique and primary key constraints, respectively. The primary key constraint can also be defined as a composite for multiple attributes and ensures both not null and unique, while unique only ensures that a value or a combination of values is unique within all instances of a class. The appropriate validation checks the database and all concurrent transactions for competing entries; (2) The annotation `@ForeignKey` specifies a foreign key constraint which ensures referential integrity. All of the possibly referenced objects are checked to ensure that referenced

⁸<http://oval.sourceforge.net/api/net/sf/oval/constraint/package-summary.html>

values exist. Only entries that are already persistent or will be persisted within the same transaction are checked; (3) The annotation `@OclConstraint` supports the definition of an arbitrary OCL expression for the validation of single or multiple attributes of an entity where profiles and penalties can be defined to express different levels of DQ as explained in Sect. 3.4.1.

The extensions `@PrimaryKey` and `@ForeignKey` are especially helpful if object databases are used since they often lack support for such constraints whereas relational databases usually already provide them.

The concept of HCs and SCs described in Sect. 3.3.1 enables a developer to validate data and enhance DQ in a more flexible way. SCs can be used to distinguish different severity levels of constraint violations by configuring them into different SC groups such as `Info` and `Warning`. Also, it is possible to add individual penalty scores to SCs such as `Low`, `Medium` and `High` which makes it possible to set the importance of the fulfilment of a constraint in a more fine-grained way. Please note that the framework is not limited to a number of constants for strings such as `Info` or constants for integers such as `Low`. Any string respectively integer can be used to define profiles respectively penalties.

The interpretations of the SCs penalty scores and their sums are identical to the ones presented in Sect. 3.3.2.

3.4.3 Use of framework

Having described the types of constraints and mechanisms supported in the framework, we now give examples to show how a developer would use it and that the requirements presented in Sect. 3.4.1 can be achieved with the framework, presented in this section, using BV. We do not describe the examples in very detail but try to give examples which are as self-explanatory as possible.

As mentioned in Sect. 3.4.1, we make use of the simplified model provided by Dresden OCL which can be configured by a Java class listing all classes for which OCL constraints are defined. List. 3.11 shows a model class `ModelProviderClass` which contains the entities `Student`, `Professor`, `Course` and `Department`.

```
public class ModelProviderClass {
    protected Student student;
    protected Professor professor;
    protected Course course;
    protected Department department;
}
```

Listing 3.11: Example of `ModelProviderClass`.

List. 3.12 and 3.13 show the usage of our constraint annotations `@PrimaryKey`, `@ForeignKey` and `@Unique`. The examples show the constraints specified for certain profile constants but it would also be possible to add additional penalties.

```

@PrimaryKey(keys="profID", profiles=Profile.PRIMARY)
public class Professor {
    private int profID;
    @ForeignKey(clazz=Department.class,
                attr = "deptID", profiles=Profile.FOREIGN)
    private int deptID;
    ...
}

```

Listing 3.12: Usage of @PrimaryKey and @ForeignKey annotations.

```

@Unique(attr="title", profiles=Profile.UNIQUE)
public class Course {
    private String title;
    ...
}

```

Listing 3.13: Usage of the @Unique annotation.

List. 3.14 shows an @OclConstraint annotation which allows constraints to be defined by means of the OCL language and to assign a severity level and profiles. To allow multiple @OclConstraint annotations to be attached to the same attribute, method or class, we provide the annotation @OclConstraints.

```

@OclConstraints(
    @OclConstraint(expr="context Student inv: self.age>18",
                  profiles=Profile.INFO, severity=Penalty.LOW)
    @OclConstraint(expr="...")
)
public class Student {
    private int age;
    ...
}

```

Listing 3.14: Usage of @OclConstraint and @OclConstraints annotations.

An example of specifying a constraint as a POJO with our OclConstraintCheck is shown in List. 3.15 for the attribute age of Student.

```

OclConstraintCheck check = new OclConstraintCheck();
check.setExpr("context Student inv: self.age>18");
check.setProfiles(Profile.INFO);
check.setSeverity(Penalty.LOW);
ConstraintManagerFactory.addChecks(
    ReflectionUtils.getField(Student.class, "age"), check);

```

Listing 3.15: Example of OclConstraintCheck as POJO.

Besides specifying constraints with annotations, it is possible to use a configuration file which defines constraints in plain OCL. An example of such a constraints.oc1 file is shown in List. 3.16.

```
package my::company::package::example
  context Person
  inv: self.age>18
  inv: self.name.size()>6
endpackage
```

Listing 3.16: Example of an OCL constraint definition file.

List. 3.17 shows the usage of an OCL constraints definition file (`constraints.ocl`). First a JDO PersistenceManagerFactory (PMF) is initialised. For each OCL file, an OCLConfig with a certain profile and penalty is created and used together with the ModelProviderClass to create a ValidatorFactory (VF). Next our ConstraintManagerFactory (CMF) is initialised with the PMF and the VF so that finally we can get our ConstraintManager (CM) from the CMF for a certain coupling mode.

```
// Initialise the JDO PMF with the database properties
PersistenceManagerFactory pmf =
    JDOHelper.getPersistenceManagerFactory(props);

// Initialise the VF with the \gls{ocl} model and the \gls{ocl} file
File modelProviderClass = new File(
    resources, "path/ModelProviderClass.class");
File oclConfig = new File(
    resources, "constraints/constraints.ocl");
ArrayList<OCLConfig> oclConfigs = new ArrayList<OCLConfig>();
oclConfigs.add(
    new OCLConfig(oclConfig, Profile.WARNING, Penalty.HIGH));
ValidatorFactory vf =
    new ValidatorFactory(modelProviderClass, oclConfigs);

// Finally initialise the CM
ConstraintManagerFactory.initialize(pmf, vf);
ConstraintManager cm =
    ConstraintManagerFactory.getConstraintManager(
        CouplingMode.DEFERRED);
```

Listing 3.17: Usage of OCL constraint definition files.

List. 3.18 shows an example of how to validate a `Student` for the constraints which are configured with the profile `INFO` with our CM. The violations `ConstraintViolation` contain the configured penalties and therefore can be used together with the profile information to calculate the DQ.

```

try{
  // begin transaction: write
  cm.begin();
  cm.disableAllProfiles();
  cm.enableProfile(Profile.INFO);
  cm.makePersistent(new Student("David", "Smith"));
  cm.commit();
} catch (ConstraintsViolatedException e) {
  // get constraint violations
  for(ConstraintViolation violation :
      e.getConstraintViolations()) {
    LOG.info(violation.getMessage());
    int severity = constraintViolation.getSeverity();
    ...
  }
  // Calculate data quality based on the profile
  // and the severities
}

```

Listing 3.18: Validation for a certain profile.

3.4.4 Implementation & evaluation

Fig. 3.14 presents an overview of our implementation where our extensions are marked with a blue background colour. We provide the annotations `@PrimaryKey`, `@ForeignKey`, `@Unique` and `@OclConstraint` as described in Sect. 3.4.2 which are implemented as Java `@interface`. The annotations are connected with our appropriate check classes by the `OVal @Constraint` annotation as shown on the right part of Fig. 3.14. For example, the `@OclConstraint` is connected with `@Constraint(checkWith = OclConstraintCheck.class)`.

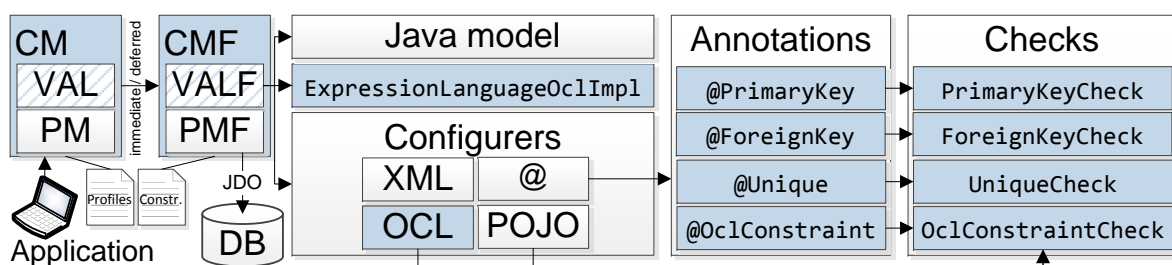


Figure 3.14: Implementation overview

Annotations are convenient for developers and the actual constraint evaluation is done in the check classes. On the left of Fig. 3.14, our `ConstraintManager` (CM) and `ConstraintManagerFactory` (CMF) are shown. A CM can be instantiated by the CMF with one of two coupling modes *immediate* or *deferred* of the ECA paradigm and gets a JDO `PersistentManager` (PM) and an extended `OVal Validator` (VAL) from the factory to handle our `OclConstraintCheck`. The CMF has a JDO `PersistentManagerFactory` (PMF) and, with it, the connection to the database. Additionally, it has a `ValidatorFactory` (VALF) extended from `OVal` which registers our

`ExpressionLanguageOclImpl` for each *OVal* validator. It is also able to be initialised with our `OCLConfig` which is not depicted in the figure since it only encapsulates an OCL file name, profiles and the severity as a helper class. By configuring different CMFs with different profiles, CMs can be produced with different profiles. The VALF is initialised with the Dresden OCL Java model `ModelProviderClass` and loads all *OVal* configurers such as `AnnotationsConfigurer` (@), `XMLConfigurer` (XML) and `POJOConfigurer` (POJO) as well as our `OCLConfigurer` (OCL) as shown in the middle of Fig. 3.14.

We already presented in Sect. 3.4.3 how the components are initialised and the validation is triggered. After the initialisation, the application only has to handle the validation with the CM(s). If multiple CMs exist, they are produced with the same constraint definitions from the CMF but, afterwards, are independent of each other. They maintain their own list of constraints (by a deep copy) for their particular profile(s). Thus, if constraints are added, removed or changed during runtime and/or profiles are enabled or disabled for one CM, it does not influence the other CMs. Besides the coupling modes, our framework uses the JDO Life Cycle Listeners which support the ECA paradigm. If a validation is triggered by an event such as an update, the managed objects of a CM which are new or in the life cycle persistent dirty are checked for constraint violations. For each of its configurers, VAL invokes the validation of all constraints which belong to its profile(s). The validation collects all violations and will eventually throw a `ConstraintsViolatedException` or, in case of deferred evaluation, commit the transaction. If an exception is thrown, it is the obligation of the programmer to decide whether to force a commit or rollback.

To validate the approach, presented in this section, we developed a simplified university information system with information about departments, professors, students and courses. We implemented it as a web application using JSF and the object database *ZooDB* which is implemented fully in Java and based on the JDO 3.0 standard. *ZooDB* is licensed under GPLv3 (GNU Public License) and available on github⁹. A detailed description of the implementation and evaluation is available as a report¹⁰ and the source code is available as the *constraints* branch of *ZooDB*.

3.4.5 Conclusions

In this section, we have presented a framework for Java developers that supports the management of DQ in information systems with JDO-aware databases. The framework builds on BV which supports a classic constraint model, but was extended with additional functionality to manage DQ. In particular, we introduced OCL as a constraint definition language for BV applicable in all definition scenarios such as annotations, configuration files and POJOs, a runtime configuration for constraints, and the concepts of SCs that can be used to associate different levels of DQ with values.

The resulting framework therefore achieves the main goals that we defined in this section. Nevertheless, there still remained several extensions which we addressed with further approaches.

We also planned to investigate more formally what types and aspects of DQ are or

⁹<https://github.com/tzaeschke/zoodb>

¹⁰<http://dx.doi.org/10.3929/ethz-a-010286836>

could be supported with this or other approaches. Moreover, to provide a thorough evaluation, we planned likewise to the approach presented in Sect. 3.3 (Data quality extended Bean Validation) to integrate this framework into our FoodCASE information system.

3.5 Data quality monitoring framework

For the investigation of how constraint models could be extended to support DQ management, the approach presented in this section provided a further missing element in exploring different approaches on DQ and appropriate DQMFs. Assume an information system with an already working constraint management facility (unified or not) where software developers and users are confident with its data validation. Nonetheless, there are no, or very limited, mechanisms to provide feedback on the DQ in the system. In this situation, a separated DQMF which analyses the constraint checking violations and provides ways of visualising and analysing the quality of the data would be very helpful.

Therefore, we investigated a general framework for analysing and giving feedback on DQ in information systems based on an independent data validation, as an extension to classic constraint models. Data validation information from a main application, as origin of constraint violations, is received and further processed in the DQ monitoring framework which provides appropriate DQ analysis and visualisation tools.

The processing of data validation information includes the association of individual constraints with fine-grained violation penalties and also DQ dimensions. In addition, it includes the measurement and calculation of DQ ratings at different levels considering also constraint inheritance. With this approach, we extended the concept of constraints to a concept of general DQIs for giving feedback on DQ.

Independent of the application and data model, the DQ monitoring framework with its data analysis tool, allows users to inspect different dimensions of DQ for single and multiple object instances, single and multiple object classes and the overall DQ. Feedback on DQ can be viewed in a variety of textual and visual forms.

Our goal was to design a general framework that could be used to analyse and provide feedback on DQ for any information system. Given that DQ requirements are highly dependent on the context in which the data is used in terms of the domain, the application and the user, it is important that the computation of DQ is highly flexible and easily configurable. In particular, the set of DQ dimensions considered should not be fixed. We therefore simply identify a chosen DQ dimension by name, and the defined constraints are mapped to one or more of those DQ dimensions.

To evaluate the approach, we conducted a user study for our DQ analysis tool with participants from both the research community and software engineering industry. The participants were asked to solve various data analysis tasks using our framework and then completed a questionnaire to give their feedback on several aspects of the framework.

3.5.1 Approach

In this approach, constraints are mapped to, potentially multiple, DQ dimensions. Individual constraint-dimension mappings are associated with violation penalties which

correspond to the subjective importance of the constraints and are used to compute and provide feedback on DQ. Since these mappings can easily be configured for different contexts, independent of the application and data model, the approach supports individual and subjective DQ analysis.

DQ can be computed at various levels of detail to provide a user with feedback on all or specified parts of data in an information system. Each constraint violation contributes its assigned penalty value to the rating score for the appropriate feedback level, enabling users to be given feedback based on a continuous fine-grained range from very good DQ to very poor DQ instead of the classical constraint checking values of true or false. The feedback levels include single and multiple object instances, single and multiple object classes as well as overall DQ. Moreover, in the case of multiple entities or classes, we offer both separated and aggregated analysis for each of the levels so the user can decide whether they want DQ feedback on each of the individual entities or classes separately, or they want the feedback aggregated into a single combined information item. Note that the computation of DQ must take into account constraint inheritance.

Our data analysis tool can provide DQ feedback for any information system using as input the static information about (1) the object hierarchy, (2) object identifications, and (3) the configuration for the mapping of constraints, dimensions and penalties, as well as (4) the dynamic information about the actual constraint violations from the running information system. The tool offers various options for providing textual and visual feedback.

It is important to provide such different feedback options, not only since individuals have different preferences, but since they enable the view on data to be adapted to answer specific questions during DQ analysis. For instance, consider the two potential questions: Which instance of the class A has the worst quality? Which DQ dimension of the whole system is the worst? To answer the first question, one would have to set the scope of the analysis to a single class type A and analyse the data for separated instances. In the case of the second question, one would have to set the scope of analysis to multiple (all) classes and analyse the data in an aggregated manner. Therefore, to give a complete overview of the DQ of a system and provide the functionality to drill down to investigate specific data causing low DQ, the various feedback options and levels have to be used as a complete set of tools.

An overview of this approach is given in Fig. 3.15. The two **prerequisites** for an information system to use the approach are depicted on the left hand side: The upper box represents (1) constraint definitions of the information system given in various forms such as BV annotations, SQL statements or business rules. The lower box, together with the connected validation box to the right of it, depicts the (2) classic constraint checking of the running information system which generates and handles constraint violations. The format of data will depend on the system and may, for example, be object instances or table rows.

The **DQ analyser tool** shown on the right hand side of Fig. 3.15 receives four files as input depicted by the connected arrows and file icons. The upper one is the static **configuration file for the mapping** of constraints, dimensions and penalties. The one in the middle represents two files, one for the static information about the **object hierarchy** and one for the **object identifications**, while the lower arrow depicts the dynamic information of the **constraint violations**. The static data is only needed once, but the constraint violations change over time in the running information system

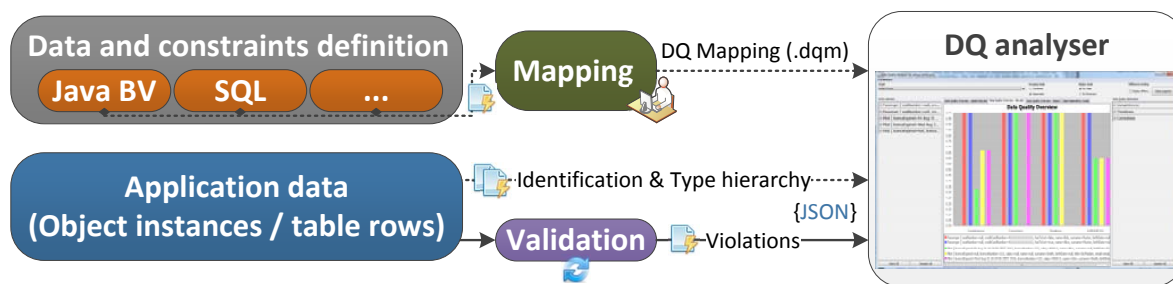


Figure 3.15: Concept overview

and therefore this information has to be resent at the time of analysis. The DQ analyser computes the DQ based on this data and gives feedback which can form the starting point for interactive data analysis.

The approach is not limited to transferring files for the input data from a main application to the DQ analyser tool. It rather provides the possibility to call the tool programmatically using the provided API which is not separately depicted in Fig. 3.15.

Thus, an information system can generate the required input according to the specified input format which will be described in detail in Sect. 3.5.4. However, this approach also supports the automatic generation of (1) a DQ mapping file stub for penalties and dimensions based on the constraint definitions, (2) the object identification and type hierarchy based on the application data, and (3) the violation representation based on the constraint checking. Each of these is marked in Fig. 3.15 with a file icon.

To configure the DQ mapping for a particular context, the data analyst only has to group constraints into dimensions and provide the individual mappings of dimensions and violation penalties for the constraints in the prepared DQ mapping file. This process is depicted by the ‘Mapping’ box in the upper middle of Fig. 3.15. Dimensions can be chosen and labelled at will, but their names have to be unique. An entry of the mapping file contains the constraint location, the constraint name and one or multiple DQ dimensions, each associated with a violation penalty. The constraint location and name together uniquely identify a constraint. The penalty values correspond to the numerical scale of $[0.0, 1.0]$ where a higher value represents a more severe constraint violation which results, if violated, in a stronger negative influence on the DQ evaluation. The general syntax of one mapping file entry can be summarised as depicted below:

```
Location Name [Dimension1 Penalty1, Dimension2 Penalty2, ...]
```

For example, a `Person` entity that is located in a Java package `model` could have a `NotNull` constraint defined on its attribute `name` which is mapped to the dimension `Completeness` and assigned a high penalty of 0.8. This would then be represented in a mapping file entry as follows:

```
model_person_name NotNull [Completeness 0.8]
```

The details of how DQ is computed in this approach are given in the subsequent subsection.

3.5.2 Computing data quality

On our numerical scale of $[0.0, 1.0]$ for penalty values, the quality of flawless data which does not violate any constraint is equal to 1.0. To compute the score of DQ for a

particular dimension, we subtract from this maximum value a normalised sum of all penalties assigned to violated constraints which are mapped to that dimension. Which constraints and object instances are taken into account and how the normalisation is applied depends on the desired abstraction level. For DQ feedback, we distinguish five different levels of granularity which either correspond to a single class or multiple classes in terms of the object data model. In particular, we provide the DQ computation for (1) a single object instance, multiple instances of one class (2) separated or (3) aggregated, and all instances of multiple classes (4) separated or (5) aggregated.

When validating constraints, not only in the database but in any object of an information system, it must be considered that objects may form subclass-superclass relationships constituting complex hierarchies. The resulting constraint inheritance affects how constraints are evaluated and which are taken into account when computing a DQ score for the selected DQ dimension at a particular level. As a consequence, during analysis of constraint violations for a single object, we take into account all constraints defined for its type and, additionally, all the constraints defined in all of its supertypes recursively. This is sufficient for analysis on the level of a single class. If multiple classes in the same type hierarchy are analysed, we also have to ensure that constraints are not taken into account multiple times, for example from a subclass and also a superclass.

We now detail how the DQ dimension scores (*dqds*) are defined on all five levels. The penalty for a particular constraint c and dimension d are given as:

$$penalty(c, d) = \begin{cases} 0.0 & \text{if constraint } c \text{ is not mapped} \\ & \text{to the DQ dimension } d \\ p \in (0.0, 1.0] & \text{if constraint } c \text{ is mapped to the DQ} \\ & \text{dimension } d \text{ with the penalty value } p \end{cases}$$

We define a violation function for a constraint c and object instance i as:

$$violation(i, c) = \begin{cases} 1 & \text{if instance } i \text{ violates constraint } c \\ 0 & \text{otherwise} \end{cases}$$

The DQ score *dqds* for a given dimension and selected object instances depends on the particular analysis level. For a **single object instance**, *dqds* is defined by Eq. 3.1:

$$dqds(d, i_t) = 1 - \frac{\sum_{c \in C_t^+} violation(i_t, c) \cdot penalty(c, d)}{\sum_{c \in C_t^+} penalty(c, d)} \quad (3.1)$$

where d is a DQ dimension, i_t is a single instance of a data object of type t (either the type t or any of its subtypes), c is a constraint and C_t^+ is the set of all constraints defined for the type t including constraints defined also for supertypes of t . Note that type t can be variable in form, for example a class in Java or a table in SQL. If a single object instance has no constraints associated with a particular DQ dimension, the penalty is 0 which does not distort the final DQ score value.

For the analysis of **multiple instances of one class separated**, we use the formula for a single object instance given above and iterate over the set of multiple instances.

For **multiple instances of one class aggregated**, *dqds* is defined by Eq. 3.2:

$$dqds(d, I_t) = 1 - \frac{\sum_{i \in I_t} \sum_{c \in C_t^+} violation(i, c) \cdot penalty(c, d)}{|I_t| \cdot \sum_{c \in C_t^+} penalty(c, d)} \quad (3.2)$$

where we use the same nomenclature as in Eq. 3.1 and additionally I_t is a set of instances of data objects of type t (either the type t or any of its subtypes) and i is a single instance of a data object. Obviously, the formula for multiple instances of one class aggregated can also be used to compute the appropriate $dqds$ for all object instances of one class aggregated.

For the analysis of **all instances of multiple classes separated**, we use the formula presented for multiple instances of one class aggregated above, taking into account all object instances of one class and iterating over the set of types.

For **all instances of multiple classes aggregated**, $dqds$ is defined by Eq. 3.3:

$$dqds(d, T) = 1 - \frac{\sum_{t \in T^+} \sum_{i \in I_t^*} \sum_{c \in C_t} violation(i, c) \cdot penalty(c, d)}{\sum_{t \in T^+} (|I_t^*| \cdot \sum_{c \in C_t} penalty(c, d))} \quad (3.3)$$

where we use the same nomenclature as in Eq. 3.1 and Eq. 3.3 and additionally T is a set of types, T^+ is the closure of the set of types, i.e. all types in the set T and all their supertypes recursively, I_t^* is a set of all instances of data objects of type t (either the type t or any of its subtypes) and C_t is the set of all constraints defined for the type t exclusively.

Finally, the overall DQ of the whole system can be computed using the formula for all instances of multiple classes aggregated for the set of all types.

A more detailed description of how the constraint inheritance is taken into account when computing DQ and explanations of all computation formulae including examples is presented in [173].

3.5.3 Data quality feedback

Our DQ analyser tool is the central component of our DQ feedback system which computes the DQ scores of dimensions for the given, uniquely identified, instances of the application data. It matches constraint violations with constraint-dimension mappings taking into account the constraints and class types hierarchy.

To give feedback on DQ, we have chosen to provide three main feedback types which can all be configured with various analysis options. The main types are a textual report and two graphical representations – bar plots and spider web plots.

Before presenting the individual feedback types, we give an overview of the tool with its components in Fig. 3.16 where the components are labelled 1 to 8.

Using the tabbed pane (1), it is possible to switch between the main feedback types which are then displayed in the centre. The last tab ‘Class Dependency Graph’ displays the hierarchy of class types.

A sample graph of such a hierarchy for a simple airline domain is presented in Fig. 3.17 where the highlighted node ‘model.Person’ represents the class that is currently being analysed. The ‘Scope’ combo box (2) allows the analysis scope to be selected i.e. either a concrete class type or multiple types. The ‘Grouping mode’ (3) is used to specify whether the DQ dimension scores should be presented for each

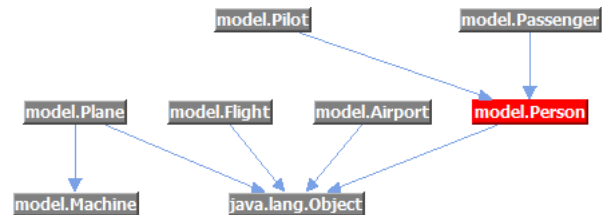


Figure 3.17: Hierarchy graph

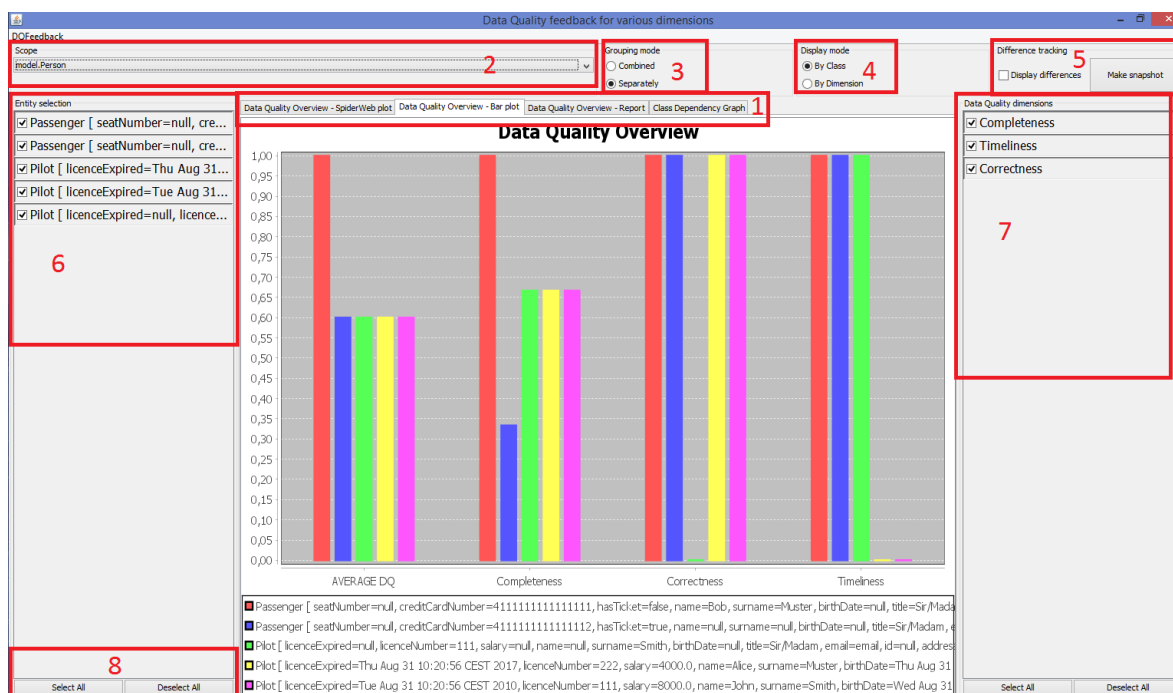


Figure 3.16: DQ analyser overview

entity separated or aggregated over the set of entities. The ‘Display mode’ (4) defines whether the DQ is presented according to the classes/types (scores of DQ dimensions grouped by entities) or DQ dimensions (scores of DQ dimensions grouped by dimensions). The ‘Difference tracking’ option (5) allows differences between two different selections to be compared in components 6 and 7. When the button ‘Make snapshot’ is clicked, the current selection of entities and DQ dimensions is temporarily stored. When the check box ‘Display differences’ is selected, not only the current selection of components 6 and 7 is visualised but also the one stored as a snapshot. Changing a choice in components 2, 3 or 4 clears the stored snapshot since the comparison for different scopes does not make sense. The ‘Entity selection’ list (6) allows multiple entities such as class types or instances to be chosen. The ‘Data Quality dimensions’ list (7) allows the selection of the DQ dimensions to be visualised. The list only shows dimensions which are actually specified for the particular scope dependent on the selection in component 2, either all specified dimensions for a concrete class or multiple classes, respectively. The selection buttons (8) enable all entries of the check box list above to be selected or none. This is possible for both the ‘Entity selection’ list and the ‘Data Quality dimensions’ list. With combinations of choices in the components 2, 3 and 4, the various analysis levels presented in Sect. 3.5.2 can be selected.

In the ‘DQ Feedback’ menu shown in the upper left corner of Fig. 3.16, the appropriate input files described in Sect. 3.5.1 can be loaded. After they are loaded, a data analyst can start their analysis using the options described above.

In the remainder of this section, we will describe the main feedback types which are visualised in the central tabbed pane of the tool.

A **textual report**, such as the example given in Fig. 3.18b, contains detailed numerical scores about DQ dimensions with regard to the selected granularity level and result aggregation options. This type of feedback is suitable for particular DQ dimensions and for the overall DQ feedback.

An example of a **bar plot** is shown in the centre of Fig. 3.16. The purpose of the bar plot is very similar to the textual report described above since they both show detailed numerical scores, but the graphical representation makes the perception of differences between scores much easier. It is suitable for both overall DQ feedback and scores of particular DQ dimensions.

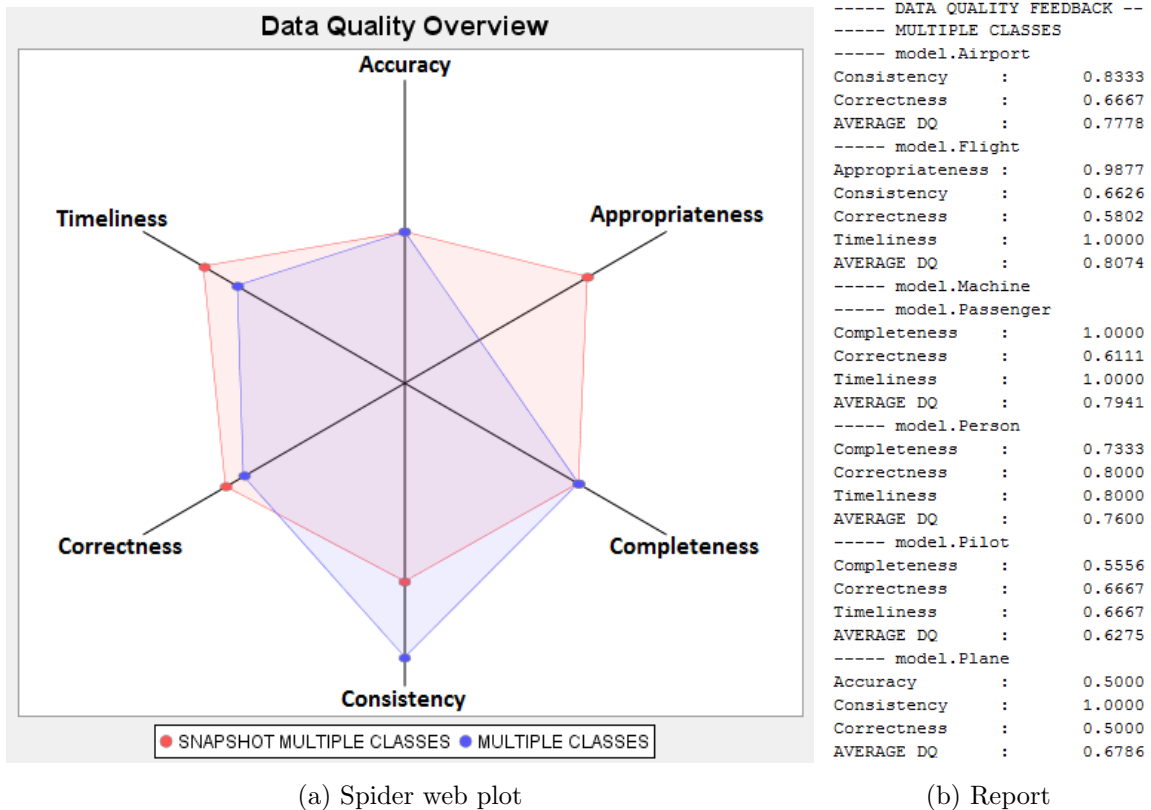


Figure 3.18: Various feedback variants

A **spider web plot**, such as that shown in Fig. 3.18a, illustrates differences in the DQ either for different classes/instances or different DQ dimensions depending on the aggregation mode. Each axis is scaled from 0.0 to 1.0. The further the point lies from the origin, the better the DQ. This plot also allows for the comparison of various DQ dimensions for many classes/instances by drawing them all in a single plot. It is not suitable for the overall DQ as it should reflect similar features, while the overall DQ is an aggregated feature of all DQ dimensions. The plot in Fig. 3.18a also shows the comparison of a stored snapshot with a current selection of components as described above in the components overview.

3.5.4 Implementation

The DQ analyser is built as a Java application using the Swing¹¹ toolkit for the GUI. It implements the Model-View-Presenter (MVP) approach which enables us to keep the code, which is processing the data (model), independent from the code responsible for the visualisation (presentation) and the view. As a result, we can easily replace the

¹¹<http://docs.oracle.com/javase/8/docs/technotes/guides/swing/>

existing GUI with another one without any alteration to the model. In the same way, we can expand the model without the necessity to change the presentation layer.

Fig. 3.19 depicts a simplified overview of the components in the framework where we abbreviate data quality with *DQ* and feedback with *FB*. The components `DQFBView`, `DQFBWindow`, `DQFBPresenter` and `DQFBModel` are the main components of the DQ analyser that implement the MVP design pattern. `DQFBView` is an interface defining all the ways in which a presenter (`DQFBPresenter`) may interact with the view. The class `DQFBWindow` is an actual implementation of the GUI. The class `DQFBPresenter` is the component that mediates the communication between the model and the view. The class `DQFBModel` is the main component of the model which is responsible for storing all the data needed to visualise the scores of the particular DQ dimensions, i.e. constraint to DQ dimension mappings (`DQMapping`), a list of constraint violations (`ConstraintViolations`), all entities for which the DQ feedback should be presented, the types hierarchy graph required for correctly computing scores for particular DQ dimensions and the name of the top level package (scope) that should be visualised. The class `DQComputer` is responsible for computing the score of a particular DQ dimension for a given set of data. It takes into account the given mapping, the types hierarchy, the list of all entities and constraint violations and contains the implementation of all formulae discussed in Sect. 3.5.2.

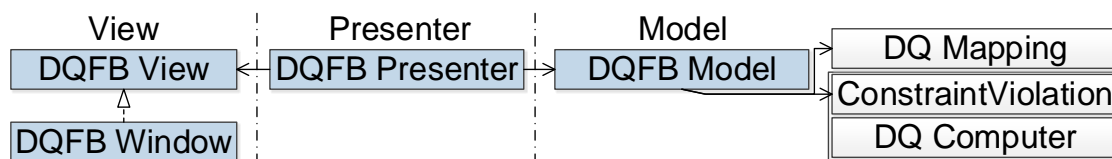


Figure 3.19: Main components

The DQ analyser is delivered as a Java Archive (JAR) and can be used either as a standalone application or called programmatically using the provided API contained in the same archive. In the first case, the input data, described in Sect. 3.5.1, is provided as files, whilst, in the second case, as Java object maps. Having these potential inputs ensures independence from the technology and validation framework, at the same time enforcing a well-defined interface usage.

As mentioned in Sect. 3.5.1, this approach also supports the automatic generation of the DQ mapping file stub, the object identification, the type hierarchy and the violation representation based on the constraint definitions and the application data. For that, we also provide an API where developers can invoke the appropriate methods in our utility classes such as `DQUtil` and `PostgresUtil`, which extract information from entities, constraint violations or by querying the database, and build simple structures that are serialised to JSON or our DQ mapping, respectively.

In the remainder of this section, we will provide examples for the input data of a simple airline domain.

List. 3.19 shows a DQ mapping file where three constraints are mapped to appropriate DQ dimensions. The custom `CorrectTime` constraint is defined as a class constraint since the constraint location only includes a class name. The `Pattern` and `NotNull` constraints are defined for an attribute `name` on the ‘Flight’ and ‘Person’ entity, respectively.


```

model_flight CorrectTime [Consistency 0.2, Timeliness 0.5]
model_flight_name Pattern [Correctness 0.2]
model_person_name NotNull [Completeness 0.2]

```

Listing 3.19: Example of DQ mapping file (`mapping.dqm`)

A constraint location is an unquoted string without white spaces that contains a path to the place where the constraint is located. The interpretation of such a path depends on the specific technology, therefore the only general requirement is to separate path segments with an underscore sign (`_`). For example, for a method parameter name in Java, this could be a package name + class name + method name + parameter name where the underscores are used instead of dots.

An example of a JSON type hierarchy is depicted in List. 3.20 where `Pilot` and `Passenger` are subclasses of `Person`, while `Person` is a subclass of `Object` and `Object` does not have a superclass. The first segment, for example `model.Pilot`, defines a class, while the second segment, for example `model.Person`, defines its superclass.

```

{
  "model.Pilot": ["model.Person"],
  "model.Passenger": ["model.Person"],
  "model.Person": ["java.lang.Object"],
  "java.lang.Object": []
}

```

Listing 3.20: Example of JSON type hierarchy

List. 3.21 presents a JSON object identification definition where a ‘Flight’ and a ‘Passenger’ object are identified. The first segment, for example `Flight 7 [id=7]`, defines a unique object ID and the second segment, for example `model.Flight`, defines its class type. Any unique string can be used as IDs, for example the return value of a `toString` method in Java called on the given object for which the ID is generated.

```

{
  "Flight 7 [id=7]": "model.Flight",
  "Passenger [id=4]": "model.Passenger"
}

```

Listing 3.21: Example of JSON object identification

List. 3.22 shows an example of a JSON constraint violation definition where three constraint violations are defined by means of the first segments `entity`, `constraintName` and `constraintLocation`.


```
{
  "entity": "Flight 7 [id=7]",
  "constraintName": "CorrectTime",
  "constraintLocation": "model_flight"
}, {
  "entity": "Flight 7 [id=7]",
  "constraintName": "Pattern",
  "constraintLocation": "model_flight_name"
}, {
  "entity": "Passenger [id=4]",
  "constraintName": "NotNull",
  "constraintLocation": "model_passenger_name"
}
```

Listing 3.22: Example of JSON constraint violations

The appropriate second segments such as `Flight 7 [id=7]`, `CorrectTime` and `model_Flight` have to correspond with the definitions in the other input data.

3.5.5 Evaluation

To evaluate how the DQ framework approach and implementation was perceived by potential users and to find out ways in which it could be improved, we conducted a user study where we focussed on participants with at least a basic knowledge of software engineering. In particular, we wanted to observe which types of feedback and analysis options were most appreciated and how users rated this approach and tool for providing feedback on DQ.

In total, twenty participants contributed to the study of whom 50% had at some time studied or worked in a university and 75% had at some time worked in industry. 75% were male and 25% female, with 10% between 20 and 25 years old, 35% between 26 and 30, 25% between 31 and 35 and 30% over 36. Only 25% of the participants stated that they already had a good knowledge of DQ in information systems with 75% stating that they had no or only poor previous knowledge.

The participants were given an introduction to our DQ analyser tool along with explanations of different modes and types of feedback that are available. They then had to solve five DQ analysis tasks with the help of our tool for a sample dataset in a simple airline domain. After solving each task, they had to fill out a questionnaire where they stated their solution to the question, which feedback type they used, which analysis scope, which grouping mode, which display mode and if they used the difference tracking. After solving all the tasks, they had to fill out an overall feedback form where they could also provide comments.

Tab. 3.2 gives an overview of the types and options, as described in Sect. 3.5.3, that were used for solving the following five tasks: (1) Which instance of the class ‘Flight’ is of the worst quality? (2) Which DQ dimension of the class ‘Flight’ is the worst? (3) Which DQ dimension of the whole system is the worst? (4) Which DQ dimension is affected the most when considering the class ‘Person’ and including only pilots instead of only passengers? (5) Which classes contain inconsistent data?

Task	FB Type			Scope		Grouping		Disp.mode		Diff.used
	Bar	Spider	Report	Single	Multiple	Comb.	Sep.	Class	Dim.	
1	20			20			20	19	1	0
2	18	2		20		12	8	14	6	0
3	19		1		20	15	5	14	6	0
4	10	9	1	15	5	17	3	16	4	18
5	16	3	1	1	19	13	7	14	6	0

Table 3.2: Chosen feedback types and analysis options

In general, the majority of participants had chosen the analysis options we expected for the individual tasks which means that the usage of the tool was intuitive and easy to understand. Nonetheless, there were some choices concerning the ‘Grouping mode’, highlighted in Tab. 3.2 with a grey background, which are worth discussing. Since task 2 asked for the rating of a certain dimension instead of an instance, we would have expected the participants to choose the combined grouping mode. Although, the majority (12) did so, several had chosen the separated mode. In task 5, we asked for certain classes and thus we would have expected that the participants would choose the separated grouping mode, but the majority (13) had chosen the combined mode. Thus, grouping seems not to be as intuitive and might need more explanation for potential users.

We did not expect a particular choice for the feedback types since all of them are suitable for solving the tasks, but there was a clear preference for the bar plots with only one participant using the raw data report. Interestingly, the spider web plots were used mostly by the users who claimed to already have good knowledge of DQ.

The snapshot feature turned out to be well understood since the large majority (18) used it to solve the fourth task which was the most natural and simple way to find the answer.

To evaluate the overall feedback, the participants had to answer five questions using a Likert-type scale with possible answers ‘totally agree’, ‘agree’, ‘neutral’, ‘disagree’ and ‘totally disagree’. For the evaluation here, we consider the first two values as agree and last two values as disagree. 90% of the participants agreed that ‘the tool was easy to use’ while 10% disagreed. 70% agreed that they ‘had no problem finding the answers’ while only 10% disagreed. 80% agreed that ‘the tool informs well about the DQ’ while no one disagreed. Even 90% affirmed that they ‘understood the plots’ while no one disagreed, and 80% agreed that ‘it was easy to learn the tool’ while only 5% disagreed.

To sum up, the feedback on the tool was very positive in terms of the information given, being useful and easy to understand. Most of the participants found it easy to use and easy to learn. Nonetheless, for a small group, which had an exclusively industrial background, the user interface turned out to be less easy to use than for the rest of the participants. Thus, a direction of future work is to consider improvements to the user interface. A couple of minor improvements mentioned in the comments were to provide indicators to show if a snapshot has already been recorded, and whether the ‘Make snapshot’ action was successful.

3.5.6 Conclusions

In this section, we have presented how we extended the concept of constraints to a concept of general DQ indicators by associating individual constraints with DQ dimensions and violation penalties. We further introduced the DQ analyser framework and tool which provides new ways of analysing DQ of an information system independent of the application and data model based on our new DQ mapping, measuring and calculating approach. The mapping can be changed between individual analysis runs and therefore provides a functionality of switching contexts easily. The conducted user study showed that potential users are able to find answers for detailed, real life DQ issues using our tool which helps to increase the reliability of information systems and to minimise operational costs. Therefore, we made a step forward towards a better management of DQ problems in information systems.

In further work, we were extending our general DQ analysis framework to support the generation of DQ mapping file stubs based on unified constraint definitions and various transformations between arbitrary constraint representations. We also planned to explore in more detail what types and aspects of DQ are, or could be, supported by the framework.

3.6 Summary

With the initial approaches, presented in this chapter, we arrived at diversified solutions to constraint-based approaches to DQ in information systems with the goal to provide a complete DQMF addressing the various challenges.

In Sect. 3.1, *Guidance on data quality feedback*, we presented a proposal of giving feedback on DQ to users without providing details on an underlying data validation and measurement of DQ. The focus was on giving answers to the question ‘How to give feedback on DQ?’ and to provide an evaluation from end-users which we used as foundation for further approaches on DQIF and DQAF. With the separated approach presented in Sect. 3.2 (*Object database data quality management*), the integrated approach presented in Sect. 3.3 (*Data quality extended Bean Validation*) and the integrated approach presented in Sect. 3.4 (*Bean Validation OCL extension*) we presented different approaches mainly concerned about data validation, unified constraint management and measures for DQ based on constraint violations. In these three approaches we provided ways of calculating DQ related to constraint compliance as a basis for giving feedback on DQ but we did not provide proposals of presenting the feedback to users. The general approach and framework presented in Sect. 3.5, *Data quality monitoring framework*, with its focus of independent analysing and giving feedback on DQ, completed our first exploration on bringing unified constraint management in relation to DQ.

In the beginning we thought that it would be desirable to provide one DQMF which is adequate for all purposes. While exploring the different approaches, we realised that there exist many different requirements to DQMFs dependent on the type of information system. For example, one existing system might want to keep its data validation and only needs functionality to analyse the quality of the data. Other systems, existing or to be built from scratch, might need data validation and DQ feedback functionality but, dependent on the technologies used, differ in the dependency to a DQMF. In particular, some systems might want to use an integrated DQMF based on BV where other might

require different integrated approaches or even separated approaches which offer more flexibility in defining constraints, i.e. during runtime. We further realised that there is also the need for keeping constraint definitions distributed in an information system by using different constraint representations/technologies in the various components.

The goal then was to build further DQMFs for different purposes based on the knowledge gained in our initial approaches. Therefore, we present in the subsequent chapter such approaches which serve for varying purposes. These approaches integrate on the one hand various concepts already explored with the approaches described in this chapter but on the other hand introduce additional features and concepts.

4

Unified data quality assurance and analysis based on Bean Validation

In our initial work, presented in the previous chapter, we built DQMFs where the focus was on providing unified constraint management, data validation and feedback on DQ either separated from an application/database (Sect. 3.2) or integrated into an information system (Sect. 3.1, 3.3, 3.4). After investigating these approaches, we recognised that, in many cases, it can be beneficial if a DQMF only deals with the DQ computation and DQ feedback, leaving the definition of constraints and the data validation to the separated information system being analysed. Thus, our initial approaches also include a separated DQMF used only for monitoring and analysing the DQ based on provided validation information (Sect. 3.5).

We continued our initial work and present in this and the following chapters, our final approaches, where we consolidate the initial approaches but also propose novel concepts for highly flexible and configurable DQMFs that can easily be adapted to the needs of different users and applications. Likewise to the initial approaches, our final approaches enable developers to ensure that certain integrity guarantees of an information system can be met through a single, extended constraint model that avoids any inconsistencies and redundancy within the constraint specification and validation processes. They aim to fulfil all of our proposed requirements of a DQMF but vary in terms of the type of the DQMF since we recognised that different initial software development scenarios call for different DQMF approaches. We describe how our approaches can be realised and how our DQMFs allow the users to validate data during input, as well as analyse, quantify and visualise the specific and overall quality of the data at any time.

One main insight from our initial approaches was the fact, that there does not exist a ‘one size fits all’ solution to provide unified constraint-based DQ management in information systems. Initial software development scenarios can differ in many aspects, such as whether an information system will be built from scratch or an existing information system is to be enhanced with appropriate functionality. In the latter case, the freedom of choosing an approach is certainly more restricted due to the existing im-

plementation. But even if a new information system is planned, choosing an approach is dependent on the requirements and goals for DQ management. Most importantly, it has to be distinguished whether the subject is a homogeneous information system built with technologies that naturally work together, or if it is a heterogeneous system where different components in various technologies interact with each other. Further, the architecture of a system, how components are working together and the chosen technologies for developers is essential to decide on a approach. For a homogeneous system, it is much simpler to provide a unified constraint representation in a specific technology. In contrast, for a heterogeneous system, an approach has to be chosen that is based on different constraint representations which have to be translated into each other, and therefore introduces more complexity into the approach. Once this general decision is taken, the details of the approach further depend on the complexity of the constraints required, the need for runtime adaptation of constraint definitions and DQ calculations, and the intended level of DQ feedback.

One goal during our TDS-Exposure project was to investigate an advanced DQMF for our FoodCASE information system. The constraints in the food science domain are inherently complex. Further, we gathered from the project participants the requirement for flexible and customisable validation that distinguishes different user groups and the need for detailed DQ feedback for input data and data analysis. Since FoodCASE is a homogeneous three tier system implemented exclusively in Java, we had the chance to investigate a homogeneous integrated approach with the focus on the Java programming language that would include complex constraints, customisable validation and detailed DQ feedback. This general approach and implementation is presented in this chapter.

Thus, our work presented in this chapter focuses on the creation of a constraint-based DQ modelling and calculation concept that can be applied to any type of (homogeneous) information system and enables all constraints to be managed in a unified way. This concept is used as the basis for creating a highly flexible DQMF that allows the users of the information system to configure the details of every quality-related aspect according to their needs. Our concept of indicating DQ and giving feedback on constraint violations is integrated with our concept of contracts and, additionally, provides the potential to configure default and constant importance weights, validation groups and an error threshold. The DQMF that we introduce consists of an input validation module, a DQ analysis module for DQ feedback and an administrative tools module for its configuration.

To address the issues of limited support for constraints in databases and distributed data validation processes with the negative consequences of redundancy, code duplication, possible inconsistencies and increased maintenance effort, we propose an extended constraint model along with a DQMF to specify and manage constraints in a single place.

By advancing the concept of DQIs, already introduced in our initial approaches, we are able to provide new measures for indicating DQ as an interpretation of constraint compliance. Thus, we are able to address a wide variety of DQ issues not limited to purely preserving integrity as with traditional database constraints. To calculate appropriate DQ scores, our DQIs are based on the importance of constraints and the severity of constraint violations which are highly configurable in the approach presented in this chapter.

For giving feedback on DQ, this approach is based on our initial approach presen-

ted in Sect. 3.1. Constraint checking and data validation is based on BV as already explored in the two initial BV approaches presented in Sect. 3.3 and 3.4, respectively. By using BV, this approach enables constraints to be specified and checked only once, even if constraint checking is used in different layers, components or technologies of an application.

To evaluate this approach, we implemented the DQMF for the FoodCASE food science data management system as an integrated framework. This implementation was presented to several food science experts from across Europe, who were then asked to assess it and provide their feedback on several aspects of the framework. An analysis of this assessment and feedback is also part of this chapter.

4.1 Approach

Our aim for this integrated approach was to provide software developers and application users as data consumers with a highly flexible and easily customisable framework. It should be possible to define and validate DQIs as complex rules in order to avoid inconsistencies and redundancies. Further, the framework should provide the basis for measuring, calculating and giving feedback on DQ both for input validation and analysis of existing data.

Specifically, it should meet the following requirements from the approach presented in Sect. 3.3: (1) The potential to specify all constraints for a data model using a single constraint model; (2) Constraints need only be defined once within a system to avoid redundancy; (3) Validation within a system is performed based on the defined constraints only; (4) Single validation of any constraint to avoid redundancy in system validation processes; (5) The use of constraints to express and indicate different levels of DQ with so-called DQIs. For this approach, we did not include the requirement that constraints can be specified and changed dynamically during runtime. But the approach includes the following two additional requirements: (6) The weighting of DQIs is highly flexible and easily configurable, and can be specified and changed dynamically during runtime; (7) DQ feedback is given during input validation as well as during data analysis.

The need for data validation in different tiers and layers of an information system usually causes the implementation of separate validation mechanisms for each tier and layer as explained in Sect. 2.5 and depicted in the left of Fig. 4.1 by the multiple validation locations. By basing data validation on BV, where constraint annotations are defined within the domain model, the approach presented in this section, inherently centralises the validation mechanism as shown in Fig. 4.1 on the right most bar. BV naturally is used for validation in the domain entities within the logic tier but can also be used in the presentation tier and therefore provides the possibility for cross-tier validation.

BV generally supports invoking the validation process manually which gives developers using this approach, the freedom to re-validate data in a multi-tier architecture in a specific layer at any given time. This is especially necessary in this approach, since the data validation is not only used for the inevitable input validation to maintain data integrity but also for the analysis of already existing data to provide DQIF as well as DQAF.

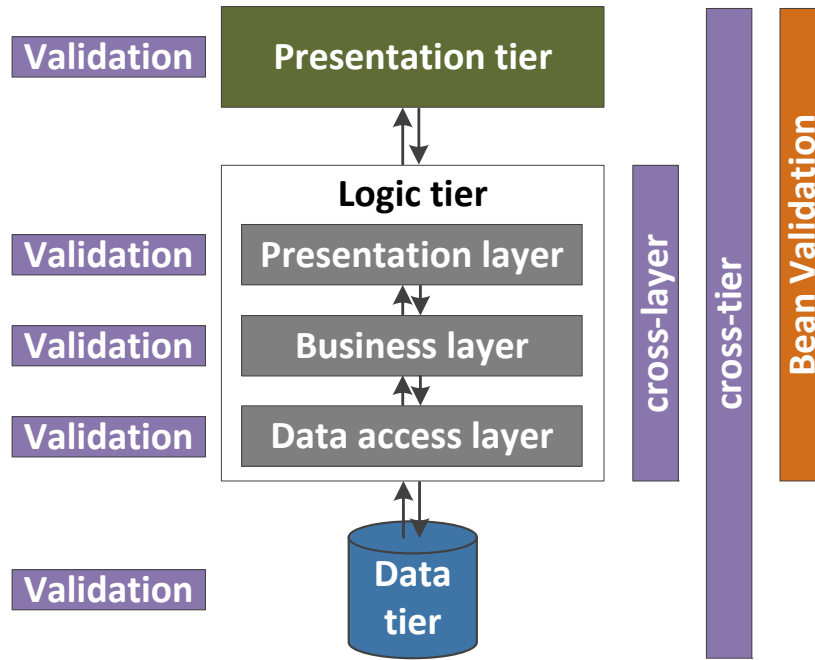


Figure 4.1: Typical three-tier architecture with unified validation mechanism using Bean Validation

Using BV, requirements 1 to 4 are already met in terms of the architectural requirement. Further, any technology that integrates BV will be able to make use of the proposed DQMF. To support requirements 5 to 7, it was necessary to introduce the new concepts of DQIs, contracts, DQ measurement and calculation, error threshold and validation groups, which we will examine in more detail in the remainder of this section.

In our concept, constraints are combined with an importance weight that is the basis for evaluating DQ in a more fine-grained way on various levels and aggregations. DQIs provide feedback on each of these levels. The weights used for the DQ calculation are defined separately from the constraints in contracts which are key-value collections assigning the importance, and potentially other parameters, to constraints and which can be exchanged at runtime. Therefore, the weights are not determined only once in an empirical objective way, but rather designed to be customisable.

4.1.1 Data quality indicators

For measuring, calculating and giving feedback concerning DQ, we first based our solutions in the initial approaches on the concept of HCs and SCs. But already with the approach presented in Sect. 3.5 from the initial approaches, we introduced the concept of general DQIs as a basis for indicating DQ alongside the traditional constraints that prohibit invalid data. Thus, our DQIs have evolved from simpler HCs/SCs definitions for single constraints to more complex general DQIs for multiple constraints, object instances and entities. We will now present how the evolved DQIs are realised in our concept for BV.

The features and benefits of BV were summarised previously, but a limitation of BV for DQ management is that it does not allow for cases where data validation requires

something more flexible than a strict true/false outcome. Assume, for example, that persons with a name and email address are stored in an information system. The name is a mandatory attribute that can be validated with a ‘not null’ constraint. Persons can exist without an email address and, therefore, the email attribute is not specified as mandatory. Nevertheless, we might want to associate such instances as having lower DQ with respect to completeness if the person is known to have an email address that has not been supplied.

To provide developers with control over the validation process, and allow them to associate DQ feedback with constraint violations for single and aggregated data items, we enhanced the notion of DQI introduced in our initial approaches.

A DQI provides a rating for one or multiple constraints applied to one or more instances of one or more entities where constraints are weighted with a certain importance measure. If multiple constraints and/or multiple instances are considered, we refer to these as aggregated Quality Indicators (QIs), while single QIs refer to cases where only one constraint and one instance are considered. In terms of an object data model, an entity is a class and an instance is an object instance of a class. A single QI assigns an importance weight to a particular constraint definition where the scope of the constraint definition can be an entity type or an entity attribute. This does not exclude that other entities are taken into account while validating the constraint, but defines that the rating for the QI is applied to the single entity. Based on that, we can aggregate multiple single QIs at several levels in order to provide a rating for various combinations of instances and constraints. We have defined the aggregated DQIs ‘OIQI’, ‘MIQI’ and ‘MEQI’, which are depicted in Tab. 4.1. The ‘OIQI’ provides a DQ measure for multiple constraints of one instance of one particular entity. Thus, it is ideal to use for the feedback on DQ during data input for a single instance, for example in a single instance detail frame. The ‘MIQI’ provides a DQ measure for one or multiple constraints of multiple instances of one particular entity. Thus, it is ideal to use for the feedback on DQ during data analysis if a DQ analyst is interested in a certain entity of an information system. The ‘MEQI’ provides a DQ measure for one or multiple constraints of multiple instances of multiple entities. Thus, it is ideal to use for the feedback on DQ during data analysis if a DQ analyst is interested in multiple entities of an information system or an overall DQ score for all data in an information system. Appropriate calculations of the QIs are described in the following subsection.

DQI calculation	Constraints	Instances and entities
OIQI	Multiple	One instance (of one particular entity)
MIQI	One or multiple	Multiple instances of a particular entity
MEQI	One or multiple	Multiple instances of multiple entities

Table 4.1: Aggregated data quality indicators overview

4.1.2 Data quality measurement and calculation

With the framework, we provide DQ measurement, calculation and feedback based on the ideas presented in our initial approaches, but calculate the DQ using the DQ measurement of our DQIs, which take the importance weights into account. This DQ

calculation results in a rating for a DQ score where we distinguish different levels of aggregation, as explained above and summarised in Tab. 4.1.

For the aggregation, we used a weighted arithmetic mean since it uses every value in the data and is a good representative of the data. Moreover, we wanted to provide an easily understandable indicator which was also affirmed by the users of our system. Nonetheless, this approach allows configuration of the formulas used and therefore it would also be possible to use the weighted geometric mean, min/max operators or other metrics.

Both the importance weights assigned to constraints, and the DQ score scale, range from 0 to 100, similar to the ‘Interval’ scale in DQ metadata formats. In terms of weights, 0 is the least and 100 the most important while, in terms of the DQ score, 0 is the lowest and 100 the highest. Corresponding to Tab. 4.1, we defined the following three aggregated DQI calculations:

The **OIQI** is a weighted arithmetic mean used to calculate the DQ score for **One Instance** of one entity and is defined by Eq. 4.1 where n is the number of constraints that are defined for the current entity, ω_i is the weight of constraint i and σ_i is either 1, if constraint i is satisfied, or 0, if constraint i is violated. Thus, a constraint with a higher importance weight has more influence on the DQ score for one entity. The weights of all satisfied constraints in one instance is summed up, divided by the total sum of all constraint weights and multiplied by 100.

$$OIQI = \left| 100 \cdot \frac{\sum_{i=1}^n \sigma_i \cdot \omega_i}{\sum_{i=1}^n \omega_i} \right| \quad (4.1)$$

The **MIQI** is an arithmetic mean used to calculate the DQ score for **Multiple Instances** of one entity and is defined by Eq. 4.2 where m is the number of object instances for the current entity and $OIQI_i$ is the OIQI of instance i . The OIQI DQ scores of all object instances for a certain entity are summed up and divided by the total number of appropriate object instances.

$$MIQI = \left| \frac{\sum_{i=1}^m OIQI_i}{m} \right| \quad (4.2)$$

The **MEQI** is a weighted arithmetic mean used to calculate the DQ score for **Multiple Entities** and is defined by Eq. 4.3 where k is the number of entities, $MIQI_i$ is the MIQI of entity i and ε_i is the weight of entity i . The weighted MIQI DQ scores for all entities of multiple entities are summed up and divided by the total sum of all entity weights.

$$MEQI = \left| \frac{\sum_{i=1}^k MIQI_i \cdot \varepsilon_i}{\sum_{i=1}^k \varepsilon_i} \right| \quad (4.3)$$

The weight of an entity is defined by Eq. 4.4 where n is the number of constraints that are defined for the current entity i of Eq. 4.3, ω_j is the weight of constraint j and θ is a custom importance value. The sum of weights is taken into account in order to give constraints with a higher total importance more influence on the DQ score while with θ it is possible to configure a custom importance for each entity. In our domain we did not need to include custom entity importance and, therefore, set θ to 1 for all entities.

$$\varepsilon = \left| \left(\sum_{j=1}^n \omega_j \right) \cdot \theta \right| \quad (4.4)$$

The smallest amount of data for which we calculate a DQ score is an entity. Therefore, we do not calculate DQ scores for single QIs. A single QI can be rated based on its assigned weight. Calculating the DQ score of a single QI, as the only constraint defined in an entity, results in the OIQI calculation being either 100, if the constraint is satisfied, or 0, if the constraint is violated.

4.1.3 Contracts

A contract is a collection of single QIs. Each single QI is represented as a key-value pair where the key is a unique constraint identifier and the value is the associated importance optionally followed by a status (`active` or `inactive`). If the status is omitted, the constraint is considered `active`. If the status is set to `inactive`, the appropriate single QI will not be taken into account for the calculation of the DQ score. The key is defined as the combination of an entity name, an optional entity attribute and a constraint name. The general definition can be summarised as

```
entityName.[attributeName].constraintName=importance[, status]
```

where the brackets `[]` denote optional parts of a contract collection item.

Several contracts can be specified that are used to define customised analysis and validation profiles for different users and applications of an information system. For input validation and data analysis, different contracts can be applied but always only one contract is configured to be used for the calculation of the DQ scores at a single point in time for each of them. Contracts can also be easily shared between users. Thus, it is possible for a user to have multiple contracts, while multiple users can share the same contract. Moreover, a default contract can be specified that serves as a fallback in cases where there is no customised contract defined. If a contract does not include all constraints that are defined in the system, a default importance (e.g. 80) is applied.

4.1.4 Error threshold

A threshold for the importance weight is a configurable framework parameter within the scale of 0 to 100 (e.g. 50), which defines the distinction between two types of violations: warnings below the threshold, which are less severe, and errors equal to or above the threshold, which are more severe. This distinction allows us to take different actions based on the type of violations encountered. Examples of these actions include the type of feedback that the user receives during data input, which will be described later, as

well as whether the user should be able to save their input when such a violation occurs. We decided to prohibit the storing of data items that contain constraint violations rated as errors. This distinction is related to the concept of hardgoals (or simply goals) and softgoals in the *Requirements Engineering* literature, as already described.

4.1.5 Validation groups

Each constraint can belong to one or more validation groups, which are used to organise constraints into logical units. For instance, by using validation groups, we can categorise constraints according to their type. Then, we can use this categorisation to distinguish between different types of constraints, such as accuracy-related constraints, completeness-related constraints, or consistency-related constraints. Validation groups allow users to limit data input validation or quality analysis to only the constraints that they are most interested in at any given time. With this concept of validation groups, it is possible to map constraint definitions/violations to any data validation categorisation, such as specific DQ dimensions. For example, arbitrary groups can be defined, such as `CompletenessChecks` or `MinimumValueChecks` which then can be assigned to a ‘not null’ or a ‘minimum value’ constraint, respectively, as explained in detail later in Sect. 4.3.

4.1.6 Constant importance weights

At the point of defining constraints, it is also possible to define a constant importance weight to each constraint that will be taken into account for the DQ calculation instead of an appropriate weight configured in a contract. The reasoning behind this concept is that some constraints might need to always have a particular weight that should stay constant and not be overridden by contracts. For instance, a constraint might be deemed extremely important and should have the highest weight of 100 in any case. However, an authorised user is allowed to disable the constant weights. For example, arbitrary representative values can be defined, such as `SevereError` or `Warning` which then can be assigned to an appropriate constraint, as explained in detail later in Sect. 4.3.

With our DQIs, and the appropriate DQ measurement and calculation, we express and indicate different levels of DQ with the use of constraints and hence fulfil requirement (5). With our contracts, the weighting of DQIs is highly flexible and easily configurable, and can be specified and changed dynamically during runtime and, thus, we fulfil requirement (6). Constant importance weights are an additional feature to limit the power of contracts if necessary, while our error threshold and validation groups in addition enhance the functionality on which our feedback to the user is based. The fulfilment of requirement (7) will be elaborated in the following section.

4.2 Data quality feedback

In our framework, we provide DQ feedback for input validation (DQIF) and data analysis (DQAF) as previously proposed in our initial approach in Sect. 3.1. We now describe how DQIF and DQAF were realised for this approach.

For **DQIF**, every screen where data can be edited has a DQ evaluation panel at the bottom. This panel lists all problems with the current data record. Problems can either be errors or warnings depending on the error threshold explained previously. Violation messages corresponding to errors are displayed in red and prevent the data record from being saved. Warning messages are displayed in orange and indicate problems with the record as shown in Fig. 4.2 at the bottom, but do not prevent it from being saved. In addition, the user is provided with a DQ score for the data record (object instance of an entity) based on the OIQI. The DQ score text changes its colour dependent on the types of violations. It is red, if there is at least one error present, orange if all discovered violations are warnings, or green if no violations were found.

The screenshot shows a software window titled "TDS Pool value detail". It contains several input fields and a data quality evaluation panel at the bottom.

Data Entry Fields:

- Pool name: Brötchen
- Value id: 5
- Substance name: Thiometon sulfoxide
- Value: 0.05 (highlighted in red)
- Value type: (MN) mean
- Unit: (mg) milligram
- Matrix unit: (TKG) per kg total food

Measurement limits:

- LOD: 0.0002
- LOQ: 0.0003
- LOR: 0.0001

Statistics:

- Mean: 0.05
- Median: 1
- Minimum: -1 (highlighted in red)
- Standard deviation: 0.01
- Maximum: 2
- n: 12
- Uncertainty: 0.1
- Maximum uncertainty: 0.2

Date:

- Verifying date: 2014-07-30
- Verifying by: (empty field)

Data quality evaluation:

- The minimum value must be greater than or equal to 0.
- The pool value must be greater than or equal to 1.
- 'Verifying by' field should not be empty.
- Current quality indicator: 48 quality points (out of 100)**

Figure 4.2: Data quality input validation

The data input fields which correspond to the identified constraint violations are also marked with the appropriate background colour, as shown in Fig. 4.2 in the top and middle part, to easily identify the input fields which are the sources of validation issues.

The DQ analysis can be run from the application, but is opened in a separate DQ analysis tool as depicted in Fig. 4.3. The **DQAF** is presented in the lower area, while the appropriate contract and validation groups can be selected in the upper region. Once the analysis has been run, the tabs in the lower part present the DQAF in various forms. The first tab provides a textual DQAF, such as the number of instances and constraints that have been validated and how many violations occurred.

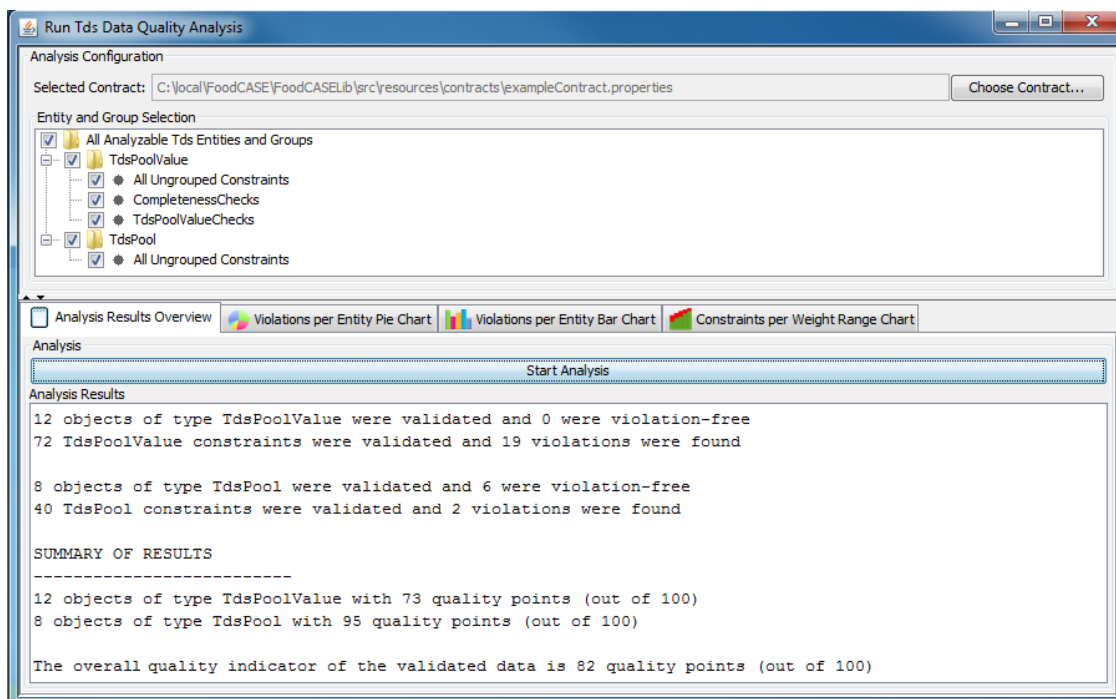


Figure 4.3: Data quality analysis tool

As a summary, the aggregated DQ score for all instances of each entity is provided followed by the overall QI of all instances of all entities. The output provided is based on MIQI and MEQI. As visualisations of the DQ, we present the number of constraint violations per entity type in a pie chart (VEPC) or bar chart (VEBC), and a graph showing the constraints per weight interval (CWRC) in the other tabs. The latter is depicted in Fig. 4.4 on the left hand side where five importance weight ranges are plotted on the x-axis.

The leftmost (0-20) represents the least important weights, while the rightmost (80-100) represents the most important weights. The y-axis represents the number of constraints where the lower parts of the stacked bars (green) represent the number of satisfied constraints, while the upper parts (red) represent the number of violated constraints. Thus, it is easy for the user to see the number of constraints violated within the highest importance range on the rightmost bar. The pie and bar chart allow a user to easily see the number and percentage of constraint violations categorised by the different entities, as depicted on the left side in Fig. 4.5 and Fig. 4.6, respectively. With these charts, a user can easily identify entities with the most constraint violations.

For all of the charts, the user can click on a part of the chart to get more information. For example, in Fig. 4.4, we show two data panels to the right of the chart. The one in the middle shows information about the total number of violations. In the case of CWRC, the items represent violations per weight range, which are sorted by entity,

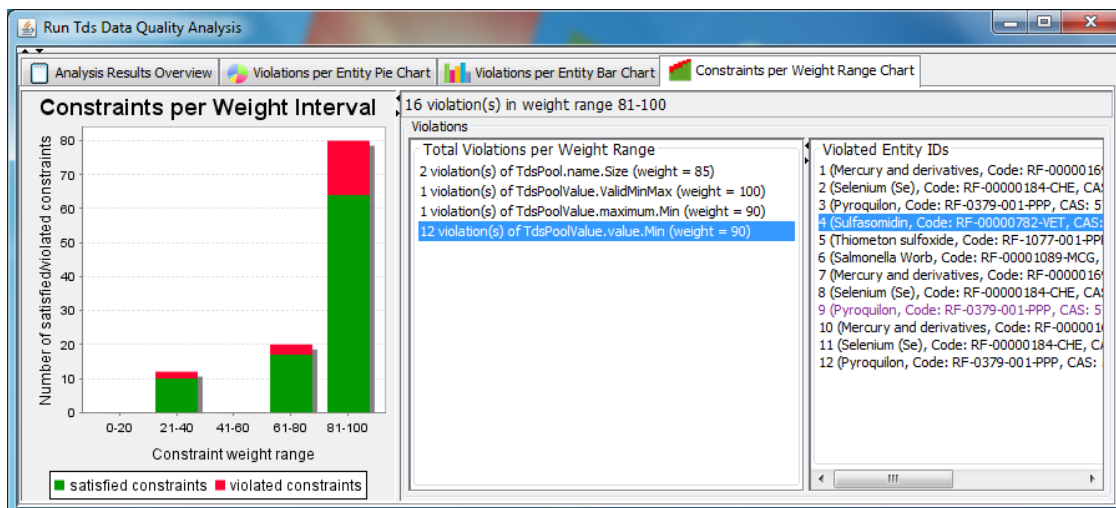


Figure 4.4: Constraints per weight interval

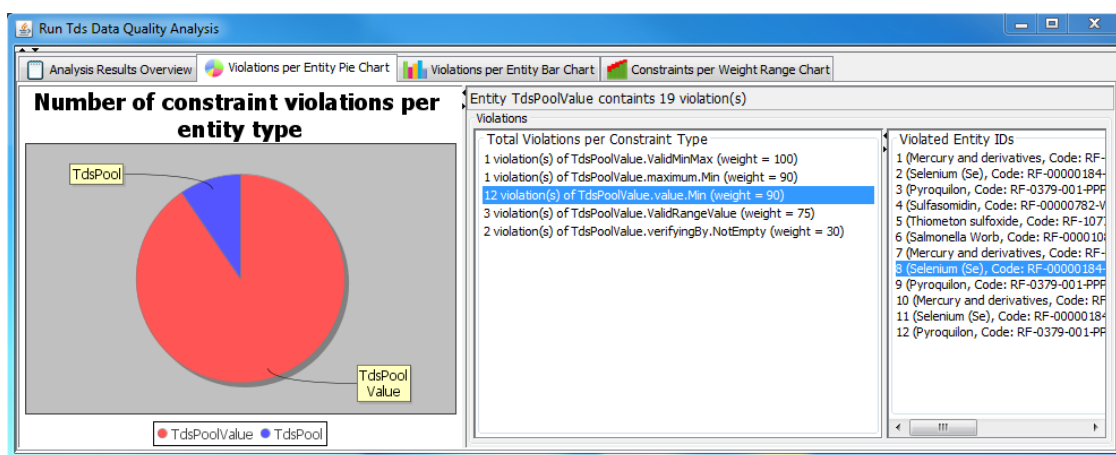


Figure 4.5: Percentage of constraint violations as pie chart

constraint and weight. In the case of VEPC and VEBC, the items represent violations per constraint type, which are sorted by constraint and weight. If the user clicks on an item in the middle panel, they can drill down to the concrete data records containing the constraint violations by double-clicking on items in the list of corresponding entity instances displayed in the rightmost panel. The data record can then be edited, enabling the user to access and correct the data record, thereby improving DQ in a straightforward manner. If a data record has already been opened, the item changes its colour in the list to violet as depicted in Fig. 4.4 for the item with ID 9.

OIQI is used for DQIF, while MIQI and MEQI are used for the textual feedback in DQAF. Note, however, that the necessary calculations for CWRC, VEPC and VEBC in DQAF are independent of those of QI.

By providing the DQ feedback during input validation as well as during data analysis, as presented above, we fulfil requirement (7) defined for this approach.

4.2.1 Validation and feedback configuration

Our administrative tools module, which is only accessible by power users, provides interfaces to configure the various validation settings, such as contract creation and

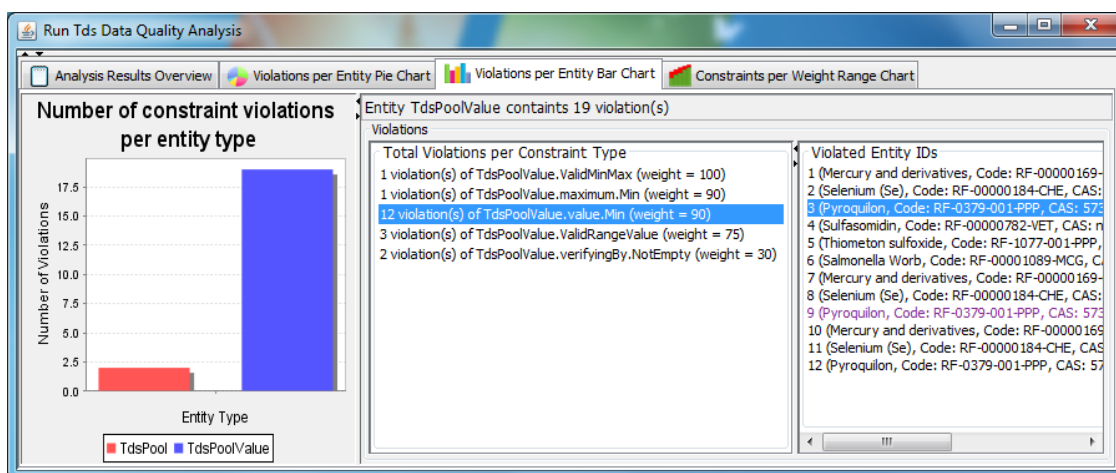


Figure 4.6: Percentage of constraint violations as bar chart

editing, general validation and analysis settings (use of constant importance weights, default constraint weight, error threshold) and input validation settings (active contract and validation groups, enabling of warnings). It consists of three parts which we will describe briefly in the remainder of this section.

With a first panel for contract editing and creation, depicted in Fig. 4.7, we provide users with the functionality to edit existing contracts and create new ones.

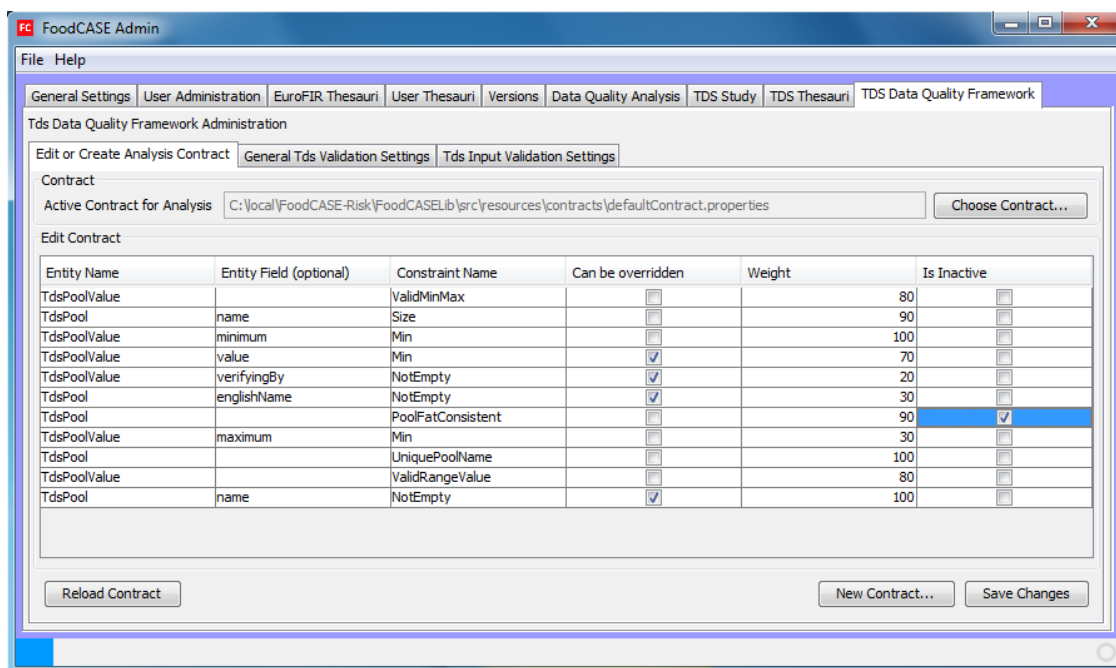


Figure 4.7: Contract editing and creation

To edit a contract, a user can select in the upper part one of the existing contracts. The contract settings are then displayed in a 6-column table presented in the centre of Fig. 4.7 where each row represents a constraint of the contract. The first four leftmost columns identify the characteristics of the constraint which are not editable. They provide information about the name of the entity where the constraint is defined, the corresponding entity field (if applicable), the name of the constraint and if a constant

importance weight is defined. The two rightmost columns are editable by the user and provide the possibility to define the individual constraint weight and to activate or deactivate the individual constraint checking. By pressing the corresponding button at the bottom, a user can create a new contract. After choosing a name for the contract, the table shows all existing constraints with blank weights where all constraints are initially activated.

With a second panel, depicted in Fig. 4.8, we provide users with the functionality to configure general settings for the validation and analysis.

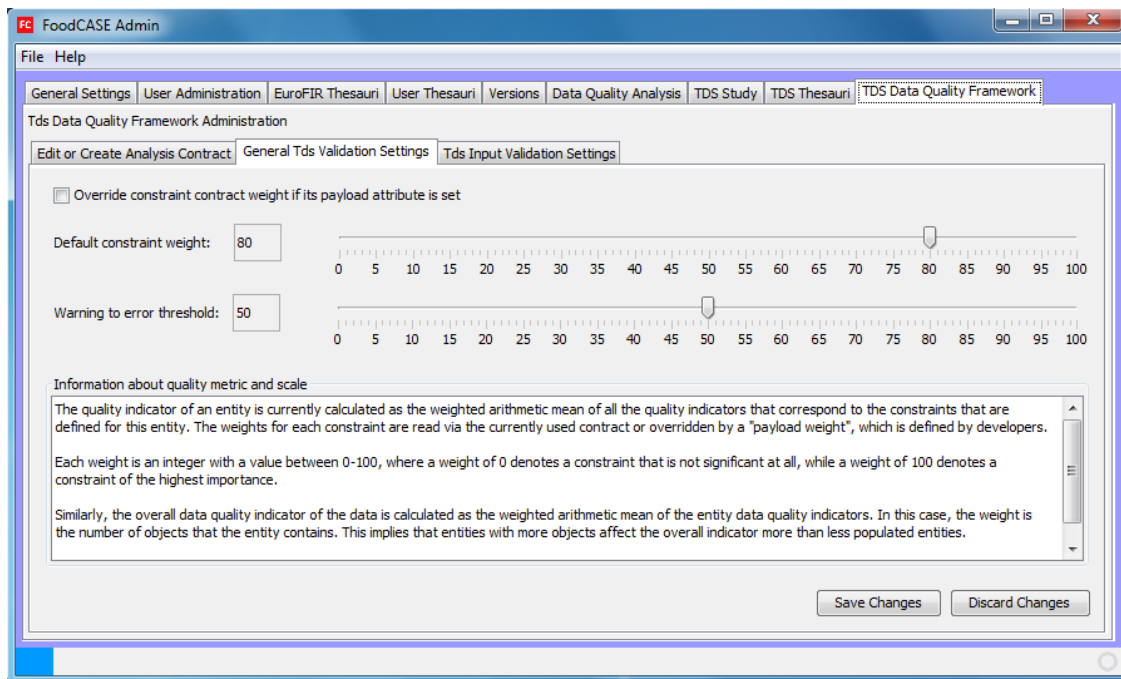


Figure 4.8: General validation and analysis settings

Here the power user is able to edit general settings which apply to both the analysis and input validation. At the top, it can be decided whether the constant importance weights should override the given weights in the contracts. Below, the user can configure the default weight which is taken for DQ calculation if neither a constraint weight is defined in the contract nor given as a constant importance weight. Further below, the warning to error threshold can be defined.

With a third panel, depicted in Fig. 4.9, we provide users with the functionality to configure settings for the input validation.

The power user can select a contract at the top that is then used for the input validation. On the left side, the entities can be chosen to which the input validation is limited. Finally, at the bottom, it can be configured whether warnings should be displayed during input validation, as described above, or if they should be omitted and only errors should be displayed.

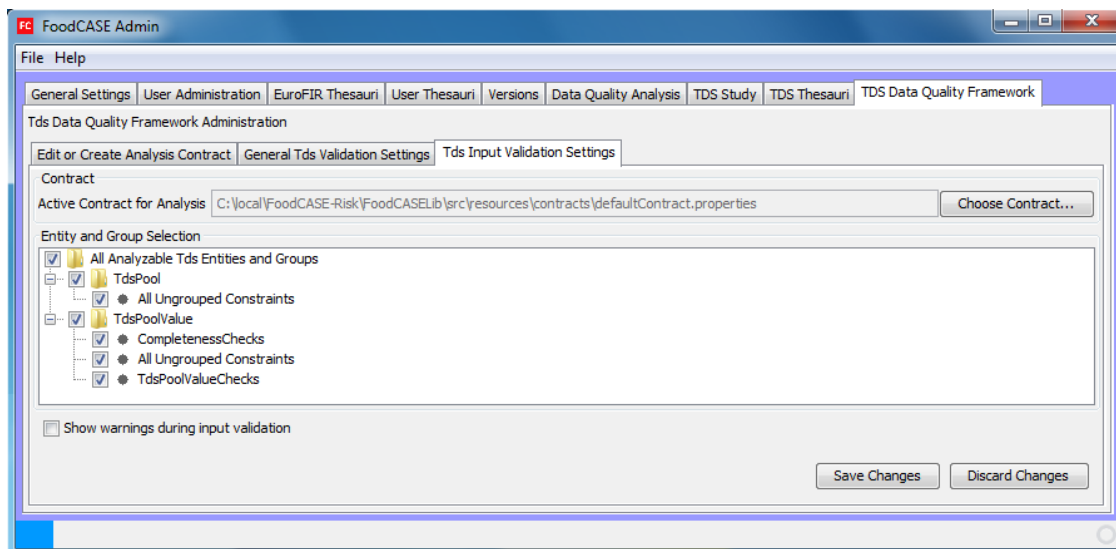


Figure 4.9: Input validation settings

4.3 Implementation

EJB is a server-side component architecture for the Java Platform Enterprise Edition. The EJB model enables the encapsulation of the business logic of an application. As our application FoodCASE is designed, based on the EJB model, the business logic of the DQMF is implemented as Stateless Session EJBs.

For BV, we use the Hibernate Validator (HV) ⁵¹, which is the reference implementation for BV 1.1. At the point of implementing the DQMF, it was the only implementation that supported BV 1.1 and provided features such as validation groups and payloads, which we could directly use for our validation groups and constant important weights, respectively. A list of built-in constraints can be found in the HV reference guide.

Our **constraints** are implemented exclusively as Java annotations in our EJBs where classes and fields may be annotated. An example of a constraint declaration is depicted in List. 4.1 in which the annotation of the declared constraint is `@Min` with four attributes specified: `value`, which is mandatory for this constraint and `message`, `groups` and `payload`, which are always part of any constraint's definition but can be omitted in its declaration. Besides using the HV built-in constraints, it is also possible to implement custom constraints which can range from relatively simple class-level constraints to more complex ones involving checks in multiple entities. Class-level constraints are particularly useful when the validation depends on a correlation between several fields of the object.

¹<http://docs.jboss.org/hibernate/validator/5.1/reference/en-US/html/>

```
@Min(  
    value = 0,  
    message = "The value must be greater than or equal to {value}.\n",  
    groups = MinimumValueChecks.class,  
    payload = Severity.SEVERE_ERROR.class  
)
```

Listing 4.1: Constraint definition

Our **contracts** are created as `.properties` files in which each property represents a constraint with its assigned importance weight. Contracts can be created and edited either via an administrative GUI, as previously presented, or by creating or editing the contract files directly using a text editor.

If we assume the `Min` constraint mentioned above for the attribute `age` of a `Person` entity, the corresponding **active** contract entry for a single QI with an importance weight of 60 would be defined as:

```
Person.age.Min=60, active
```

We chose to define contracts as `.properties` files instead of persisting the corresponding information in the database in order to ensure that the contracts can be shared easily amongst users and applications.

Validation groups are defined as simple tagging interfaces, i.e. interfaces without any methods. HV provides the default group `javax.validation.groups.Default`, which contains all the constraints where no other group is specified. Custom groups can be defined and added to the `groups` attribute in the constraint definition, as presented in List. 4.1, which can then be used for configuring the input and analysis validation during runtime.

The concept of payloads is part of the BV specification where they are typically used to associate metadata information with a given constraint declaration. In our framework, payloads have been repurposed as a way of specifying **constant importance weights**. In our `Severity` class, we predefined six distinctive payloads as interface extensions of the HV `Payload` tagging interface. Each of these payloads corresponds to a different predefined importance weight (`Payload=weight`): `Info=0`, `Suggestion=20`, `Recommendation=40`, `Warning=60`, `Error=80`, and `SevereError=100`.

An example of a constant importance weight definition is depicted in List. 4.1 with `payload = Severity.SEVERE_ERROR.class`

where the `Severity.SEVERE_ERROR.class` corresponds to the predefined importance weight 100.

Our business logic, which is additional to functionality already provided by BV such as custom constraint and constraint group validation, quality analysis and calculation of all types of QIs, is implemented in an EJB session bean and a few helper classes. To create new contracts, for example, we provide the logic to scan all analysable entities with BV annotations for declared constraints and create a new contract `.properties` file with no weights predefined. Not all functionalities are described here and, for the interested reader, we refer to [67, p.40-50] where a detailed description is provided.

4.4 Evaluation

As part of an annual General Assembly meeting of the TDS-Exposure project, we organised a workshop in February 2015 with several food science data experts from different countries in Europe. The participants had a variety of roles including food compiler, project coordinator, exposure assessment researcher, food database manager, research assistant, postdoctoral researcher, and PhD student. The half-day workshop consisted of an explanation of the main concepts and ideas behind the framework, followed by a demonstration of the three modules for input validation, DQ analysis and administration in our information system FoodCASE. In order to evaluate how our DQMF concept and implementation was perceived by its target users, and understand ways in which it could be improved, the participants were asked to complete a questionnaire about several aspects of the framework at the end of the workshop. The questionnaire was answered by 15 of the participants, which, although a relatively small number, represents experts from 5 of 7 European countries that have experience in this particular project. In this section, we present the most interesting results revealed by our analysis of the questionnaire answers. The full list of questions contained in our questionnaire are presented in Appendix A and a detailed evaluation of each single question is provided in Appendix B.

Contracts: Around 90% of the participants endorsed the concept of assigning importance weights to constraints and rated the scale (0 to 100) as intuitive. Almost 50% considered having different contracts for different users and applications of the system useful, while no one disagreed. 60% welcomed the opportunity to share and exchange contracts with other users, with only around 7% disagreeing. Being able to deactivate and activate constraints at runtime had both supporters (40%) and opponents (30%).

Constraint definition: For most of the participants, it was acceptable for new constraints to be defined by developers with around 13% disagreeing. While 40% thought that the definition of constraints at runtime was not necessary, around 27% disagreed.

Groups: Most participants liked being able to organise constraints into groups and rated it as acceptable if constraints could only be assigned to groups by developers. Nonetheless, 20% believed that assigning constraints to groups should be available to users too and around 53% would like to be able to define customised groups of constraints during runtime.

Constant importance weights: Only 40% of the participants rated the constant importance weight as useful with almost all the remaining participants being neutral. Almost 70% did not mind if only developers could define the constant importance weight, but around 27% disagreed.

Error threshold: Almost 70% of the participants believed that only authorised users should be able to change the error threshold.

DQIF: All of the participants considered having a QI useful and around 73% liked also having a visual representation of quality during data input. Around 87% agreed that, when an important constraint is violated, it should not be possible to store the data, while 80% agreed that storing should be possible if less important constraints are violated. When violations occur, 80% would like the system to provide an example of appropriate data and only around 13% would like it to insert the most probable value by default.

DQAF: 80% of the participants rated the overall DQ analysis as useful and almost 70% as desirable that the DQ analysis is available to any user. Many also welcomed being able to select the entities (60%) and constraint groups (73%) to be analysed. While the participants like having both a textual DQ representation with QIs (53%) and a visual DQ representation (80%) of the results, they indicate a greater preference towards the visual representation. Overall, the results of the questionnaire were very encouraging for our framework, as most of the findings indicated that the participants of the workshop agreed with the majority of the framework's aspects. The answers also indicated that some participants are sceptical about the degree of freedom that should be available for the analysis and input validation, and would prefer to restrict access to important settings to power-users only. Finally, most of the participants were satisfied if constraints, constant important weights and validation groups could only be defined by developers, while a minor, but significant, proportion of the participants would prefer to be able to define these during runtime.

4.5 Conclusions

With the approach presented in this chapter, we have introduced a novel concept and framework for managing DQ in information systems. We understand DQ as a constraint problem, which means that we measure and quantify DQ in an information system based on the degree to which data complies with certain constraints. While this approach cannot cover all aspects of DQ, it enables constraint checking methods to be used to simplify the evaluation of DQ for a significant proportion. Our concept takes into account that not all constraint violations should be considered equal when it comes to calculating DQ. We, therefore, introduce the notion of contracts that allow weights to be assigned to constraints that are then used to calculate DQ. This is done in a highly flexible way, allowing contracts to be defined on a fine-grained scale, adapted at runtime and shared between different users and applications. Further, an error threshold is used to distinguish warnings from errors. By using our concept and framework, (1) developers can benefit from unified constraint definition and validation to avoid constraint inconsistencies and maintenance overhead, (2) data consumers can benefit from flexible configuration, allowing them to decide exactly what they wish to validate and analyse as well as how important each constraint should be and (3) data consumers can benefit from our novel DQ measurement and calculation scheme that results in a quantitative representation of DQ.

We note that the framework is extensible, enabling the proposed DQI calculations to be replaced if required and new visualisations to be added for data input and data analysis feedback. Moreover, our contracts could be extended with additional para-

meters such as the custom importance of entities (θ) for the DQI calculations. On the implementation side, so far, we have only used bean constraints but it would be possible to expand the framework to also support method constraints.

The implementation of our framework is actually embedded in our information system FoodCASE which is why the evaluation was done using our system. Consequently, future evaluation should also be done by applying our concepts to other information systems and conducting studies with a larger group of participants. Our validation groups can be used to link requirements expressed as constraint definitions with DQ dimensions which can then be included for the DQ feedback. Nonetheless, further investigation should be done in mapping between constraints and concrete dimensions, for example mapping a ‘not null’ constraint to the completeness dimension.

For many information systems, it might not be desirable that constraints are defined only in the business layer if, for example, they should also inhibit invalid data or low quality data from being entered directly into the database. Nonetheless, it is desirable that, if possible, the definition and maintenance of constraints is done in a unified way in a single place, such as with BV. Thus, another focus of our work was to enable developers to generate various constraint representations, such as BV annotations, SQL, JavaScript or Business Rules, from a single unified representation which we describe in the approach presented in the subsequent chapter.

5

System-wide constraint representations with UnifiedOCL

In the previous chapter, we have presented an approach based on a single technology (BV) that is used for managing constraints in all components of an information system. There, the presented DQMF is dependent on the information system where the unified constraint definitions are an integrated part of the application components. This is a convenient solution, if feasible, but as already motivated in Sect. 1.2.1, constraint definitions also tend to be distributed across the components of an information system using a variety of technology-specific representations to guarantee consistency and data correctness in databases, GUIs and business-logic components. Therefore, in this chapter, we propose an approach and unified constraint representation that enables software developers to manage all constraints in a single place based on OCL with extensions for technology-specific concepts. These constraints are then mapped to the technology-specific representations which can be validated at runtime. Since the unified constraint definitions are clearly managed separately from the information system, the DQMF, presented in this approach, is separated from the information system.

To achieve that constraint definitions can be managed in a single place and are automatically mapped to component-specific implementations for runtime validation, we have developed an approach in line with MDA where a technology-independent model serves as a basis for technology-specific code generation. In our model, constraints are expressed in our *UnifiedOCL*, a DSL which extends OCL with capabilities to represent technology-specific constraints. To cater for the integration of different technologies, the grammar of *UnifiedOCL* is extensible by means of so-called *labels* which are grouped into dictionaries. Furthermore, our concept enables the constraints defined in *UnifiedOCL* to be associated with violation penalties and mapped to specific DQ dimensions.

We studied alternative approaches and technologies for bi-directional translations between various models and representations. This resulted in an extensible toolkit capable of efficiently translating constraints to and from *UnifiedOCL*. As a consequence,

the system allows for translation from any source representation to any target representation, which we call multi-translations. Our proof-of-concept implementation supports three technology-specific representations: relational databases (*SQL*), an object-oriented language (*Java*), and business rules (*Drools*), which let us explore and cover a broad range of constraints.

Consider the example from Sect. 1.2.1, where the `Food` entity `name` attribute of the type `String` must be at least 3 characters long. The appropriate technology-specific representations for *SQL*, *BV* and *Drools* are given in List. 5.1, 5.2 and 5.3, respectively.

```
CREATE TABLE Food (
  name VARCHAR NOT NULL,
  CONSTRAINT ck_food_name CHECK (length(name::text) >= 3)
);
```

Listing 5.1: SQL translation example

```
package org.example;
public class Food {
  @Length(min = 3)
  public String name;
}
```

Listing 5.2: Java BV translation example

```
rule "Food_name_minlength"
when Food (name.length < 3)
then
  ...
end;
```

Listing 5.3: *Drools* translation example

These three diverse technology-specific representations are used in different components of an information system, for example in the data tier, logic tier and presentation tier, specifying the same constraint. With our non-technology-specific representation in *UnifiedOCL* and our constraint translations, it is possible to define the constraint once in a unified way and generate all required technology-specific representations accordingly. Further, there is no need of a separated domain model for constraint definitions, as in *OCL*, since the model is also part of the *UnifiedOCL* definition. Constraints in *UnifiedOCL* can be defined with pure *OCL* syntax or with our labels. The attribute length constraint exists in *OCL* (as `size`) and therefore the minimum length constraint of the `name` attribute from the example above could be defined in *UnifiedOCL* with *OCL* syntax as

```
invariant NameMinLength : self.name -> size() > 2;
```

which can be considered as a convenient way of defining a constraint. With our labels, we provide even more convenient definitions, also for constraints which do not exist in *OCL* as such. Consider, for example, the email constraint of an `email` attribute for a `Person` entity, as described in Sect. 2.4.5, which ensures a valid email address. The appropriate constraint in *OCL* syntax could be defined as

```
invariant Email:
  self.email.matches('^([a-z0-9_+-.]@[a-z0-9-\.]+\.([a-z]{2,4})$')
```

which we consider as an inconvenient way of defining a constraint due to the complex pattern definition, especially if it has to be defined multiple times for different attributes and/or entities. In the *HV* implementation for *BV*, a specific notation `@Email` exists

for an email constraint which we also provide in *UnifiedOCL*. Thus, the appropriate constraint in *UnifiedOCL* could be defined in a very convenient way with the help of our label `email`, as depicted below. Where the label is simply included in curly brackets after the type.

```
public attribute email: String {email};
```

For a primary key constraint, which exists in SQL but does not exist as such in OCL, the according constraint in *UnifiedOCL* could be defined with the help of our label `primarykey`, as shown below.

```
public attribute id: Integer {primarykey};
```

By providing bi-directional translations of constraint definitions between our *UnifiedOCL* representation and any technology-specific representation, existing components can easily be integrated into the system and developers are not required to use our representation. We present an implementation of the approach along with the results of a user study of the resulting system carried out with developers in industry and research.

The work presented in the remainder of this chapter is an extended version of a previous publication [193].

5.1 Approach

The goals of the approach presented in this section are: (1) To support Model-Driven Software Development with regard to constraint definition; (2) To provide a constraint representation from which various technology-specific constraint representations can be generated that can be validated at runtime; (3) To provide a multi-translation mechanism which allows any constraint representation within an information system to be translated into another, and which is also easily extensible; (4) To support DQ by providing an association between constraint definitions and constraint violation penalties related to DQ dimensions.

With (2) we enable all constraints to be managed in a single place, guaranteeing constraint coherence and reducing the effort of defining and maintaining constraints across components and multiple layers in an information system. With (3) we support the translation of legacy constraint representations and allow developers to choose their main constraint representation.

Fig. 5.1 presents an overview of this approach. The various technology-specific constraint representations from the example discussed in Sect. 1.2.1 are depicted at the bottom (BV, SQL, *Drools*). Each representation could be used in one or more components of the information system.

To fulfil (2), we propose a unified non-technology-specific assertion representation (*UnifiedOCL*) depicted in the centre of Fig. 5.1 in combination with individual translations to the technology-specific representations. The syntax and constraint representation capabilities of *UnifiedOCL* will be presented in Sect. 5.2, while details of the constraint translations will be given in Sect. 5.3.

We created *UnifiedOCL* to overcome limitations of constraint representation imposed by the OCL syntax with regard to technology-specific representations as explained in Sect. 2.5.5. It is implemented as a new DSL and designed as a textual

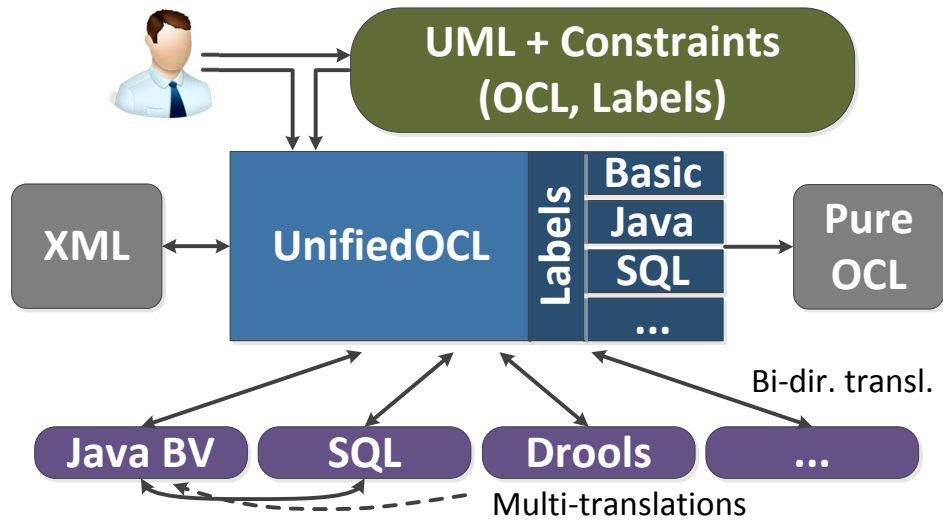


Figure 5.1: Approach overview

representation which should provide human readability also for non-programmers while enabling efficient processing for our translation needs. *UnifiedOCL* combines structural domain information (UML), constraint definitions from OCL and additional constraint definitions, so-called labels. Thus, the model is a part of the *UnifiedOCL* definition. Labels provide a convenient way of defining constraints, which are either not available in OCL (such as primary key in SQL) or not convenient to define in OCL (such as email and credit card constraints). With *UnifiedOCL*, developers can define a great variety of constraints in a textual representation, with the labels providing a compact way of specifying common constraints by simply labelling elements without providing details of the validation implementation.

Having the notation of *UnifiedOCL* already supports MDSD, but, to fulfil (1), we also introduced the conversion of a UML model including constraint definitions to our *UnifiedOCL* representation. It is therefore possible to model an information system using a UML editor and enhancing the model with constraints defined in the standard OCL notation or with our labels notation which offers a seamless integration with UML.

The UML model enriched with constraints is depicted at the top of Fig. 5.1. According to MDE, this model constitutes the PIM. To convert the PIM to PSMs, a developer has two options: One option is to stick to the UML graphical representation enhanced with constraints and convert it first to *UnifiedOCL* and then to the desired representation(s). Another option is to choose the textual representation of *UnifiedOCL*, which is also a PIM, and convert it in a single step to the desired representation(s).

This unified approach also enables (3) to be fulfilled by additionally introducing individual translations from all technology-specific representations to *UnifiedOCL* as indicated by the bi-directional arrows in Fig. 5.1. To add an extension for a technology-specific representation, only one additional bi-directional translation to and from *UnifiedOCL* has to be provided in order to achieve multi-translations as depicted at the bottom of Fig. 5.1. A multi-translation is always carried out with a translation to and from *UnifiedOCL*.

Furthermore, it is possible to translate our *UnifiedOCL* representation to pure OCL and exchange *UnifiedOCL* in an XML format.

With *UnifiedOCL*, we were focussing on system-wide constraint representations but

also integrated concepts of relating constraint definitions/violations to DQ feedback, as already proposed in the initial approach presented in Sect. 3.5.1. Consequently, our approach also fulfils our goal (4).

5.2 UnifiedOCL

UnifiedOCL specifies both the constraints and the data structure which provides the context in which constraints exist, such as an SQL table schema or a Java class.

An example of *UnifiedOCL* is presented in List. 5.4 where the OCL syntax together with our concept of labels are used to define constraints.

```
package org.example {
  public abstract class Employee {
    public attribute id: Integer {primarykey};
    public attribute name: String {notnull};
    public attribute birthDate: Date {past};
    public attribute email: String {email};
    public attribute salary: Real {range(min=2500, max=5300)};
    public attribute bonus: Real;

    public reference subordinates: Employee[0..*]{unique};

    invariant BonusNotExceeded: self.bonus<self.salary;
    invariant PersonId: not self.id.oclIsUndefined() and self.allInstances()
      ->forall(a : Person|a<>self implies a.id<>self.id);

    public operation checkIn (time: in Date,
      checkedIn: inout Boolean): String {
      precondition NotAlreadyCheckedIn: checkedIn=false;
      body CheckInBody: checkedIn=true;
      postcondition SuccessfullyCheckedIn: result=true;
    };
    ...
  }
}
```

Listing 5.4: Example of *UnifiedOCL*

In List. 5.4, the data structure of a class `Employee` is given in a certain package (`org.example`) with the attributes (`id`, `name`, etc.) of various data types (`Integer`, `String`, etc.). *UnifiedOCL* labels are used to define constraints associated with these. For example, the attribute `id` is labelled as a `primarykey` based on the primary key concept from SQL. The labels `past` and `email` define pattern matches that are not supported in OCL, while the `notnull`, `range` and `unique` labels provide a convenient way of defining the appropriate constraints. The `past` label represents a time-related constraint which ensures that a date is in the past, such as the date of birth for a person.

Additionally, the `reference` (`subordinates`) has a label `unique` specifying that duplicates are not allowed in the collection along with cardinality constraints specifying that any number of subordinates can exist. The cardinality constraints syntax such as `[0..*]` is part of the *UnifiedOCL* grammar. It defines in this example that an `Employee`

can have zero or more `Employees` as subordinates.

In contrast, the `invariant` constraints `BonusNotExceeded` and `PersonId` are defined using OCL notation. The `BonusNotExceeded` invariant defines that the bonus of an `Employee` has to be less than the salary of an `Employee`. The `PersonId` invariant defines that the `id` of an `Employee` cannot be undefined and that the `id` must be different to the `ids` of all other `Employee` instances. This is a deliberately included duplicate definition of the `primarykey` constraint defined as a *UnifiedOCL* label on the public attribute `id` just to emphasise the convenience of using *UnifiedOCL* labels over OCL notation.

UnifiedOCL also allows directionalities (`in`, `out`, and `inout`) to be specified for the parameters of operations as shown in the definition of the `checkIn` operation. The `time: in Date` defines that the parameter `time` of the type `Date` can be used for input only in the method. The `checkedIn: inout Boolean` defines that the parameter `checkedIn` of the type `Boolean` is used for input and output (the result) in the method. In addition, pre-, body- and post-conditions are defined for this operation, which means here that the `checkedIn` parameter has to have the `Boolean` value `false` when entering the method and `true` within the method body and at the end of the method.

As seen in the example of List. 5.4, constraints can be specified either using OCL syntax or our *UnifiedOCL* labels in cases where the constraint cannot be expressed in OCL or for reasons of convenience. Labels not only provide a shorthand for specifying common forms of constraints, but also provide a way of extending the language with technology-specific concepts. Thus, each time support for a new technology-specific representation is integrated, the set of labels may need to be expanded. This is not the case, if all of the constraints in the new technology-specific representation are already included in the set of labels and the appropriate OCL representation is inconvenient. Many technology-specific representations use a similar set of constraint definitions, such as ‘not null’, ‘size’, ‘pattern’, and ‘maximum’ constraints. Therefore, new labels would only have to be introduced if completely new constraint definitions would be introduced with an additional technology-specific representation. We note that the use of the OCL syntax to define constraints is especially useful where constraints are easily expressed using imperative statements or involve graphs of interrelated objects.

The OCL expressions are embedded into the data structure representation of *UnifiedOCL* in the same way as in *OCLinEcore*[137]. They are specified within the scope of *classifiers*, *methods* or *structural features*. Therefore, the original OCL scope specification – the *context* clause – is no longer needed. As a result, the use of the OCL syntax is significantly simplified. To specify OCL constraints within the body of one of the allowed data structure representation components, one just needs to use the keyword (type of the OCL statement), optionally followed by its name, and the body of the OCL expression. The syntax of the *UnifiedOCL* labels can be expressed using *EBNF notation* as follows:

```
UNIFIEDOCL-EXPRESSION := '{', { label-name,
                               [ '(', {PARAMETER-EXPRESSION}, ')' ]
                               }, '}'
PARAMETER-EXPRESSION := parameter-name, '=',
                        parameter-value, [ ', ' ]
```

where `label-name` and `parameter-name` are *unquoted strings*, i.e. just a name or an identifier, and `parameter-value` is one of the following literals: an integer number, a

real number, a Boolean value, or a string (unquoted or single/double quoted).

For standard BV, we provide labels for the following constraints: @AssertTrue, @AssertFalse, @DecimalMin, @DecimalMax, @Min, @Max, @Size, @Future, @Past, @Digits, @Pattern, @Null, and @NotNull. For specific HV constraints, we provide additional labels for the following constraints: @NotEmpty, @NotBlank, @Length, @Range, @Email, @CreditCardNumber, @EAN, and @URL. Moreover, we provide labels for the following SQL constraints: PRIMARY KEY, FOREIGN KEY, NOT NULL, NULL, UNIQUE, SERIAL, column type NUMBER, column type VARCHAR, and CHECK. A list of all supported labels with their parameters can be found in Appendix C.

When specifying constraints in *UnifiedOCL* using labels, the developer only has to name commonly used constraints and not care about the implementation in terms of specific, potentially complex, validation code. The technology-specific representation and validation of these constraints is the concern of the constraint translation which is discussed in the subsequent section.

UnifiedOCL is based on the *OCLinEcore* syntax which allows an *EMF Ecore model* [172] to be combined with OCL statements. The *OCLinEcore* grammar extends the *EssentialOCL*[55] grammar, which allows OCL expressions to be specified but has no relation to the *Ecore* (or any other) data model. The *OCLinEcore* language and the *Ecore* metamodel fitted already well for our needs, but we aimed for *UnifiedOCL* being more suitable for UML concepts than for *Ecore* ones and further were in need of additional components to fulfil our requirements as explained below.

Thus, we derived *UnifiedOCL* by extending the *EssentialOCL* language and re-using the *OCLinEcore* language components and *Ecore* metamodel. We changed the metamodel by extending *OCLinEcore* concepts with attributes or relationships required to reflect the structure of a UML class diagram. For example, this involved introducing (1) a grammar element to reflect the concept of encapsulation levels (modifiers `public`, `private`, `protected`, `package`), (2) simple exceptions, (3) a directionality meaning for method parameters (`in`, `out`, `inout`), (4) user-defined primitive types, (5) a `Date` built-in primitive type aimed to store time and date related values (not supported in *OCLinEcore*), and (6) a grammar element to allow operations to be specified as multithreaded. Additionally, and most important, we introduced in the *UnifiedOCL* metamodel support for the special constraint representation labels described above. We further extended the metamodel to support the association of constraints to DQ dimensions with appropriate constraint violation penalties as described below.

To support DQ, we extended the *UnifiedOCL* syntax in order to represent DQ mappings directly in *UnifiedOCL*. With the following grammar, it is possible to represent any mapping from a *label* or OCL invariant, respectively, to DQ dimensions and constraint violation penalties.

```
DQ-MAPPINGS_DEFINITION := '[' , DQ-MAPPING,
                          '[' , ']' DQ-MAPPING } , ']'
DQ-MAPPING := dimension-name, penalty-value
```

This grammar element can be placed directly after the definition of a *label* or the invariant specification, respectively, as shown in the example below.

```
public attribute salary : Real { range(min=2500, max=5300)
  [Correctness 0.2] } ;
```

Consequently, our framework is able to generate mapping file stubs for a given specific technology (e.g. Java BV annotations) as part of the translation from *UnifiedOCL* to the concrete technology-specific representations. The mapping file stubs can then be used for an appropriate mapping file (`mapping.dqm`). The constraint violations from the translated technology-specific representation can then be sent, for example, to a DQ monitoring framework such as the one presented in our initial approaches in Sect. 3.5, where the information from the mapping file is used for the DQ calculations and appropriate feedback on DQ is given.

In this section, we explained the language components available in *UnifiedOCL* by focussing also on extensions to *EssentialOCL* and *OCLinEcore*. For the complete detailed language definition, we refer the interested reader to [173, p.27-48].

5.3 Constraint translations

We distinguish between a *non-technology-specific constraint definition* and a *technology-specific representation*. In *UnifiedOCL*, the only information required about a constraint is the non-technology-specific constraint definition and its particular related component(s) of the data structure. A technology-specific representation is dependent on the particular programming language where it is used as validation code. Our bi-directional translations between *UnifiedOCL* and the technology-specific representations are concerned with generating the validation code from the non-technology-specific constraint definition and vice versa.

We will first show an example of the various representations before delving into the details of the translation process. For the example from Sect. 1.2.1, we could model a *Food* entity with a *name* attribute of type string in a UML editor, add a constraint that the string length must be at least 3 and then translate this to *UnifiedOCL* from which technology-specific representations could be generated.

Thus the non-technology-specific constraint definition would be represented in *UnifiedOCL* as shown in List. 5.5.

```
package org.example {
  public abstract class Food {
    public attribute name: String {length(min=3)};
  }
}
```

Listing 5.5: *UnifiedOCL* translation example

The *technology-specific representations* for SQL, BV and *Drools* were already given in the beginning of this chapter in List. 5.1, 5.2 and 5.3, respectively.

All translations, except for pure OCL extractions, use the model-to-text (M2T)[195] approach and consist of three main steps: (1) Obtaining a traversable in-memory model instance from the source file(s); (2) Model discovery and analysis; (3) Performing a model to text serialisation.

To get an in-memory model such as an AST, we used three different approaches. In some cases such as *Drools*, an existing parser that accompanies the technology was used. For SQL, a parser was generated with the help of the modelling framework *Xtext*¹

¹<https://eclipse.org/Xtext>

which is based on the *ANTLR*² parser. BV is an example where a model discovery tool such as *MoDisco*³ was used.

Usually, it is not possible to generate an intermediary model which can later be serialised to text from a traversable model in a single pass. Therefore, serialisation to the technology-specific representation cannot run in parallel to model discovery. Frequently, language concepts are divided among various constructs or scopes, may be nested or embedded in some external scopes, and may include circular dependencies. In such cases, we need to analyse the entire model instance, gathering the relevant information which is then processed and used to produce the output. For example, constraints in SQL can be defined as part of a column definition, at the table level or as an `ALTER TABLE` statement defined completely outside of the table, possibly in a different script. They all can express the same constraint and we may not be aware of the existence of a constraint when it is defined in a different posterior location. Therefore, serialisation is postponed until after the entire intermediary model is discovered and analysed.

Once the intermediary model is complete, it has to be converted to the target textual representation. This is non-trivial since it requires knowledge about where constraints can be used in the model and how data structure definitions are affected by them.

In the following two paragraphs, we describe the bi-directional translation process in more detail before presenting the remaining three translations from UML to *UnifiedOCL*, OCL extraction and the XML conversion.

When analysing a source model, we identify constraint occurrences with regular expressions and create Abstract Constraint Representations (ACRs) which contain all the information about the constraint in a technology independent way. Therefore, we introduced dictionaries that define which positions of a regular expression string correspond to which parameters of the constraint. An ACR is the intermediary model and includes the constraint location, the type of constraint, a constraint name and the constraint parameters and values. To generate a target representation, we use specific Target Representation Producers (TRPs) which are also defined in dictionaries and able to serialise the ACRs to the target textual representation.

If we consider the BV example from List. 5.2, the technology-specific constraint `@Length(min = 3)` would be matched with an abstract *size* constraint from our `Java2UnifiedOCL` dictionary. The resulting ACR would then have the location `org.example.Food.name`, the type *size*, a unique name and the *lowerBound* parameter with a value of 3. Our `General2UnifiedOCL` dictionary can be used to translate any ACR to a specific *UnifiedOCL* representation. Thus, it would be used here to generate the representation in List. 5.5. If we would consider the opposite translation from *UnifiedOCL* to BV, our `UnifiedOCL2Java` dictionary would be used for creating both the ACR and the specific Java representation.

There exist many UML editors and, unfortunately, they do not all use the same UML file format. Our translation from the graphical UML representation (including constraints) to *UnifiedOCL* is based on the UML2 format⁴ which is the *Ecore*-based implementation of the OMG UML standard. It is stored in the XML Metadata Interchange (XMI) format, that uses the UML2 schema definition. We do not support

²<http://www.antlr.org/>

³<https://eclipse.org/MoDisco>

⁴<http://wiki.eclipse.org/MDT-UML2>

the reverse translation since it requires anticipation about the graphical aspects of the model (colours, positioning, layers, etc.).

Besides our translation from the graphical UML representation to *UnifiedOCL* and the bi-directional main translations between *UnifiedOCL* and the technology-specific representations, we also support two additional translations. The extraction of OCL statements from *UnifiedOCL* increases the usefulness of our system since there exist tools such as *DresdenOCL*⁵ which allow objects such as *JavaBeans* to be validated based on the pure OCL specification. Further, we ensure portability with a bi-directional conversion between the *UnifiedOCL* and the XML format.

5.4 Evaluation

To evaluate how this approach and implementation was perceived by potential users and find ways in which it could be improved, we conducted a user study with software engineers from both research and industry. We had 20 participants (75% male, 25% female) of whom 50% had at some time studied or worked in a university and 75% had at some time worked in industry.

The participants had to solve three tasks with the help of our tool. Before each task, any additional features of the tool relevant to the task were explained. A feedback questionnaire based on a 5-point Likert-type scale ('totally agree', 'agree', 'neutral', 'disagree' and 'totally disagree') was used. In the summary of results given here, we combine the first two values into 'agree' and the last two values into 'disagree'.

In the first task, participants were presented with a UML diagram showing two simple, connected entities without any constraints that had a total of eight attributes. They had to choose a representation from *Java*, *SQL* or *Drools* and manually convert the entity specifications into code. They were then presented with the same UML diagram with seven constraint definitions using the *UnifiedOCL* label notation and had to enrich their code with the corresponding code without providing an introduction to the notation. Finally, they had to generate the same technology-specific representation using our tool, analysing the generated code and editing it if it did not meet their expectations.

For the second task, participants were given a short introduction to *UnifiedOCL* and then asked to add a field to one of the classes from task 1 along with a specific constraint.

In the third task, participants were given one definition in *Java* and one in *SQL* with ten attributes and eight constraints and given the task of combining these to come up with a single representation in *Java*, *SQL* or *UnifiedOCL*. They were first asked to compare the definitions manually and then use the tool to perform any necessary translations to their choice of representation. With this task, we not only wanted to find out their preferences and test how well they remembered features of the system from the first task, but also to allow them to explore use cases and discover the potential of the system.

We now present a summary of the results and note that full details of the results are provided in [173, p.136-143].

⁵<https://github.com/dresden-ocl>

85% of the participants chose Java in task 1, with only 15% choosing SQL. Nobody used *Drools*. Only 30% of the participants were able to manually solve the task completely, while 70% solved it partially. Moreover, almost half of those who completed the task partially (43%) declared their knowledge in the selected technology as very good. The great majority of users (80%) declared their understanding of UML as at least good. Therefore, it is unlikely that the main reason for not solving the task completely would be the misunderstanding of the diagram, especially since it was very simple. The average time users spent on writing code in the selected technology was 185 sec (with a standard deviation of 70 sec). Enriching the representation with constraints took users on average 224 sec (with a standard deviation of 73 sec). The average time needed to solve the task using our tool was 46 sec (with a standard deviation of 21 sec). Taking into account the fact that the system was new to them, this is a good indication that the system could save developer time. Moreover, all participants agreed that the generated code met their expectations. Only five of them needed to slightly adjust the code by applying formatting, organising imports, or altering names. The great majority of the participants (80%) agreed that our tool solved all of the difficulties encountered during the manual code creation. All participants agreed that the tool was easy to learn and use. Also all agreed that it makes the process of code generation faster and nobody disagreed that it simplifies the process of the technology-specific code generation.

Task two was successfully solved by almost all participants (90% knew how to define the field and 85% the constraint). The reason for failing this task was misunderstanding what to do rather than not knowing how to do it. 90% of users agreed that the *UnifiedOCL* syntax is intuitive. Similarly, almost all (75%) agreed that they would remember the *UnifiedOCL* syntax required to solve this task in the future.

For task three, all participants knew how to translate files from the source representation into the target representation (80% totally agree, 20% agree). Moreover, we observed that almost all the participants knew immediately what action was required to execute the translation which shows a good level of memorability for the system. Also, there were no errors in the use of the system and none of the participants stated that they did not understand the concepts of constraint translations and our unified constraint model. 85% of users considered the manual comparison of two representations to be time-consuming, with 55% agreeing that it is not an easy task. 75% agreed that the translation between two representations is difficult, while 95% agreed that it is time-consuming.

Most users (60%) chose *UnifiedOCL* as the common format used for the comparison. They motivated their choice with statements such as ‘*UnifiedOCL* can represent all constraints whereas Java and SQL will have some problems with certain constraints’, ‘this translation is easier and more obvious’ and ‘this representation benefits from syntax highlighting with *UnifiedOCL* labels for a better perception of constraints’. This indicates that *UnifiedOCL* has a user-friendly representation and a convenient code editor. It is interesting to note that their choice of *UnifiedOCL* came after just a short introduction in task 2, which suggests that it is both intuitive and easy to learn. Most participants selecting the conversion to Java or SQL justified it in terms of familiarity, but one of them stated that *UnifiedOCL* would be the preferred choice if they had more experience with it.

95% of participants agreed that the system really helps in a comparison of various constraint translations (70% totally agree, 25% agree). 85% also agreed that transla-

tions would be helpful for solving various software engineering tasks, while 90% agreed that it would be helpful in system modelling.

All participants agreed that the tool was efficient and convenient to use, and 85% were satisfied with the results. Those who were not satisfied were missing some sort of order of the attributes in the target representation. We decided to maintain the same order of attributes as in the source representation, but nevertheless it is one of the extensions that should be considered since it was mentioned by seven participants (35%).

5.5 Conclusions

With the approach described in this chapter, we have presented *UnifiedOCL*, a unified constraint representation for constraints that is expressive enough to cover most concepts from UML class diagrams together with their constraints. A developer can define constraints using a mix of OCL syntax and *UnifiedOCL* labels which is a convenient way of labelling common forms of constraints, some of which cannot be expressed in OCL. Labels can be used to express constraints that are technology-specific such as those supported in BV, SQL and *Drools*. *UnifiedOCL* is extensible as new labels can be introduced at any time. Moreover, constraints defined in *UnifiedOCL* can be associated with constraint violation penalties and mapped to specific DQ dimensions. Therefore, *UnifiedOCL* provides the basis for DQ measures which can be used to calculate DQ scores and give feedback on DQ.

By providing constraint translation from UML to *UnifiedOCL* as well as bi-directional translations between *UnifiedOCL* and various technology-specific representations, we offer a technology-independent repository for constraint definitions that are validated in different components of an information system, thereby facilitating model-driven management of constraints. The bi-directional translations can be extended by means of plugins with new representations such as JML or *JavaScript*.

We also discussed a general architecture for translations, elaborating common steps required to construct a code generator for translating a constraint from one representation to another. We implemented all translations along with the *UnifiedOCL* representation as a set of *Eclipse* plugins. A user study supported our claims that the problems addressed by our system are important and can reduce developer effort.

So far, *UnifiedOCL* labels are applicable only to attributes, but they could be extended to entire classes, method parameters and return values. Further, our constraint translations generate complete target representations without regard to previously generated output. It could be useful to regenerate just parts of the output, so that existing pieces of code in the target representation could be preserved. This could be achieved by annotating parts of the source or target code specification, as well as the intermediate representation *UnifiedOCL*.

Currently, our implemented system relies heavily on Eclipse Modeling Framework (EMF) and hence is dependent on the *Eclipse* Integrated Development Environments (IDE). To use our system outside of the *Eclipse* IDE, it is possible to create a *Headless Build* which includes all the required Open Service Gateway Initiative (OSGi) bundles.

Finally, we note that, although our system provides a translation from *UnifiedOCL* to pure OCL, we do not support Round Trip Engineering (RTE) since we only handle

a unidirectional translation from UML to *UnifiedOCL*.

Another focus of our work was to also include implicit and explicit feedback of users to the quality of data items which enriches our concepts to cover additional aspects of DQ which cannot be covered by constraint compliance. We present such an approach in Chap. 7.

6

Data quality management in mobile applications

Smartphones and the use of mobile applications nowadays play an essential role in our modern world. Beyond dispute, users around the globe spend a large amount of time accessing information via their mobile devices. Often, mobile applications are dependent applications which communicate with other applications and transfer data to and from information systems. With this interdependence, and since requirements and constraints usually change over time, the question arises how sets of constraints, which exist in information systems and their dependent mobile applications, can be managed in a uniform way and kept consistent during software evolution.

Further, mobile applications shape new ways of accessing, displaying and editing information, since they usually provide simpler interfaces than classical desktop applications in order to provide convenient and appropriate data management for small screens. Nonetheless, managing data on a portable device must not negatively influence the DQ such as the correctness of input data. Thus, it is important, besides having a consistent constraint set, to provide appropriate feedback on DQ also for mobile applications.

In this chapter, we present an approach for synchronising constraint sets between a main and a mobile application and explore ways of giving feedback on DQ for portable devices. With this approach, a constraint set is managed in the main information system and a subset of this constraint set is injected into the dependent mobile application without requiring the developer to take care of it. We translate the constraint subset from the main application to business rules which are validated in the mobile application by a rule engine. Further, we propose a responsive GUI which adapts automatically to the synchronised constraint subset. In general, this approach is not limited to injecting a constraint set into a mobile application but could be applied to any dependent information system with business rules.

As an extension of our FoodCASE information system, we developed a mobile application for managing food item purchases within a TDS. The resulting scientific shopping

data is transferred to FoodCASE where it is further used as the food consumption data. Since the final risk assessment in a TDS is also based on this data, it is essential to provide a high level of DQ.

We investigated and implemented different DQ feedback presentations based on the device size and evaluated our approach with a user study where participants had to compare the different presentations. At the end of the chapter, we report on our findings.

6.1 Related work

Adaptive interfaces have been investigated in research for multiple domains and various needs. For example, an adaptive interface in a tool for passive persuasion of participants is suggested in [155]. With the adaptive interface, the authors intend to make feedback continually effective within their tool. Other examples include [71, 92, 83, 26].

In [153], the authors specifically conducted an analysis for small screens by using ideas from rule-based programming. With their approach, it is possible to match any source pattern with several transformation rules and to determine which rules should be applied by using *conflict resolution*. They use multiple models which are subsequently transformed from one to another, while different interface parts are matched with various proposed rules. The result of the matching is the positioning of all interface parts to a predefined screen size. The following three main objectives are proposed to optimise this process: (1) Maximum use of the available space; (2) Minimum amount of navigation clicks; (3) Minimum scrolling.

The adaptive user interface of a personalised transportation system was investigated in [129]. The usual routes that users have build the basis for providing an interface. As soon as a new route is added, the user interface is extended. The authors compare their adaptive interface solution with two other approaches showing decreased click and time costs for the adaptive interface.

A more general method for providing an adaptive interface, which combines a user interface runtime architecture and runtime user interface models, is presented in [160]. In their model-based approach, the user interface can be adapted at runtime to numerous contexts-of-use. Their system is able to generate the interface based on the dynamic context information from the environment.

Adaptive UIs are concerned about providing easy-to-use UIs which adjust to specific preferences of users and/or changing data. In [26, 25], a technique, called Rich Entity Aspect/Audit Design (READ), is proposed that provides adaptive UIs for production enterprise systems. Their aim was to reduce development and maintenance effort to a degree which is equivalent to the development of a single UI. With this technique, it is not necessary for developers to design tables or forms for each panel or page. ‘Instead, they design generic and reusable transformation rules capable of presenting application data instances in the UI while considering the runtime context.’ [25] Their approach considers existing standards for application frameworks and AOP. To face the efforts and complexity related to adaptive UI design, their approach employs code-inspection, where they propose the inspection and reuse of application business rules. The work also includes the development of the library ‘AspectFaces’ for Java systems that implements their READ framework. In their work, aspects are added based on annotations.

An approach for generating UIs from domain objects is presented in [96, 97]. Their framework, called ‘Metawidget’, is based on MDD and capable of inspecting an application’s back-end architecture. Based on this information, native UI subcomponents are created which match the back-end (model).

In [168], the design of a framework for adaptive user interfaces on mobile devices for the Android¹ platform is proposed. Their focus is context-aware computing, involving hardware elements of devices, such as Global Positioning System (GPS) and light sensors. They propose a concern-separating style of AOP which provides developers with access to the context of device sensors to adapt individual application aspects.

6.2 Approach

In this section, we present our approach to unified constraint management and DQ feedback for two dependent applications. To illustrate the approach, we will use the example of the FoodCASE main application (client, server and database) and mobile application (with its own database) already introduced in Chap. 1 on pages 4 and 6. A simplified depiction of this scenario information system is presented in Fig. 6.1.

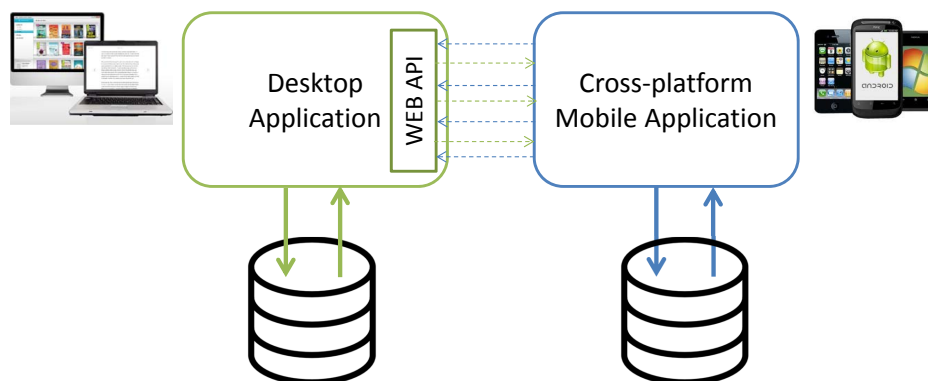


Figure 6.1: Scenario information system

We distinguish between a source system, for example the FoodCASE main application depicted on the left side, and a target system, for example the FoodCASE mobile application depicted on the right side.

In our scenario, the FoodCASE client is used to manage all data, while the mobile application is used to manage a subset of data related to buying food samples in grocery stores. Both systems contain a set of constraints which differ from each other and therefore it is necessary to keep track of the set of relevant constraints for each system and keep them synchronised.

The source system is the main application for which a set of constraints is defined and managed. The target system is the mobile application which is connected with the source system by transferring data, as depicted in the centre of Fig. 6.1. Since the target system delivers input data to the source system, it shares a subset of the constraint set from the source system. This subset of constraints is used in the target system for two purposes: (1) Data validation where the constraint violations are the basis for

¹<https://developer.android.com/index.html>

calculating and giving feedback on DQ to the users of the target system; (2) Generating the UI which adapts to the set of constraints. Different levels of constraint importance can be considered for the UI generation, for example, to place attribute fields with more important constraints at a prominent position. Further, in our approach, the way of giving feedback on DQ to the users of the target system is dependent on the device screen size. Thus, the way of giving feedback on DQ differs for different devices such as desktop screens, tablets and mobile phones.

In summary, our approach consists of three components which work hand in hand based on the same constraint set: (1) Constraint injection/synchronisation which targets the problem of unified constraint management; (2) Adaptive user interface which simplifies the creation and maintenance of GUIs for changing constraints over time; (3) DQ feedback appropriate to the device screen size which brings constraint compliance in relation to DQ feedback especially for different display devices. In the remainder of this section, we will first present how these components are integrated before going on to describe each of these components in detail.

Fig. 6.2 shows an overview of the three components. The source constraint set, depicted on the left, is initially defined by the developer or afterwards adapted by the developer. A modification and translation process generates the appropriate subset of target constraints for the source set which is depicted by the circle arrow between the systems. The subset is then ready for injection into the target system.

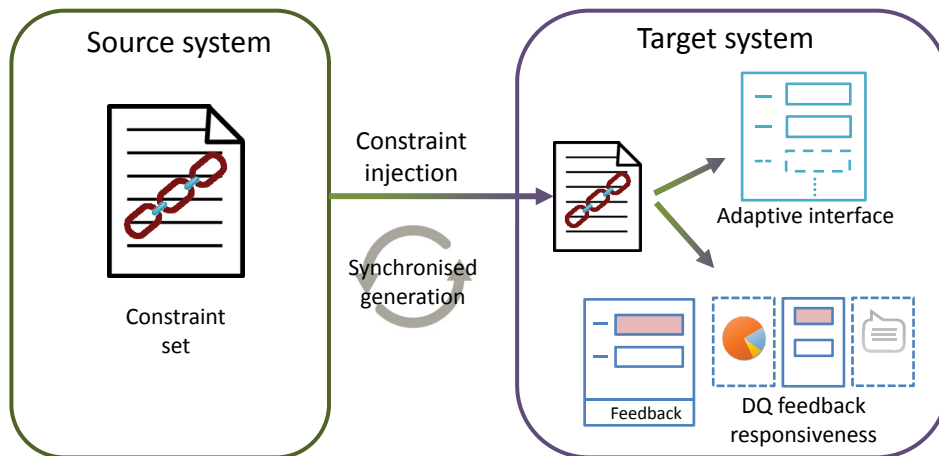


Figure 6.2: Components integration

The target system receives the constraint subset and updates its constraint repository. This set is then used to generate the adaptive user interface, depicted in the upper right corner of the target system, and, the data validation and the dependent DQ feedback, depicted in the lower right corner.

This approach therefore enables unified constraint management with the development of a flexible target system that adapts its data validation and GUI according to changing constraints. With this and the responsive DQ feedback, it reduces development and maintenance effort.

6.2.1 Constraint injection

Constraint injection denotes the transferring of a set of constraints from a source system to a target system, where the target system's data validation is then based on the

injected constraint set without being influenced in its other runtime processes. If the source and target system use the same constraint representations, the constraints can be taken as defined in the source system. There solely might be a filtering of unnecessary constraints which do not apply to any data in the target system. We consider here the more complex case where the source and target system contain different mechanisms for data validation and consequently the constraint representations in the two systems are different. Therefore, a translation process is needed which translates the constraints from the source representation to the target representation. This is related to the approach presented in the previous chapter, but here only one particular translation has to be implemented, instead of multiple bi-directional translations and consequent multi-translations.

An overview of the constraint injection is presented in Fig. 6.3. The source constraint set on the left is given in a certain representation, such as BV. Constraints in the set are divided into and ordered by different groups/levels of importance, indicated by the leftmost colour graded triangle. More important constraints are indicated with a larger red bar in the triangle while less important constraints are indicated with a smaller green bar in the triangle.

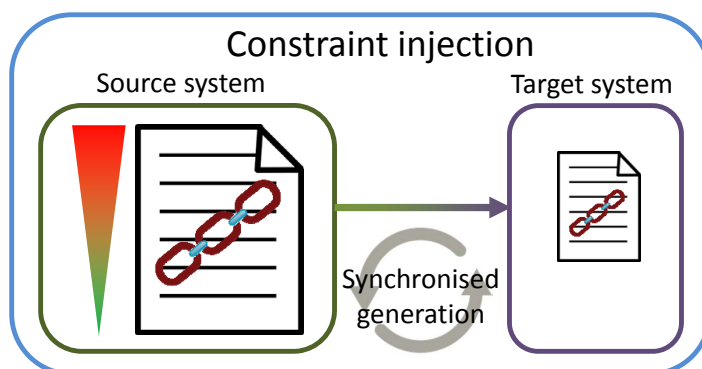


Figure 6.3: Constraint injection overview

An example of a small constraint set with an importance ordering is given in Fig. 6.4. We use an example of our FoodCASE system for food items. Three constraints are depicted there, where the type of constraint and the name are presented in the boxes on the left side, the constraint itself is presented in the boxes in the centre and the association to a constraint group is presented in the boxes on the right side. Consequently, a food item (1) must have a name, (2) must have a total weight of at least 100 grams, and (3) should state the origin of the product.

The background colour of the constraints distinguishes the different levels of importance. The first two constraints, ‘name’ and ‘weight’, are of high importance and therefore coloured in red. The third constraint, ‘origin’, is only of medium importance and therefore coloured in orange. Thus, users should be forced to input the first two attributes but only be recommended to input the third attribute. This example shows the distinction between two importance groups, nonetheless, our approach is not limited to a certain number of groups. All subsequent operations and modifications on the set of constraints is based on the information as given in Fig. 6.4.

Based on the source constraint information, the set of constraints for the target system is then generated, which is indicated in the centre of Fig. 6.3 by the arrow from the source to the target system. This includes in a first step the grouping in

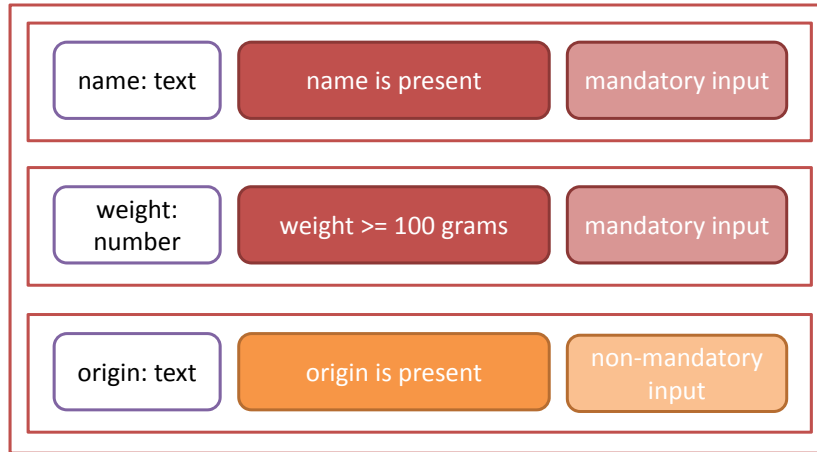


Figure 6.4: Example constraint set

different importance levels and the filtering of constraints due to various criteria. One essential criteria is to only keep constraints which are related to appropriate attributes in the target data. But it is also possible to filter certain types of constraints or certain importance levels. If we consider the example from above and, if we do not filter any of the constraints, the information about the constraint importance is used to generate two constraint sets: Highly important constraints (name and weight), where we consider the associated attributes as ‘mandatory input’ and less important constraints (origin), where we consider attributes as ‘non-mandatory input’.

The next important step is to translate each constraint of the filtered source constraint set to the appropriate target constraint representation. In the given scenario, we translate the source constraints to business rules which are then validated in the target mobile application with a rule engine. A rule file is generated for each importance group. Thus, the generation for the example above results in two rule files for constraints related to mandatory and non-mandatory input data, respectively. The target system, depicted in Fig. 6.3 on the right, uses these target constraint sets for its data validation. For this approach, it is assumed that the target constraint set is stored and updated in a way that does not influence other target system processes. This is guaranteed, for example, by using a rule engine that validates data based on the different rule sets, for the different constraint importances, provided as different files.

The constraint set can be injected more than once to handle changes to the original set or the filtering process. Thus, constraint synchronisation from the source system to the target is provided which is indicated in the centre of Fig. 6.3 with the two circling arrows. The synchronisation can be achieved by either manually triggering the generation process again, or by automatically running the generation process upon update of the original constraint set or filtering.

6.2.2 Adaptive user interface

Over the lifetime of software, requirements can change which means that not only the functionalities may need to be adapted but also the data validation. The source constraint set can bear the following changes: (1) New constraints might be added; (2) Existing constraints might be deleted; (3) Constraint importances might be changed; (4) Constraints themselves might be changed. With constraint synchronisation, as

described previously, all these changes in the constraint set are taken into account, which already reduces the maintenance effort. Nonetheless, the target system must also take these changes into account by adapting its input forms. Otherwise, the mapping between constraints and appropriate attributes for the data input will be broken.

Consider, for example, the adapted constraint set depicted in Fig. 6.5. The colouring of the boxes again indicates the importance of the constraints as in the example above. Two changes were made to the original source constraint set: (1) A new mandatory constraint for the attribute ‘brand’ was introduced which ensures that the brand name is present; (2) The importance for the constraint related to the ‘weight’ attribute is now associated to the non-mandatory input constraint group.

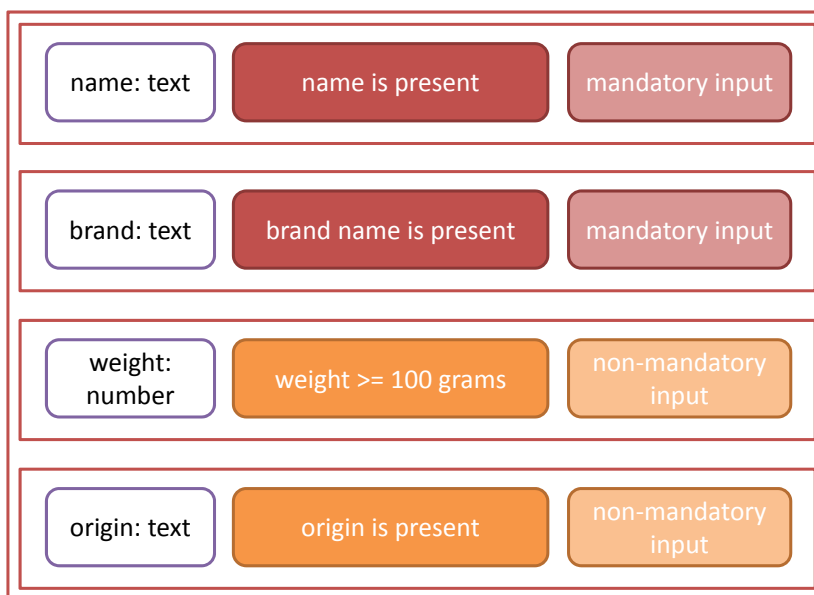


Figure 6.5: Adapted example constraint set

The data validation in the target system should now only recommend the input of the ‘weight’ which is already guaranteed by the constraint synchronisation. For the constraint concerning the attribute ‘brand’, a new input field must be generated and placed appropriately in the GUI and the according mandatory constraint checking must be applied. Thus, an adjustment in the GUI might be needed each time the constraint set changes. We therefore recommend that the GUI should be built automatically based on the set of constraints in the target system. This avoids manual code modification for the GUI and ensures the correct mapping between input attributes and constraints. Such an adaptive GUI in the dependent target system further reduces the maintenance effort not only for constraint checking but also in terms of provided input fields. Nevertheless, a developer has to take care of all other functionality concerning the changed attributes, such as further processing of the input data in addition to data validation.

Fig. 6.6 gives an overview of the adaptive interface generation. The generation is based on the target constraint set depicted on the far left side.

This set was processed during the constraint injection and extracted into different constraint set files according to the importance levels, depicted on the left in the target system. The target system analyses the types and groups of constraints before generating, the appropriate input forms in the GUI. This is presented in Fig. 6.6 on the right. The input forms are placed in the GUI according to the importance of their constraint

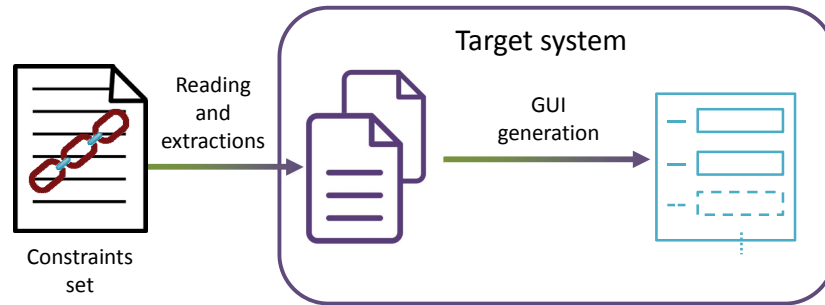


Figure 6.6: Adaptive interface generation

checking association.

We generate interfaces for each of the importance groups, placing fields of the same importance in separate panels and present the panels in order of decreasing importance. In the case of the example depicted in Fig. 6.5, a user therefore will be presented first with the mandatory input fields for ‘name’ and ‘brand’. After having entered the appropriate data without constraint violations, the user is presented with the non-mandatory input fields ‘weight’ and ‘origin’ which they are only recommended to fill in. This helps a user focus on the main data input fields and ensures that data which is required by the system cannot be skipped. In the case of shortage on time or other reasons, a user can skip the less important input fields or return to them later. In our scenario application, users are entering data for many input attributes of one entity (a food item) where they first focus on the most important attributes due to time limitation and are able to enter less important attributes to a later point in time.

6.2.3 Data quality feedback responsiveness

For guaranteeing the high quality of data, it is crucial to provide users of an information system with the result of data validation, such as the constraint violations of appropriate input forms. It is essential that users are aware of the state of DQ in the system and encouraged to improve it. How feedback on DQ can be given was previously discussed but it also depends on the capabilities of a system and, within the context of mobile devices, this is also related to the screen size of a device. Small screens call for different feedback representations due to the limited available screen space. Therefore, we present here an approach which provides a responsive DQ feedback for mobile devices.

Since we already adapt the GUI of the target system according to the target constraint set, as described in the subsection above, we can also take the screen size into account to generate a convenient GUI for DQ feedback. We distinguish two screen sizes: (1) Big screens as depicted in Fig. 6.7 on the left side; (2) Small screens as depicted in Fig. 6.7 on the right side. Big screens encompass tablets or other big screen devices while small screens encompass mobile phones. This distinction can be configured since different application domains, content, goals and design of applications can influence the way in which feedback should be given.

For both big and small screens, we provide results of the constraint checking directly on the data input fields to ensure that users are always aware of constraint violations. For big screens, we provide DQ feedback as part of the main screen, for example, below input forms, as already discussed in previous approaches. For small screens, we provide DQ feedback on separated side-panels which can be access by swiping to the left and

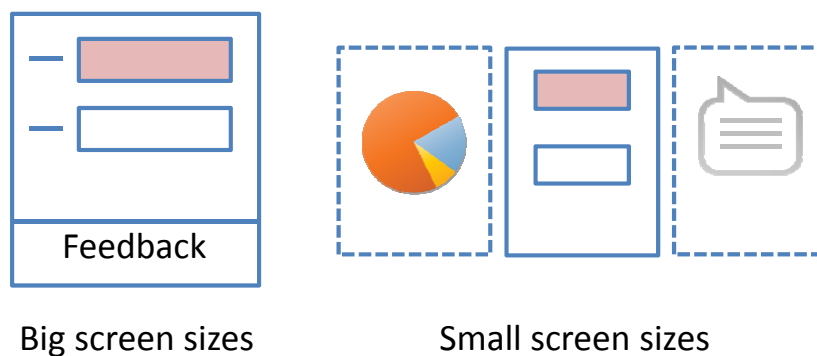


Figure 6.7: Different screen sizes

right from an input form panel. Therefore, users can use the full small screen for their main interaction with the application and access the additional panels for DQ feedback if needed, with the side-panels providing the most relevant feedback on DQ. Further, we provide additional feedback, such as detailed statistics, on a separate view which can be accessed via a navigation button. Dependent on the screen size type, DQ feedback is presented in different ways, such as with plots, tables or both plots and tables.

6.3 Implementation

To realise our approach in the TDS-Exposure project, we developed a mobile application as part of the FoodCASE information system, to support the sampling step where subsamples (concrete food items) have to be bought in shops. Shopping lists for subsamples can be created in the main FoodCASE desktop application which are then downloaded onto mobile devices. Thus, the mobile application contains only a subset of the FoodCASE main application data, namely subsample data for buying concrete food items in grocery stores. The shopping lists on the mobile devices are used by the appropriate personnel to find and buy the correct food items and to enter detailed information about each food item through the mobile application. This subset of shopping data is then transferred to the FoodCASE main application where it is further processed. How the sampling step is embedded in the whole TDS process was already introduced in Chap. 1 on pages 4 and 6.

Since a significant amount of input data is transferred from the mobile application to the FoodCASE main application, it is inevitable that the constraint checking and a high level of DQ is already guaranteed while entering data into the mobile application. Constraints on the subsample entity in the mobile application are identical to the constraints on the subsample entity in the FoodCASE main application. But no constraints from other entities of the FoodCASE main application are necessary in the mobile application. Thus, one main requirement was to synchronise the constraint set from the FoodCASE main application, as the source system, with the constraint set in the mobile application, as the target system. Further, besides the pre-defined business processes, we defined the requirements that the mobile application should be connected to the desktop application through a Web API, that it should work on multiple mobile operating systems (cross-platform) and that it should provide DQ measurements and DQ feedback based on constraint compliance.

To enable cross-platform development, we decided for the open-source cross-platform

framework *PhoneGap*² which provides a large amount of plugins and has a large community. For the implementation of the GUIs we used the *Ionic Framework*³ which is based on Cordova⁴ and AngularJS⁵ and supports the building of native-feeling mobile applications using web technologies such as HTML, CSS and JavaScript.

For the constraint management in the mobile application, it was required that the data validation could be performed solely in the mobile application without a connection to the source system, since it must be possible to work completely offline during a shopping trip. For mobile applications lightweight solutions are always favoured to reduce the application package size, amount of memory used and battery consumption. To realise our approach, it was most important that constraint definition was de-coupled from the source code and possible to update during runtime of the mobile application. We therefore decided on rule-based validation implemented with *Nools* which is a *Rete algorithm* based rule engine written entirely in JavaScript. The Nools rule engine provides the following important features for our approach: (1) It is possible to define rules in a separate file using the Nools language; (2) It is possible to express very complex rules; (3) Rules can be compiled and used on-the-go which enables changing rules during runtime without negatively influencing the application.

The mobile application had to be able to store data persistently, including data received from the source system, input data and high resolution photographs of food items. We decided to use the SQLite⁶ database, since it met our requirements for data storage, is well supported, is compatible with PhoneGap, has a large developer community, and provides stable releases as well as detailed documentation.

Within the FoodCASE main application, we implemented a Web API for data transfer with the mobile application and a constraint generation component to translate constraints from the source system to the target system representation. Here, we will only explain the constraint generation which is essential for our constraint synchronisation approach.

6.3.1 Constraint injection

To perform the constraint translations, first all constraints from the FoodCASE entity `TdsSubSample` are collected, since it is the only entity required for the mobile application. List. 6.1 shows the BV constraint definitions from the example presented in Fig. 6.5. The constraint definitions with BV annotations in the example are similar to previously presented BV examples, where the annotation type, such as `@NotNull` and `@Min`, defines the constraint type and the `payload` element, such as `Mandatory.class` and `NonMandatory.class`, defines the importance of the constraint. We currently distinguish only two groups of importance in our implementation but it would also be possible to define other importance classes.

²<http://phonegap.com/>

³<http://ionicframework.com/>

⁴<https://cordova.apache.org/>

⁵<https://angular.io/>

⁶<https://www.sqlite.org/>

```
public class TdsSubSample {
    @NotNull(payload = Mandatory.class)
    private String name;
    @NotNull(payload = Mandatory.class)
    private String brand;
    @Min(value = 100, payload = NonMandatory.class)
    private Integer weight;
    @NotNull(payload = NonMandatory.class)
    private String origin;
}
```

Listing 6.1: BV example

Our implementation analyses the available constraints and extracts the following details: (1) The importance, such as mandatory or non-mandatory; (2) The validated field, such as name, brand, weight and origin; (3) The field type, such as `String` and `Integer`. Our `ValidationGenerator` class then translates the constraint definitions to appropriate Nools rules and writes separated rule files for each importance level. In our case, two files are created for mandatory and non-mandatory rules, respectively. List. 6.2 shows the example for the generated mandatory rules file where the two rules for the name and brand attributes are defined.

```
rule CheckNameRequired {
    when {
        m : Model isUndefinedOrNull(m.name) or m.name.length eq 0;
    }
    then {
        m.errors.push("name is required");
    }
}

rule CheckBrandRequired {
    when {
        m : Model isUndefinedOrNull(m.brand) or m.brand.length eq 0;
    }
    then {
        m.errors.push("brand is required");
    }
}
```

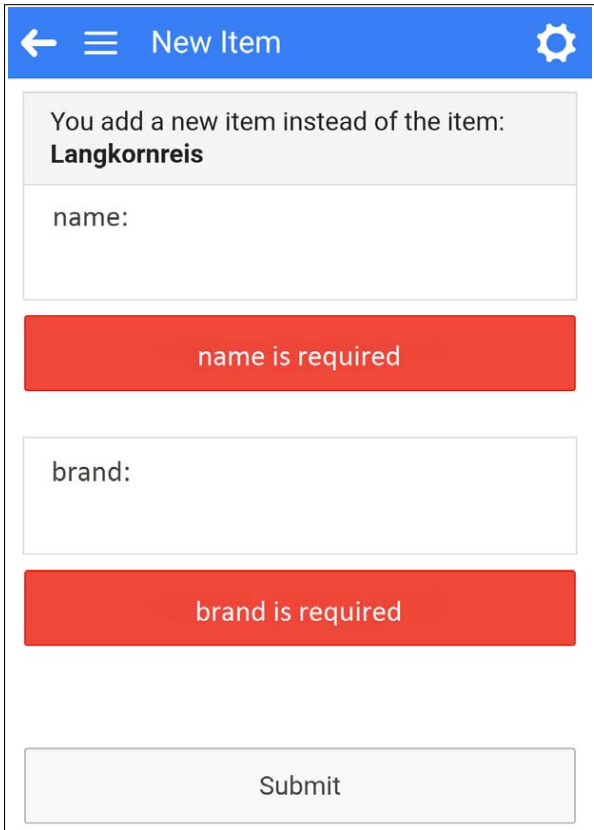
Listing 6.2: Nools example

With the generation of the Nools rules files, the source side constraint synchronisation part is finished and the rule files are ready for the constraint injection process to the target system. The rule files are then sent to the target system as described in Sect. 6.2.1.

6.3.2 Adaptive user interface

In our mobile application, there exist three use cases where constraint checking is needed: (1) Editing data of an existing item in the shop before purchase; (2) Adding a new item with input data in the shop for purchase; (3) Input of additional information after purchase outside of the shop. For all these cases, the constraint checking within the mobile application, based on the injected rule files, is implemented as follows. If a user invokes one of the three use cases, the rules from the rule files are extracted, and the appropriate GUI with all associated attribute fields and a local model object is created. The local model is then used for all operations that require a constraint check.

The constraint checking process is implemented as follows: (1) For each change in an attribute field of the GUI, a check function is called based on the local model; (2) The match function for the model is executed, which triggers the appropriate constraint checking, and returns either a set of errors in the case of constraint violations, or an empty set if no constraint violations occurred; (3) If errors exist, the corresponding error message(s), defined in the rule file, is/are displayed beneath the attribute field in an error message box, as depicted in Fig. 6.8. If check functions are called multiple times without changes to the rule files, the model does not have to be re-created again.



The screenshot shows a mobile application interface for adding a new item. The title bar is blue and contains a back arrow, a hamburger menu, the text "New Item", and a settings gear. Below the title bar, there is a grey box with the text "You add a new item instead of the item: **Langkornreis**". Below this, there are two input fields. The first is labeled "name:" and has a red error message box below it that says "name is required". The second is labeled "brand:" and has a red error message box below it that says "brand is required". At the bottom of the form is a grey "Submit" button.

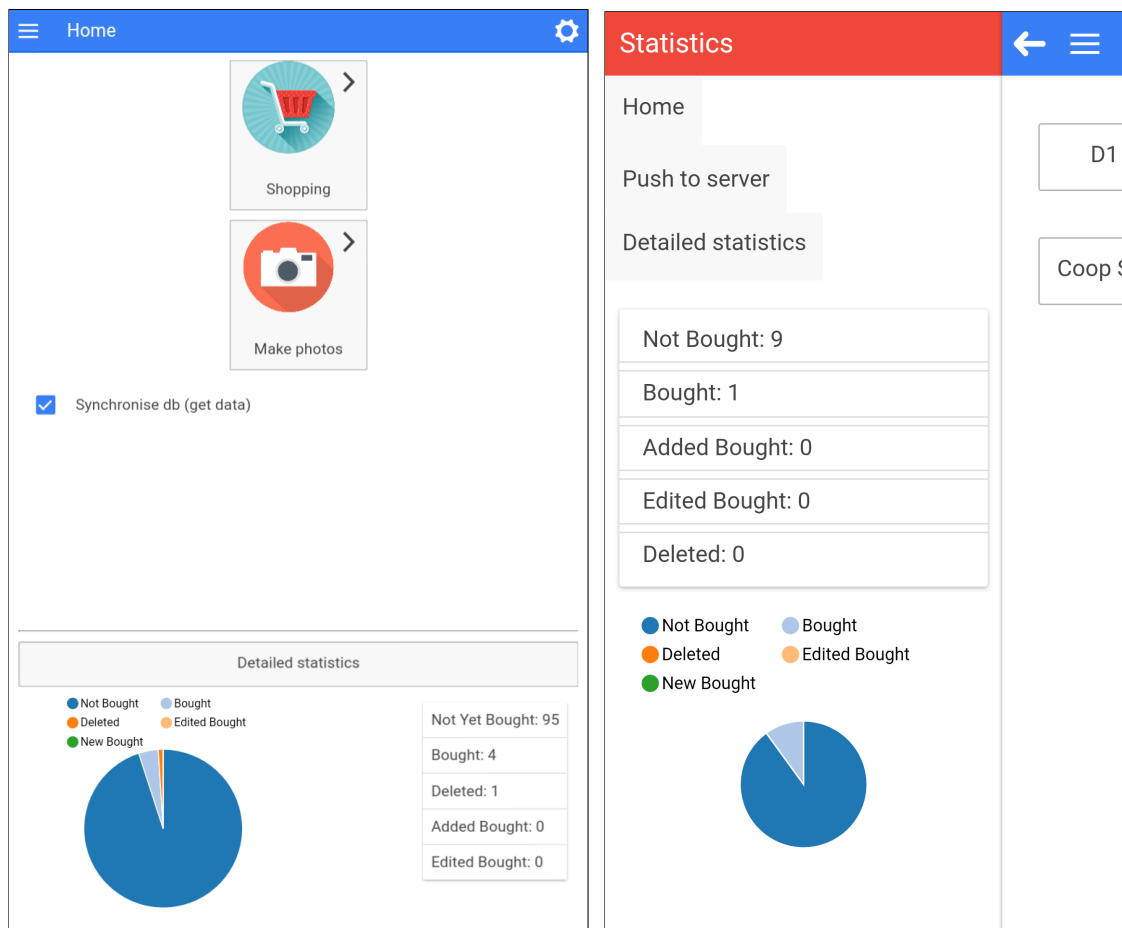
Figure 6.8: Mandatory input validation

6.3.3 Data quality feedback

The error message boxes which appear upon constraint violations beneath the appropriate attribute fields in the GUI, as described in the previous subsection, are a direct way of giving feedback on DQ during data input.

To give additional feedback on DQ, we implemented three different feedback versions which were then evaluated in a user study, described in the next section. The first version, named *basic*, is a responsive DQ feedback which realises our approach presented in Sect. 6.2.3. We considered a screen height below 650 pixels as a small screen size device, while others were considered as large screen size devices.

Fig. 6.9a shows the DQ feedback for large screens, where the feedback with a pie chart and detailed information in a table is positioned at the bottom of the GUI panel, taking 35% of the screen size. It provides an overview of the buying status of all food items. Fig. 6.9b shows the DQ feedback for small screens, where the same feedback content is available on a side panel of the application, visible if a user swipes right.



(a) Large screen

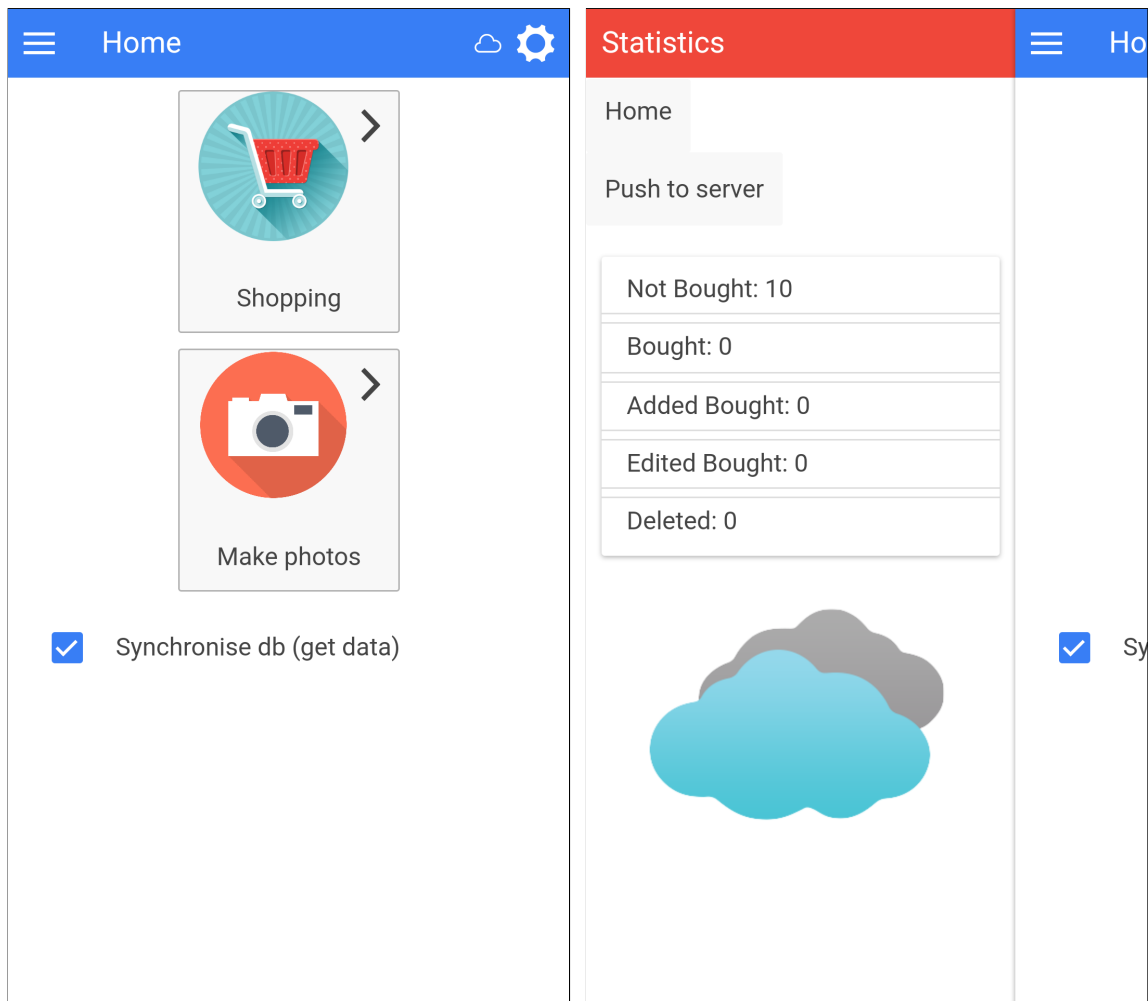
(b) Small screen

Figure 6.9: Basic version: responsive data quality feedback

In both cases, more detailed feedback information is available on a separate application page which can be opened by clicking on the button ‘Detailed statistics’. It contains bar charts representing the DQ levels in each shop. DQ is calculated as a relation of filled and empty data input fields in each shop for the most relevant attributes,

which provides a measure of completeness.

The second feedback version, named *weather*, introduces a metaphorical representation of DQ, where five different weather conditions (thunderstorm, rain, cloudy, intermediate, sunny) represent different levels of DQ. Again the completeness for the most relative attributes per food item is calculated as a measure for DQ. The appropriate mapping to a certain weather condition is displayed as a small icon on the top panel of the application, as shown in Fig. 6.10a. Additionally, on the available side panel the weather condition icon replaces the pie chart from the *basic* feedback approach, as shown in Fig. 6.10b.



(a) Icon feedback

(b) Side panel feedback

Figure 6.10: Weather version

The third feedback version, named *modal*, provides again an overview of the buying status of all food items with a pie chart and a table, but available in a special pop-up window, as shown in Fig. 6.11. The pop-up window only appears if the status of a food item was changed, for example when it was bought. In this version, the status change is the only event that triggers the DQ feedback. During data input, no detailed information on DQ is available for a food item.

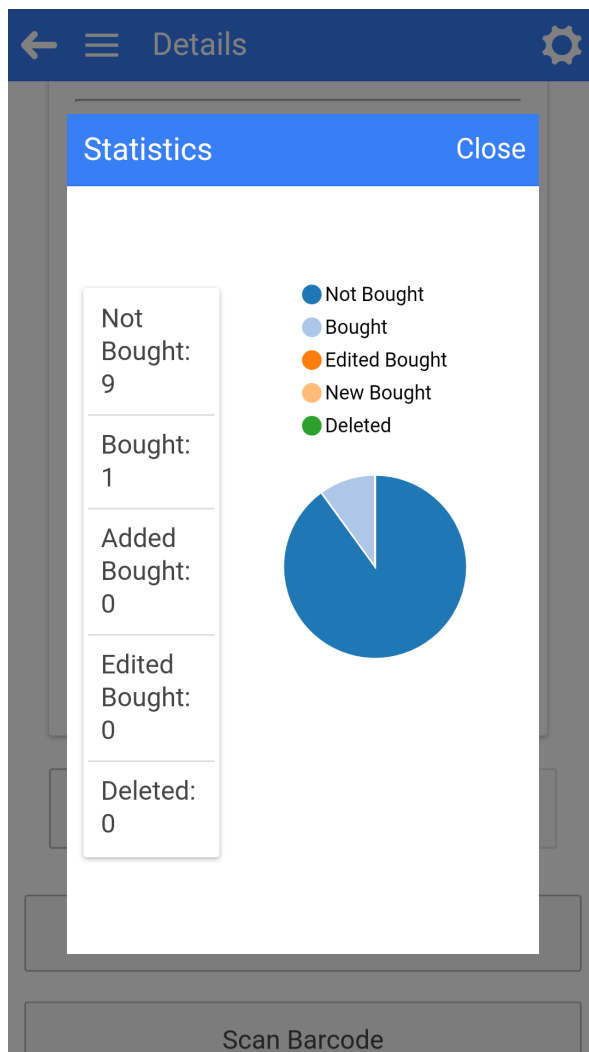


Figure 6.11: Modal version

6.4 Evaluation

To evaluate our responsive DQ feedback, we carried out a user study with thirteen participants, mainly studying or working in the computer science area, to investigate the three different versions of DQ feedback. The goal of the study was to identify the user's preferences and assess the usability of the mobile application also concerning DQ feedback.

We were simulating a shop visit where the users were asked to buy five different food items which we presented to the users. They had to buy, edit or delete food items with the help of our mobile application provided in all three different versions *basic*, *weather* and *modal*. Besides the DQ feedback which they received from the application during runtime, we also presented screenshots of different constraint violation messages that could occur during the use of the application. In order to avoid a learning effect in using our mobile application, the users could familiarise themselves first by using the application without any DQ feedback and were then presented with the different versions in a shuffled order for each participant.

During the study, the users had to fill out a questionnaire related to the following

four parts: (1) DQ feedback; (2) Usability; (3) Data validation; (4) Personal user feedback. In all parts, except the personal user feedback, the participants answered the questions by ticking a checkbox on a Likert-type scale with possible answers ‘strongly agree’, ‘agree’, ‘neutral’, ‘disagree’ and ‘strongly disagree’.

In the DQ feedback part, the participants purchased the food items with each mobile application version and answered the following three questions for each version: ‘The data quality feedback representation was clear for me’, ‘I found the feedback useful’, and ‘The feedback was helpful in improving the data quality’. The usability part contained thirteen questions about the general usability of the different versions, such as ‘The navigation was clear and consistent’, ‘It was clear how to use this application’, ‘I was able to complete the tasks efficiently using this application’, ‘It was easy to add the data’, ‘The help provided by the application was clear’ and ‘The application’s error messages helped me to fix the problem’. We presented three screenshots in the data validation part with different error messages for which the participants were asked if they could correct the errors. Concerning that, they had to answer the following three questions: ‘It was easy for me to understand where incorrect data was entered’, ‘It was easy for me to understand how I need to change the input data in order to satisfy the validation constraints’, and ‘The validation errors were displayed clearly’. In the personal user feedback part, the participants had to answer the following three questions with their own words: ‘Please specify, which application version you like the most and why’, ‘Please add any feedback that you think might be valuable for the application development’, and the optional question about paper prototyping ‘Please compose the application interface in a way you see its ideal version’.

For the sake of brevity, we will present only a summary of the most important findings in the remainder of this section. The detailed description of the study setup and the full list of questions and results are provided in [15] on pages 43 to 50 and 66 to 75, respectively.

For the majority of the users, all the DQ feedback representations were clearly understandable. Thus, our pie charts, tables and metaphorical weather depiction are generally good solutions for visualising DQ feedback. The *basic* and *weather* versions were rated more useful than the *modal* version, possibly due to the simple representations and their non-invasive nature. The *weather* version was rated as the most helpful in improving DQ and giving the strongest motivation to improve DQ. This perception could be due to its simplicity and the immediately visible effect as soon as DQ is affected. On the contrary, the *modal* version was not appreciated that much since it seems to show not enough information to perform DQ improvements and is less motivating for the users. All users were generally satisfied with the usability of the application, the majority rated the application as easy to use and that they were able to complete the tasks efficiently. The error messages which were encountered during the study were rated as helpful in fixing the problems and only two participants stated that the help provided by the application is not enough. The error representation for the validation feedback was also rated as good and sufficiently detailed for the majority of the participants to cope with the issues. From the personal feedback of the participants, it can be seen that less invasive and easy to understand DQ representations are preferred. Further, providing motivating ways to improve DQ is important to encourage users to instantly enhance DQ. Overall, the *basic* and *weather* versions were rated as comparably good in regard to the usefulness of feedback. The *weather* version was the most appreciated

but even it could be improved by combining it with features from the *basic* version, such as additional configurable charts. Further, it could be indicated how much DQ improvement is needed to reach the next higher DQ level or better weather condition, respectively.

6.5 Conclusions

With the approach presented in this chapter, we have introduced a novel constraint injection concept which allows for a one-way synchronisation of constraints including individual constraint importance levels between dependent information system applications. Thus, changes to the constraints in a source system can be propagated automatically to a target system. We further introduced an adaptive user interface generation based on the injected constraint set which reduces manual creation and adjustments of GUI code in the target system. Input forms for the GUI are generated and grouped dependent on the constraint set and the individual constraint importances, such as mandatory and non-mandatory constraints. More important attributes for input data are placed in a prominent position to facilitate maintaining a high level of DQ. We presented different ways of providing data validation and DQ feedback in a mobile application which includes an approach on responsive DQ feedback that automatically adjusts feedback content and position depending on the device screen size. With this, DQ feedback is customised to the available data input device and therefore further enhances the possibilities to motivate users to improve DQ while entering data. Our concepts were implemented within our FoodCASE information system, as a desktop and dependent mobile application and a user study was carried out to evaluate the different DQ feedback representations. The feedback from the study participants was positive and provides insights on giving DQ feedback in mobile applications.

The presented approach is explicitly applicable for the injection of a constraint set from one main system to a dependent system which is designed to manage a subset of the main data or at most the same data. The constraints are injected in one direction from the source to the target system where the constraint definitions in the target system cannot exceed the definitions of the source system. Although the input forms of the GUI in the target system are generated automatically, a developer has to take care of the additional business logic which handles the appropriate input attributes.

7

Hybrid approach with subjective data quality assessment

So far, our final approaches tackle the problem of unified constraint management by covering (1) homogeneous usage of constraint definitions throughout an information system, such as BV, (2) a unified constraint representation combined with bi-directional translations to and from technology-specific constraint representations for the different components in an information system, and (3) a specific constraint injection from a main system into a dependent application. For giving feedback on DQ, we have presented various measurement, calculation and visualisation concepts, always based only on constraint compliance. While the appropriate DQMFs for (1) and (3) are implemented as part of an information system, and hence are considered as *integrated* approaches, the DQMF for (2) is implemented external to an information system, and hence is considered as a *separated* approach.

With the approach described in this chapter, we aimed to provide a convenient unified constraint management and data validation mechanism for systems with heterogeneous constraint definitions following a hybrid approach, as a combination of previously introduced integrated and separated concepts. The validation logic is implemented in validation components for different tiers as part of the DQMF. Therefore, the approach could be considered as a *separated* approach. But the components are integrated in an information system for data validation independent of the DQMF. Due to this validation component injection, the approach enables validation functionality from the separated DQMF to be integrated into an information system, and hence is considered as a *hybrid* approach.

The hybrid dependency between the DQMF and an information system in combination with an architecture that allows for separated DQ analysis monitoring, as introduced in the initial approach in Sect. 3.5, was the driving factor for this approach. The further aim was to incorporate previously presented concepts and provide additional new features for assessing DQ with a combination of objective and subjective DQ metrics/measures.

Therefore, the approach incorporates all previously presented concepts concerning unified constraint management, input data validation (DQIF), and DQ analysis (DQAF). The presented DQMF also provides unified constraint definitions, measures and calculations to analyse the quality of data, and feedback on how to improve DQ. It further supports different constraint definitions for different user groups and offers flexible data validation with multiple constraint violation severities. The constraint checking is based on constraint repositories and AOP where we also provide the freedom to modify, add and delete constraints during the runtime of an information system.

We distinguish between a monitored information system and our DQMF. A monitored information system is the main system which provides input validation based on unified constraint definitions stored in constraint repositories central to the monitored information system. Our DQMF provides constraint checking components (a module and a library) which can be used in different tiers of a monitored information system, such as the presentation and the logic tier, to guarantee input validation in multiple tiers based on the central constraint repositories. At the same time, the constraint checking components are used to send validation and feedback metadata from the monitored information system to our DQMF which processes the metadata as a separately running application for DQ analysis and DQ monitoring. Thus, our components are integrated into the monitored information system for data validation but the DQ analysis is processed in our separated DQ management application. Therefore, the DQ analysis and DQ monitoring can be performed in the separated application without influencing the runtime of the main system. Nonetheless, the identification of data is guaranteed to enable single data instances in the monitored information system to be improved upon indication of deficient DQ in the DQ management application. The DQ management application further provides functionality to maintain the constraints in the constraint repositories, where changes to the constraints take immediate effect on data validation in the monitored information system.

With this approach, we base DQ measures and feedback not only on constraint compliance but provide new concepts to include implicit and explicit user feedback for our DQ calculations to incorporate both objective and subjective DQ assessment. As already indicated in Sect. 2.6.2, not all DQ issues can be identified by objective DQ assessment expressed by constraint checking. There exist, for example, semantic issues which cannot be identified by a machine and therefore call for additional subjective DQ assessment. Moreover, within objective DQ assessment based on constraint checking, a set of constraints might also include incorrect constraint definitions or definitions might be missing which could be discovered by comparing objective and subjective DQ assessment.

Therefore, we propose a DQ vector space model which represents DQ in the three dimensions *constraint compliance*, *explicit user feedback* and *importance*. *Constraint compliance* is the fulfilment of the constraint checking which means that the more constraints are fulfilled, the better the DQ is assumed to be. *Explicit user feedback* is a rating of perceived quality from the application users concerning data instances and constraints. *Importance* is a combination of the user's interest in a data instance, measured by number of clicks, and the importance of an entity as defined by domain experts, who know the domain and information system well enough to rate the entity. Explicit user feedback and importance are interesting additional measures which enable DQ to be indicated as a verification of the constraint compliance DQ indicator or where

constraint compliance is not applicable for indicating DQ. It is especially promising to use explicit user feedback in an information system since, due to the Web 2.0, Internet users turned from data consumers to data producers by editing and rating information, given for example in images, social networks and product ratings. With this approach, we also investigated how motivated users of an information system could be to rate the quality of data they work with.

Based on the DQ vector space model, we provide further measures, so-called DQ improvement tendencies and a DQ improvement urgency score, to support a DQ analyst in identifying and enhancing data instances of bad DQ. Thus, our approach provides measures and visualisations for DQ analysis that go beyond solely constraint compliance, as well as enhanced suggestions for DQ improvement.

To evaluate our approach, we implemented a scenario application *FoodCASEWeb* as a web-based counterpart to our FoodCASE desktop application in the food science domain. *FoodCASEWeb* provides the most important functionalities of FoodCASE and operates on the same scientific data. It represents the monitored information system which sends input validation and user feedback metadata to our separated DQ management application, named *DQAnalytics*. In order to evaluate the usefulness and usability of *DQAnalytics*, we conducted a user study and present its results where one outcome was that explicit user feedback is perceived as an important and promising DQ indicator.

7.1 Approach

Fig. 7.1 gives an overview of the components of our DQMF presented in this chapter. The monitored information system, as introduced above, is depicted on the left side in a three-tier architecture with its presentation tier, logic tier and database. The components of our DQMF are highlighted with a grey background. It consists of four components: A *user input module* used in the presentation tier, a *business logic library* used in the logic tier, potentially multiple constraint repositories used by both the module and the library, and a DQ management application depicted on the right side.

The user input module in the presentation tier handles the validation of the data input in a client application based on the constraint repositories central to the monitored information system. Further, it provides the functionality to collect implicit and explicit user feedback on data instances. If a user enters or changes data in the client application, the defined constraints are checked and immediate validation feedback is given to the user. Based on the validation results, it can be decided if a data instance can be stored or needs further DQ improvement. Additionally, information concerning constraint violations, as well as implicit and explicit user feedback is sent to the connected business logic library.

The business logic library in the logic tier handles the validation of the data available in a server application. Further, it provides the functionality to generate metadata on the constraint checking results for the data validation and the implicit and explicit user feedback which is used by the DQ management application for DQ analysis and DQ monitoring. Both the module and the library base their data validation on the same constraint repositories, available in the monitored information system.

The DQ metadata is then sent to the DQ management application, as depicted in Fig. 7.1 in the centre with an arrow from the monitored information system to the DQ

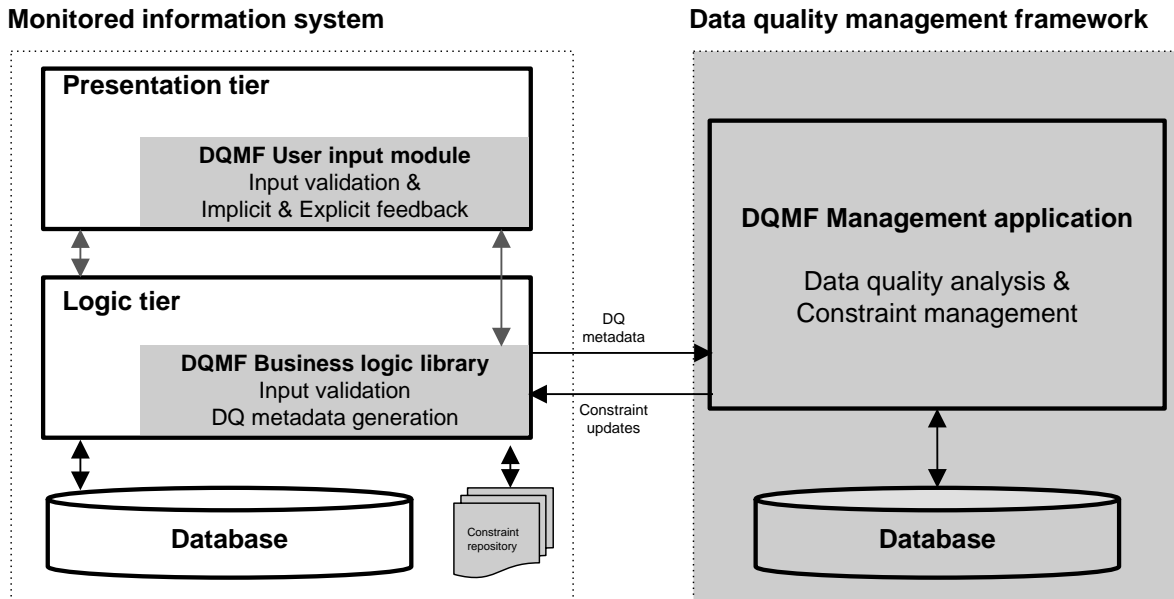


Figure 7.1: Data quality management framework components overview

management application. There, the metadata can be processed independently of the runtime of the monitored information system to provide feedback on DQ analysis and DQ monitoring to users of the DQ management application. The DQ management application stores the metadata in its own database and also provides functionality to manage the set of constraints in the monitored information system. When a user of the DQ management application adapts the constraints, the appropriate changes are propagated to the constraint repositories through the business logic library in the monitored information system, as depicted in Fig. 7.1 in the centre with an arrow from the DQ management application to the monitored information system. In the monitored information system, the constraint repositories are then adapted accordingly and with immediate effect, with the changed set of constraints being used for validation in the logic tier and presentation tier.

Thus, the DQMF provides three main capabilities to the monitored information system: (1) Unified constraint management; (2) Constraint checking; (3) DQ analysis. In the remainder of this section, we will summarise these capabilities briefly before providing more details on each of these capabilities in the subsequent sections.

Unified constraint management is achieved by defining constraints in constraint repositories located at a central place in the monitored information system. The constraints are composed in a non-technology-specific way, similar to *UnifiedOCL*, presented in Sect. 5.2, extended with the possibility to define individual constraint sets for specific user groups, which we previously proposed as ‘contracts’ in Chap. 4. This enables different feedback on DQ to be configured for different user groups. Each constraint set of a user group is stored in a separate constraint repository. These constraint sets are used for all validation components, such as the module and the library in the monitored information system, and are maintained centrally to guarantee constraint management without redundancy. In addition to previously presented work, this DQMF supports constraint repository inheritance. Thus, a repository can extend a similar repository which further reduces redundancy and simplifies constraint maintenance. Moreover,

the management of constraints in repositories enables the full functionality of adding, modifying and deleting constraints at runtime. Even if code changes in a logic tier would need a compilation phase, constraints become active in the monitored information system as soon as they are adapted in a repository.

With the module and library, **constraint checking** is guaranteed in multiple tiers which is necessary in many cases, as motivated earlier. As in the previous approaches, constraints are associated with a fine-grained penalty scale which enables multiple different constraint violation severities to be distinguished. We also differentiate between constraints with a high penalty which must be fulfilled in order not to reject the data, and constraints with a lower penalty which are only recommended to be fulfilled to increase the DQ. The different severity levels are again the basis for giving appropriate DQ feedback to a user, such as error messages and warnings, respectively. The constraints are stored in constraint repository files and are not hard-coded in a specific programming language. This enables the constraint repository files to be used in both the module and library where the files are parsed and prepared for data validation during the runtime. Therefore, changes to the constraints in the repository files during the runtime can be reflected in the data validation without further effort. For the real-time parsing of the constraint repository files, and checking of the constraints, we use an AOP approach in Java for the business logic library, which is applied to a Java-based logic tier. For the user input module, we use a constraint translation approach in JavaScript, which is applied to a web-based presentation tier.

The **data quality analysis** in the DQ management application, separated from the monitored information system, is based on the DQ metadata which is sent from the business logic library to the DQ management application whenever a data instance is persisted to the database. The metadata contains information about which constraints were violated and which were not, from which the constraint compliance of the instances can be calculated.

With the metadata and the user feedback information, all instances of the monitored information system can be mapped to our DQ vector space model. Thus, the data quality analysis is based on our DQ vector space model from which we derive measures and indicators for DQ concerning the three dimensional space with the dimensions constraint compliance, explicit feedback and importance, as introduced above. In previous approaches, we always only considered constraint compliance as the basis for indicating DQ. By taking the two additional dimensions into account, a more comprehensive analysis of DQ can be provided in an information system.

The explicit user feedback can be used to detect semantic issues in the data which cannot be indicated by constraint checking. For example, consider a constraint which defines that a person's name must not be null and must have a minimum length of 2 characters. If the input data contains 'Paul', the constraint is valid although the person's real name might be 'Peter'. Or consider that food items with a certain name, e.g. 'Apple, Belle de Boskoop', are supplemented with a mandatory appropriate image of the food in FoodCASE. If someone uploads an image of a different species of apple, e.g. an 'Apple, Gala', it is complex or impossible for a system to verify the apple in the image. For a human, semantic issues might be obvious to discover in the data and therefore explicit user feedback can be given to indicate the low DQ.

Further, explicit user feedback can be used to control the correctness of the set of constraints. Users can give explicit user feedback on the quality of data they are accessing or that they do not agree with the DQ feedback based on the constraint checking. By comparing the indication on DQ based on constraint compliance with that based on explicit user feedback, constraint compliance can be verified and additional indication on DQ provided.

The importance can be used to give an entity a certain weight within the comprehensive DQ assessment which helps to indicate what data instances of bad DQ should be improved first. Our new measures, DQ improvement tendencies and DQ improvement urgency score, base their indication for DQ on two or three dimensions, respectively, from our DQ vector space model. DQ improvement tendencies help to indicate whether there exist issues in the data and/or the constraints. The DQ improvement urgency score helps to rank data instances in terms of improvement urgency and to decide where DQ should be enhanced first. In our DQMF, we provide ways of collecting additional feedback information, measures and visualisations to analyse data and indicate DQ based on our DQ vector space model. In any case where constraints give a different indication on DQ to the explicit user feedback, a detailed analysis is appropriate where our DQ management application provides the necessary tools.

7.2 Unified constraint management

We propose a non-technology-specific constraint definition which is therefore independent of the programming languages with which it is used. The constraint repository files which contain the constraint definitions should be stored as close as possible to the monitored information system for fast read access. Defining and adapting constraints can be done directly in the repository files or with our DQ management application which provides a GUI with the appropriate functionalities.

As constraints, we consider *field constraints* and *class invariants*. Field constraints are defined for one specific attribute in an entity, for example, to check if an `email` attribute of a `Person` entity complies to the correct email pattern. Class invariants are defined for multiple attributes in an entity and can include multiple instances for the constraint checking. For example, if an entity `Employee` has the attributes `salary` and `bonus`, a class invariant can be defined to check that the `bonus` value is less than the `salary` value.

We decided to define the constraint repository files in JSON since it fulfils the following requirements: (1) The definitions should be in machine-readable format which can be consumed by different platforms and programming languages; (2) Humans should be able to easily modify the definitions and add new constraints; (3) Built-in parsers, for a variety of programming languages, should exist for the constraint definition format. An example content of a JSON constraint repository file is provided in List. 7.1. It is defined for a user group of a food science institution in Britain (`"groupId": "BRITAIN"`) and contains constraints for the entities `food` (`TdsFood`) and `sample` (`TdsSample`), with field constraints (`size`, `required`) for various fields (`TdsFood.name`, `TdsFood.englishname`, `TdsSample.name`) and a class invariant for the sample entity. A food (e.g. red cherries) is an entity from a TDS food list which is pooled to a food pool, a so-called sample entity (e.g. cherries).

```
1 { "groupId": "BRITAIN",
2   "extends": "DEFAULT",
3   "versioning": {"version": 128},
4   "settings": {"severityThreshold": 80},
5   "entities": {
6     "TdsFood": {
7       "name": {
8         "size": {
9           "identifier": "TdsFood.name.size#1",
10          "min": 2,
11          "severity": 60,
12          "message": "A food name should have a size of >= 2 characters",
13          "dimensions": [["accuracy",60]]
14        },
15        "required": {
16          "identifier": "TdsFood.name.required#1",
17          "severity": 100,
18          "message": "A food must have a name",
19          "dimensions": [["completeness",100]]
20        }
21      },
22      "englishname": {
23        "required": {
24          "identifier": "TdsFood.englishname.required#1",
25          "disabled": true
26        }}, ... // more fields
27    },
28    "TdsSample": {
29      "name": {
30        "required": {
31          "identifier": "TdsSample.name.required#1",
32          "severity": 100,
33          "message": "A sample must have a name",
34          "dimensions": [["completeness",100]]
35        }
36      },
37      "_invariants": [
38        {
39          "identifier": "TdsSample._invariants.sampleediblequantity#1",
40          "severity": 60,
41          "minWeight": 100,
42          "name": "sampleediblequantity",
43          "message": "The subsamples that contribute to the sample should
44            have an edible weight of > 100g",
45          "dimensions": [["completeness",60]]
46        } ... // more invariants
47      ]
48    }
49  }
50 }
```

Listing 7.1: JSON constraint repository example

A repository is organised in a tree structure and contains several repository-specific attributes and constraints defined for a set of entities, of the information system. A conceptual overview of the constraint repository structure is given in Fig. 7.2 for the tree levels 0 to 2 and Fig. 7.3 for the tree levels 2 to 5.

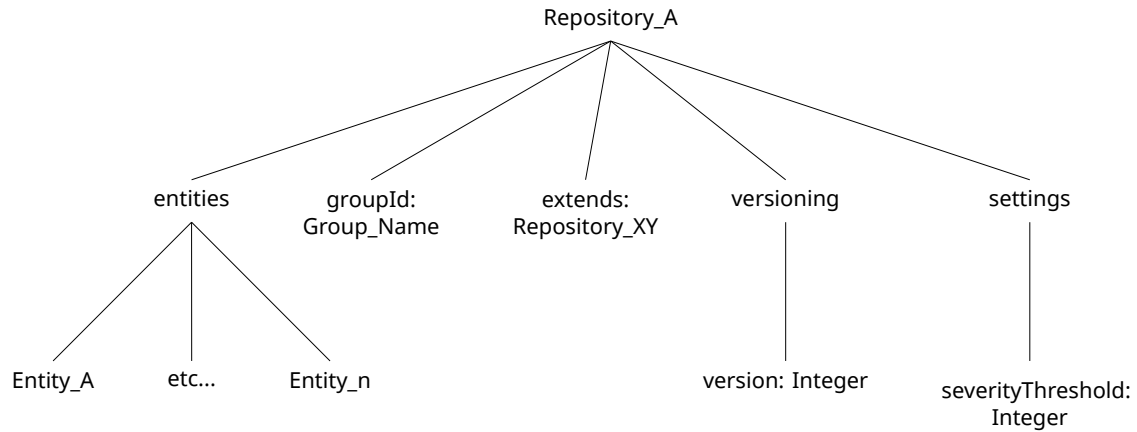


Figure 7.2: Constraint repository structure [level 0-2]

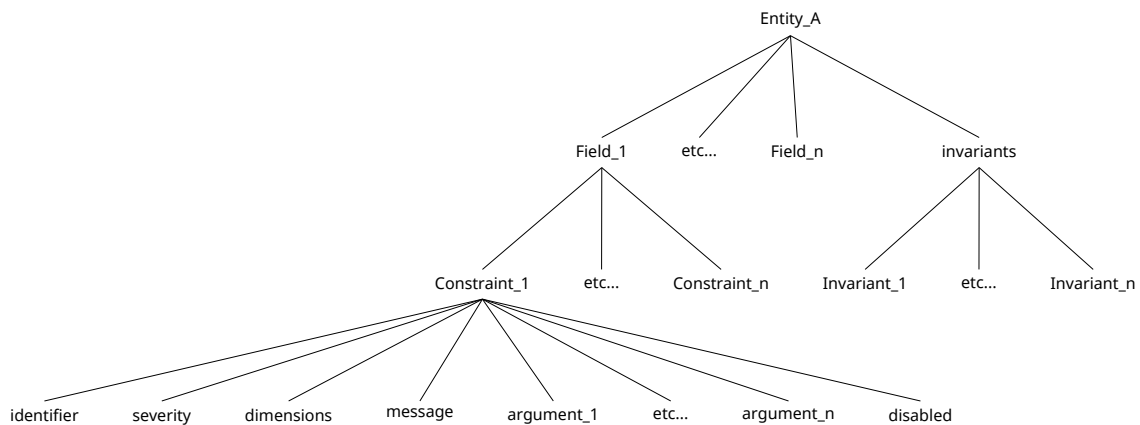


Figure 7.3: Constraint repository structure [level 2-5]

We will now describe the conceptual structure exemplified by the elements in List. 7.1.

The root node of the tree defines the repository name which is also used for the JSON repository file name. The general attributes, specific to a repository, are direct subnodes of the root node, depicted in Fig. 7.2 on the first level by the four rightmost attributes. These general attributes are defined in the first four lines of the JSON repository file, given in List. 7.1. The *groupId* node defines the group name for which the repository is applicable. The *extends* node defines from which repository the actual repository is extended (inheritance). The *versioning* node with its *version* value defines the actual repository version number. The version number is needed so that the DQ analysis can keep track of changes in the repositories to be able to recalculate appropriate DQ measures. The *settings* node with its *severityThreshold* value defines a threshold value above which constraint severity a constraint violation is considered an error.

The remaining direct root subnode *entities*, depicted in Fig. 7.2 on the first level by the leftmost attribute and in List. 7.1 on the fifth line, contains all entity nodes for which constraints are defined. List. 7.1 shows the entities `TdsFood` and `TdsSample` on lines 6 and 28, respectively. The subtree of an entity node *Entity_A* is depicted in Fig. 7.3.

An entity node, such as *Entity_A/TdsFood/TdsSample*, contains on a subnode for each of the appropriate entity fields and one subnode *invariants* for all class invariants (lines 7, 22, 29, 37 in List. 7.1). A field node (e.g. *Field_1/name*) contains subnodes for each field constraint definition where the *invariants* node contains subnodes for each class invariant. A field constraint is of a certain constraint type, such as **size** and **required** shown in List. 7.1 on the lines 8, 15, 23 and 30. The type determines the built-in validation logic which is explained later. A field constraint node (e.g. *Constraint_1/size*) and a class invariant node (e.g. *Invariant_1/lines 38-45* in List. 7.1) both require certain constraint definition attributes which are listed below. The appropriate attributes are also shown in Fig. 7.3 on the lowest level and in List. 7.1, for example, on lines 9-13 and 25.

- identifier *String*
- disabled *Boolean* (optional)
- severity *Integer*[0,100]
- message *String*
- dimensions *Array*
 - dimension_{1–n} *String*
 - severity_{1–n} *Integer*[0,100]
- argument_{1–k} (*String, any*)

The attribute **identifier** uniquely identifies a constraint which is used for constraint repository inheritance together with the optional attribute **disabled**. A constraint definition is ignored for the validation if **disabled** is set to true. The **severity** attribute defines the influence, in the range of [0,100], of the constraint violation for the DQ calculation, as already presented in previous approaches. The attribute **message** defines the message used for the constraint violation feedback in the user input module. Depending on the **severity** attribute and the *severityThreshold* value, it can be decided upon constraint violation, if an appropriate data instance can be stored and what kind of feedback (error/warning) is given to the user to provide a higher DQ. The attribute **dimensions** is used to associate one or multiple DQ dimensions (dimension_n) with an individual **severity** to a certain constraint, such as the ‘accuracy’ dimension with a severity of 60 to the **size** field constraint in List. 7.1 on line 13. Note that the severity of the constraint is independent of the severity/severities of the dimension(s). Dependent on the constraint type, additional attributes (argument₁ to argument_k) may need to be provided. Consider, for example, the **size** field constraint for the field `TdsFood.name` in List. 7.1 on lines 8 to 14. A built-in size field constraint type accepts the argument **min** as an integer and consequently is violated during validation if the given field value is below the **min** value. A **required** field constraint, for example, does not need any additional arguments and is violated if the given field value is **null**.

In our DQMF, multiple built-in field constraint types are available, where we selected the set of predefined constraint types based on the available constraints in BV. The complete list of supported built-in field constraint types is available in Appendix D.

In contrast to the built-in field constraint types for which our DQMF provides the

appropriate predefined validation, class invariants need a custom implemented validator. The according constraint checking logic is indicated in the class invariant definition with the `type` attribute and loaded by our business logic library upon validation.

Constraint repository inheritance is achieved in our DQMF by defining a single inheritance from one repository to another with the `extends` attribute, as described above. If *Repository_B* extends *Repository_A*, the constraints from *Repository_A* are included in *Repository_B*. Constraints are excluded from inheritance in two cases: (1) If the same constraint from *Repository_A* is already defined in *Repository_B* with the same entity, same field and same constraint type; (2) If a constraint in *Repository_B* has the same `identifier` defined as a constraint in *Repository_A* and the constraint in *Repository_B* has the attribute `disabled` set to `true`. Thus, case (1) enables constraints from *Repository_A* to be overwritten in *Repository_B*, and case (2) enables certain constraints to be explicitly excluded from *Repository_B*, as shown in List. 7.1 on lines 22 to 26.

7.3 Constraint checking

Our DQMF provides constraint checking with our user input model in the presentation tier and the business logic library in the logic tier. We further provide, as introduced in previously presented approaches, a fine-grained scale to associate constraints with a constraint violation severity to calculate and give appropriate feedback on DQ. The severity scale and `severityThreshold` were already described in the previous section. The checking of a constraint always results in a validation to `true` or `false`. Thus, a constraint is either violated or not. Only the association with a severity and the interpretation based on a severity threshold provides the possibility to give a differentiated feedback on DQ.

Based on the `severityThreshold`, we can also distinguish between ‘error’ messages and ‘warning’ messages when giving feedback to the users in the GUI. Thus, on mandatory fields where a constraint with a severity equal or above the `severityThreshold` is defined, we provide ‘error’ messages in red colour while on non-mandatory fields where a constraint with a severity below the `severityThreshold` is defined, we provide ‘warning’ messages in orange colour. We distinguish similar severity levels, as presented in the approach in Chap. 4, namely (Constraint violation type=[Severity range]):

Notice=[0-19], Warning Low=[20-39], Warning Medium=[40-59], Warning High=[60-79], Error=[80-100]

As in previous approaches, the severity levels can be customised and are not limited to our choice of levels.

We base our **constraint checking in the logic tier** on domain entities as is usual in OO programming. In the business logic, instances should comply with the constraints defined on each entity. It is important to prohibit instances from being stored in the database due to deficient DQ if at least one mandatory constraint with a severity higher than the threshold is violated. This is ensured by our business logic library which prevents modifications of an instance which would cause the violation of a mandatory constraint. Our library catches all events in the business logic which try to modify instance attributes and injects the appropriate validation logic, based on AOP, prior to the modification. If a mandatory constraint would be violated, the

modification is aborted.

With AOP it is possible to inject *cross-cutting* concerns in the code at defined *join points* which are controlled by *aspects*, as described in Sect. 2.4.6. We inject constraint checking at two join points in the business logic: (1) Before a setter method of an instance is called in order to validate field constraints; (2) Before an instance is persisted to the database in order to validate class invariants.

In accordance with widely accepted Java style guidelines, we defined two catch method calls in the form of `Entity.setField(*)` for the first join point, and method calls in the form of `Entity.save()` for the second join point. Aspects are stored in files and contain the join point definition and the code that is injected at the places that match the join points.

Consider the first constraint example from List. 7.1 where a `name` attribute of a `TdsFood` entity should be at least two characters long, which is defined with a `size` constraint. The join points would be before the setter method `setName` and the entity save method `TdsFood.save` are invoked. There, the validation code for all appropriate constraints, such as the `size` constraint for the `name` attribute, is injected.

The data that is needed for constraint checking in the validation logic are the entity, the field, input parameters and the appropriate constraint. For example, if a join point matches the method call `TdsFood.setName('Apple')`, the data 'TdsFood', 'Name' and 'Apple' is exposed by the aspect. The business logic library uses this information from the aspect during the parsing of constraint repository files for data validation. It traverses the tree structure, explained in the previous section, to find the appropriate constraints which have to be checked, depending on which field or class is involved in the validation.

For the example from above, the library would parse the repository file and find the `size` and `required` constraints that are defined for the field `name` of the entity `TdsFood`. Then it checks these constraints with the input data 'Apple' to validate that the string is not `null` and that it is at least two characters long. Since no constraints would be violated, the `TdsFood.setName('Apple')` method would be invoked. If the input data is not `null` but contains only one character, the setter would be invoked since the severity of the `size` constraint is set to 60 which is below the `severityThreshold`, of 80. If the input data is `null`, the validation would abort the invocation of the setter method.

For **constraint checking in the presentation tier**, we focus on the scenario of a web interface for data input, but the concepts presented here are not restricted to web interfaces. With a presentation tier implemented as web interface in HTML/JavaScript and connected to a logic tier implemented in a programming language such as Java, heterogeneous constraint checking would be needed. Thus, the constraint definition and data validation logic would be duplicated for the client and server application, as motivated earlier.

Our DQMF overcomes this issue with the injection of our user input module to the presentation tier. Fig. 7.4 shows the logic tier of the monitored information system which uses our business logic library in the lower part. The presentation tier of the monitored information system which uses our user input module is shown in the upper part.

The same constraint repositories, used in the business logic library are passed through the library to the module which uses them for its data validation. Instead

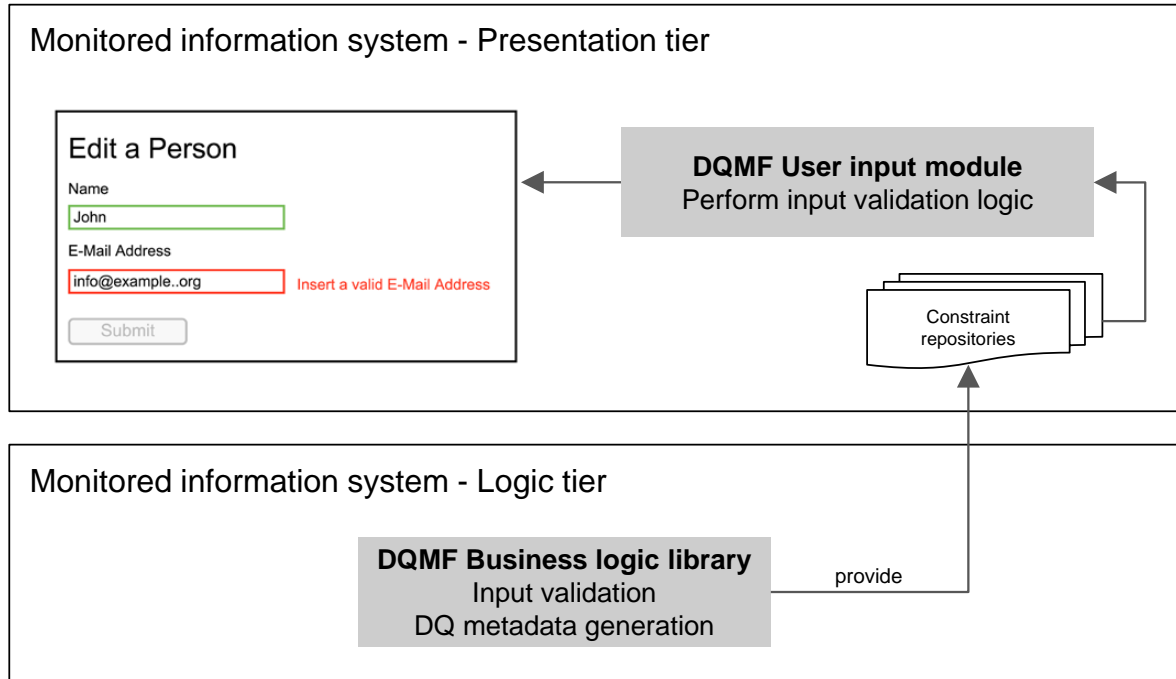


Figure 7.4: Client-side constraint checking

of adding the validation logic explicitly, the module parses the constraint repositories and applies constraint checking to the HTML form field for the input data validation. Depending on the constraint violation severities and the severity threshold, the appropriate feedback can be given to the user directly at the form fields. The example in the upper left part of Fig. 7.4 shows the input of an incorrect email address, which violates a mandatory constraint, and therefore the form field is marked with a red border with an appropriate message alongside and the ‘Submit’ button is disabled. Thus, the user is informed about the constraint violation on a specific field, and prevented from submitting the data with deficient DQ to the logic tier.

The `name` HTML attribute defines the association between the constraint repositories and the HTML forms for web interfaces. For example, the form name (e.g. `Person`) with an input field (e.g. `email`) is associated by the user input module with an entity (e.g. `Person`) and the appropriate entity field (e.g. `email`).

7.4 Data quality analysis

Our DQ management application provides means to quantify and analyse DQ which helps to solve DQ issues. The provided information and tools are helpful for a DQ analyst, who can use it as a base for his decisions on enhancing the quality of data in an information system. The DQ management application supports a DQ analyst in answering three essential questions for improving DQ: (1) How good is the quality of the data in the information system? (2) Did the DQ significantly change over time? (3) What data items need quality improvements most urgently, and how can they be improved?

Answering the first question gives an indication of how reliable the data is for decision taking. The better the DQ, the more valuable is the data for basing further

processes on it. Moreover, the underlying measures for answering the first question build the foundation for answering the other two questions.

The second question is concerned about the evolution of DQ. It aims to identify changes to the information system which negatively influenced the DQ in a large scale. Basically, four types of potential changes exist: (1) Changes to the data; (2) Changes to the constraints; (3) Changes to the DQ measures; (4) Time-dependent constraints turn from valid to invalid or vice versa (e.g. future and past constraints). A drop in DQ based on such changes during time and their causes should be detected by answering the second question. Issues, for example, could be wrong or missing DQ constraints, failed data import processes and bugs in the information system. Note that a fifth type would be the receipt of negative subjective DQ assessment feedback, if it is included for indicating DQ in an information system, as described later in this section. Nonetheless, for this type, the DQ would already be lowered, but not identified before the subjective DQ assessment.

Answers to the third question should help identify the data instances that contribute the most to low DQ. In the DQ management application, we can indicate DQ issues on aggregated levels but also down to an entity and data instance level, as already proposed in previous approaches. It is possible to detect and correct instances with the most issues concerning DQ but also to give an indication of potential error sources that lead to bad DQ. This is based on objective DQ assessment by analysing the constraint compliance and also on subjective DQ assessment by analysing implicit and explicit user feedback.

As depicted in Fig. 7.1 in the centre, our business logic library continuously sends data validation metadata from the monitored information system to our DQ management application. We will now describe what is included in the metadata before we describe the concept of our DQ vector space model. The metadata contains information about the (1) constraint compliance of the instances, (2) explicit feedback provided by the users, and (3) implicit feedback. It is stored in the DQ management application database and builds the basis for our DQ vector space model and all DQ monitoring and analysis support. The metadata consists of single metadata objects which contain DQ information specific to an instance and entity in the monitored information system. Due to privacy and redundancy reasons, the metadata does not contain any data from the information system.

(1) The **constraint compliance** information is generated whenever an object instance is persisted to the database in the monitored information system. The AOP aspect in the business logic library gathers the list of all constraints of the instance including their severity and the information whether a constraint was violated or not.

(2) The user input module continuously sends the **explicit user feedback** for object instances to the business logic library.

(3) As soon as an instance is viewed in the presentation tier, an appropriate log entry of the **implicit user feedback** is sent from the user input module to the business logic library.

The metadata about constraint compliance, explicit and implicit user feedback is continuously sent from the business logic library to the DQ management application.

7.4.1 Data quality vector space model

Based on the metadata of the object instances, DQ indications concerning the three dimensions **constraint compliance**, **explicit user feedback**, and **importance** can be calculated and each instance can be mapped to a data point in our DQ vector space model. We will now define our DQ vector space model, depicted in Fig. 7.5, and each dimension in more detail.

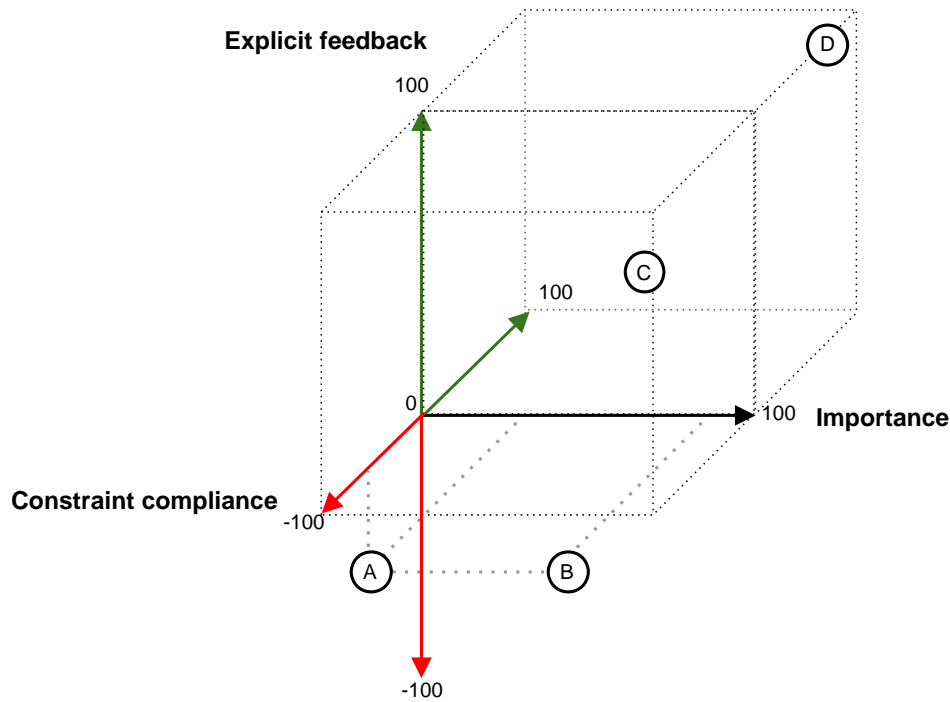


Figure 7.5: Data quality vector space model

The possible values in the dimensions **constraint compliance** and **explicit user feedback** are in the range of $[-100, 100]$, while the range for the **importance** dimension is $[0, 100]$, which together define the space of our DQ vector space model. A value of 100 for constraint compliance expresses the highest DQ in this dimension since all constraints are fulfilled. In contrast, a value of -100 for constraint compliance expresses the lowest DQ in this dimension since all constraints are violated. A value of 0 for constraint compliance expresses that the fulfilled and violated constraints are cancelling each other out or that there are no constraints defined. A value of 100 for explicit user feedback expresses the highest DQ in this dimension where only positive feedback is received. In contrast, a value of -100 for explicit user feedback expresses the lowest DQ in this dimension where only negative feedback is received. A value of 0 for explicit user feedback expresses that the positive and negative feedback is in balance. A value of 100 for importance expresses the highest importance for an instance while a value of 0 for importance expresses the lowest importance for an instance.

Four exemplary instance mappings are depicted in Fig. 7.5 where the data points for each instance are marked in the DQ vector space model with the characters (A), (B), (C), and (D). Instance A and B, both have a low constraint compliance and negative explicit feedback. This means that users indicate DQ issues which is in line with the low DQ indication from the constraint violations or that users indicate issues with the

constraint definitions. Thus, the data and constraint details of both instances should be analysed to find the sources that lower the DQ. However, instance *B* has a much higher importance in the system than instance *A*. Therefore, its influence on DQ is stronger and should be considered first for the improvement of DQ. Instance *C* has a very high importance and received very positive feedback but a very low DQ indicated by the constraint compliance. Thus, the negative constraint compliance is not in line with the positive user feedback which is an indication that issues exist in the constraint definitions for this instance, if it is assumed that the feedback is correct. Instance *D* has a high importance, a very positive feedback and a very high constraint compliance. Thus, there is no indication that there exist any issues concerning DQ with instance *D*.

Constraint compliance is calculated on an instance level which results in a constraint compliance value for each instance. Eq. 7.1 defines the constraint compliance cc_i of an instance *i* with constraints c_1, \dots, c_n .

$$cc_i = 100 \cdot \left[\frac{\sum_{j=1}^n severity_j \cdot valid_j}{\sum_{j=1}^n severity_j} \right] \in [-100, 100] \quad (7.1)$$

The severity attribute of constraint *j*, as defined in the constraint repository, is represented by the term $severity_j$. The term $valid_j$ is either 1 if the constraint c_j is not violated or -1 if the constraint c_j is violated. Taking into account the non violated constraints as well differentiates this calculation from constraint compliance in previous approaches.

Concerning the **explicit user feedback**, a user *k* can provide an explicit rating value $ef_rating_{i,k}$ of the perceived DQ of an instance *i*. The rating value is in the range of $[-2,2]$, which represents a five point Likert-type scale for the rating values: very bad = -2, bad = -1, neutral = 0, good = 1, very good = 2. Additionally to the explicit rating value, a user can provide details on the rating with a text comment, where for example hints can be given on how to improve an instance with corrected data or missing constraints.

The calculation of the explicit user feedback is presented in Eq. 7.2, which is based on the collected explicit ratings $ef_rating_{i,k}$ of an instance *i* by all users where we also consider a reputation score $reputation_k \in \mathbb{N}_{>0}$ for each user *k*. The rating of a user with a higher reputation score has a stronger influence on the overall rating than a user with a lower reputation score.

$$ef_i = 100 \cdot \left[\frac{\sum_{k=1}^n reputation_k \cdot ef_rating_{i,k}}{\max |ef|} \right] \in [-100, 100] \quad (7.2)$$

To normalise the ratings to the range $[-1,1]$ we use the highest absolute feedback rating available in the system $\max |ef|$ in the denominator.

The **importance** dimension is not an indicator for DQ but used to classify instances and entities due to their importance in the information system. The higher the relevance of an instance/entity is, to users and the system's purpose, the higher is its importance. We consider instances/entities with a higher importance as also more important to be corrected, if the other dimensions indicate bad DQ. In Eq. 7.3 our calculation of the importance imp_i of an instance *i* of entity type *e* is presented, where we distinguish the two factors *interest* and *entity importance* which contribute to the importance.

$$imp_i = \max(100, \alpha \cdot entityImportance_e + (1 - \alpha) \cdot 100 \cdot \frac{interest_i}{\max |interest|}) \in [0, 100] \quad (7.3)$$

The $interest_i$ value represents how often the instance i was viewed by the users. It is derived from the implicit user feedback, as explained above. Thus, an instance is considered to be of higher interest if it is often opened. The $entityImportance_e \in [0, 100]$ is the importance indication due to the relevance of instance i to the information system, which can be configured for each entity, as explained above, and which is adopted for all instances of an entity.

The interest in data instances is tracked only for instances which a user can directly view in the presentation tier. Nonetheless, there can exist instances which are never opened by a user but are of high importance to the system's purpose. Consequently, such instances will have a **zero** interest but can contribute with their entity importance. Further, instances could contain wrong data which could cause that they are never fetched by correct database queries and consequently would not be shown in the presentation tier.

We use the parameter $\alpha \in [0, 1]$ to weight the impact of the interest compared to the entity importance. Further, we normalise with $\max |interest|$ so that the potentially widespread interest values result in a value in the range $[0, 1]$. This denominator represents the highest absolute interest value of any instance in the system.

If α is set to 1, the importance imp_i of an instance equals the entity importance, $entityImportance_e$. This can be useful, for example, if an information system receives a lot of automated requests which cause a lack of trust in the data of interest.

7.4.2 Data quality improvement tendencies

In the previous subsection, we have already presented examples of how single data points in the DQ vector space model can be interpreted. Since it would be a tedious task for a DQ analyst to interpret each data point, we provide further measures, which support DQ analysts in identifying data instances with low DQ and consequently help increasing the DQ in an information system. In this subsection, we present our *DQ improvement tendencies* and in the subsequent subsection our *DQ improvement urgency score*, which are both measures based on the DQ vector space model.

If an instance receives negative explicit user feedback and/or its constraint compliance is low, we generally can assume that its DQ is low. However, several cases can cause low DQ: (1) The data instance is of low quality; (2) The data instance is of good quality, but the constraints defined on the instance are wrong; (3) The data instance is of low quality, and the constraints defined on the instance are wrong.

We assume that the explicit user feedback is a trustworthy source of information, at least if multiple users have given their consistent feedback. Thus, we can use the explicit user feedback as a comparison to our constraint compliance results, as explained above. This comparison can provide indications on the different causes of low DQ, presented in the previous paragraph. To provide such indications, we introduce our DQ improvement tendencies. Although we assume that the feedback is trustworthy, it should always be considered that there is the possibility of incorrect feedback. Therefore, in case of doubt, the feedback should be checked and deleted if wrong.

In any case, it should be possible to improve the DQ by correcting data instances and/or constraint definitions. Correcting actions should always not only correct data but also correct or add constraints, if possible, to prohibit the input of data with low DQ in the future. However, constraints cannot be used to identify low DQ in all cases, especially in case of semantic issues. Therefore explicit user feedback is the only identification possibility of low DQ in such cases.

With our DQ improvement tendencies, we compare the subjective (explicit user feedback) and objective (constraint compliance) DQ assessments, as proposed by Pipino et al. [144]. (1) We gather subjective and objective DQ assessment information from the explicit user feedback and constraint compliance, respectively; (2) We compare the results of the assessments by placing a data instance on the surface of our DQ vector space model with the dimensions *Constraint compliance* and *Explicit feedback* as depicted in Fig. 7.6 on the x-axis and y-axis, respectively; (3) We identify discrepancies, and determine root causes of discrepancies with the help of our DQ improvement tendencies; (4) We determine necessary actions, such as *improve data* or *adapt constraints* based on the outcome of (3). A DQ analyst can use our DQMF to follow these steps and carry out the necessary actions for DQ improvement.

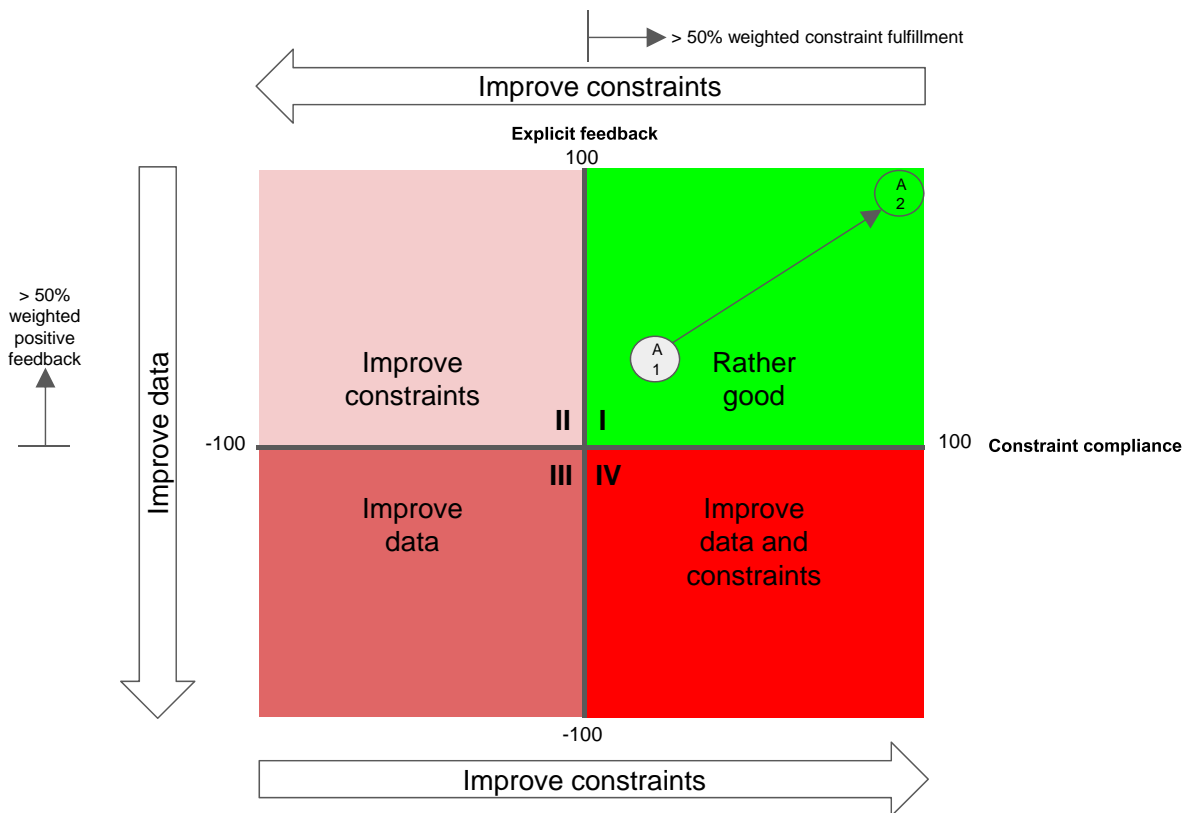


Figure 7.6: Data quality improvement tendencies

Similar to the approach of Pipino et al. [144], described in Sect. 2.6.2, we propose the categorisation of four combinations of objective and subjective DQ assessments, depicted in Fig. 7.6 by the four quadrants. As an extension to the work of Pipino et al. [144], we provide for each category/quadrant the analysis of the root causes and recommendations for corrective actions, expressed by our DQ improvement tendencies. An instance is located in one of the four quadrants based on its dimension values for ex-

PLICIT feedback and constraint compliance, and consequently assigned to the appropriate tendency.

We describe now our distinction of the four tendencies, which are presented in Fig. 7.6 in quadrants I to IV. Each tendency indicates a different prioritisation for DQ improvement, which is depicted by the different background colours. The prioritisation order is consistent with the quadrant numbering, where quadrant I has the lowest DQ improvement priority and quadrant IV the highest DQ improvement priority.

Rather good (quadrant I): The feedback and constraint compliance are rather high. Both dimensions are indicating good DQ, which is the desirable state. We cannot give clear improvement recommendations, if at all necessary. This is the regular case of high DQ. The priority to improve DQ is rather low.

Improve constraints (quadrant II): The feedback is rather high but the constraints compliance is rather low. The dimensions are indicating a different DQ. Since we trust the feedback, we assume that the DQ is good, but the constraints should be improved to better reflect the data requirements. This is the case of good DQ where the constraints give a wrong lower DQ indication. The priority to improve DQ is still rather low since the DQ is assumed to be good and the data validation is rather too strict, which prohibits entering data of low DQ in the future.

Improve data (quadrant III): Both the feedback and constraint compliance indicate low DQ. This is not a desirable state, but at least there are no contradicting indications. The data should be improved to increase the DQ. This is the regular case of low DQ. The priority to improve DQ is rather high since there is a strong indication that the DQ in the information system is low.

Improve data and constraints (quadrant IV): The feedback is rather low but the constraints compliance is rather high. The dimensions are indicating a different DQ. Since we trust the feedback, we assume that the DQ is low, and therefore the data should be improved to increase the DQ. Further, the constraints should be improved since they indicate a contradicting good DQ. This is the case of low DQ where the constraints give a false higher DQ indication. We rate the priority to improve DQ for this tendency even higher than in quadrant III, since the DQ is assumed to be low and the data validation is rather too loose, which enables data of low DQ to be entered in the future. Not only the data but also the constraints must be enhanced, if possible, to ensure correct data validation in the future. This tendency can also indicate that there are semantic issues, which cannot be expressed by constraints. In such a case, it would be even more important to identify the issues and find ways, beyond constraint checking, to ensure a high DQ.

These tendencies give recommendations on how to improve DQ but they are not to be understood as clear categorisations since the transitions within and between the quadrants/tendencies are smooth and open to interpretation. For example, an instance at the data point A_1 in Fig. 7.6 is assumed to be of much lower DQ than the data point A_2 , since fewer constraints are fulfilled and the feedback is less positive. Thus, A_1 needs more improvement than A_2 , although they are assigned to the same tendency.

Therefore, we also indicate general tendencies for the transitions within and between the DQ improvement tendencies, depicted by the arrows in the upper, lower and left part of Fig. 7.6. The arrow on the left *Improve data* indicates that the lower the explicit feedback value is, the lower the DQ is and the higher the likelihood that the data has to be improved. This is consistent for both comparing the upper quadrants I and II

with the lower quadrants III and IV and within a quadrant for single data points. The arrows labelled with *Improve constraints* at the top, valid for quadrants I and II, and the bottom, valid for quadrants III and IV, indicate the likelihood that the constraints have to be improved. For quadrants I and II it increases from right to left, since the DQ is assumed to be rather good but the constraints compliance gets more contradictory to the left. For quadrants III and IV, it increases in the opposite direction, from left to right, since the DQ is assumed to be rather low but the constraints compliance gets more contradictory to the right.

7.4.3 Data quality improvement urgency score

Our DQ improvement urgency score is a measure for DQ which includes all three dimensions from our DQ vector space model, including the importance dimension which is not considered in our DQ improvement tendencies. Based on the values for each dimension, we calculate a score which indicates how important it is to improve DQ for a certain instance. The DQ improvement urgency score is in the range of $[0, 100]$. An instance with an urgency score of 0 does not need improvement, while it is very urgent to improve an instance with an urgency score of 100. This enables a DQ analyst to distinguish data instances with a high priority from ones with lower priority. The DQ improvement tendencies, presented in the previous subsection, already give an indication of the improvement priority. DQ analysis based on the tendencies can be seen as a tool to quickly categorise instances into one of the four improvement priorities, also as an indication of the root causes with recommendations on what actions have to be taken to improve DQ. The DQ improvement urgency score goes a step further, to provide more distinct analysis for each data point than the DQ improvement tendencies, also including the third dimension *importance*.

We will explain first, how we derived our new measure DQ improvement urgency score before presenting the complete calculation. Therefore, consider again the surface of our DQ vector space model with the dimensions *constraint compliance* and *explicit feedback* as depicted in Fig. 7.7.

The green arrows, show in which direction generally the DQ increases considering the x-axis/y-axis but not the divergence between the dimensions. The higher the constraint compliance and/or the higher the explicit feedback, the higher also the DQ. The red arrows close to the dashed angle bisections in the four quadrants, symbolise how the urgency to improve the quality of constraints and/or data increases in the direction of the arrows, taking both dimensions into consideration. In quadrant I and III, the feedback and constraint compliance correspond on the dashed angle bisections and the urgency to improve DQ increases the further down left a data point is located. Thus, a point in the top right corner has a very low urgency to improve DQ, a point at the origin of the coordinate system a higher urgency and a point at the lower left corner a very high urgency. By deviating from the angle bisections in quadrants I and III, the divergence between the two dimensions increases.

In contrast, the maximal divergence between the dimensions resides in quadrants II and IV on their angle bisections. There, the urgency to improve the quality of constraints and/or data is lowest at the origin of the coordinate system and increases towards the upper left and the lower right corner, if the divergence between the dimensions is considered. Deviating from the angle bisections in quadrants II and IV will

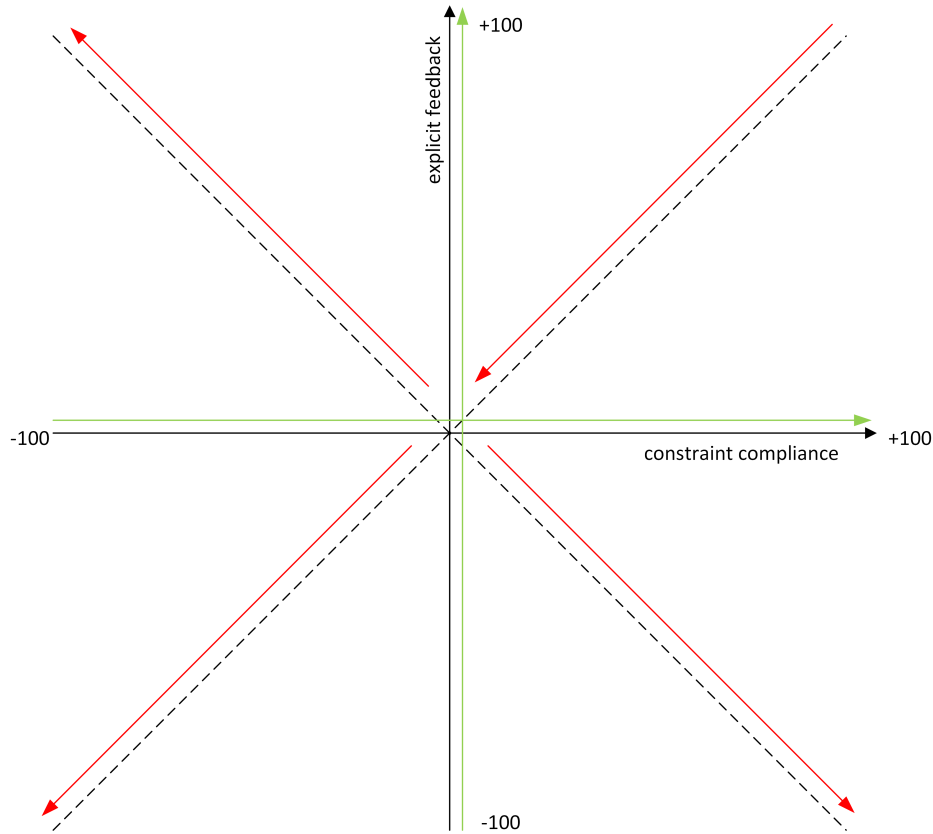


Figure 7.7: Coordinate system with angle bisections

decrease the divergence between the two dimensions.

Since the divergence of the two dimension values is an indication for issues in the quality of constraints and/or data we consider the angle from the origin to a data point in a quadrant for determining the urgency score. We use the polar form of the coordinate system, as known from complex numbers, depicted in Fig. 7.8.

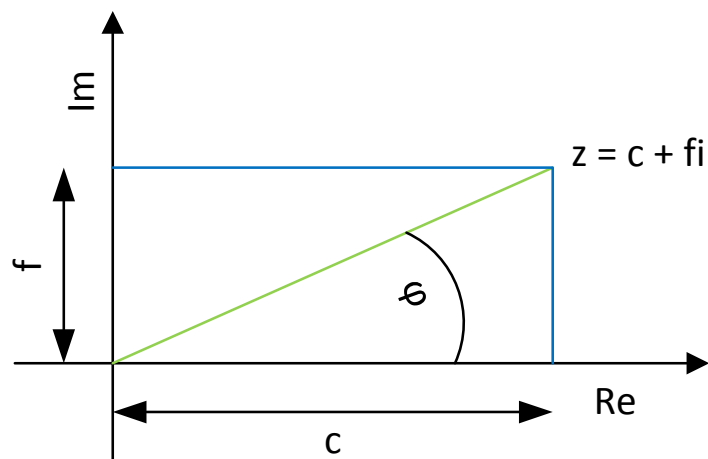


Figure 7.8: Complex plane transformation

As the real part, we consider the constraint compliance c , while we consider the feedback f as the imaginary part, as shown in Eq. 7.4 for the complex number z .

$$z = c + fi \quad (7.4)$$

The values c and f are the values for constraint compliance and explicit user feedback, respectively, from a data point in the DQ vector space model.

According to Eq. 7.5, we calculate the absolute value or magnitude r of the complex number.

$$r = |z| = \sqrt{c^2 + f^2} \quad (7.5)$$

Further, we calculate the angle or phase φ with Eq. 7.6.

$$\varphi = \arg(z) = \begin{cases} \arctan\left(\frac{f}{c}\right) & \text{if } c > 0 \\ \arctan\left(\frac{f}{c}\right) + \pi & \text{if } c < 0 \text{ and } f \geq 0 \\ \arctan\left(\frac{f}{c}\right) - \pi & \text{if } c < 0 \text{ and } f < 0 \\ \frac{\pi}{2} & \text{if } c = 0 \text{ and } f > 0 \\ -\frac{\pi}{2} & \text{if } c = 0 \text{ and } f < 0 \\ \text{indeterminate} & \text{if } c = 0 \text{ and } f = 0 \end{cases} \quad (7.6)$$

For the calculation of an overall DQ improvement urgency score, we combine four measures: (1) The urgency of improving DQ based on the explicit feedback; (2) The urgency of improving DQ based on the constraint compliance; (3) The urgency of improving the constraints based on the divergence of explicit feedback and constraint compliance; (4) The importance. The polar form, as explained above, enables us to determine the urgency to improve the quality of constraints and/or data, based on the two dimensions explicit user feedback and constraint compliance.

To extract the imaginary part from the polar form, i.e. the explicit user feedback, we use Eq. 7.7.

$$y = \text{Im}(z) = r \cdot \sin(\varphi) \quad (7.7)$$

We further map the imaginary part from the range $[-100, 100]$ to the range $[0, 100]$ and invert it to get the Eq. 7.8 which represents the urgency based on the explicit feedback u_f .

$$u_f = \frac{100 - r \cdot \sin(\varphi)}{2} \in [0, 100] \quad (7.8)$$

Likewise, we extract the real part from the polar form, i.e. the constraint compliance, with Eq. 7.9 and transform it in a similar manner as above to get Eq. 7.10 which represents the urgency based on the constraint compliance u_c .

$$x = \text{Re}(z) = r \cdot \cos(\varphi) \quad (7.9)$$

$$u_c = \frac{100 - r \cdot \cos(\varphi)}{2} \in [0, 100] \quad (7.10)$$

If the indication of DQ diverges for constraint compliance and explicit feedback, the constraints have to be improved, since we assume precedence of explicit user feedback over constraint compliance, as previously explained.

With the help of the polar form, we can express an appropriate penalty score by multiplying the scaled amplitude with the absolute sine of the phase. We consider the offset $-\frac{1}{4}\pi$ for the phase, so that the phase reaches its maximum at the angle bisections of the quadrants II and IV at $\frac{3}{4}\pi$ and $\frac{7}{4}\pi$ and its minimum at the angle bisections of

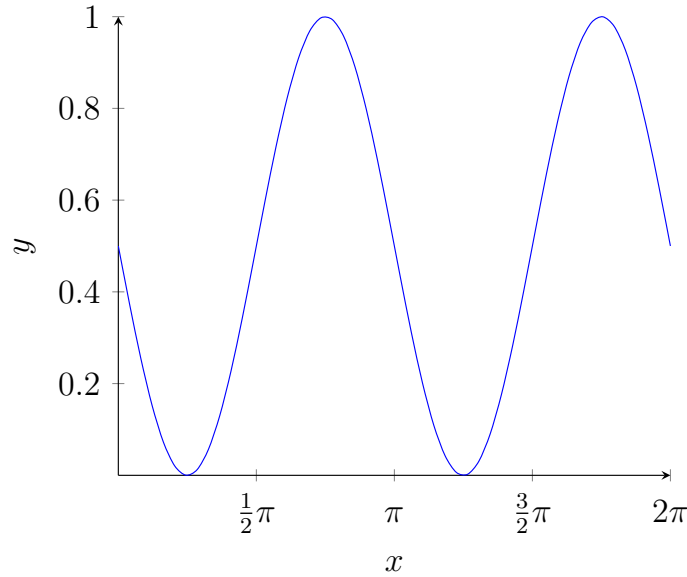


Figure 7.9: Shifted absolute sine for constraint compliance

quadrants I and III at $\frac{1}{4}\pi$ and $\frac{5}{4}\pi$. The resulting shifted absolute sine for constraint compliance is depicted in Fig. 7.9.

The penalty for the urgency of improving the constraints u_p , based on the divergence of explicit feedback and constraint compliance, is consequently calculated with Eq. 7.11.

$$u_p = \frac{r}{\sqrt{2}} \cdot |\sin(\varphi - \frac{1}{4}\pi)| \in [0, 100] \quad (7.11)$$

The last measure that we take into account for the DQ improvement urgency score is the importance dimension. Its value range is $[0, 100]$, where 0 represents a *not important* and 100 a *very important* instance, as described in Sect. 7.4.1.

We define the importance imp with Eq. 7.12, where i is the importance value from the DQ vector space model and w_i is a weight to control the influence of the importance on the final DQ improvement urgency score.

$$imp = \frac{1 + w_i \cdot i/100}{1 + w_i} \in (0, 1] \quad (7.12)$$

We normalise the overall DQ improvement urgency score to the range $[0, 100]$. Therefore, the importance imp , as defined in Eq. 7.12, has the upper bound $i = 100$ while its lower bound is determined by the weight of the importance w_i . Thus, the higher the weight of the importance w_i is, the closer the importance imp can get to zero.

Now we can sum up the urgency based on the explicit feedback u_f , the urgency based on the constraint compliance u_c , and the penalty for the urgency of improving the constraints u_p to define an overall *urgency* which does not include the importance yet, as depicted in Eq. 7.13.

$$urgency = w_f \cdot u_f + w_c \cdot u_c + w_p \cdot u_p \quad (7.13)$$

When considering again the decreasing and increasing characteristics for the urgency described above with Fig. 7.7, u_f and u_c represent the characteristics of the green arrows,

while u_p represents the characteristics of the red arrows.

We multiply each measure part (u_f, u_c, u_p) with an appropriate weight (w_f, w_c, w_p) to be able to control the contribution of each measure part to the *urgency*. To scale the urgency to the range $[0, 100]$, it has to be divided by the maximal possible weighting factor.

Since u_c, u_f and u_p are in the same range $[0, 100]$ we can sum up the three weighting factors, as defined in Eq. 7.14, to calculate the *scale*.

$$scale = sum(w_c, w_f, w_p) \quad (7.14)$$

As a last step, we combine the importance *imp* by multiplying it with the *urgency* and dividing it by the *scale* to derive the final formula for the DQ improvement urgency score *urgencyScore*, defined in Eq. 7.15. The importance *imp* is in the range $(0, 1]$ and therefore reduces the *urgency* accordingly.

$$urgencyScore = \frac{imp \cdot urgency}{scale} \quad (7.15)$$

A list of all parameters for configuring the calculation of the DQ improvement urgency score *urgencyScore* is provided in Tab. 7.1.

Parameter	Description
$c \in [-100, 100]$	constraint compliance
$f \in [-100, 100]$	explicit feedback
$i \in [0, 100]$	importance
$w_c \in \mathbb{R}$	weight of constraint compliance
$w_f \in \mathbb{R}$	weight of explicit feedback
$w_i \in \mathbb{R}$	weight of importance
$w_p \in \mathbb{R}$	weight of penalty function

Table 7.1: Urgency score parameters

The weights w_f, w_c, w_p , and w_i can be any rational number. They do not have to be in a certain range, such as $[0, 1]$, since only the ratio between the weights is important. Our default configuration of the weights is depicted below, where the precedence of explicit user feedback over constraint compliance is adhered.

$$\begin{aligned} w_c &= 1 \\ w_p &= 2 \cdot w_c = 2 \\ w_f &= 2 \cdot w_p = 4 \\ w_i &= 1 \end{aligned}$$

With a default weight of 1 for the importance weight w_i , the urgency can be reduced with the factor 0.5 to 50% at most for the minimum importance value i of 0. The higher the weight, the closer to zero the importance can get and therefore the more reduced the urgency can be.

With Fig. 7.10, we want to provide a clearly understandable visual representation of the DQ improvement urgency score and enable a comparison with the DQ improvement tendencies, presented in Fig. 7.6. Therefore, the *urgencyScore* in Fig. 7.10 is reduced to the two dimensions *explicit feedback* and *constraint compliance* by setting the weight of the importance w_i to zero. Defining w_i larger than zero would decrease the DQ improvement urgency according to the importance value.

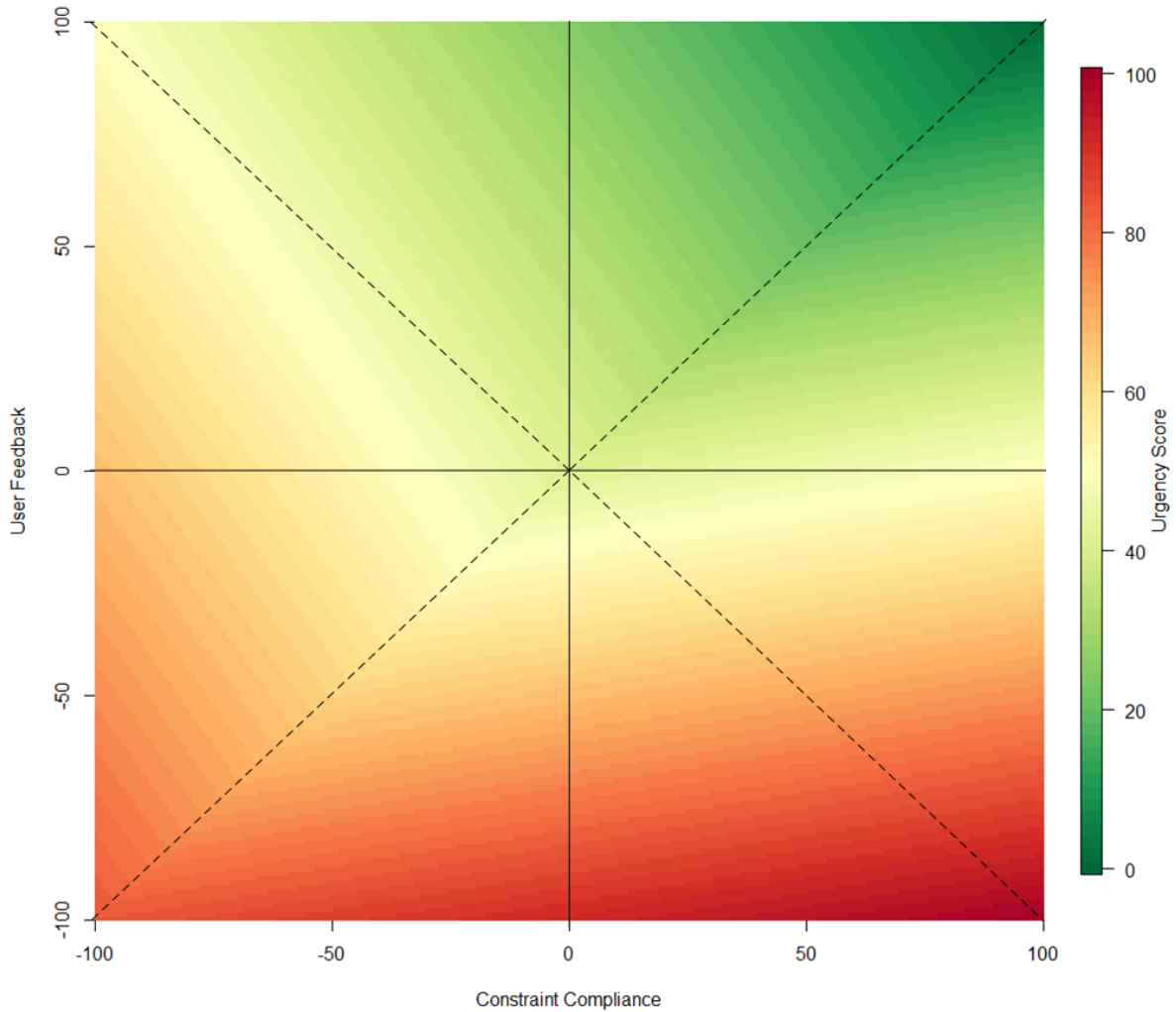


Figure 7.10: Urgency score map for $w_i = 0$, $w_c = 1$, $w_p = 2$, $w_f = 4$

To exemplify the influence of the three measure parts u_f , u_c , and u_p , we describe below the appropriate values in the four corner points of Fig. 7.10:

Upper right corner: $c=100$, $f=100$, $u_f=0$, $u_c=0$, $u_p=0$

Upper left corner: $c=-100$, $f=100$, $u_f=0$, $u_c=\max$, $u_p=\max$

Lower left corner: $c=-100$, $f=-100$, $u_f=\max$, $u_c=\max$, $u_p=0$

Lower right corner: $c=100$, $f=-100$, $u_f=\max$, $u_c=0$, $u_p=\max$

Fig. 7.10 is intentionally plotted with only 64 colours to visualise the different gradients and to be able to compare data points which are located in different quadrants but have the same urgency score. The colour representation of the urgency score depicts data points with low urgency in dark green, rather low urgency in light green, fair urgency in yellow, rather high urgency in orange, and high urgency in red. The general priority precedence of the different quadrants is still adhered, as explained with Fig. 7.6 in Sect. 7.4.2. Further, the decreasing and increasing characteristics for the urgency within the quadrants and the quadrant transitions is achieved, as described earlier with Fig. 7.7.

By combining the views of the DQ improvement urgency score with the DQ improvement tendencies, we are able to present a consolidated view on improvement urgency,

improvement priorities, root causes of issues and recommended DQ improvement. This is depicted in Fig. 7.11 where the four improvement tendencies are shown on the horizontal axis and the urgency of improvement of an instance inside an improvement tendency is represented on the vertical axis. By trend, the tendencies on the right side have a lower improvement urgency than the ones on the left side, but this is not the case for all data points within the tendencies, as explained above with the DQ improvement urgency score.

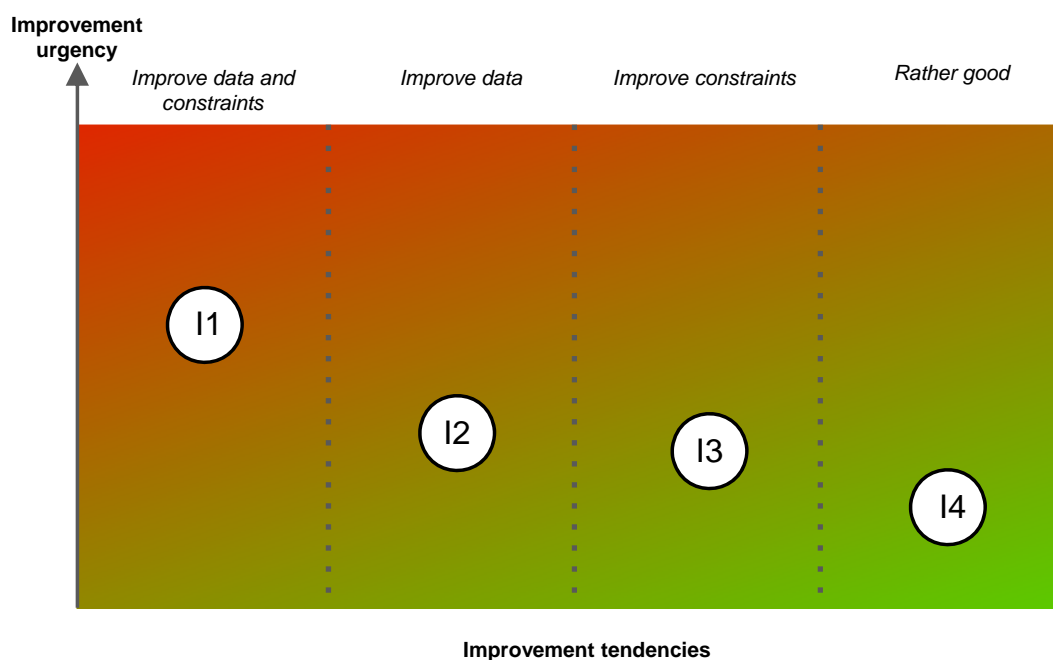


Figure 7.11: Data quality improvement urgency score

This visualisation enables us to provide DQ feedback to a DQ analyst, where the urgency of improving instances, such as I1, I2, I3, and I4 in Fig. 7.11, is presented. The more an instance is located towards the top, the higher is the urgency/need for its improvement. Moreover, it can be concluded from the tendency where the instance is located, what causes the low DQ and what countermeasures could be taken. The background gradient colouring assists the perception from higher urgency with red to lower urgency with green.

The measures presented above support the analysis of DQ of an information system at a particular point in time. Different types of changes in an information system can cause the quality of data to change and it is important to track their impact on DQ during the evolution of DQ over time, as described earlier. For example, by identifying certain events in the time line that introduced changes to the system, more indications on what has caused a lower DQ can be provided. Moreover, the DQ trend for certain instances, entities and the whole information system can be tracked and visualised to provide alerts that report on a sudden drop in the quality of the data.

With our DQMF, we also support the tracking of changes to the DQ over time. Our business logic library is continuously generating metadata objects corresponding to changes in the data and user feedback which have a timestamp assigned. With our DQ management application, it is possible to request the re-computation of the constraint

compliance in the business logic library. Thus, it is possible to trigger the validation at various points in time and receive the appropriate generated new metadata, for example, when a constraint repository was updated.

Based on the metadata with various timestamps, we provide an overview of the DQ evolution, as presented in Fig. 7.12. This builds the basis for further means of identifying issues, as described above.

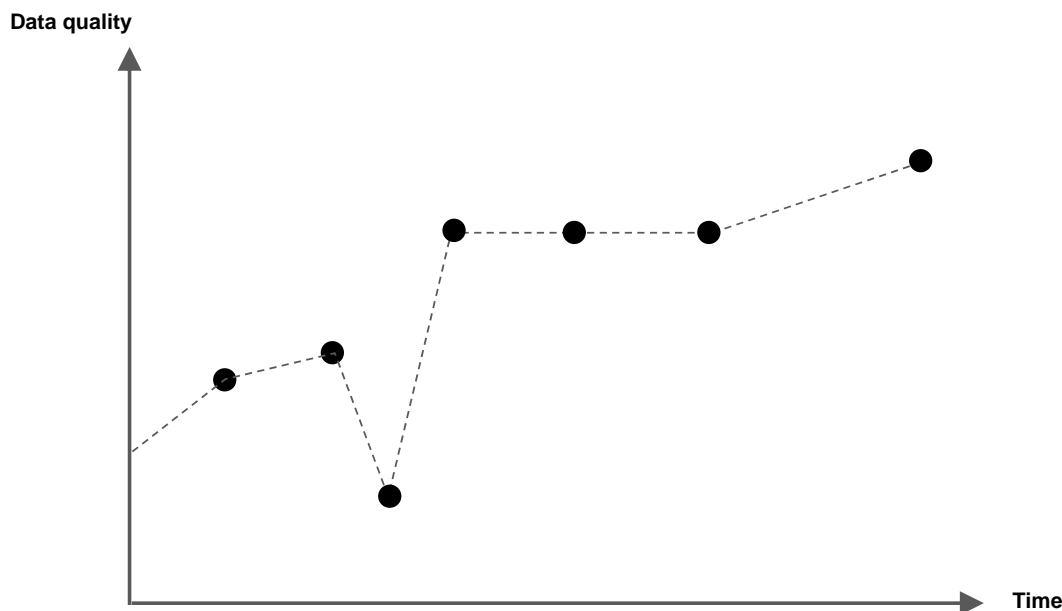


Figure 7.12: Evolution of the data quality of an instance over time

7.5 Implementation

We now present certain details of our *DQAnalytics* DQMF and our monitored information system *FoodCASEWeb* which together build our case scenario and implement the concepts of our DQMF, as introduced above. The scenario architecture was already presented with Fig. 7.1. The implementation of our DQMF includes the *DQAnalytics application* (DQ management application), the *DQAnalytics Library* (business logic library), and the *DQAnalytics Web module* (user input module). In the remainder of this section, we refer to the *DQAnalytics Library* as *library* and to the *DQAnalytics Web module* as *web module*.

Both GUIs of the *DQAnalytics* application and *FoodCASEWeb* are implemented as web interfaces, using AngularJS/JavaScript and HTML5. The *DQAnalytics* application is connected to the library via a REST interface. Its business logic is implemented with the Express¹ web framework and Node.js². The *FoodCASEWeb* business logic is implemented in Java. Since the *FoodCASE* desktop application is based on a PostgreSQL database, the *FoodCASEWeb* data is also received from a PostgreSQL database. The *DQAnalytics* application uses MongoDB for the persistence, since its data model is suitable for storing the DQ metadata in JSON format. All communication between the web module, the library and the *DQAnalytics* application is done over HTTP.

¹<https://expressjs.com/>

²<https://nodejs.org/en/>

7.5.1 DQAnalytics Library

FoodCASEWeb provides user authentication which assigns an authority, specifying a user group and a reputation, to each user. According to the authority, the appropriate group-specific constraint repository is used by the library for the data validation. To rate the relevancy of the explicit user feedback, the reputation is taken into account.

The library provides the constraint checking functionality in the logic tier for data which is received from FoodCASEWeb. It uses AspectJ³ for AOP to inject the validation code based on the constraint repository, at the appropriate places in the business logic of FoodCASEWeb.

Performance analysis

The library not only provides the validation injection into the monitored information system, FoodCASEWeb, but also the generation of DQ metadata for the appropriate validation results and received explicit/implicit user feedback, which is then analysed in our separated DQAnalytics application. This is a convenient solution for data validation and DQ analysis for the monitored information system by a separated DQMF. Nonetheless, the question arises how the required additional computation time influences the monitored information system.

The execution time of the validation and metadata generation depends on two parameters: (1) The number of field constraints and invariants that have to be validated; (2) The number of constraint repositories for which the metadata has to be generated.

Therefore, we have conducted a preliminary benchmark for creating a new instance in FoodCASEWeb with three scenarios: (I) No AOP injection with consequently neither constraint checking nor metadata generation; (II) AOP injection, with a constraint repository, containing five field constraints (two of type size, three of type required) defined on three fields, used for only constraint checking; (III) The same setup as in the second scenario, but with additional metadata generation.

Inserting an instance in scenario (I) took around 15ms, while the execution time increased by 60% for scenario (II) and by 130% for scenario (III), compared to scenario (I). Thus, there is a clear indication on the influence to the monitored information system, but further extensive benchmarks would have to be conducted to present a detailed analysis.

In the current implementation, constraint repositories are loaded and parsed for each constraint that is validated and for each metadata object that is generated. Further, if a constraint is inherited from another repository, both the original constraint and inherited constraint are checked. Thus, there is potential for improving the execution time in the library by caching constraint repositories and enhancing the checking of inherited constraints.

7.5.2 DQAnalytics Web module

For the validation in the FoodCASEWeb GUI, our web module uses a modified version of the *Valdr AngularJS module*. The original Valdr module provides data input validation of HTML input elements based on constraints defined in a JSON file. We adapted the

³<https://eclipse.org/aspectj/>

Valdr module to support the format of our constraint repository files, introduced in Sect. 7.2. Further, we extended the constraint checking logic of Valdr to introduce the distinction between errors and warnings, in case of a constraint violation.

Data quality input feedback

Based on the validation in our web module, we provide DQIF as proposed in previous approaches. An example of a data input detail dialogue for a food instance with constraint violation is presented in Fig. 7.13. A data input detail dialogue is used in FoodCASEWeb to view/modify detailed attributes of an instance or to create/delete an instance. According to the constraint example, presented in List. 7.1, a **required** constraint is defined on the food name attribute with a severity above the error threshold. Consequently, the appropriate error message ‘A food must have a name’ is shown below the ‘Food name’ input field and the ‘Save’ button is disabled.

The screenshot shows a web dialog titled "Edit Tdsfood (3)". It contains several input fields: "ID" (3), "Food code" (F201100), "Food name" (empty), "English name" (Apricot), and "Scientific name" (empty). Below the "Food name" field, there is a red error message: "Error A food must have a name". The "Study Name" is set to "BFR pilot 2014" and the "Sample Name" is "Aprikose". A "Data Quality" section at the bottom shows a "Data Quality Rating (Constraint Compliance)" of 47%, represented by an orange progress bar. At the bottom right, there are three buttons: "Delete" (red), "Cancel" (grey), and "Save" (blue, disabled).

Figure 7.13: FoodCASEWeb GUI – Detail dialogue for a food instance with constraint violation

Further, we provide a DQ indicator bar in a DQ panel below the input fields, which presents a DQ rating in the range of $[0, 100]$ based on the constraint compliance. The rating is adapted in real-time according to the user input.

Collecting explicit and implicit user feedback

To collect explicit user feedback, we designed a feedback form, which is attached to the data input detail dialogue. The form is opened on the right hand side of the dialogue, upon request. Both the dialogue and the feedback form are depicted in Fig. 7.14.

The image shows two overlapping windows from the FoodCASEWeb GUI. The left window, titled 'Edit Tdsfood (6)', displays a form for editing a food item. It includes fields for ID (6), Food code (F503100), Food name (Banana raw), English name (Banana), Scientific name (Musa acuminata), Study Name, Food Info, and Sample Name (Banane). A 'Data Quality' section shows a 'Data Quality Rating (Constraint Compliance)' of 100% with a green progress bar. At the bottom are 'Delete', 'Cancel', and 'Save' buttons. The right window, titled 'Data Quality Feedback instance (6)', contains three sections: 1. 'Provide feedback to instance fields' with a blue highlighted area for the 'code' field and radio buttons for 'has wrong/incomplete data' and 'has wrong constraints', followed by a text input and 'add'/'Cancel' buttons; 2. 'Rate the overall quality of this instance' with a five-point Likert scale from 'Very Bad' to 'Very Good', where 'Neutral' is selected; 3. 'Provide general feedback to this instance' with a text input area. A large green 'Submit Explicit Feedback' button is at the bottom.

Figure 7.14: FoodCASEWeb GUI – Explicit feedback form attached to the food detail dialogue

The feedback form contains three sections: (1) A structured field-specific feedback; (2) A DQ rating for the instance; (3) General feedback to an instance as free text. By selecting the blue area, covering a specific form field, issues with the appropriate field value in the *first section* can be reported. A user on the one hand can choose, if a field ‘has wrong/incomplete data’ and/or ‘has wrong constraints’, and on the other hand can use free text to describe the issues in more detail and/or to provide correct data. In the *second section*, a general DQ rating for the instance can be selected on a five point Likert-type scale, according to the description in Sect. 7.4.1. This rating is used to calculate the explicit feedback dimension in our DQ vector space model. The text area in the *third section*, can be used to give any further comments that provide general feedback to an instance, for example, if an additional instance attribute would be required.

Once the ‘Submit Explicit Feedback’ button is clicked, the feedback is submitted in a JSON object to the library. The feedback form, as presented in Fig. 7.14, is an enhanced version of our initial feedback form, which was less intuitive concerning the field-specific feedback.

Whenever a data input detail dialogue is opened, the web module sends an appropriate notification through the library to the DQAnalytics application, where an instance-specific counter is incremented. This implicit feedback is then used to calculate the interest of users for the various instances.

7.5.3 DQAnalytics application

The DQAnalytics application fulfils four main purposes in our DQMF: (1) Persisting the DQ metadata, received from the library; (2) Computing the DQ information based on the metadata; (3) Providing tools to visualise and analyse DQ; (4) Providing a GUI to manage the constraint repositories. In this subsection, we are predominantly presenting the tools to visualise and analyse DQ.

The web interface of our DQAnalytics application provides three main modules for analysing DQ: (1) The *scoreboard* gives a brief overview of the DQ in the system; (2) Different views for the *DQ enhancement analysis* provide means to identify DQ issues and to recommend countermeasures; (3) The *DQ evolution analysis* can be used to monitor DQ over time. From the main DQ enhancement analysis views, a user can deepen the analysis by using multiple additional subviews.

The **scoreboard** is the entry point of the DQAnalytics application. It provides an overview of the DQ based on the constraint compliance, for each entity in the monitored system in a small chart. An example of two charts for a TDS food and subsample are presented in Fig. 7.15. In the centre of each chart, the constraint compliance for the appropriate entity is presented for the three DQ calculation snapshots ‘1 week’ ago, ‘three days’ ago, and ‘today’. These charts enable the user to see how the DQ has evolved in the past week at a glance. Additionally, a trend indicator in the upper right corner of each chart, visualises with upwards, sideways or downwards pointing arrows whether the constraint compliance has increased, remained constant or has decreased in the past three days. Below the charts (not shown in the figure), a system log is presented which keeps a user informed about recent events in the system, such as an update of the constraint repository that triggered the recalculation of the constraint compliance.

Scoreboard



Figure 7.15: DQAnalytics scoreboard – Evolution of the constraint compliance on an entity level

The **DQ enhancement analysis** provides main DQ visualisations for the DQ improvement tendencies and the vector space model. From these visualisations, it is

possible to navigate to further visualisations, so-called subviews, which provide further details on different aspects of the DQ, for example, the DQ improvement urgency analysis.

In all visualisations, we distinguish between an analysis on an instance or on an entity level. On the instance level, all single instances are presented individually, while on the entity level all appropriate instance values are aggregated to an entity value. Aggregation of single instance values is always calculated with the arithmetic average. Further, we provide faceted filtering, where it can be chosen for which groups (constraint repositories) and entities the analysis should be refined.

Data quality improvement tendencies chart

An example visualisation of the DQ improvement tendencies is presented in Fig. 7.16, where the instances/entities are represented with blue circles. Instances/entities are assigned and located in the appropriate quadrants according to their constraint compliance and explicit feedback values, as described in Sect. 7.4.2. Additionally to the two dimensions constraint compliance and explicit feedback, we visualise in this chart the importance value of instances/entities which is reflected by the radius of the circles. The larger the radius, the more important is the appropriate instance/entity. For calculating the importance, we use Eq. 7.3. By hovering with the mouse over a circle, the appropriate DQ indication values are shown for the single dimensions constraint compliance, explicit feedback and importance.

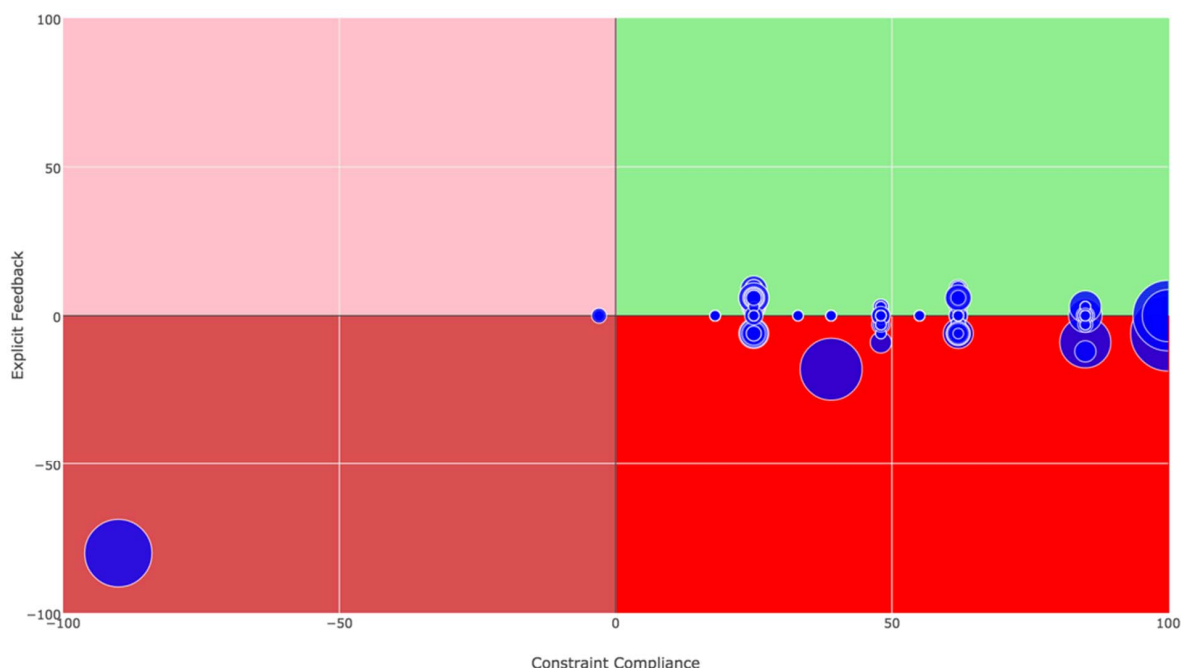


Figure 7.16: DQAnalytics – Improvement tendency chart

We also provide a vector space model chart based on the same data used for the DQ improvement tendencies chart. There, the data is visualised in a three dimensional chart, that can be moved and turned in all directions, where also zooming and panning is possible. The user study, presented later, showed that the DQ improvement tendencies chart is more intuitive for analysing DQ regarding the three dimensions, than the vector

space model chart. Note that we also provided an example visualisation with Fig. E.1 in Appendix E.

Subviews

To have tools at hand for providing more detailed DQ information on instances/entities that are presented in the main DQ analysis views, we implemented four additional views, which are loaded upon clicking on the instances/entities circles. Only upon opening a subview, is the required metadata loaded and the appropriate calculations performed.

(1) The DQ improvement urgency analysis provides the visualisation for the DQ improvement urgency score, as described in Sect. 7.4.3. This is a new visualisation concept, which we explain with an example below.

(2) The DQ dimensions analysis provides visualisations to analyse data concerning the DQ dimensions, assigned to the constraint definitions, such as accuracy and completeness. For example, a bar chart shows the constraint compliance for instances/entities associated to and grouped by DQ dimensions. The visualisation concept is adopted from our previous approaches. Note that we also provided an example visualisation with Fig. E.2 in Appendix E.

(3) The constraint compliance analysis provides a list overview of the single constraints, where it shows the fulfilment rate of the constraints for all instances, grouped by constraint repositories. The fulfilment rate is visualised with bars that are coloured in red or green, depending on the fulfilment rate. The visualisation concept is adopted from our previous approaches. Note that we also provided an example visualisation with Fig. E.3 in Appendix E.

(4) The explicit feedback analysis provides a list overview of the explicit user feedback which is a new visualisation concept. Note that we also provided an example visualisation with Fig. E.4 in Appendix E. All the details of the single explicit user feedback entries can be viewed and entries can be marked as *resolved* on a field and instance level. Before marking an entry as resolved, the appropriate issue should be fixed. A resolved explicit user feedback, which was probably a negative feedback, is then excluded from further DQ calculations.

An example chart for the **DQ improvement urgency analysis** is depicted in Fig. 7.17. The visualisation is based on the DQ improvement urgency score. The example shows different instances of the TDS food located on the chart according to their DQ improvement urgency score. On the horizontal axis, the DQ improvement tendencies are distinguished, while the vertical axis represents the DQ improvement urgency score. The numbers, depicted in the blue circles, represent the number of instances which are located at the appropriate urgency score. The size of the circles also depends on this number. The higher the number of instances, the larger the circle. For example, the top left circle gives an indication on the twelve instances with the highest DQ improvement urgency score.

Note that the range of the urgency score is only within $[0, 60]$ instead of $[0, 100]$, since there exist no instances in the example with a higher score than 60. A user can further click on a circle to open a dialogue which presents the identifiers of all instances, associated to the appropriate circle. Based on this information, a user can enhance the DQ of the individual instances in the monitored information system.

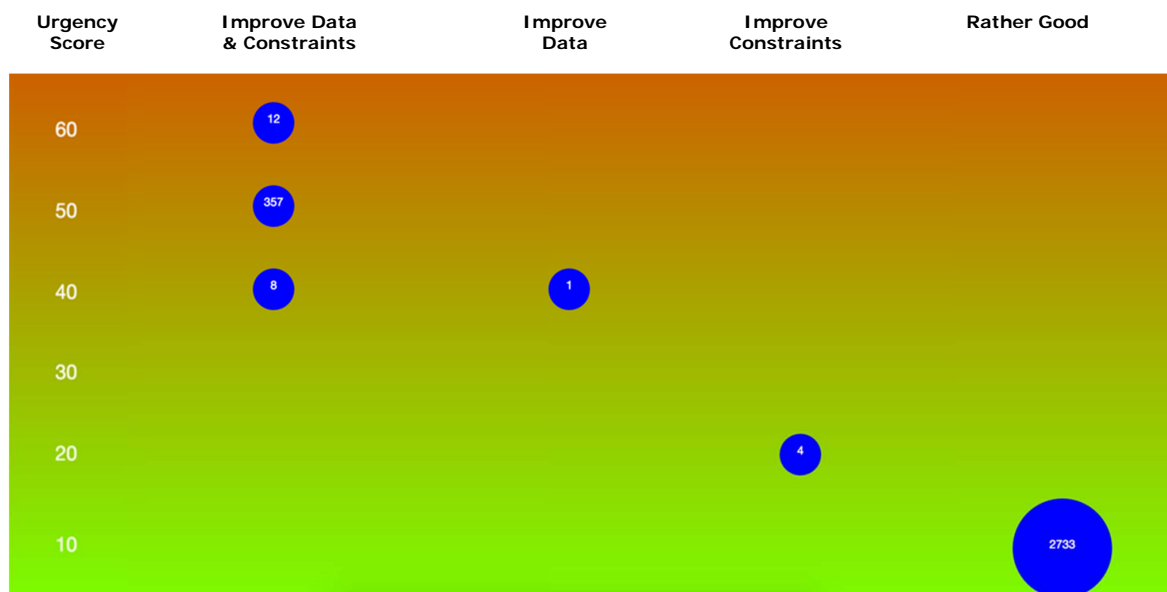


Figure 7.17: DQAnalytics – Urgency score chart

Further details of the implementation of the DQAnalytics DQMF are provided in [14] on pages 45 to 99.

7.6 Evaluation

To evaluate the usefulness and usability of the DQAnalytics DQMF, we conducted a user study, where the participants had to solve three tasks using the DQAnalytics DQMF functionality in FoodCASEWeb and the DQAnalytics application. The participants provided their feedback in a questionnaire, where we asked task-specific questions and general questions on the usefulness and usability of the DQAnalytics DQMF, but also general questions about issues in the management of distributed constraints and DQ monitoring.

Besides getting feedback on the usefulness and usability of the DQAnalytics DQMF and on how it could be improved, the goal of the study was also to obtain insights from computer scientists and software engineers on questions, such as: (1) Do they perceive issues in the management of distributed constraints in practice, and if so, what are the issues? (2) Is it desired to provide unified constraint definitions in an information system? (3) Is it desired to provide means of measuring and tracking DQ, i.e. DQ monitoring, in an information system? (4) What kind of constraint definitions, in terms of paradigms and technologies, are most often used? (5) Are the participants willing, as users of an information system, to provide explicit feedback on DQ? The full list of questions contained in our questionnaire is presented in Appendix F.

In total, 13 people with experience in general computer science or software engineering participated in the study, of which 12 were male. The ages were between 26 and 41 years, with an average age of 30.5. All participants hold a degree in computer science, 69.2% a master, 23.1% a PhD and 7.7% a bachelor. At the time of conducting the study, all but two of the participants were working at least part-time as software engineers, 61.5% full-time and 23.1% part-time.

Participants had to rate their skills with respect to five computer science topics, on a numeric 7-point Likert-type scale from 1 to 7, where 1 means no experience and 7 means expert. We considered all values from and above 4 as at least good knowledge in the appropriate topic. Consequently, we present the percentage of participants, which had at least good knowledge in the following topics: General software engineering (100%), Web engineering (75%), JSON (70%), Data quality (62%), Constraint modelling (62%).

We will now briefly describe the three tasks that the participants had to solve using FoodCASEWeb and the DQAnalytics application, before presenting the results for the task-specific and general questions. The complete task description sheet is available in Appendix F.

In task 1, the participants were asked to randomly choose five food instances in a list provided by the FoodCASEWeb GUI, and inspect the data for each of the instances by opening the detail dialogue, as presented in Fig. 7.13. They then had to provide their explicit feedback on the DQ for the instances using the feedback form presented in Fig. 7.14. With this task, we aimed to evaluate the usability of the explicit feedback form, and to emphasise the effort for providing explicit feedback.

In task 2, the participants had to modify a constraint repository file, where they had to add a `required` and `size` constraint to the `scientificname` attribute of a TDS food entity. With this task, we aimed to evaluate, to what degree the participants understood the structure of the constraint repositories, and how difficult they perceive the modification of the repositories.

In task 3, the participants had to analyse the DQ of the monitored information system, using various tools of the DQAnalytics application. They had to answer specific analysis questions, such as: ‘When you look at the improvement tendency chart, which of the entities that are available for analysis would you improve first?’ The full list of specific analysis questions is provided in Appendix F. With this task, we aimed to evaluate the usability of the tools provided in the DQAnalytics application.

After finishing each task, the participants answered the appropriate task-specific questions. Most of the answers to task-specific and general questions were based on a 5-point Likert-type scale (‘totally agree’, ‘agree’, ‘neutral’, ‘disagree’ and ‘totally disagree’). For the evaluation here, we consider the first two values as agree and last two values as disagree.

For the sake of brevity, we will present only a summary of the most important findings in the remainder of this section. A detailed description of the study setup and the results are provided in [14] on pages 102 to 112.

The participants had to state, if they had experience with distributed constraint management, where constraints are used in different representations. 76.9% of the participants had such experience, while the remaining 23.1% stated that they either need or use no constraints at all in their projects, or that they do not need distributed constraints specifically. The 10 participants with experience, had to answer further questions about distributed constraint management, while the 3 participants without experience skipped them. There is a clear indication, that input validation is important to ensure DQ (100% agree) and that maintaining multiple constraint repositories is time consuming (100% agree), hard (90% agree), and a frequent source of failures (80% agree). Also 60% agree that there is lack of supporting tools and paradigms to develop and maintain multiple constraint representations in distributed systems, where the remaining 40% are neutral. Almost all participants (90%), define constraints

in various components of the application, such as GUI, business logic, and database. Complex constraints are rather defined in the business logic than in the database (70% agree), while 80% of the participants usually define no or only a few constraints in the database.

The remaining task-specific and general questions were again answered by all participants.

For task 1, we can conclude that most of the participants were satisfied with the features provided by the feedback form. It is easy to use (100% agree), and enabled the participants to express their feedback (85% agree, 15% neutral). The 15% neutral answers probably relate to the issue that, during the study, it was only possible to give free text comments to the overall instance. We added the specific text field comment and other suggested enhancements to the final version of our feedback form which was presented in the previous section.

For task 2, we can conclude, that our JSON format for the constraint repositories is easy to learn (92.5% agree, 7.5% neutral). Further, the format enables an intuitive definition of constraints since 100% of the participants agreed that the task was easy to solve, and that the obtained result is what they expected. Nonetheless, we received multiple suggestions for enhancement, such as a more advanced constraint repository management application, where constraints can be added and moved in a drag-and-drop manner, or an additional syntax-check for the format.

For task 3, we can conclude, that our tools support the analysis of DQ since 100% of the participants agreed that the task was easy to solve. Further, the usage of our improvement tendency chart is intuitive (100% agree), but only 54% of the participants agree that the vector space model chart is intuitive to use (31% neutral, 15% disagree). We also received feedback that it takes some time to understand the filters, different colours and symbols that represent the instances/entities. This could probably be improved by adding additional descriptions to the specific features. Further, it was suggested that instances and entities should be displayed at the same time in the charts, which could enhance the interpretation of the distributed instances, and provide better insights on how the entity value is calculated.

The next results correspond to questions related to the usability of DQAnalytics. The overall feedback for the usability of DQAnalytics was very good. 85% of the participants rated it as easy to use (15% neutral), and also 85% felt confident using the system (7.5% neutral). 77% of the participants think that they would not have to learn a lot before being able to use the system (15% neutral). Further, 85% of the participants agreed that the functionality in the system is well integrated (15% neutral). Thus, the participants were satisfied with DQAnalytics but the understandability of its capabilities can still be improved. We generally assume that such an application, to analyse and monitor DQ would be used rather frequently by users who would therefore be willing to invest time to familiarise themselves with the application. Therefore, good documentation and/or short tutorial would be helpful. We also received feedback from two participants that it would be convenient to be able to correct data directly after clicking on an instance in the charts, as we provide it in previously presented integrated approaches.

The remaining results correspond to general questions. Almost all of the participants agreed that there is a need for tools that provide information about the DQ in an information system (92.5% agree, 7.5 neutral), and that DQAnalytics provides a good

understanding of the quality of data of the monitored information system (92.5% agree, 7.5 neutral). Further, almost all of the participants (92.5% agree, 7.5 neutral) stated that they would invest time to provide explicit feedback to an information system in order to help increase the DQ. All participants agreed that explicit user feedback can be used to better understand the DQ in an information system. The participants would use DQAnalytics in their software products due to the provided unified constraint management (92.5% agree, 7.5 neutral), and due to the provided DQ analysis tools (100% agree).

Overall, the results are very promising for our approach, especially the fact that our integration of explicit user feedback was rated as an important indicator for DQ, and that our approach seems to provide the appropriate convenient tools.

7.7 Conclusions

We have presented a DQMF which provides components that can be used by an information system to inject data validation at different tiers, based on central constraint repositories and AOP. The constraint repositories are convenient to maintain, and are used for all validations, which guarantees unified constraint management. Further, the DQMF provides a separated application with tools to analyse and monitor the DQ of an information system. It provides means of receiving DQ metadata, such as validation results and explicit/implicit DQ feedback from the users of an information system, and concepts to analyse the DQ in an information system as a combination of objective and subjective DQ assessment.

Therefore, we have introduced a DQ vector space model which expresses the DQ of data instances with three different indications on DQ, namely compliance, explicit user feedback and importance. Based on this vector space model, it is possible to associate data instances to different DQ improvement tendencies and calculate a DQ improvement urgency score, which both take the objective and subjective DQ assessment into account. We have shown how to use these measures to evaluate causes of low DQ and to determine actions to improve the DQ.

Further, we showed that user feedback, as subjective DQ assessment, is very useful, not only to indicate DQ and recognise semantic errors in the data, but also to identify mistakes in the constraint definitions, when it is combined with objective DQ assessment.

To evaluate our approach, we implemented a case scenario, that consists of a monitored information system, FoodCASEWeb, a DQ management application, DQAnalytics application, and the two components DQAnalytics Web module and DQAnalytics Library. The DQAnalytics application, the web module and the library represent the implementation of our DQMF DQAnalytics. Based on this implementation, we conducted a user study which indicated very good usability and usefulness of DQAnalytics and its provided tools. An important result is that the user study participants would invest time and effort to provide explicit feedback to an information systems they work with, to help improve its DQ.

The injected functionality from our DQMF into a monitored information system is a great advantage compared to previously presented integrated approaches. Nonetheless, it also comes with an increase of runtime in the monitored information system. We

conducted a brief benchmark to provide an indication on the runtime overhead but further benchmarks would be needed to give a clear indication. The approach should also be compared with other integrated validation mechanisms, such as BV, since they also cause an increase of the runtime.

There still exist many ways of optimising the runtime of our approach. For example, the constraint repositories could be cached to avoid loading and parsing runtime for the checking of constraints and the generation of DQ metadata.

On purpose, our approach does not propose the transferring of data instances between the monitored information system and the DQ management application. Thus, the data instances with low DQ always have to be corrected in the monitored information system, although we provide the necessary tools to identify them in the DQ management application. Directly adapting the data instances in the DQ management application could be achieved by adding appropriate functionality to the monitored information system, or by a concrete data transfer of the instances between the monitored information system and the DQ management application. Nonetheless, this is a trade-off between integrated functionality and independence of a DQMF.

8

Conclusion & outlook

In this thesis, we have presented alternative approaches for managing constraints in a unified way in information systems to reduce the constraint maintenance effort and the likelihood of incorrect constraint definitions as well as constraint inconsistencies. Moreover, we have introduced concepts which enable individual data validation for different user groups. Further, we have shown, how constraints can be associated with additional attributes, such as constraint violation severities and DQ dimensions, and presented alternative measures and concepts to indicate DQ based on constraint compliance. Additionally, we have presented measures and concepts that combine such objective DQ assessment with subjective DQ assessment, such as explicit and implicit user feedback, to provide an extended analysis of DQ. As part of our approaches, we have presented diverse tools to analyse DQ, and diverse ways of giving feedback to users of an information system, both during data input and data analysis.

Our research was motivated from the experience of the author in large, partly agile, projects in both the financial sector and in the food science domain with the extension of the FoodCASE system described in this thesis.

To investigate our concepts and approaches, we mainly focussed on contexts involving object-oriented programming languages.

From our analysis of FoodCASE, we identified several requirements, issues and solutions which we implemented and evaluated with our alternative approaches. At the beginning, we had the vision that there could be one final approach to meet all requirements. During the work on our initial approaches, we recognised that there is no ‘one size fits all’ solution, due to the different scenarios of software development and the diverse requirements for different information systems. Consequently, we have shown with our alternative approaches how to cope with different scenarios of software development concerning unified constraint management and DQ assessment.

In the remainder of this chapter, we will first briefly outline the main contributions and present an overview and brief assessment of our alternative final approaches. We will then conclude with a discussion on the criteria for deciding on an approach, before giving an outlook on future work.

With the explanation of our alternative approaches and findings, we aimed to provide guidance for software architects and developers who have to decide on a suitable solution for constraint-based data validation, potentially in relation to DQ assessment, for their individual information system scenario. There are four main contributions contained within these approaches by providing concepts concerning (1) unified constraint management for software developers, (2) expressing DQ by constraints and beyond constraint compliance, (3) extended constraint checking for different users/applications, and (4) giving feedback on DQ during data input and data analysis for different end-users of an information system.

With our approaches, we satisfy **unified constraint management** in different ways for different requirements and scenarios. This includes both existing systems, which should be improved with unified constraint management, and systems that are planned from scratch. We further bridge the gap between formal methods in software engineering and the practice of software development with an MDA approach.

How to analyse and improve the DQ of an information system has been investigated from many different viewpoints by many different researchers. We provide one of the few works focusing on **expressing DQ by means of constraints**. Our approaches show how to support giving indications on DQ with DQMFs which are part of an information system, or which provide analysis tools in a monitoring application, external to the information system. With our concept of indicating DQ based on constraint compliance, we provide objective DQ assessment in all approaches. Moreover, we have shown how subjective DQ information can be collected in an information system, and used in combination with objective DQ assessment to provide new measures and indications on DQ.

Based on the requirements obtained in the TDS-Exposure project, we have presented concepts that enable data validation specific to user groups as well as unified constraint management. In combination with our concepts of expressing DQ by means of constraints, this **extended constraint checking** provides highly customisable data validation and indication on DQ during data input and data analysis. Further, we have pointed out the difficulties of defining constraints during the runtime of an application, and provided solutions which tackle these issues.

Our approaches also give guidance on how to give **feedback on DQ** by providing many tools for DQIF and DQAF. We have shown that the alternative approaches offer different possibilities for a user to analyse and improve DQ. For example, with DQMFs which are integrated into an information system, it is simpler to provide tools to drill down to single instances which have DQ issues. This becomes more complicated with DQMFs which are separated from an information system since these DQMFs, by themselves, might not have access to the data.

Further, we have presented different user studies for our alternative approaches, which not only aimed to evaluate the usability and usefulness of our implementations, but also to provide insights into the issues which developers and users of information systems have, concerning distributed constraint management, data validation, DQ analysis and feedback on DQ.

8.1 Final approaches summary

Similar to the overview of our initial approaches in Chap. 3, we give here an overview of all final approaches where we also point out the strengths and weaknesses of each approach. But first, we present the criteria used to categorise our alternative approaches.

We distinguish the two main criteria (1) *constraint representation uniformity* for the categorisation of information systems concerning their homogeneity of constraint representations, and (2) *level of dependence* for the categorisation of DQMFs according to their extent of integration with an information system.

With the constraint representation uniformity, we categorise into (1-A) *homogeneous* information systems, where uniform constraint definitions are used for data validation in all components, and into (1-B) *heterogeneous* information systems, where diverse constraint representations are used for data validation in the different components.

With the level of dependence, we categorise into (2-A) *integrated* DQMFs, where the DQMF is implemented as part of the information system, into (2-B) *separated* DQMFs, where the DQMF is separately implemented from the information system, and into (2-C) *hybrid* DQMFs which combine criteria from integrated and separated DQMFs.

The resulting categorisation for our final approaches is presented in Tab. 8.1, where the row labels represent the criteria for the *constraint representation uniformity*, and the column labels represent the criteria for the *level of dependence*.

Note that this categorisation only considers different solutions for fulfilling the requirements of unified constraint management and not the different concepts for accomplishing DQ assessment.

	integrated (2-A)	separated (2-B)	hybrid (2-C)
homogeneous (1-A)	Bean Validation approach (Chap. 4)		
heterogeneous (1-B)	Constraint injection approach (Chap. 6)	<i>UnifiedOCL</i> approach (Chap. 5)	<i>DQAnalytics</i> approach (Chap. 7)

Table 8.1: Approaches categorisation

For all approaches, we summarise now their main features and provide a brief assessment for each approach. In Tab. 8.2 all final approaches are listed with their approach name (Approach), a brief description of the main topic (Topic), their main features (Features), their categorisation for the level of dependence (Dep.) where ‘Int.’ indicates an *integrated* approach, ‘Sep.’ a *separated* approach, and ‘Hyb.’ a *hybrid* approach, and a reference to the chapter (Chap.) where the approach is described in detail. Please note that for each approach, an individual, detailed discussion is provided at the end of the corresponding chapter.

In Chap. 4, we presented an approach for integrated DQMFs using BV. It was based on our experiences gained in the initial approaches Guidance on data quality feedback (Sect. 3.1), Data quality extended Bean Validation (Sect. 3.3) and Bean Validation OCL extension (Sect. 3.4). The approach was implemented as the second version of

Approach	Topic	Features	Dep.	Chap.
<i>Unified data quality assurance and analysis based on Bean Validation</i>	Integrated and extended constraint management based on BV	fine-grained severity; new measures for general DQIs; DQ score; customised validation with contracts, error threshold and validation groups; new DQIF and new DQAF; second version of the FoodCASE DQMF for TDS	Int.	4
<i>System-wide constraint representations with UnifiedOCL</i>	Achieving unified constraint representations with a new DSL (UnifiedOCL) and constraint translations	MDA support; fine-grained severity; new measures; powerful unified model and constraint definition based on OCL; pluggable label dictionaries for convenient constraint definition; mapping from constraints to DQ dimensions; bi-directional constraint translations and multi-translations	Sep.	5
<i>Data quality management in mobile applications</i>	Constraint synchronisation between different applications	mobile focus; constraint injection/synchronisation; adaptive GUI; rule engine based; FoodCASE extension	Int.	6
<i>Hybrid approach with subjective data quality assessment</i>	Combination of integrated validation and separated monitoring including objective and subjective DQ assessment based on AOP	consolidation of approaches; validation library and web module; validation logic translation; fine-grained severity; DQ dimensions mapping; runtime constraint definition; AOP based; new measures; including subjective DQ assessment; new DQ monitoring; new DQIF and new DQAF	Hyb.	7

Table 8.2: Overview of the final approaches

a DQMF within FoodCASE for the TDS-Exposure project. It provides a fine-grained severity scale for constraint violations where a threshold can be defined that discriminates warnings from errors. Moreover, constraint definitions can be associated with customisable validation groups, which enable the mapping of constraint violations to any data validation categorisation. We extended our DQ measurement and calculation concepts and presented measures of general DQIs for different levels in the DQAF as well as DQ scores for entities in the DQIF. Based on these measures, we also provided new tools to support the analysis and enhancement of DQ in an information system. Further, this approach enables highly customisable constraint checking and data validation with validation profiles, so-called contracts, configurable for different user groups or applications. We also presented an evaluation of this approach conducted with various

food science experts participating in the TDS-Exposure project.

This approach is well suited to homogeneous systems where BV constraints conveniently can be defined in the logic tier. Further, BV also can be used in the presentation tier, where it can be integrated with appropriate technologies. Predefined constraint types cope with a wide range of constraints, but the static definition of parameters reduces their flexibility. Custom constraints, which enable complex business rules to be defined, partly compensate for this lack of flexibility. Defining custom constraints always adds more complexity but is still convenient here, since it is an integrated approach. Implementing our concepts is highly dependable on the given functionality of BV, which reduces the potential for custom extension.

In Chap. 5, we proposed an approach where constraints are managed in a single place using OCL with extensions for technology-specific concepts. Constraints are defined with our powerful DSL *UnifiedOCL* which includes pluggable label dictionaries for convenient constraint definition. These constraints are then mapped to technology-specific representations which are validated at runtime. Bi-directional translations of constraint definitions then allow existing components to be easily integrated into the system. With this approach, we support MDA and the need for intentionally distributed constraints across an information system. It further integrates the association of penalties for constraint violations on a fine-grained severity scale and the mapping of constraint definitions to DQ dimensions. As proof of concept, we implemented this approach as a set of *Eclipse* plugins and presented the results of a user study which support our claims that the problems addressed by our system are important and can reduce developer effort.

This approach is well suited to heterogeneous systems, where constraints have to be used in potentially many different technology-specific representations. Since the DQMF is completely separated from the information system, where the technology-specific constraint representations are used for data validation, it is applicable in many cases and provides high flexibility. Nonetheless, although this approach provides a translation from *UnifiedOCL* to pure OCL, it does not support RTE since it only handles a unidirectional translation from UML to *UnifiedOCL*.

In Chap. 6, we introduced an approach for interconnected applications which share constraint definitions and have to synchronise their separated constraint (sub)sets on a regular basis. We used the scenario of a main application as a *source system* from which constraint definitions are injected to a mobile application as a *target system*. The *constraint injection* includes the translation from the source to the target constraint representation as well as the transfer and update of the constraint set in the target system without influencing the runtime of both systems, besides the constraint sets and data validation. In this approach, the target constraint set is represented by business rules and validated with a rule engine. With the focus on mobile applications, this approach also proposed an adaptive GUI which automatically adjusts the input forms of the target system according to the received constraint set. In terms of DQIF for mobile applications, we also presented a responsive DQ feedback interface which informs users about the quality of data in different ways, dependent on the screen size of their device. As proof of concept, we extended our FoodCASE information system with a mobile application with appropriate constraint synchronisation functionality. To evaluate different ways of giving DQ feedback in mobile applications, we also conducted a user study.

This approach is well suited to heterogeneous systems where one component/application is dependent on another component/application. Only one constraint translation is needed which is integrated in the source system and is therefore highly customisable. The constraint checking, based on a rule engine, provides a high flexibility to define business rules. Nonetheless, with this approach it is only possible to inject constraints from a source system to a target system and not vice versa.

The approach proposed in Chap. 7, is a combination of an integrated validation framework and an independent monitoring framework for DQ including not only objective DQ assessment, represented as constraint compliance, but also subjective DQ assessment. This approach and its DQMF provide unified constraint management based on AOP and consolidates our integrated and separated approaches. The constraint checking mechanisms can be injected into different tiers of a software system by including our validation library and/or web module and applying appropriate validation translations. Constraint definitions can be created, adapted and deleted during runtime while also being associated with penalties on a fine-grained constraint violation scale and mapped to certain DQ dimensions. Besides including constraint compliance for our DQ measures, as already proposed in our other approaches, we also included explicit/implicit user feedback and the importance of individual domain entities as a subjective DQ assessment in terms of a DQ vector space model. Based on this model, we provided new measures, referred to as DQ improvement tendencies and a DQ improvement urgency score, which give further indications on enhancing DQ. Consequently, this approach provides additional features and visualisations for DQIF, DQAF and monitoring DQ. Our case scenario implementation of the resulting DQMF, called *DQAnalytics*, was realised as a web application in the domain of food science, complementing our FoodCASE information system. We also presented the results of a user study which we conducted to evaluate the concepts included in this approach and the usefulness/usability of *DQAnalytics*.

This approach is well suited to heterogeneous systems where aspects can be used for data validation in the logic tier and one constraint translation is sufficient for the data validation in the presentation tier. It provides high flexibility for all our concepts, also due to the custom constraint definition representation in JSON. The hybrid injection of validation and DQ assessment logic into a monitored information system through this DQMF is very convenient. Nonetheless, this is a trade-off between integrated functionality and separation of a DQMF which causes, for example, the problem of accessing concrete data during the DQ assessment.

8.2 Discussion

With the initial approaches, we investigated the different requirements and goals presented in Chap. 1 by exploring alternative solutions to fulfilling them. The specific aim of these approaches was to explore various ways of (1) defining and checking simple and complex constraints, (2) measures of constraint compliance, (3) relating constraint compliance to DQ, and (4) giving DQIF and DQAF. Having explored the different initial approaches, we recognised that instead of trying to achieve a single combined approach, it was better to offer different solutions for different scenarios. Thus, the initial approaches built the knowledge and decision basis for our final approaches, which

consolidate and extend concepts from the initial approaches to fulfil all of our proposed requirements of a DQMF for various initial software development scenarios.

In this thesis, we have described how our approaches can be realised and how our DQMFs allow the users to validate data during input, as well as analyse, quantify and visualise the specific and overall quality of the data at any time.

The initial software development scenario for an information system, in terms of whether parts of it already exist or it is to be developed from scratch, is crucial for the choice of the approach and its implementation. Therefore, additional to the architecture of an information system, the constraint requirements, and the targeted goals for constraint definition, constraint checking, DQ measuring, DQ feedback and DQ monitoring have all to be taken into account when deciding on appropriate concepts and approaches.

In the remainder of this section, we will provide a set of questions, which aim to help software architects/developers clarify the requirements of an information system concerning constraint checking and DQ assessment, and choose the appropriate combination of concepts presented in our approaches. We therefore hope that this thesis can contribute to constraint-based approaches to DQ in information systems, both for the research community and industry.

First, it is necessary to gather the **constraint requirements** or business rules, from the different stakeholders by getting answers to the two basic questions: (1) What kind of simple and complex constraints are needed? (2) How do the constraint violations differ in terms of severity? The answer to the first question can then be used to define accurate constraint checking for the domain entities and additional business rules. The answer to the second question determines whether a distinction between different constraint violation severities is needed. If it is not needed, all constraints have the same importance and can be traditionally validated to true/false. If different constraint violation severities are needed, the answer to the second question provides the information on the individual constraint importance, also taking into account the potential need of different user groups. To get the information for these two questions, it is possible to conduct an appropriate evaluation with the different stakeholders, such as domain experts and potential end users of the information system.

In the TDS-Exposure project, for example, we identified in a first step, a set of entities with their manifold attributes relevant to the TDS-Exposure domain which built the data model of our extended FoodCASE. Further, we defined multiple simpler consistency constraints and multiple complex rules as a proposal for the evaluation. In the evaluation with our project partners and domain experts, we received feedback on our already defined simple and complex constraints and gathered additional constraint and rule definitions which we incorporated into FoodCASE. Based on the feedback, it was possible to rate constraint violations with individual different severities by also distinguishing different users groups, such as different institutions from different countries.

In combination with the answers to the two basic constraint requirements questions from above, answers to further questions concerning **implementation requirements and technical details** provide the evaluation basis for deciding on appropriate concepts and approaches. Please note, that we provide here a list of possible questions, without the claim for completeness. Below, from top left down to bottom right, the questions are group by the topics, constraint definitions (1-6), system components (7-10), DQ – requirements (11-13), DQ – technical aspects (14-15), various (16-19).

1. Where have the constraints to be defined? Central possible? Many components?
2. What constraint definition is convenient/needed for developers? Close to the code, as part of the programming language, separated as configuration, etc.
3. How complex are the constraint definitions? Standard checking, customised validators, checking with metadata/statistics, depending on data source reputation, reusing of constraints, etc.
4. Who will define/maintain the constraint definitions?
5. Are customised user group-specific constraints needed?
6. Is it needed to change/apply constraint definitions during runtime?
7. How many and what kind of components are involved? Database, server, GUI, mobile (online/offline, responsive), multiple applications/databases etc.
8. How many and what kind of technologies are involved?
9. What architecture is possible/needed? From scratch? Legacy system? Federated solution or consolidation?
10. Is there a need for constraint synchronisation/injection?
11. What are the needs for DQIF?
12. What are the needs for DQAF and reporting?
13. Is an objective DQ assessment (constraint compliance) sufficient as an indicator for DQ or is additional subjective DQ assessment needed?
14. Is an information system integrated or separated DQMF necessary/desired?
15. Is it needed to change measurements for DQ during the information system's life cycle?
16. Is there a need for an adaptive GUI dependent on constraint definitions, and device screen sizes, and/or users?
17. Is there a need for data cleaning?
18. Is there a need for constraints interdependence/contradictions checking (validation of constraint sets themselves)?
19. Is data imported on a regular basis?

Answering these questions should provide insights on which concepts are needed to which level of detail. Since it should be obvious for the reader which questions target which decisions, we do not discuss the questions in further detail. We rather want to provide general guidance. For homogeneous systems, if possible, it is convenient to define constraints in a unified way, such as with BV. If necessary, legacy or heterogeneous constraint definitions can be transformed once into a unified representation, for example with or via *UnifiedOCL* and appropriate translations. If using a single constraint representation is not possible, since the system is heterogeneous, it is necessary to use a constraint transformation approach, such as *UnifiedOCL* with translations for multiple technology-specific representations, a single translation approach with constraint injection, or constraint repositories with AOP and constraint translation. Depending on the requirements for constraints, implementation, technical details and DQ assessment, appropriate concepts from our approaches can then be realised for a specific solution.

8.3 Outlook

With our approaches, we have tackled the issue of defining constraints during the runtime of an application, with immediate effect on the data validation. While this is accomplished for predefined constraint types, the definition of complex custom constraints still provides issues for unified constraint management, applied to different components in an information system. For custom constraints, it seems to be inevitable in certain cases to recompile at least parts of the application or validation code. A solution could be to provide hot deployment of validation code and/or specific custom class loaders, which we did not address in this thesis.

We provide the concepts for associating DQ dimensions with constraint checking to enable DQ analysis also based on DQ dimensions. On purpose, we leave it open to the user, which DQ dimensions are associated to constraint definitions in order to provide flexible configuration. Further, more formal investigations should be carried out to establish what types and aspects of DQ are, or could be, supported with these approaches. This would allow guidelines to be provided about which constraint types can be related to what DQ dimensions. For example, the objective DQ assessment could also consider the data source reputation, such as different laboratories, for an indication of DQ. Reputation is an important factor, especially when handling scientific data, as with FoodCASE. If analytical values are received from different laboratories with different levels of reputation, the DQ assessment could indicate the data from a laboratory with low reputation as of lower quality. Such a DQ assessment could be realised by associating the DQ dimension *reputation* with an appropriate severity to analytical results.

With our *UnifiedOCL* approach, our constraint translations generate complete target representations without regard to previously generated output. It could be useful to regenerate just parts of the output, so that existing pieces of code in the target representation could be preserved. This could be achieved by annotating parts of the source or target code specification, as well as the intermediate representation *UnifiedOCL*.

There still exist many ways of optimising the runtime of our AOP-based approach, presented in Chap. 7. For example, the constraint repositories could be cached to avoid loading and parsing runtime for the checking of constraints and the generation of DQ metadata.

We think that DQIF and DQAF can always provide more possibilities giving feedback on DQ. For example, it would be possible to give the user an indication of constraint dependencies between data input fields in the GUI, for example caused by class constraints. It would also be convenient to present DQIF by adding short DQ feedback close to the appropriate fields, if need be by an additional accompanied symbol such as a light bulb with attached tooltip.

The work in this thesis provides numerous opportunities for future research in combination with our approaches and we outline a few possible directions below.

We have not addressed the problem of *constraint validation* for sets of constraints. Sets of constraints, which are used for unified constraint management, should also be validated themselves, to identify and resolve interdependences and contradictions between constraint definitions.

In our approaches, constraint checking is always done on the basis of precisely defined constraints. It would also be possible to define constraints in a more generic way,

checking them against available metadata, statistics, or existing data in the database. For example, a range constraint for an analytical value could be defined, which checks if a new value is in the range of already stored values. If not, it could be an indication of incorrect data or an outlier value. Note that such constraint checks would have to be delayed until a number of reliable values are stored in the database. Based on metadata, the objective DQ assessment could also consider the specific type of data, such as different substances, for an indication of DQ. For example, if different substances, such as selenium and mercury, are checked for certain value ranges, the validation could depend on the type of substance, even if the substances represent the same entity. Such a DQ assessment could be realised by including additional metadata for the substances during the assessment process.

In our approaches, we give indications of how DQ can be improved, but we always leave it up to the user to actually correct the data. A reasoning system could support the data correction process by semi-automatically, or even automatically, adapting data. Since automatically adapting data can be disastrous for the DQ in case of faulty reasoning, at least recommendations could be provided which then could be approved by users.

Constraint checking and DQ assessment on data items is the basis for accumulating DQ information for higher purposes. The accumulated numbers could be used for charts and reports to give an overview for business processes and business summaries. Based on such overviews, decisions are taken in business. We assume, that there is a lack of including such detailed DQ information in management reports which are the basis for important decisions. For example, it would be beneficial if the concepts provided in this work could be included in enterprise management software, such as the SAP Information Steward¹. There, the focus lies on data cleaning, such as deduplication and address data validation.

¹<https://www.sap.com/products/data-profiling-steward.html>

A

TDS-Exposure workshop 2015 – Questionnaire

We provide here the list of questions from the questionnaire that was completed by the participants of the TDS-Exposure workshop 2015 in Zurich. The introductory questions asked for the participants' position and skills and aimed to provide some degree of context to their responses. For the questions in Sect. A to Sect. A, the participants were asked to choose from a five-level Likert-type scale if they 'Strongly Disagree', 'Disagree', are 'Neutral', 'Agree' or 'Strongly Agree'. The remainder of the questions (Sect. A to Sect. A) could be answered in own words, with the exception of questions A-1 and A-3, which accepted a 'Yes' or 'No' answer.

Intro and user skills

- Position
- Name (optional)
- How would you estimate your computer skills? (Choose from 1-5, where 1=beginner, 3=average user, 5=professional)

Contracts

1. I find it useful to be able to assign a weight (or 'importance score') to each data constraint
2. I find specifying the constraint weights using a scale of 0-100 (where 0=not important and 100=most important) intuitive

3. I would like the option to have different ‘contracts’ or quality profiles (different constraint importance scores and different active/inactive constraints) for different users and applications of the system
4. I would like to be able to share and exchange contract files with other users
5. I would like to be able to easily activate/deactivate constraints at runtime
6. For some constraints, it would be useful to have a weight (importance score) that would override the contract weights of the constraint
7. It is acceptable if the weight can only be defined by a developer

Input validation

1. During input validation, I find a quality indicator percentage useful
2. I would like to have a visual representation of quality during input validation
3. If I don’t specify a contract for input validation, I would like all constraints to be of equal weight
4. If I don’t specify a contract for input validation, I would like a default contract to be used
5. When some important constraints are violated, it should not be possible to save the data
6. When some less important constraints are violated, it is OK to be able to save the data
7. In some cases I would like to be able to save the data even if important constraints are violated, as long as I am aware of the violations and can provide an explanation about their occurrence (e.g. data not available yet)
8. When a constraint violation occurs during data input, I would like the system to provide an example of data that satisfies the constraint
9. When a constraint violation occurs during data input, I would like the system to insert the most probable value that satisfies the constraint

Analysis

1. I think that being able to perform an overall data quality analysis is a useful feature
2. I would like to be able to analyse the data quality according to certain constraint groups (e.g. only perform data completeness checks or value correctness controls)
3. I would like to be able to only analyse certain data entities (e.g. samples)

4. I find it useful to have the data quality analysis results in textual form
5. I would like to have visual representations (e.g. graphs, pie charts) of data quality analysis results

Access rights

1. Anyone should be able to perform a data quality analysis
2. Anyone should be able to decide on the weight threshold that distinguishes errors from warnings
3. Anyone should be able to disable warnings during input validation

Constraints

1. It is acceptable that new constraints can only be defined by developers and not any user of the application
2. It would be useful to be able to organize constraints into groups (logical units) in order to be able to validate/analyse only these groups
3. It is acceptable if constraints can only be assigned to groups by developers
4. It would be necessary that the constraints could be defined during the running application by a non-developer such as a power user
5. It would be helpful for users of the application (non-developers) to be able to define customized groups of constraints during runtime

Constraint grouping

1. Can you think of additional ways to group constraints that would be useful to you? If so, please describe.

Comparison with existing food composition data quality framework

1. Have you had experience using the existing Food Composition input validation framework? (Yes/No)
2. If yes, how would you compare it to the Contaminant input validation framework that was presented today?
3. Have you had experience using the existing Food Composition data quality analysis framework? (Yes/No)

4. If yes, how would you compare it to the Contaminant data quality analysis framework that was presented today?

Additional comments or suggestions

1. Do you have any comments or suggestions related to the data quality framework? If so, please describe.

B

TDS-Exposure workshop 2015 – Questionnaire findings

We present here our detailed findings for the questionnaire answered at the TDS-Exposure workshop 2015 in Zurich. The findings are organised according to the general question category that they correspond to. In all the presented answers, the participants could choose in the appropriate question from a five-level Likert-type scale if they ‘Strongly Agree’ (SA), ‘Agree’ (A), are ‘Neutral’ (N), ‘Disagree’ (D) or ‘Strongly Disagree’ (SD). Next to each finding, we indicate the percentages of answers that led to its conclusion.

Contracts

#	Finding	SA%	A%	N%	D%	SD%
1	Almost 90% of the participants consider assigning a weight to each data constraint useful	40	46.67	13.33	-	-
2	Almost 90% find the 0-100 weight scale intuitive	33.33	53.33	6.67	6.67	-
3	Almost half of the participants consider having different contracts for different users and applications of the system useful	6.67	40	53.33	-	-
4	60% would like to be able to share and exchange contract files with other users	13.33	46.67	33.33	6.67	-

Continued on next page

Table B.1 – *Continued from previous page*

#	Finding	SA%	A%	N%	D%	SD%
5	While 40% of the participants would like to be able to activate or deactivate constraints at runtime, about 30% disagree	13.33	26.67	33.33	13.33	13.33
6	40% find the existence of payload weights useful	6.67	33.33	53.33	6.67	-
7	Almost 70% find it acceptable if the payload weight can only be defined by developers, while more than 25% disagree	6.67	60	6.67	20	6.67

Table B.1: Findings related to the concept of contracts

Input validation

#	Finding	SA%	A%	N%	D%	SD%
1	100% of the participants find a quality indicator during input validation (<i>IVQI</i>) useful	13.33	86.67	-	-	-
2	Almost 75% would like to have a visual representation of quality during input validation	13.33	60	13.33	13.33	-
3	While about 35% of the participants would like all constraints to be considered equivalent if a contract is not selected, 40% disagree	6.67	26.67	26.67	26.67	13.33
4	Almost 70% agree that if a contract is not selected for input validation, a default contract should be used	6.67	60	26.67	6.67	-
5	More than 85% of the participants agree that when an important constraint is violated, it shouldn't be possible to save the data	13.33	73.33	6.67	6.67	-
6	80% agree that if less important constraints are violated, it should be possible to save the data	20	60	6.67	13.33	-

Continued on next page

Table B.2 – *Continued from previous page*

#	Finding	SA%	A%	N%	D%	SD%
7	While more than 50% would like to be able to save the data even if important constraints are violated, as long as they are aware of the violations and can explain their occurrence, 30% disagree	-	53.33	13.33	33.33	-
8	If a violation occurs during input validation, 80% would like the system to provide an example of appropriate data	-	80	20	-	-
9	If a violation occurs during input validation, almost 70% would not like the system to insert the most probable value that satisfies the constraint	6.67	6.67	20	46.67	20

Table B.2: Findings related to input validation

Analysis

#	Finding	SA%	A%	N%	D%	SD%
1	80% of the participants agree that being able to perform an overall data quality analysis is a useful feature	20	60	20	-	-
2	More than 70% would like to be able to analyse the data quality according to certain constraint groups	6.67	66.67	20	6.67	-
3	60% would like to be able to only analyse certain data entities	6.67	53.33	20	20	-
4	More than half of the participants find having the analysis results in textual form useful	-	53.33	46.67	-	-
5	80% would like to have visual representations of the analysis results	33.33	46.67	20	-	-

Table B.3: Findings related to data quality analysis

Access rights

#	Finding	SA%	A%	N%	D%	SD%
1	Almost 70% of the participants believe that anyone should be able to perform a data quality analysis	6.67	60	26.67	6.67	-
2	Almost 70% of the participants don't think that the weight threshold that distinguishes errors from warnings should be decided by any type of user	-	13.33	20	53.33	13.33
3	More than 70% disagree with allowing any type of user to disable input validation warnings	-	13.33	13.33	73.33	-

Table B.4: Findings related to user access rights

Constraints

#	Finding	SA%	A%	N%	D%	SD%
1	More than half of the participants find it acceptable if new constraints can only be defined by developers	20	33.33	33.33	13.33	-
2	More than 70% find organizing constraints into groups useful	-	73.33	26.67	-	-
3	While 40% of the participants find it acceptable if constraints can only be assigned to groups by developers, 20% disagree	-	40	40	20	-
4	While about 25% of the participants believe it would be necessary that the constraints could be defined during the running application by a non-developer (such as a power user), 40% disagree	-	26.67	33.33	33.33	6.67
5	More than 50% would like to be able to define customized groups of constraints during runtime	6.67	46.67	46.67	-	-

Table B.5: Findings related to constraints

C

List of supported constraints with UnifiedOCL labels

We present here the list of all constraints supported by *UnifiedOCL* labels originating from BV, HV and *PostgreSQL* including the appropriate parameters.

BV annotation	UnifiedOCL label	UnifiedOCL label parameters
@AssertTrue	asserttrue	
@AssertFalse	assertfalse	
@DecimalMin	decimalmin	value (STRING) inclusive (BOOLEAN)
@DecimalMax	decimalmax	value (STRING) inclusive (BOOLEAN)
@Min	decimalmin	value (INTEGER) inclusive (BOOLEAN)
@Max	decimalmin	value (INTEGER) inclusive (BOOLEAN)
@Size	size	min (INTEGER) max (INTEGER) inclusive (BOOLEAN)
@Future	future	
@Past	past	
@Digits	digits	integer (INTEGER) fraction (INTEGER)
@Pattern	pattern	regexp (STRING) flag (STRING) – partially supported
@Null	null	
@NotNull	notnull	

Table C.1: Bean Validation – Supported constraints

HV annotation	UnifiedOCL label	UnifiedOCL label parameters
@NotEmpty	notempty	
@NotBlank	notblank	
@Length	length	min (INTEGER) max (INTEGER)
@Range	range	min (INTEGER) max (INTEGER)
@Email	email	
@CreditCardNumber	creditcard	ignoreNonDigitCharacters (BOOLEAN) – partially supported
@EAN	ean	type (INTEGER)
@URL	url	protocol (STRING) host (STRING) port (STRING) regexp (STRING) – partially supported flags (STRING) – partially supported

Table C.2: Hibernate Validator – Supported constraints

SQL constraint	UnifiedOCL label or UnifiedOCL concept	UnifiedOCL label parameters
PRIMARY KEY	primarykey	
FOREIGN KEY REFERENCES	foreignkey <i>UnifiedOCL structural feature that is of class or interface type</i>	table (IDENTIFIER) column (IDENTIFIER)
NOT NULL	notnull	
NULL	null	
UNIQUE	unique	
SERIAL	autoincrement	
column type NUMBER with defined precision of num- bers	digits	integer (INTEGER) fraction (INTEGER)
column type VARCHAR with defined max length	length	max (INTEGER)
CHECK constraints that cov- ers most of the correspond- ing concepts from BV and HV	see UnifiedOCL labels from Tab. C.2 and Tab. C.1	

Table C.3: SQL (PostgreSQL) – Supported constraints

D

List of supported built-in field constraint types of the DQAnalytics framework

We present here the list of the built-in field constraint types available in our DQMF, described in Chap. 7. The choice of predefined constraint types, presented in Table D.1, was motivated by the supported constraints of standard BV and HV, respectively.

The supported field constraints are mostly consistent with BV/HV but we provide an additional field constraint type `compare`. The `compare` field constraint allows to compare a field value with the value of another field. Consider, for example, an analytical value in food science which contains multiple attributes to store the analytical results, such as single analytical results from different portions of a sample, a mean value for on average of all results, and a minimum (`min`) and maximum (`max`) value for the lower and upper analytical results. Thus, the `compare` field constraint can be defined on the `max` field of the analytical value entity, the `compareop` set to ‘ \geq ’ and `compareto` can refer to the `min` field of the analytical value entity. Consequently, the `compare` field constraint guarantees that the `max` value is greater or equal to the `min` value.

Constraint Type	Description	Arguments
required	True if the validated value is not null	-
size	True if the validated value has a character length between (and including) arguments <i>min</i> and <i>max</i> . If <i>min</i> or <i>max</i> is not available, any upper or lower bound is accepted	min (Integer), max (Integer)
max	True if value to be stored is smaller or equals the value defined in argument <i>value</i> .	value (Numeric)
min	True if value to be stored is greater or equals the value defined in argument <i>value</i> .	value (Numeric)
pattern	True if value to be stored fulfils the pattern defined in argument <i>value</i> .	value (String)
future	True if date to be stored is in the future	-
past	True if the date to be stored is in the past	-
digits	True if number to be stored consists of argument <i>integer</i> numbers of digits before the decimal point, and argument <i>fraction</i> numbers of digits after the decimal point	integer (Integer), fraction (Integer)
null	True if the value to be stored is null	-
false	True if the value to be stored is Boolean false	-
true	True if the value to be stored is a Boolean true	-
url	True if the value to be stored is a valid url	-
email	True if the value to be stored is a valid e-mail address	-
compare	True if the value to be stored fulfils the compare operator defined in argument <i>compareop</i> with the value in field defined by argument <i>compareto</i> .	compareop (String), compareto (String)

Table D.1: Built-in field constraint types

E

Various data quality analysis visualisations of the DQAnalytics framework

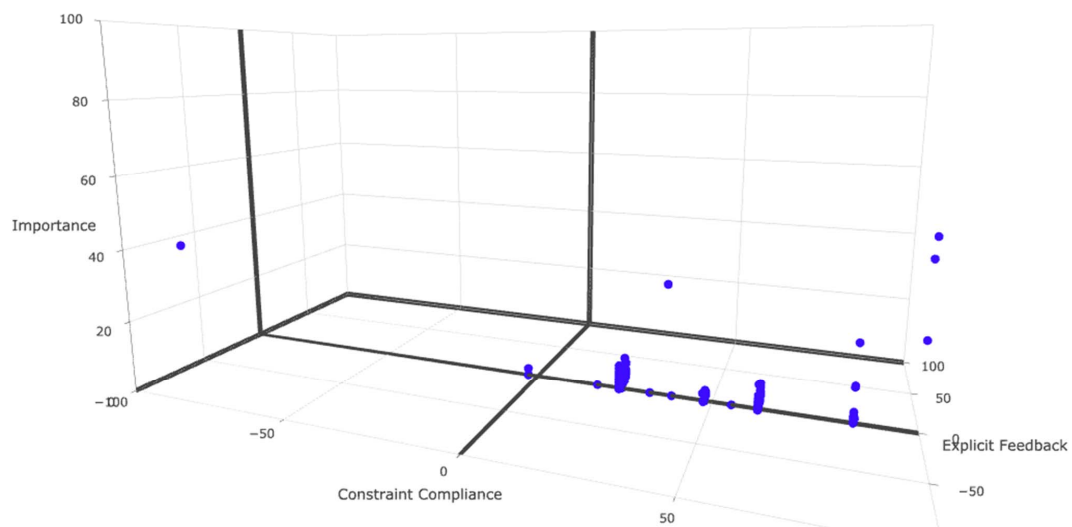


Figure E.1: DQAnalytics – Vector space model chart

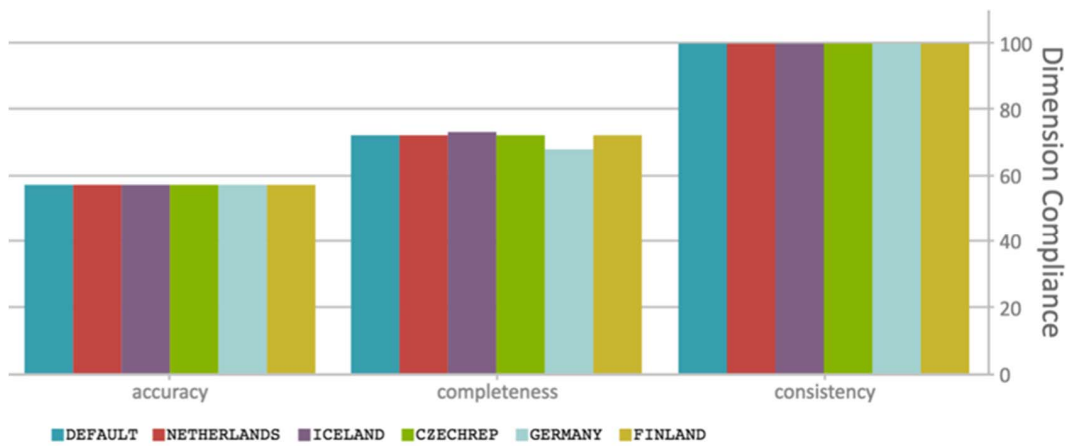


Figure E.2: DQAnalytics – Dimension compliance chart

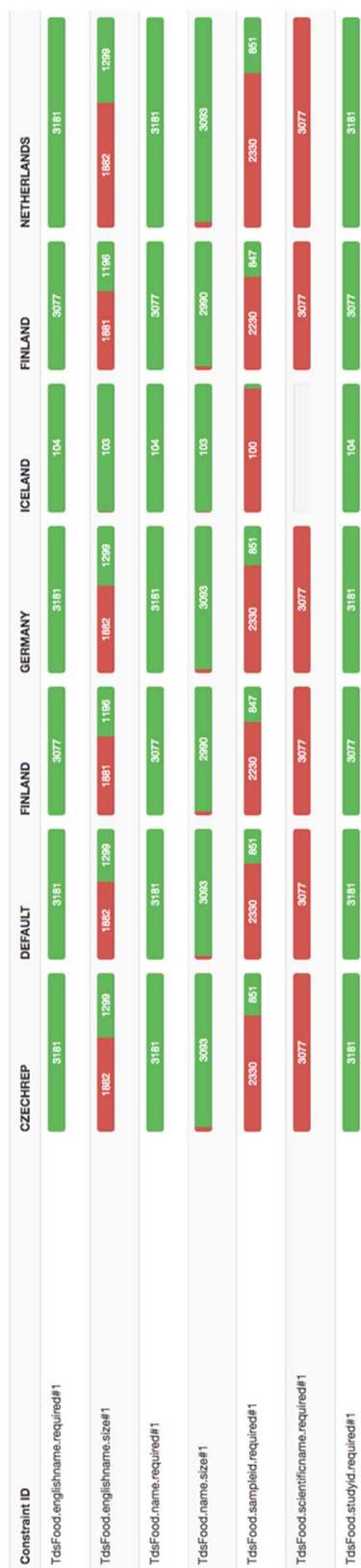
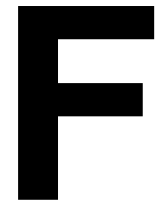


Figure E.3: DQAnalytics – Constraint compliance analysis chart

Timestamp	User Reputation	Instance Rating	Field Feedback	Data is wrong	Constraints are wrong	Comment	Resolve field	Instance Feedback	Resolve Instance
2016-07-23T20:59:56.213Z	3	-2	Field Name scientificname	yes	no	-	resolved	Field should not be empty, please add 'Musa paradisiaca'	resolved
2016-07-14T15:15:32.826Z	3	-2	Field Name scientificname	yes	no	-	resolved	Should be removed	resolved
2016-07-14T15:13:47.172Z	3	-1	Field Name scientificname	yes	no	-	resolved	Name and scientific name are missing!	resolved
2016-07-04T17:36:14.924Z	3	-1	Field Name name scientificname	yes yes	no no	- -	resolved resolved	scientific name looks wrong	resolved

Figure E.4: DQAnalytics – List of explicit feedback



User study of the DQAnalytics framework

Questionnaire

Study Section 1 - Basic Information

1. **Gender** female/male
2. **Year of Birth** Numeric Input
3. **What is the highest level of education you have achieved?**
 - Bachelor in CS (or similar)
 - Master in CS (or similar)
 - PhD in CS (or similar)
 - Bachelor in applied sciences
 - Master in applied sciences
 - Apprenticeship
 - Other
4. **Are you working in Software Engineering**
 - Yes, Full Time
 - Yes, Part Time
 - No
5. **Rate your general Software Engineering Skills** 1-7 Likert-Scale (from *no experience* to *expert*)

6. **Rate your Web Engineering Skills** 1-7 Likert-Scale
7. **Rate your experience in data quality topics** 1-7 Likert-Scale
8. **Rate your experience in constraint modelling** 1-7 Likert-Scale
9. **Rate your experience with JSON** 1-7 Likert-Scale
10. **Do you have experience with Distributed Constraint Management (i.e. Constraints in different representations, like JavaScript on client-side and database constraints on database)**
 - Yes
 - No
11. **If you answered the previous question with no, please specify**
 - I do not need any constraints
 - I do not need distributed constraints
 - I never worked as a Software Engineer
 - Other (Text Input)

Study Section 2 - Distributed Constraints in Client/Server Applications

Note: this section is only filled out if the participant answered question number 10 from section 1 with **yes**.

1. **Input validation is important to ensure the quality of data** 1-5 Likert-Scale (from *totally disagree* to
2. **Maintaining multiple constraint representations is time-consuming** 1-5 Likert-Scale
3. **Multiple constraint representations are hard to maintain** 1-5 Likert-Scale
4. **Multiple constraint representations are a frequent source of failures** 1-5 Likert-Scale
5. **There is a lack of supporting tools/paradigms to develop and maintain multiple constraint representations in heterogeneous Client/Server Applications** 1-5 Likert-Scale
6. **In software development we (company) usually define constraints in various components of our applications (such as GUI, business logic and database)** 1-5 Likert-Scale
7. **In software development we (company) usually define complex constraints in the business logic rather than in the database** 1-5 Likert-Scale

8. In software development we (company) usually define no or just a very few constraints in the database 1-5 Likert-Scale
9. List the paradigms/technologies which you are mainly using for constraint definitions in the various components of your (company) applications (such as JavaScript in the GUI, JavaBeans Validation in the business logic and SQL in the database) Text Input

Study Section 3 - Answers to Task 1

1. Have you solved the task?
 - Yes
 - Partially
 - No
2. Did you encounter any problems? If so, state them. Text Input
3. The feedback tool was easy to use 1-5 Likert-Scale
4. I was able to express my feedback through the form provided 1-5 Likert-Scale
5. How would you simplify or improve the feedback process? Text Input

Study Section 4 - Answers to Task 2

1. Have you solved the task?
 - Yes
 - Partially
 - No
2. Did you encounter any problems? If so, state them. Text Input
3. The task was easy to solve 1-5 Likert-Scale
4. The obtained result is what I expected 1-5 Likert-Scale
5. The constraint definition format (JSON) is easy to learn Text Input
6. How would you simplify or improve the constraint definition process? Text Input

Study Section 5 - Answers to Task 3

1. When you look at the improvement tendency chart, which of the entities that are available for analysis, would you improve first? Text Input
2. What is the most problematic constraint of the entity you have chosen in 1.? Text Input
3. Give the ID of the instance of the entity you have chosen in 1. that needs data quality improvements with the highest urgency. Text Input
4. Spatial View: Of all instances in the system, which one needs data quality improvements with the lowest urgency? Text Input
5. Have you solved the task?
 - Yes
 - Partially
 - No
6. Did you encounter any problems? If so, state them. Text Input
7. The task was easy to solve 1-5 Likert-Scale
8. In Data Quality Enhancement: Flat view is intuitive to use 1-5 Likert-Scale
9. In Data Quality Enhancement: Spatial view is intuitive to use 1-5 Likert-Scale
10. Do you suggest another form of visualization besides Flat and Spatial view? Text Input

Study Section 6 - Usability Evaluation DQAnalytics

1. I think that I would like to use this system frequently 1-5 Likert-Scale
2. I found the system unnecessarily complex 1-5 Likert-Scale
3. I thought the system was easy to use 1-5 Likert-Scale
4. I think that I would need the support of a technical person to be able to use this system 1-5 Likert-Scale
5. I found the various functions in this system were well integrated 1-5 Likert-Scale
6. I thought there was too much inconsistency in this system 1-5 Likert-Scale
7. I felt very confident using the system 1-5 Likert-Scale

8. I needed to learn a lot of things before I could get going with this system 1-5 Likert-Scale
9. Did you encounter any undesirable or unexpected limitations? If so, please mention or describe them. Text input
10. Please give an overall rating of the system 1-10 Likert-Scale
11. What features do you recommend to improve the future system design? Text input
12. What services do you recommend to be integrated into the system? Text input
13. Do you have any other comments or suggestions? Text input

Study Section 7 - Closing Questions

1. Data Quality monitoring helps to increase the quality of data 1-5 Likert-Scale
2. There is a need for tools which provide information about the quality of data in an Information System 1-5 Likert-Scale
3. DQAnalytics provides a good understanding of the quality of data of the underlying system 1-5 Likert-Scale
4. How would you simplify or improve DQAnalytics? Text Input
5. As a user of an Information System, I would invest my time and provide feedback to data instances in order to help increasing the data quality 1-5 Likert-Scale
6. As a System Administrator, I think user feedback can be used to better understand the quality of data in an Information System 1-5 Likert-Scale
7. As a developer I would use DQAnalytics in my software products because of the unified constraint management 1-5 Likert-Scale
8. As a developer I would use DQAnalytics in my software products because of the data quality analysis tools 1-5 Likert-Scale
9. Any other Feedback you want to provide to DQAnalytics Text input

Task Description Sheet

The following task description sheet was handed out to the study participants to solve the practical tasks.

Task 1

In this task you simulate an end-user of FoodCaseWEB. Browse through the TdsFood instances and look at the data available. Rate the perceived data quality of five instances, where you mainly look whether:

- The instance has a correct looking name
- The instance has a correct looking scientific name
- The instance has a correct looking english name

Express your feedback as accurately as possible.

Task 2

The administrators of FoodCASEWeb decided that TdsFood instances must have set a scientific name (importance: 100), and that the minimum length of the scientific name should be 3 characters (importance: 60). Both constraints should contribute to dimension “completeness” with a value of 100.

General JSON Constraint Structure

ConstraintName	Description	Arguments
required	True if validated value is not null	-
size	True if the validated value has a character length between (and including) min and max . If min or max is not available, any upper or lower bound is accepted	min (INTEGER, optional) max (INTEGER, optional)

Your Task

1. Modify the constraint repository such that the mentioned requirements are enforced by the constraints.
2. Test whether your changes are applied correctly when modifying instances in FoodCASEWeb

Task 3

Use DQAnalytics to answer the questions in Google Form Task 3.

Figure F.1: User study task description sheet



Student contributions

The following students, which all have been personally supervised by this thesis' author, have made a significant contribution to this thesis as part of their Master's or Bachelor's project.

- **Oliver Probst** has developed the DQMF presented in Sect. 3.3
- **Benjamin Oser** has developed the DQMF presented in Sect. 3.4 and contributed to the measure presented in Sect. 7.4.3
- **Konstantina Gemenetzi** has developed the DQMF presented in Chap. 4
- **Jakub Szymanek** has developed the DQMFs presented in Sect. 3.5 and Chap. 5
- **Diana Birenbaum** has developed the DQMF presented in Chap. 6
- **Michael Berli** has developed the DQMF presented in Chap. 7

Bibliography

- [1] T.L. Alves, P.F. Silva, and J. Visser. Constraint-Aware Schema Transformation. *Electr. Notes Theor. Comput. Sci.*, 290:3–18, 2012. Cited on page 41.
- [2] N. Askham, D. Cook, M. Doyle, H. Fereday, M. Gibson, U. Landbeck, R. Lee, C. Maynard, G. Palmer, and J. Schwarzenbach. The Six Primary Dimensions for Data Quality Assessment: Defining Data Quality Dimensions. Technical report, DAMA UK Working Group, October 2013. Cited on page 20.
- [3] C. Avila, G. Flores, and Y. Cheon. A Library-Based Approach to Translating OCL Constraints to JML Assertions for Runtime Checking. In *Proc. SERP*, pages 403–408, 2008. Cited on page 39.
- [4] C. Avila, A. Sarcar, Y. Cheon, and C. Yeep. Runtime Constraint Checking Approaches for OCL, A Critical Comparison. In *Proc. SEKE*, pages 393–398, 2010. Cited on pages 34 and 39.
- [5] S. Bagui. Achievements and Weaknesses of Object-Oriented Databases. *Object Technology*, 2(4):29–41, 2003. Cited on page 36.
- [6] I.S. Bajwa and M.G. Lee. Transformation Rules for Translating Business Rules to OCL Constraints. In *Proc. ECMFA*, pages 132–143, 2011. Cited on page 38.
- [7] D.P. Ballou and H.L. Pazer. Modeling Completeness versus Consistency Tradeoffs in Information Decision Contexts. *IEEE Trans. Knowl. Data Eng.*, 15(1):240–243, 2003. Cited on page 21.
- [8] D.P. Ballou, R. Wang, H.L. Pazer, and G.K. Tayi. Modeling Information Manufacturing Systems to Determine Information Product Quality. *Manage. Sci.*, 44(4):462–484, 1998. Cited on pages 46 and 47.
- [9] S. Balzer, P.T. Eugster, and B. Meyer. Can Aspects Implement Contracts? In *Proc. RISE*, pages 145–157, 2005. Cited on page 30.
- [10] L. Baresi and M. Young. Toward Translating Design Constraints to Run-Time Assertions. *Electr. Notes Theor. Comput. Sci.*, 116:73–84, 2005. Cited on page 40.
- [11] C. Batini, C. Cappiello, C. Francalanci, and A. Maurino. Methodologies for Data Quality Assessment and Improvement. *ACM Comput. Surv.*, 41(3), 2009. Cited on pages 1, 41, and 44.

- [12] C. Batini and M. Scannapieco. *Data and Information Quality – Dimensions, Principles and Techniques*. Data-Centric Systems and Applications. Springer, 2016. Cited on pages 1, 2, 20, 21, and 41.
- [13] D. Beneventano, S. Bergamaschi, S. Lodi, and C. Sartori. Consistency Checking in Complex Object Database Schemata with Integrity Constraints. *IEEE Trans. Knowl. Data Eng.*, 10(4):576–598, 1998. Cited on pages 24 and 36.
- [14] M. Berli. Monitoring Data Quality Evolution based on Explicit and Implicit User Feedback. <https://doi.org/10.3929/ethz-b-000175991>, 2016. Cited on pages 183 and 184.
- [15] D. Birenbaum. Data Quality Assurance in a Mobile Application for Food Science Data. <http://dx.doi.org/10.3929/ethz-a-010655396>, 2016. Cited on page 148.
- [16] R. Blake. Identifying the Core Topics and Themes of Data and Information Quality Research. In *Proc. AMCIS*, 2010. Cited on page 22.
- [17] M. Book, T. Brückmann, V. Gruhn, and M. Hülder. Specification and Control of Interface Responses to User Input in Rich Internet Applications. In *Proc. ASE*, pages 321–331, 2009. Cited on page 50.
- [18] M. Bouzeghoub and E. Métais. Semantic Modeling of Object Oriented Databases. In *Proc. VLDB*, pages 3–14, 1991. Cited on page 35.
- [19] C. Brabrand, A. Møller, M. Ricky, and M.I. Schwartzbach. PowerForms: Declarative Client-Side Form Field Validation. *World Wide Web*, 3(4):205–214, 2000. Cited on pages 33 and 50.
- [20] E.H. Bradley, E.S. Holmboe, J.A. Mattera, S.A. Roumanis, M.J. Radford, and H.M. Krumholz. Data Feedback Efforts in Quality Improvement: Lessons Learned from US Hospitals. *Quality & Safety in Health Care*, 13(1):26–31, 2004. Cited on page 51.
- [21] I. Caballero, E. Verbo, C. Calero, and M. Piattini. A Data Quality Measurement Information Model Based On ISO/IEC 15939. In *Proc. ICIQ*, pages 393–408, 2007. Cited on pages 45 and 46.
- [22] J. Cabot and E. Teniente. Constraint Support in MDA Tools: A Survey. In *Proc. ECMDA-FA*, pages 256–267, 2006. Cited on page 34.
- [23] C. Cappiello, C. Francalanci, and B. Pernici. Data Quality Assessment from the User’s Perspective. In *Proc. IQIS*, pages 68–73, 2004. Cited on page 47.
- [24] K. Cemus and T. Cerný. Aspect-Driven Design of Information Systems. In *Proc. SOFSEM*, pages 174–186, 2014. Cited on page 38.
- [25] T. Cerný, K. Cemus, M.J. Donahoo, and E. Song. Aspect-Driven, Data-Reflective and Context-Aware User Interfaces Design. *SIGAPP Appl. Comput. Rev.*, 13(4):53–66, 2013. Cited on pages 38 and 134.

- [26] T. Cerný, M.J. Donahoo, and E. Song. Towards Effective Adaptive User Interfaces Design. In *Proc. RACS*, pages 373–380, 2013. Cited on pages 38 and 134.
- [27] I.N. Chengalur-Smith, D.P. Ballou, and H.L. Pazer. The Impact of Data Quality Information on Decision Making: An Exploratory Analysis. *IEEE Trans. Knowl. Data Eng.*, 11(6):853–864, 1999. Cited on page 45.
- [28] Y. Cheon, C. Avila, S. Roach, C. Munoz, N. Estrada, V. Fierro, and J. Romo. An aspect-based approach to checking design constraints at run-time. In *Proc. ITNG*, pages 223–228, 2009. Cited on page 39.
- [29] M.-Y. Choi, E.-A. Cho, D.-H. Park, J.-Y. Bae, C.-J. Moon, and D.-K. Baik. A Synchronization Algorithm of Mobile Database for Ubiquitous Computing. In *Proc. Joint Conf. INC, IMS and IDC*, pages 416–419, 2009. Cited on page 53.
- [30] V. Cosentino and S. Martínez. Extracting UML/OCL Integrity Constraints and Derived Types from Relational Databases. In *MODELS Int'l Workshops*, pages 43–52, 2013. Cited on page 39.
- [31] U. Dayal, A.P. Buchmann, and D.R. McCarthy. Rules are Objects Too: A Knowledge Model for an Active, Object-Oriented Database System. In *Proc. OODBS*, pages 129–143, 1988. Cited on page 27.
- [32] R. De La Harpe. A Semiotic View on Paper and Mobile Care Data Quality. *Stud. Health Technol. Inform.*, 6:180–442, 2012. Cited on page 52.
- [33] H. Decker. Modeling and Monitoring the Quality of Data by Integrity Constraints and Integrity Checking. In *Proc. ICSoft*, pages 207–214, 2009. Cited on pages 41 and 42.
- [34] H. Decker. Quantifying the Quality of Stored Data by Measuring Their Integrity. In *Proc. ICADIWT*, pages 796–801, 2009. Cited on page 42.
- [35] H. Decker. Causes of the Violation of Integrity Constraints for Supporting the Quality of Databases. In *Proc. ICCSA*, pages 283–292, 2011. Cited on page 42.
- [36] H. Decker. Data Quality Maintenance by Integrity-Preserving Repairs that Tolerate Inconsistency. In *Proc. QSIC*, 2011. Cited on page 42.
- [37] H. Decker. New Measures for Maintaining the Quality of Databases. In *Proc. ICCSA*, pages 170–185, 2012. Cited on page 42.
- [38] H. Decker and D. Martinenghi. Modeling, Measuring and Monitoring the Quality of Information. In *ER Workshops*, pages 212–221, 2009. Cited on page 42.
- [39] A. Dedeke. A Conceptual Framework for Developing Quality Measures for Information Systems. In *Proc. IQ*, pages 126–128, 2000. Cited on page 20.
- [40] W.E. Deming. *Out of the Crisis*. MIT/CAES, 1986. Cited on page 43.

- [41] A. Demuth, R.E. Lopez-Herrejon, and A. Egyed. Constraint-Driven Modeling Through Transformation. *Software and System Modeling*, 14(2):573–596, 2015. Cited on page 39.
- [42] N. Do, I.J. Choi, and M.K. Jang. A Structure-Oriented Product Data Representation of Engineering Changes for Supporting Integrity Constraints. *Adv. Manuf. Technol.*, 20(8):564–570, 2002. Cited on page 36.
- [43] N.C. Do, S.M. Bae, and I.J. Choi. Constraint Maintenance in Engineering Design System: An Active Object-Oriented Approach. *Computers & Industrial Engineering*, 33(3-4):643–647, 1997. Cited on pages 31 and 36.
- [44] W.J. Dzidek, L.C. Briand, and Y. Labiche. Lessons Learned from Developing a Dynamic OCL Constraint Enforcement Tool for Java. In *MODELS Int'l Workshops*, pages 10–19, 2005. Cited on page 39.
- [45] Z. Dzolkhifli, H. Ibrahim, and L.S. Affendey. Analyzing Integrity Tests for Data Caching in Mobile Databases. In *Proc. ICDCIT*, pages 157–165, 2008. Cited on page 52.
- [46] Z. Dzolkhifli, H. Ibrahim, L.S. Affendey, and P. Madiraju. A Framework for Caching Relevant Data Items for Checking Integrity Constraints of Mobile Database. *iJIM*, 3(S2):18–23, 2009. Cited on page 53.
- [47] J. Edwards, A. Bahjat, Y. Jiang, T. Cook, and T.F. La Porta. Quality of Information-Aware Mobile Applications. *Pervasive and Mobile Computing*, 11:216–228, 2014. Cited on page 52.
- [48] M. Egea, C. Dania, and M. Clavel. MySQL4OCL: A Stored Procedure-Based MySQL Code Generator for OCL. *Commun. EASST*, 36, 2010. Cited on page 39.
- [49] C.F. Eick and P. Werstein. Rule-Based Consistency Enforcement for Knowledge-Based Systems. *IEEE Trans. Knowl. Data Eng.*, 5(1):52–64, 1993. Cited on pages 32 and 36.
- [50] R. Elmasri, S. James, and V. Kouramajian. Automatic Class and Method Generation for Object-Oriented Databases. In *Proc. DOOD*, pages 395–414, 1993. Cited on page 35.
- [51] L.P. English. *Improving Data Warehouse and Business Information Quality: Methods for Reducing Costs and Increasing Profits*. John Wiley & Sons, Inc., 1999. Cited on page 46.
- [52] L.P. English. *Information Quality Applied: Best Practices for Improving Business Information, Processes and Systems*. Wiley Publishing, 2009. Cited on page 1.
- [53] M.J. Eppler. *Managing Information Quality*, volume Second Edition. Springer, 2006. Cited on pages 20 and 44.
- [54] E. Escott, P.A. Strooper, P. King, and I.J. Hayes. Model-Driven Web Form Validation with UML and OCL. In *Proc. ICWE*, pages 223–235, 2011. Cited on page 34.

- [55] EssentialOCL. <http://help.eclipse.org/neon/index.jsp?topic=%2Forg.eclipse.oc1.doc%2Fhelp%2FessentialOCL.html>. Accessed: 2017-02-21. Cited on page 125.
- [56] K.P. Eswaran and D.D. Chamberlin. Functional Specifications of a Subsystem for Database Integrity. In *Proc. VLDB*, pages 48–68, 1975. Cited on page 27.
- [57] C. Fahrner, T. Marx, and S. Philippi. DICE: Declarative Integrity Constraint Embedding Into the Object Database Standard ODMG-93. *Data Knowl. Eng.*, 23(2):119–145, 1997. Cited on pages 35 and 36.
- [58] W. Fan. Dependencies Revisited for Improving Data Quality. In *Proc. PODS*, pages 159–170, 2008. Cited on page 42.
- [59] W. Fan and F. Geerts. *Foundations of Data Quality Management*. Synthesis Lectures on Data Management. Morgan & Claypool Publishers, 2012. Cited on pages 21 and 22.
- [60] W. Fan, F. Geerts, and X. Jia. Semandaq: A Data Quality System Based on Conditional Functional Dependencies. *PVLDB*, 1(2):1460–1463, 2008. Cited on page 42.
- [61] D.D. Fehrenbacher and M. Helfert. Contextual Factors Influencing Perceived Importance and Trade-offs of Information Quality. *Commun. AIS*, 30:8, 2012. Cited on page 22.
- [62] C. Forgy. Rete: A Fast Algorithm for the Many Patterns/Many Objects Match Problem. *Artif. Intell.*, 19(1):17–37, 1982. Cited on page 30.
- [63] A. Formica. Finite Satisfiability of Integrity Constraints in Object-Oriented Database Schemas. *IEEE Trans. Knowl. Data Eng.*, 14(1):123–139, 2002. Cited on page 36.
- [64] C.J. Fox, A. Levitin, and T. Redman. The Notion of Data and its Quality Dimensions. *Information Processing & Management*, 30(1):9–20, 1994. Cited on page 20.
- [65] L. Frohofer, G. Glos, J. Osrael, and K.M. Göschka. Overview and Evaluation of Constraint Validation Approaches in Java. In *Proc. ICSE*, pages 313–322, 2007. Cited on page 34.
- [66] V. Ganti and A.D. Sarma. *Data Cleaning: A Practical Perspective*. Synthesis Lectures on Data Management. Morgan & Claypool Publishers, 2013. Cited on pages 2, 49, and 50.
- [67] K. Gemenetzi. Data Quality Assurance and Analysis for Food Science Data. <http://dx.doi.org/10.3929/ethz-a-010510135>, 2015. Cited on page 115.
- [68] J. Gray, T. Bapty, S. Neema, and J. Tuck. Handling Crosscutting Constraints in Domain-Specific Modeling. *Commun. ACM*, 44(10):87–93, 2001. Cited on page 37.

- [69] D.M. Groenewegen, Z. Hemel, L.C.L. Kats, and E. Visser. WebDSL: a Domain-Specific Language for Dynamic Web Applications. In *Proc. OOPSLA*, pages 779–780, 2008. Cited on page 34.
- [70] D.M. Groenewegen and E. Visser. Integration of Data Validation and User Interface Concerns in a DSL for Web Applications. *Software & Systems Modeling*, 12(1):35–52, 2013. Cited on pages 34 and 50.
- [71] F. Gullà, L. Cavalieri, S. Ceccacci, M. Germani, and R. Bevilacqua. Method to Design Adaptable and Adaptive User Interfaces. In *Proc. HCI International*, pages 19–24, 2015. Cited on page 134.
- [72] A. Hamie. Translating the Object Constraint Language into the Java Modelling Language. In *Proc. ACM SAC*, pages 1531–1535, 2004. Cited on page 27.
- [73] A. Hamie. Using Patterns to Map OCL Constraints to JML Specifications. In *Proc. MODELSWARD*, pages 35–48, 2014. Cited on page 39.
- [74] M. Hammer and D. McLeod. The Semantic Data Model: A Modelling Mechanism for Data Base Applications. In *Proc. SIGMOD*, pages 26–36, 1978. Cited on page 35.
- [75] K. Hanada, K. Okano, S. Kusumoto, and K. Miyazawa. Practical Application of a Translation Tool from UML/OCL to Java Skeleton with JML Annotation. In *Proc. ICEIS*, pages 389–394, 2012. Cited on page 39.
- [76] F. Heidenreich, C. Wende, and B. Demuth. A Framework for Generating Query Language Code from OCL Invariants. *Commun. EASST*, 9, 2008. Cited on page 39.
- [77] B. Heinrich, M. Kaiser, and M. Klier. How to Measure Data Quality? - A Metric-Based Approach. In *Proc. ICIS*, page 108, 2007. Cited on page 46.
- [78] M. Helfert and B. Heinrich. Analyzing Data Quality Investments in CRM: a Model-Based Approach. In *Proc. IQ*, pages 80–95, 2003. Cited on page 46.
- [79] Z. Hemel, D.M. Groenewegen, L.C.L. Kats, and E. Visser. Static Consistency Checking of Web Applications with WebDSL. *Symbolic Computation*, 46(2):150–182, 2011. Cited on page 34.
- [80] H. Hinrichs. *Datenqualitätsmanagement in Data-warehouse-Systemen*. PhD thesis, University of Oldenburg, Germany, 2002. Cited on page 46.
- [81] J.M. Holden, S.A. Bhagwat, and K.Y. Patterson. Development of a Multi-Nutrient Data Quality Evaluation System. *Food Composition and Analysis*, 15(4):339–348, 2002. Cited on page 51.
- [82] K.-T. Huang, Y.W. Lee, and R.Y. Wang. *Quality Information and Knowledge*. Prentice Hall PTR, 1999. Cited on page 47.

- [83] J. Hussain, W.A. Khan, M. Afzal, M. Hussain, B.H. Kang, and S. Lee. Adaptive User Interface and User Experience Based Authoring Tool for Recommendation Systems. In *Proc. UCAM'I*, pages 136–142, 2014. Cited on page 134.
- [84] H. Ibrahim. Maintaining the Integrity Constraints of Mobile Databases with Event-Condition-Action (ECA) Rules. In *Proc. MoMM*, pages 87–96, 2007. Cited on page 38.
- [85] C. Indiono, J. Mangler, W. Fdhila, and S. Rinderle-Ma. Rule-Based Runtime Monitoring of Instance-Spanning Constraints in Process-Aware Information Systems. In *Proc. Joint Conf. CoopIS, C&TC and ODBASE*, pages 381–399, 2016. Cited on page 38.
- [86] ISO/TS 8000-1:2011 Data quality – Part 1: Overview, 2011. http://www.iso.org/iso/catalogue_detail.htm?csnumber=50798. Cited on page 1.
- [87] ISO 9000:2005 Quality Management Systems – Fundamentals and Vocabulary, 2005. http://www.iso.org/iso/catalogue_detail?csnumber=42180. Cited on page 20.
- [88] ISO/IEC 15939:2007 Systems and Software Engineering – Measurement Process, 2007. <https://www.iso.org/obp/ui/#iso:std:iso-iec:15939:ed-2:v2:en>. Cited on page 46.
- [89] I. Jacobson and P.W. Ng. *Aspect-Oriented Software Development with Use Cases*. Addison Wesley Object Technology Series. Addison-Wesley, 2005. Cited on page 29.
- [90] V. Jayawardene, S.W. Sadiq, and M. Indulska. An Analysis of Data Quality Dimensions. Technical report, School of Information Technology and Electrical Engineering, The University of Queensland, Brisbane, Australia, October 2013. Cited on page 22.
- [91] T. Joachims, L.A. Granka, B. Pan, H. Hembrooke, and G. Gay. Accurately Interpreting Clickthrough Data as Implicit Feedback. In *Proc. SIGIR*, pages 154–161, 2005. Cited on page 48.
- [92] M. Jovanovic, D. Starcevic, and Z. Jovanovic. Bridging User Context and Design Models to Build Adaptive User Interfaces. In *Proc. HCSE*, pages 36–56, 2014. Cited on page 134.
- [93] J.M. Juran and A.B. Godfrey. *Quality Handbook*. McGraw-Hill, 1999. Cited on pages 20, 43, and 46.
- [94] B.K. Kahn, D.M. Strong, and R.Y. Wang. Information Quality Benchmarks: Product and Service Performance. *Commun. ACM*, 45(4):184–192, 2002. Cited on page 20.
- [95] S. Kandel, A. Paepcke, J.M. Hellerstein, and J. Heer. Wrangler: Interactive Visual Specification of Data Transformation Scripts. In *Proc. SIGCHI*, pages 3363–3372, 2011. Cited on page 50.

- [96] R. Kennard and J. Leaney. Towards a General Purpose Architecture for UI Generation. *Systems and Software*, 83(10):1896–1906, 2010. Cited on pages 38 and 135.
- [97] R. Kennard and R. Steele. Application of Software Mining to Automatic User Interface Generation. In *Proc. SoMeT*, pages 244–254, 2008. Cited on pages 38 and 135.
- [98] S. Kent. Model Driven Engineering. In *Proc. IFM*, pages 286–298, 2002. Cited on pages 7 and 27.
- [99] M.U. Khan, N. Arshad, M.Z. Iqbal, and H. Umar. AspectOCL: Extending OCL for Crosscutting Constraints. In *Proc. ECMFA*, pages 92–107, 2015. Cited on page 39.
- [100] A. Kleppe, J. Warmer, and W. Bast. *MDA Explained – the Model Driven Architecture: Practice and Promise*. Addison-Wesley, 2003. Cited on pages 14 and 27.
- [101] M. Klier. Metriken zur Bewertung der Datenqualität - Konzeption und praktischer Nutzen. *Informatik Spektrum*, 31(3):223–236, 2008. Cited on page 46.
- [102] S. Knight and J.M. Burn. Developing a Framework for Assessing Information Quality on the World Wide Web. *Informing Science*, 8:159–172, 2005. Cited on page 22.
- [103] K.C. Laudon. Data Quality and Due Process in Large Interorganizational Record Systems. *Commun. ACM*, 29(1):4–11, 1986. Cited on page 47.
- [104] G.T. Leavens, A.L. Baker, and C. Ruby. JML: A Notation for Detailed Design. In *Behavioral Specifications of Businesses and Systems*, pages 175–188. Springer, 1999. Cited on page 39.
- [105] Y.W. Lee, L. Pipino, J.D. Funk, and R.Y. Wang. *Journey to Data Quality*. MIT Press, 2006. Cited on pages 46 and 48.
- [106] Y.W. Lee, L. Pipino, D.M. Strong, and R.Y. Wang. Process-Embedded Data Integrity. *J. Database Manag.*, 15(1):87–103, 2004. Cited on page 43.
- [107] Y.W. Lee, D.M. Strong, B.K. Kahn, and R.Y. Wang. AIMQ: A Methodology for Information Quality Assessment. *Information & Management*, 40(2):133–146, 2002. Cited on pages 2 and 22.
- [108] S. Liu and K.J. Kochut. A Semantic End-to-End Process Constraint Modeling Framework. In *Proc. ICSC*, pages 203–210, 2014. Cited on page 22.
- [109] D. Loshin. *Enterprise Knowledge Management: The Data Quality Approach*. Morgan Kaufmann Pub., 2001. Cited on pages 1 and 46.
- [110] D. Loshin. Data Quality and Business Rules. In *Information and Database Quality*, pages 111–133. Springer, 2002. Cited on page 38.

- [111] D. Loshin. Rule-Based Data Quality. In *Proc. CIKM*, pages 614–616, 2002. Cited on page 38.
- [112] D. Loshin. Monitoring Data Quality Performance Using Data Quality Metrics. Technical report, Informatica Corporation, November 2006. Cited on page 22.
- [113] S.E. Madnick, R.Y. Wang, Y.W. Lee, and H. Zhu. Overview and Framework for Data and Information Quality Research. *Data and Information Quality*, 1(1), 2009. Cited on pages 1, 22, and 41.
- [114] S.E. Madnick and R.Y. Wang. Introduction to Total Data Quality Management (TDQM) Research Program. No. TDQM-92-01. Total Data Quality Management Program, MIT Sloan School of Management, Sloan, 1992. Cited on page 43.
- [115] J.I. Maletic and A. Marcus. Data Cleansing: Beyond Integrity Analysis. In *Proc. IQ*, pages 200–209, 2000. Cited on page 51.
- [116] A. Marotta and A.A. Vaisman. Rule-Based Multidimensional Data Quality Assessment Using Contexts. In *Proc. DaWaK*, pages 299–313, 2016. Cited on page 38.
- [117] D. McGilvray. *Executing Data Quality Projects: Ten Steps to Quality Data and Trusted Information*. Elsevier, 2008. Cited on page 20.
- [118] S.J. Mellor, K. Scott, A. Uhl, and D. Weise. Model-Driven Architecture. In *Advances in Object-Oriented Information Systems*, pages 290–297. Springer, 2002. Cited on pages 14 and 27.
- [119] B. Mesing, C. Constantinides, and W. Lohmann. Limes: An Aspect-Oriented Constraint Checking Language. In *Proc. ECMDA-FA*, pages 299–315, 2006. Cited on page 37.
- [120] B. Meyer. Applying ”design by contract”. *IEEE Computer*, 25(10):40–51, 1992. Cited on page 29.
- [121] G.A. Mihaila, L. Raschid, and M.-E. Vidal. Using Quality of Data Metadata for Source Selection and Ranking. In *Proc. WebDB*, pages 93–98, 2000. Cited on page 45.
- [122] R. Mock. Data Quality Analysis for Food Composition Databases. <http://dx.doi.org/10.3929/ethz-a-006660133>, 2011. Cited on page 59.
- [123] H.-T. Moges, K. Dejaeger, W. Lemahieu, and B. Baesens. A Multidimensional Analysis of Data Quality for Credit Risk Management: New Insights and Challenges. *Information & Management*, 50(1):43–58, 2013. Cited on pages 1, 20, and 21.
- [124] H.-T. Moges, V. Van Vlasselaer, W. Lemahieu, and B. Baesens. Determining the Use of Data Quality Metadata (DQM) for Decision Making Purposes and its Impact on Decision Outcomes - An Exploratory Study. *Decision Support Systems*, 83:32–46, 2016. Cited on page 45.

- [125] R. Moiseev, S. Hayashi, and M. Saeki. Generating Assertion Code from OCL: A Transformational Approach Based on Similarities of Implementation Languages. In *Proc. MODELS*, pages 650–664, 2009. Cited on page 40.
- [126] R. Moiseev, S. Hayashi, and M. Saeki. Using Hierarchical Transformation to Generate Assertion Code from OCL Constraints. *IEICE Transactions*, 94-D(3):612–621, 2011. Cited on page 40.
- [127] J. Mylopoulos, L. Chung, S. Liao, H. Wang, and E. Yu. Exploring Alternatives During Requirements Analysis. *IEEE Software*, 18(1):92–96, 2001. Cited on page 15.
- [128] J. Mylopoulos, L. Chung, and E. Yu. From Object-Oriented to Goal-Oriented Requirements Analysis. *Commun. ACM*, 42(1):31–37, 1999. Cited on page 15.
- [129] H. Nakamura, Y. Gao, H. Gao, H. Zhang, A. Kiyohiro, and T. Mine. Adaptive User Interface for Personalized Transportation Guidance System. In *Tourism Informatics*, pages 119–134. Springer, 2015. Cited on page 134.
- [130] B. Narasimhan, S.B. Navathe, and S. Jayaraman. *On Mapping ER and Relational Models into OO Schemas*, pages 402–413. Springer, 1994. Cited on page 35.
- [131] F. Naumann, U. Leser, and J.C. Freytag. Quality-Driven Integration of Heterogeneous Information Systems. In *Proc. VLDB*, pages 447–458, 1999. Cited on page 45.
- [132] F. Naumann and C. Rolker. Assessment Methods for Information Quality Criteria. In *Proc. IQ*, pages 148–162, 2000. Cited on pages 20 and 46.
- [133] M.C. Norrie. An Extended Entity-Relationship Approach to Data Management in Object-Oriented Systems. In *Proc. ER*, pages 390–401, 1993. Cited on page 35.
- [134] M.C. Norrie and A. Würigler. OM Framework for Object-Oriented Data Management. *INFORMATIK Journal of the Swiss Informaticians Society*, 3, 1997. Cited on page 23.
- [135] H. Oakasha, S. Conrad, and G. Saake. Consistency Management in Object-Oriented Databases. *Concurrency and Computation: Practice and Experience*, 13(11):955–985, 2001. Cited on page 33.
- [136] N. Obrenovic, A. Popovic, S. Aleksic, and I. Lukovic. Transformations of Check Constraint PIM Specifications. *Computing and Informatics*, 31(5):1045–1079, 2012. Cited on page 39.
- [137] OCLinEcore. <http://help.eclipse.org/neon/index.jsp?topic=%2Forg.eclipse.oc1.doc%2Fhelp%2FOCLinEcore.html>, 2017. Accessed: 2017-01-19. Cited on page 124.
- [138] X. Oriol and E. Teniente. Incremental Checking of OCL Constraints with Aggregates Through SQL. In *Proc. ER*, pages 199–213, 2015. Cited on page 39.

- [139] L. Orman, V.C. Storey, and R.Y. Wang. Systems Approaches to Improving Data Quality. In *Proc. IQ*, pages 117–126, 1996. Cited on page 45.
- [140] R.K. Pandey. Object Constraint Language (OCL): Past, Present and Future. *SIGSOFT Software Engineering Notes*, 36(1):1–4, 2011. Cited on page 27.
- [141] N.W. Paton and O. Díaz. Active Database Systems. *ACM Comput. Surv.*, 31(1):63–103, 1999. Cited on page 36.
- [142] J. Peckham and F.J. Maryanski. Semantic Data Models. *ACM Comput. Surv.*, 20(3):153–189, 1988. Cited on page 35.
- [143] L. Pipino. Information Quality Assessment. In *Encyclopedia of Database Systems*, pages 1512–1515. Springer US, 2009. Cited on pages 44 and 48.
- [144] L. Pipino, Y.W. Lee, and R.Y. Wang. Data Quality Assessment. *Commun. ACM*, 45(4):211–218, 2002. Cited on pages 2, 21, 22, 46, 47, 48, and 167.
- [145] L. Pipino, R. Wang, D. Kopsco, and W. Rybolt. Developing Measurement Scales for Data-Quality Dimensions. In *Information Quality*, pages 37–51. Routledge, 2005. Cited on page 46.
- [146] K. Presser. *A Requirement-Oriented Data Quality Model and Framework of a Food Composition Database System*. PhD thesis, ETH Zurich, 2012. Cited on pages 5, 19, 42, 51, 52, 56, 58, 61, and 63.
- [147] K. Presser, D. Weber, and M.C. Norrie. A Study of Data Quality Requirements for Empirical Data in the Food Sciences. In *Proc. ECIS*, 2014. Cited on pages 21, 51, and 58.
- [148] K. Presser, D. Weber, and M.C. Norrie. FoodCASE: A System to Manage Food Composition, Consumption and TDS Data. *Food Chemistry*, 238:166–172, 2018. Cited on page 5.
- [149] R.J Price and G.G. Shanks. A Semiotic Information Quality Framework: Development and Comparative Analysis. *Information Technology*, 20(2):88–102, 2005. Cited on page 22.
- [150] R.J Price and G.G. Shanks. The Impact of Data Quality Tags on Decision-Making Outcomes and Process. *J. AIS*, 12(4):1, 2011. Cited on page 45.
- [151] O. Probst. Investigating a Constraint-Based Approach to Data Quality in Information Systems. <http://dx.doi.org/10.3929/ethz-a-009980065>, 2013. Cited on pages 73 and 76.
- [152] R. Rahm and H.H. Do. Data Cleaning: Problems and Current Approaches. *IEEE Data Eng. Bull.*, 23(4):3–13, 2000. Cited on page 20.
- [153] D. Raneburger, R. Popp, S. Kavaldjian, H. Kaindl, and J. Falb. *Optimized GUI Generation for Small Screens*, pages 107–122. Springer Berlin Heidelberg, 2011. Cited on page 134.

- [154] B.R. Rao. *Object-Oriented Databases: Technology, Applications, and Products*. McGraw-Hill, 1994. Cited on page 36.
- [155] S. Reddy, J. Burke, D. Estrin, M.H. Hansen, and M.B. Srivastava. A Framework for Data Quality and Feedback in Participatory Sensing. In *Proc. SenSys*, pages 417–418, 2007. Cited on pages 52 and 134.
- [156] T.C. Redman. *Data Quality for the Information Age*. Artech House, 1997. Cited on page 20.
- [157] T.C. Redman. *Data Quality: The Field Guide*. Digital Press, 2001. Cited on page 46.
- [158] H. Rocha and M.T. Valente. How Annotations are Used in Java: An Empirical Study. In *Proc. SEKE*, pages 426–431, 2011. Cited on page 26.
- [159] D. Rodríguez, E. García, S. Sánchez, and C. Solano. Defining Software Process Model Constraints with Rules Using OWL and SWRL. *Software Engineering and Knowledge Engineering*, 20(4):533–548, 2010. Cited on page 20.
- [160] D. Roscher, G. Lehmann, V. Schwartz, M. Blumendorf, and S. Albayrak. *Dynamic Distribution and Layouting of Model-Based User Interfaces in Smart Environments*, pages 171–197. Springer Berlin Heidelberg, 2011. Cited on page 134.
- [161] R.G. Ross. Are Integrity Constraints Business Rules? Not! *Business Rules Journal*, 8(3), 2007. Published online, URL: <http://www.brcommunity.com/b335.php>, Accessed: 2017-01-17. Cited on page 24.
- [162] J. Rothenberg. Metadata to Support Data Quality and Longevity. In *Proc. IEEE Metadata Conf.*, 1996. Cited on page 45.
- [163] S.W. Sadiq. *Handbook of Data Quality: Research and Practice*. Springer, 2013. Cited on pages 20 and 42.
- [164] S.W. Sadiq, N.K. Yeganeh, and M. Indulska. 20 Years of Data Quality Research: Themes, Trends and Synergies. In *Proc. ADC*, pages 153–162, 2011. Cited on pages 3 and 22.
- [165] S.W. Sadiq, N.K. Yeganeh, and M. Indulska. Cross-Disciplinary Collaborations in Data Quality Research. In *Proc. ECIS*, 2011. Cited on page 2.
- [166] S. Salvini, M. Oseredczuk, M. Roe, A. Møller, and J. Ireland. Guidelines for Quality Index Attribution to Original Data from Scientific Literature or Reports for EuroFIR Data Interchange. Cited on page 51.
- [167] C. Scaffidi, B.A. Myers, and M. Shaw. Topes: Reusable Abstractions for Validating Data. In *Proc. ICSE*, pages 1–10, 2008. Cited on page 35.
- [168] J. Sebek and T. Cerný. AOP-Based Approach for Local Data Management in Adaptive Interfaces. In *Proc. ICITCS*, pages 1–5, 2016. Cited on pages 38 and 135.

- [169] G. Shankaranarayanan and Y. Cai. Supporting Data Quality Management in Decision-Making. *Decision Support Systems*, 42(1):302–317, 2006. Cited on page 21.
- [170] H. Shimba, K. Hanada, K. Okano, and S. Kusumoto. Bidirectional Translation between OCL and JML for Round-Trip Engineering. In *Proc. APSEC*, pages 49–54, 2013. Cited on page 40.
- [171] H.M. Sneed, B. Demuth, and B. Freitag. A Process for Assessing Data Quality. In *Proc. ICSTW*, pages 114–119, 2013. Cited on page 22.
- [172] D. Steinberg, F. Budinsky, M. Paternostro, and E. Merks. *EMF: Eclipse Modeling Framework 2.0*. Addison-Wesley Professional, 2nd edition, 2009. Cited on page 125.
- [173] J. Szymanek. Achieving Unified Data Quality Representation by Constraints Transformation. <http://dx.doi.org/10.3929/ethz-a-010510131>, 2015. Cited on pages 92, 126, and 128.
- [174] J. Teboul. *Managing Quality Dynamics*. Prentice Hall, 1991. Cited on page 46.
- [175] D.C. Tsichritzis and F.H. Lochovsky. *Data Models*. Prentice Hall Professional Technical Reference, 1982. Cited on page 23.
- [176] C. Türker and M. Gertz. Semantic Integrity Support in SQL:1999 and Commercial (Object-)Relational Database Management Systems. *VLDB J.*, 10(4):241–269, 2001. Cited on pages 24 and 33.
- [177] S.D. Urban and M. Desiderio. CONTEXT:A CONstrainT EXplanation Tool. *Data & Knowledge Engineering*, 8(2):153–183, 1992. Cited on page 36.
- [178] S.D. Urban and A.M. Wang. The Design of a Constraint/Rule Language for an Object-Oriented Data Model. *Systems and Software*, 28(3):203–224, 1995. Cited on page 28.
- [179] P. Vassiliadis, M. Bouzeghoub, and C. Quix. Towards Quality-Oriented Data Warehouse Usage and Evolution. *Information Systems*, 25(2):89–115, 2000. Cited on page 21.
- [180] B. Verheecke and R. Van Der Straeten. Specifying and Implementing the Operational Use of Constraints in Object-Oriented Applications. In *Proc. CRPIT*, pages 23–32, 2002. Cited on page 34.
- [181] I.E. Vermeulen and D. Seegers. Tried and Tested: The Impact of Online Hotel Reviews on Consumer Consideration. *Tourism Management*, 30(1):123–127, 2009. Cited on page 48.
- [182] Y. Wand and R.Y. Wang. Anchoring Data Quality Dimensions in Ontological Foundations. *Commun. ACM*, 39(11):86–95, 1996. Cited on pages 1, 20, 21, 41, and 69.

- [183] Q. Wang and A.P. Mathur. Interceptor Based Constraint Violation Detection. In *Proc. ECBS*, pages 457–464, 2005. Cited on page 34.
- [184] R.Y. Wang. A Product Perspective on Total Data Quality Management. *Commun. ACM*, 41(2):58–65, 1998. Cited on pages 45 and 47.
- [185] R.Y. Wang, M.P. Reddy, and H.B. Kon. Toward Quality Data: An Attribute-Based Approach. *Decision Support Systems*, 13(3-4):349–372, 1995. Cited on pages 1, 41, and 45.
- [186] R.Y. Wang, V.C. Storey, and C.P. Firth. A Framework for Analysis of Data Quality Research. *IEEE Trans. Knowl. Data Eng.*, 7(4):623–640, 1995. Cited on page 41.
- [187] R.Y. Wang and D.M. Strong. Beyond Accuracy: What Data Quality Means to Data Consumers. *Management Information Systems*, 12(4):5–33, 1996. Cited on pages 1, 20, 22, 45, and 69.
- [188] T. Wang and J. Wang. Visualisation of Spatial Data Quality for Internet and Mobile GIS Applications. *Spatial Science*, 49(1):97–107, 2004. Cited on page 52.
- [189] Y.R. Wang and S.E. Madnick. A Polygen Model for Heterogeneous Database Systems: The Source Tagging Perspective. In *Proc. VLDB*, pages 519–538, 1990. Cited on page 2.
- [190] J. Warmer and A. Kleppe. *The Object Constraint Language: Getting Your Models Ready for MDA*. Addison-Wesley Longman Publishing Co., Inc., 2nd edition, 2003. Cited on page 27.
- [191] D. Weber, S. Leone, and M.C. Norrie. Constraint-Based Data Quality Management Framework for Object Databases. In *Proc. ECIS*, 2013. Cited on pages 22, 35, and 65.
- [192] D. Weber, K. Presser, and M.C. Norrie. How to Give Feedback on Data Quality: A Study in the Food Sciences. In *Proc. ECIS*, 2015. Cited on pages 58 and 62.
- [193] D. Weber, J. Szymanek, and M.C. Norrie. UnifiedOCL: Achieving System-Wide Constraint Representations. In *Proc. ER*, 2016. Cited on pages 35 and 121.
- [194] C. Wilke. Java Code Generation for Dresden OCL2 for Eclipse. Technische Universität Dresden, Germany, 2009. Cited on page 39.
- [195] M. Wimmer and L. Burgueño. Testing M2T/T2M Transformations. In *Proc. MODELS*, pages 203–219, 2013. Cited on page 126.
- [196] G. Witt. *Writing Effective Business Rules*. Elsevier, 2012. Cited on page 22.
- [197] A.P. Würzler. *OMS Development Framework: Rapid Prototyping for Object-Oriented Databases*. PhD thesis, ETH Zurich, 2000. Cited on pages 66 and 69.

-
- [198] Y. Xiao, L.Y.Y. Lu, J.S. Liu, and Z. Zhou. Knowledge Diffusion Path Analysis of Data Quality Literature: A Main Path Analysis. *Informetrics*, 8(3):594–605, 2014. Cited on pages 1, 2, and 22.
- [199] N.K. Yeganeh, S.W. Sadiq, and M.A. Sharaf. A Framework for Data Quality Aware Query Systems. *Information Systems*, 46:24–44, 2014. Cited on page 1.
- [200] B. Zaqaibeh and E. Al Daoud. The Constraints of Object-Oriented Databases. *Open Problems in Computer Science and Mathematics*, 1:11–17, 2008. Cited on pages 31 and 36.
- [201] B. Zaqaibeh, H. Ibrahim, A. Mamat, and M.N. Sulaiman. Enforcing User-Defined Constraints during the Run-Time in OODB. *Int. Arab J. Inf. Technol.*, 5(4):341–348, 2008. Cited on pages 36 and 53.
- [202] B.M. Zaqaibeh, F.A. Albalas, and W.A. Awajan. Designing a New Assertion Constraints Model for Mobile Databases. *iJIM*, 6(2):39–46, 2012. Cited on page 53.
- [203] R. Zhang, V. Jayawardene, M. Indulska, S.W. Sadiq, and X. Zhou. A Data Driven Approach for Discovering Data Quality Requirements. In *Proc. ICIS*, 2014. Cited on page 1.

Acronyms

- ACR** Abstract Constraint Representation. 127, 239
- AMIC** Assertion Model of Integrity Constraints. 36, 239
- AOIM** Application-Oriented Integrity Maintenance. 31, 239
- AOP** Aspect-Oriented Programming. 10, 14, 28–30, 37–39, 134, 135, 152, 155, 160, 161, 163, 177, 186, 192, 194, 196, 197, 239
- API** Application Programming Interface. 30, 79, 80, 82, 90, 95, 141, 142, 239
- AST** Abstract Syntax Tree. 40, 126, 239
- ATL** ATL Transformation Language. 40, 239
- BRE** Business Rules Engine. 30, 31, 38, 239
- BRMS** Business Rules Management System. 30, 31, 38, 239
- BV** Bean Validation. 6, 7, 10, 14, 26, 27, 29, 30, 33, 36, 37, 39, 41, 56, 57, 64, 68–73, 75–83, 87, 89, 99, 103, 104, 114, 115, 118–121, 125–127, 130, 137, 142, 143, 151, 159, 187, 191–193, 196, 207, 208, 210, 211, 239
- CDM** Constraint-Driven Modelling. 39, 239
- CFD** Conditional Functional Dependency. 42, 239
- CIM** Centralised Integrity Maintenance. 36, 239
- CMS** Content Management System. 239
- ConfC** Confidence Code. 51, 239
- CSS** Cascading Style Sheets. 40, 142, 239
- DbC** Design by Contract. 26, 29, 30, 34, 239
- DBMS** Database Management System. 23, 42, 239
- DDL** Data Definition Language. 24, 35, 39, 66, 69, 239
- DQ** Data Quality. 1–3, 5, 7–22, 32, 36, 38, 41–49, 51–53, 55–65, 67, 69, 70, 72, 73, 78–80, 82, 83, 85, 87–99, 101–111, 113, 116–119, 121, 123, 125, 126, 130, 131, 133–136, 140, 141, 145–149, 151–156, 158–160, 162–169, 171–187, 189–198, 239

- DQAF** Data Quality Analysis Feedback. 17, 18, 49–51, 53, 55, 56, 58–61, 64, 99, 103, 108, 110, 111, 117, 152, 190, 192, 194, 196, 197, 239
- DQI** Data Quality Indicator. 15, 16, 56, 57, 70, 72, 88, 102–106, 108, 117, 118, 192, 239
- DQIF** Data Quality Input Feedback. 17, 18, 49–51, 53, 55, 56, 58, 59, 99, 103, 108, 109, 111, 117, 152, 178, 190, 192–194, 196, 197, 239
- DQMF** Data Quality Management Framework. 10–13, 17–19, 22, 41, 48, 52, 53, 55–57, 69, 71, 76, 79, 80, 88, 99–104, 114, 116, 119, 151–154, 156, 159–161, 167, 175–177, 180, 183, 186, 187, 190–196, 211, 223, 239
- DQMIM** Data Quality Measurement Information Model. 45, 46, 239
- DSL** Domain-Specific Language. 13, 14, 39, 119, 121, 192, 193, 239
- ECA** Event Condition Action. 14, 27, 35, 36, 38, 56, 57, 64, 65, 68, 82, 86, 87, 239
- EJB** Enterprise JavaBean. 6, 114, 115, 239
- EMF** Eclipse Modeling Framework. 130, 239
- ERM** Entity-Relationship-Model. 35, 58, 239
- EuroFIR** European Food Information Resource. 51, 239
- FDTP** EuroFIR Food Data Transport Package. 239
- FSVO** Federal Food Safety and Veterinary Office. 3, 239
- GIS** Geographic Information System. 52, 53, 239
- GPS** Global Positioning System. 135, 239
- GUI** Graphical User Interface. 37, 62, 64, 94, 95, 115, 119, 133, 136, 139, 140, 142, 144, 145, 149, 156, 160, 176, 177, 180, 184, 185, 192, 193, 196, 197, 239
- GWT** Google Web Toolkit. 37, 239
- HC** Hard Constraint. 56–60, 62–69, 72, 73, 75, 76, 78, 82, 83, 104, 239
- HV** Hibernate Validator. 114, 115, 120, 125, 207, 209–211, 239
- IDE** Integrated Development Environments. 130, 239
- IQ** Information Quality. 19, 239
- IQA** Information Quality Assessment. 48, 239
- JAR** Java Archive. 95, 239

- JDO** Java Data Objects. 79–82, 85–87, 239
- JML** Java Modeling Language. 26, 30, 39, 40, 130, 239
- JPA** Java Persistence API. 37, 71, 75–77, 80, 239
- JPQL** Java Persistence Query Language. 239
- JSF** JavaServer Faces. 37, 80, 87, 239
- JSON** JavaScript Object Notation. 39–41, 95–97, 156–158, 176, 177, 179, 184, 185, 194, 239
- JSR** Java Specification Request. 36, 37, 80, 239
- M2T** model-to-text. 126, 239
- MDA** Model-Driven Architecture. 14, 27, 39, 40, 119, 190, 192, 193, 239
- MDD** Model-Driven Development. 14, 135, 239
- MDE** Model-Driven Engineering. 27, 122, 239
- MDIE** Model-Driven Integrity Engineering. 239
- MDSD** Model-Driven Software Development. 7, 122, 239
- MOF** Meta-Object Facility. 27, 239
- MVP** Model-View-Presenter. 94, 95, 239
- OALIC** Object Assertion Language for Integrity Constraints. 53, 239
- OCL** Object Constraint Language. 10, 14, 27–29, 34, 35, 37–40, 56, 57, 79–85, 87, 119–128, 130, 192, 193, 239
- ODB** Object Database. 13, 56, 239
- OMG** Object Management Group. 20, 27, 80, 127, 239
- OO** object-oriented. 12, 13, 160, 239
- OODBMS** Object Oriented Database Management System. 24, 36, 53, 239
- OSGi** Open Service Gateway Initiative. 130, 239
- PCL** Process Constraint Language. 22, 239
- PIM** Platform-Independent Model. 27, 122, 239
- POJO** Plain Old Java Object. 80–82, 84, 87, 239
- ProContO** Process Constraint Ontology. 22, 239

- PSM** Platform-Specific Model. 27, 122, 239
- QI** Quality Indicator. 105, 107, 110, 111, 115, 117, 239
- QIX** Quality Index. 51, 239
- QVT** Query/View/Transformation. 40, 239
- RCP** Eclipse Rich Client Platform. 37, 239
- RDB** Relational Database. 13, 25, 239
- RDBMS** Relational Database Management System. 24, 25, 35, 239
- READ** Rich Entity Aspect/Audit Design. 134, 239
- RMI** Remote Method Invocation. 6, 239
- RODQ** Requirement-Oriented Data Quality. 19, 58, 239
- RTE** Round Trip Engineering. 130, 193, 239
- SC** Soft Constraint. 56–60, 62–69, 72, 73, 76, 78, 79, 82, 83, 87, 104, 239
- SQL** Structured Query Language. 6, 7, 14, 25, 27, 33, 39, 41, 89, 91, 118, 120–123, 125–130, 210, 239
- SWT** Standard Widget Toolkit. 37, 239
- TDQM** Total Data Quality Management. 43, 239
- TDS** Total Diet Study. 3–5, 8, 133, 134, 141, 156, 180, 182, 184, 192, 239
- TDS-Exposure** Total Diet Study Exposure. 5, 8, 16, 17, 62, 64, 102, 116, 141, 190, 192, 193, 195, 199, 203, 239
- TQM** Total Quality Management. 43, 239
- TRP** Target Representation Producer. 127, 239
- UI** User Interface. 2, 38, 49, 50, 60, 134–136, 239
- UML** Unified Modeling Language. 14, 27, 28, 34, 35, 37, 39, 58, 67, 77, 81, 122, 125–131, 193, 239
- USDA** US Department of Agriculture. 51, 63, 239
- XMI** XML Metadata Interchange. 127, 239
- XML** Extensible Markup Language. 37, 41, 45, 80, 82, 87, 122, 127, 128, 239

Curriculum vitae

Particulars

Name	David Weber
Date of Birth	April 29, 1976
Birthplace	Zürich, Switzerland
Citizenship	Küsnacht (ZH), Switzerland and Germany

Education

2011–2017	Doctoral studies (PhD student) and research / teaching assistant in the Global Information Systems group, ETH Zürich, Switzerland
2006	Master of science in computer science with concentration in information systems / business informatics (Dipl. Inform.), University of Zürich, Switzerland
2000–2005	Study of computer science with concentration in information systems / business informatics, University of Zürich, Switzerland
1997–1999	Study of mechanical engineering, ETH Zürich, Switzerland
1997	Matura type E (economics), Wirtschaftsgymnasium Hottingen grammar school, Zürich, Switzerland
1992–1997	Wirtschaftsgymnasium Hottingen grammar school, Zürich, Switzerland
1989–1992	Realgymnasium Rämibühl grammar school, Zürich, Switzerland
1983–1989	Primary school Itschnach, Küsnacht, Switzerland

Work experience

- 2012–2016 Software engineer ('FoodCASE' food science data management system) at ETH Zürich, Switzerland
- 2006–2010 Software engineer at SIX Card Solutions AG, Zürich, Switzerland
- 2006 Freelance software engineer ('Ecosolvent' life-cycle assessment tool) at ETH Zürich, Switzerland
- 2005 Internship as software engineer at Telekurs Card Solutions AG, Zürich, Switzerland
- 2001–2005 Part-time employment as software engineer at WeCoMa, Küsnacht, Switzerland
- 2000 Internship as mechanical engineer at ABB Turbo Systems AG, Baden, Switzerland
- 1997 Internship / workshop training at Bucher AG, Fällanden, Switzerland
- 1995–1999 Part-time employment for data processing at Regional Compact Car AG, Zürich, Switzerland
- 1992–1994 Part-time employment for data processing at Überseebank AG, Zürich, Switzerland

Publications

- 2018-01 'FoodCASE: A System to Manage Food Composition, Consumption and TDS Data' by K. Presser, D. Weber, M.C. Norrie in *Food Chemistry*, January 2018
- 2016-11 'UnifiedOCL: Achieving System-Wide Constraint Representations' by D. Weber, J. Szymanek, M.C. Norrie in *Proc. 35th Intl. Conf. on Conceptual Modeling (ER)*, Gifu, Japan, November 2016
- 2016-10 'An Assessment Impact Classification of Data Quality Requirements in Food Composition Database Systems' by K. Presser, D. Weber, M.C. Norrie in *Proc. 2nd IMEKOFOODS Intl. Conf.*, Benevento, Italy, October 2016
- 2016-02 'A Scope Classification of Data Quality Requirements for Food Composition Data' by K. Presser, H. Hinterberger, D. Weber, M.C. Norrie in *Food Chemistry*, February 2016
- 2015-05 'How to Give Feedback on Data Quality: A Study in the Food Sciences' by D. Weber, K. Presser, M.C. Norrie in *Proc. 23rd European Conf. on Information Systems (ECIS)*, Munster, Germany, May 2015
- 2014-06 'A Study of Data Quality Requirements for Empirical Data in the Food Sciences' by K. Presser, D. Weber, M.C. Norrie in *Proc. 22nd European Conf. on Information Systems (ECIS)*, Tel Aviv, Israel, June 2014
- 2013-06 'Constraint-Based Data Quality Management Framework for Object Databases' by D. Weber, S. Leone, M.C. Norrie in *Proc. 21st European Conf. on Information Systems (ECIS)*, Utrecht, The Netherlands, June 2013