Data Quality Assurance and Analysis for Food Science Data

Data Quality Assurance and Analysis for Food Science Data

Master Thesis

Konstantina Gemenetzi
<kgemenet@student.ethz.ch>

Prof. Dr. Moira C. Norrie
David Weber

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

Monday 6th April, 2015

ETH
Eidgenössische Technische Hochschule Zürich
Swiss Federal Institute of Technology Zurich

globis

Abstract

Data is of paramount significance in every information system. Even though data quality might have a different meaning for different users and applications, it is always recognized that the quality of the data in an information system can have a large impact to the quality and value of every other aspect of the system.

In this master thesis, we model and quantify data quality in information systems with the use of *constraints*. We propose a novel concept for a *highly flexible and configurable data quality management framework* that can easily be adapted to the needs of different users and applications. For this purpose, we introduce the notion of *contracts*, which are simple, interchangeable files that allow each user and application of the system to specify an individual importance weight for each quality-related constraint, thus enabling the creation of unique *quality profiles*.

Our framework allows the user to validate data during input, as well as analyze, quantify and visualize the overall quality of the data at any time. For both the *data input validation* and the *data quality analysis*, the user is able to select the data entities and constraint groups that should be validated and analyzed, as well as the contract (quality profile) that should be used for the validation and analysis. To measure quality, we defined a novel *quality calculation scheme and scale*. For the definition, validation and management of the constraints in a unified way, we use the bean validation[1] specification.

For evaluating our approach, we implemented our concept within the FoodCASE food science data management system. FoodCASE is used for the administration of nutrients for food composition studies and contaminants for Total Diet Studies (TDS). Currently, FoodCASE is used to manage the data of the Swiss Food Composition Database as well as the data of TDS-Exposure[2] studies for several countries across Europe. Our data quality management framework and its FoodCASE implementation prototype were presented to the participants of the TDS-Exposure workshop that took place during the third TDS-Exposure General Assembly in February 2015 and received positive feedback, which is also discussed in this work.

[1] http://beanvalidation.org/
[2] http://www.tds-exposure.eu/

Contents

1 Introduction

In this chapter, we present the motivation behind this work and provide a short outline of the structure of this report.

1.1 Motivation

The main goal of any information system can be abstracted to the extraction, collection, processing, analysis and distribution of useful information derived from data. Therefore, without correct, consistent and overall appropriate data that is fit to its intended use, the information system cannot properly serve its purpose.

Despite the indisputable importance of data quality in any type of information system, the definition of a data quality modeling and measuring framework that can be applied to any type of system has not yet been proposed. In this thesis, we attempt to model the data quality of a system with the use of *constraints*, which are the rules that the data has to follow in order to be considered of high quality. Thus, constraints can be regarded as an indicator of quality, as the more constraints are satisfied the higher the quality of the system is.

A potential drawback of a constraint-based data quality framework approach in large information systems is the fact that constraints are often defined in several different layers of the information system, while their associated validators might be distributed in the system's interface, application and database levels. This can often lead to redundancies and inconsistencies and result in maintenance and performance issues. For this reason, a central motivation of this thesis is to investigate ways in which quality constraints and validation checks can be managed in a *unified way*.

Another important focus of this work is the creation of a data quality management framework that is highly *flexible and configurable*. Indeed, quality might entail different aspects of the system for different users or applications. On the other hand, even for the same user, different constraints are most likely of different importance. As the end users of a system are usually different than its developers, it is crucial to enable authorized users to tune these quality aspects at any time via an intuitive and easy-to-use interface. For this purpose, we introduce the concept of *contracts*, which allow authorized users to decide on the importance of each constraint for each application as well as deactivate constraints that are not deemed relevant.

In summary, this thesis focuses on the creation of a *constraint-based data quality modeling and calculation concept*, which can be applied to any type of information system and enables managing all constraints in a unified way. This concept is used as a base to create a *highly flexible data quality management framework* that allows the users of the information system to configure the details of every quality-related aspect according to their needs. The data quality management framework that we introduce consists of an input validation module, a data quality analysis module and an administrative tools module.

To evaluate our approach, we implemented our data quality management framework for the FoodCASE food science data management system. 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 work.

1.2 Structure

Chapter 2 contains the fundamental background that is needed to place this thesis into context, starting with a description of the notion of data quality and other related concepts. It continues with an overview of some examples of existing data quality management frameworks, both conceptual as well as industrially applied. Next, we describe FoodCASE, which is the food science data management system within which we implemented our data quality management framework.

Chapter 3 presents the main contribution of this thesis, which includes a novel concept for a highly flexible data quality management framework. It also describes the three composing parts of this framework, which are a data quality analysis module, a data input validation module as well as an administrative tools module for configuring the different aspects of the framework.

Chapter 4 deals with the details of the implementation of our framework within FoodCASE.

Chapter 5 concentrates on a comparison of our framework with the previously existing food composition data quality management framework that was implemented within FoodCASE.

In chapter 6 we display the results of a first evaluation of our system by the participants of the TDS-Exposure workshop, which took place in Zurich in February 2015.

Chapter 7 discusses some of the possible ways in which the current framework could be extended in order to cover additional requirements of the users.

Finally, this report concludes with chapter 8, which summarizes the work and contributions of this thesis and discusses ways in which it could be continued and improved.

2
Background

This chapter contains a summary of the results of the background research related to this thesis. It begins with a description of the concept of data quality, as well as an outline of the dimensions that are most commonly used to characterize it. This is followed by a definition of data constraints and their relation to data quality, which is one of the central themes in our work.

The second part of this chapter consists of an overview of some existing data quality management frameworks (DQMFs), which originate from both the research and the industry realms.

The chapter continues with a description of FoodCASE, the food science data management system within which our data quality management framework is implemented, and concludes with a summary of the open related issues that are addressed by this thesis.

2.1 Data Quality

Data quality is a complex concept that can bear different meanings depending on the context in which it is being used. Therefore, it is nearly impossible to come up with a single, generic definition to describe it. However, as Juran and Blanton [9] note, when it comes to quality management, the two most critical aspects of quality are "the features of products which meet customer needs and thereby provide customer satisfaction" as well as the products' "freedom from deficiencies", with data-derived information being the product in our case. These two aspects comprise what is often characterized as the data's *fitness of use*, which is commonly used as an example of what data quality is about.

2.1.1 Data versus Information

Before further discussing data quality, it is very useful to make a distinction between data and information. The terms *data* and *information* are closely related but not equivalent. According to Beynon-Davies [3], data is a string of symbols, while information occurs when the symbols are used to refer to something. When it comes to information systems, data can be seen as the 'raw material' that the system uses in order to derive its product, which is information of some kind.

2.1.2 Data Quality Dimensions

In [2], data quality is defined as a multifaceted concept, which means that different dimensions occur in its definition. Indeed, most studies agree about the multi-dimensional nature of data quality [11], [14], [15]. Recognizing these dimensions can be easier in some cases than others and might depend on the types of data and information systems that are involved.

Each dimension represents a different aspect of data quality. Batini and Scannapieco [2] make a distinction between dimensions that refer to the data's *extension*, i.e. their values, or their *intension*, i.e. their schema. In this work, we focus mostly on the data extension quality. Next, we will present some of the most commonly referred-to data quality dimensions.

Accuracy

In [2] *accuracy* is defined as "the degree of closeness between a value `v` and a value `v'`, considered as the correct representation of the real-life phenomenon that `v` aims to represent". Accuracy can be distinguished between syntactic accuracy and semantic accuracy.

Syntactic accuracy is "the closeness of a value `v` to the elements of the corresponding definition domain `D`". In other words, syntactic accuracy is not concerned with whether `v` is the same as its real value `v'`, but rather with whether the value of `v` is any of the existing values of the domain `D`. For example, let's consider the months of the year. `Movember` in that case would be syntactically inaccurate, as (officially) there is no such month in the year. However, `December` is syntactically accurate, even if `v'` is actually `January`.

On the other hand, *semantic* accuracy is "the closeness of the value `v` to the true value `v'`". So, to continue with the previous example, if we were looking for the first month of the year, the only semantically accurate value would be `January`. In this case, `March` would be syntactically accurate, but semantically inaccurate.

Completeness

Wang and Strong [15] define *completeness* as "the extent to which data are of sufficient breadth, depth, and scope for the task at hand". Pipino et al. [11] recognize that completeness

can have different aspects and therefore make a distinction between schema, column and population completeness.

Schema completeness is defined as the degree to which entities and attributes are not missing from the schema. *Column completeness* deals with the amount of missing values in a particular table column (for relational data). Finally, *population completeness* refers to the amount of missing values in relation to a reference population.

Consistency

Consistency deals with "the violation of semantic rules defined over (a set of) data items, where items can be tuples of relational tables or records in a file" [2]. As with the previously described dimensions, consistency can also have different aspects. A well-known form of consistency in the relational database world is what is known as *referential integrity*: if a table field is declared as a *foreign* key, it can either contain a *null* value, or an existing and valid *primary key* from a parent table. If the above condition is not satisfied, then the consistency of the data is compromised.

Time-related dimensions

When regarding data quality, it is important to consider how the data evolves with time. Batini and Scannapieco [2] identify three time-related dimensions: currency, timeliness and volatility.

Currency refers to how instantly the data is updated. *Timeliness* regards how actual the data is for the current task. Finally, *volatility* deals with the frequency with which data varies with time.

Other data quality dimensions

Some additional data quality dimensions that are frequently encountered in the bibliography [2], [4], [11] are the following:

- Accessibility
- Appropriate amount of data
- Believability
- Interpretability
- Understandability
- Uniqueness

2.1.3 Data Constraints

We use the term *data constraints* to refer to specific rules that the data needs to comply to in order to be considered of high quality or even meaningful in the context where it is being used. In this work, we also use the terms *business rules* or just *rules* to refer to the same concept. By definition, constraints are closely related to data quality, as they provide a concrete frame to the sometimes abstract concept of data quality and equip us with ways to measure and quantify it in an intuitive way.

2.1.4 Data Quality as a Constraint Problem

After recognizing the close relation of data quality to data constraints, deciding to regard data quality as a constraint problem seems to be a very reasonable approach. This approach has, however, its disadvantages. As Dasu and Johnson [4] discuss, defining the constraints and rules that are required for high-quality data can be extremely domain-specific. This means that in order to define the constraints for a system, it might be necessary to consult the top-level experts of the corresponding field. In addition, due to their domain-specific nature, constraints cannot be easily generalized and transferred to other systems, which means that the labor-intensive and time-consuming task of defining new constraints has to be repeated for every system whose quality has to be managed.

Another disadvantage of this approach is the fact that constraints are often defined in different layers of the application, which, as mentioned in section 1.1, can lead to maintenance and performance issues, as well as inconsistencies and redundancies. In addition, even though domain experts take part in the constraint definition process, after the constraints have been defined, the users of the system, who might also be domain experts, can usually have little further influence and configuration options for the already defined constraints. These last two issues are largely addressed in this work and will be discussed again in later chapters of this report.

2.2 Data Quality Management Frameworks

A *data quality management framework* (DQMF) has been described as "a comprehensive checklist of all best practices and desirable requirements for an optimal management of data quality" along with (self-)assessment tools, with the ultimate goal of controlling and improving data quality [6].

In this section, we present a non-exhaustive overview of some of the data quality management frameworks that we encountered during our background search. This overview ranges from purely conceptual to non-abstract industry-adopted frameworks.

2.2.1 Conceptual Frameworks

This subsection presents two of the most prominent conceptual data quality management frameworks that have been proposed by the research community.

Beyond Accuracy: What data quality means to data consumers

One of the most acknowledged conceptual frameworks for data quality is described in [15]. The authors of [15] underline the importance of data quality and note the multidimensionality and broadness that characterize it. In order to capture the data quality aspects that are most important to data consumers, they conduct a series of surveys and studies which identify, sort and rank these aspects according to their importance to consumers. The product of this work is a hierarchical framework for organizing data quality dimensions into four data quality categories (intrinsic, contextual, representational and accessibility data quality) and is summarized by figure 2.1.

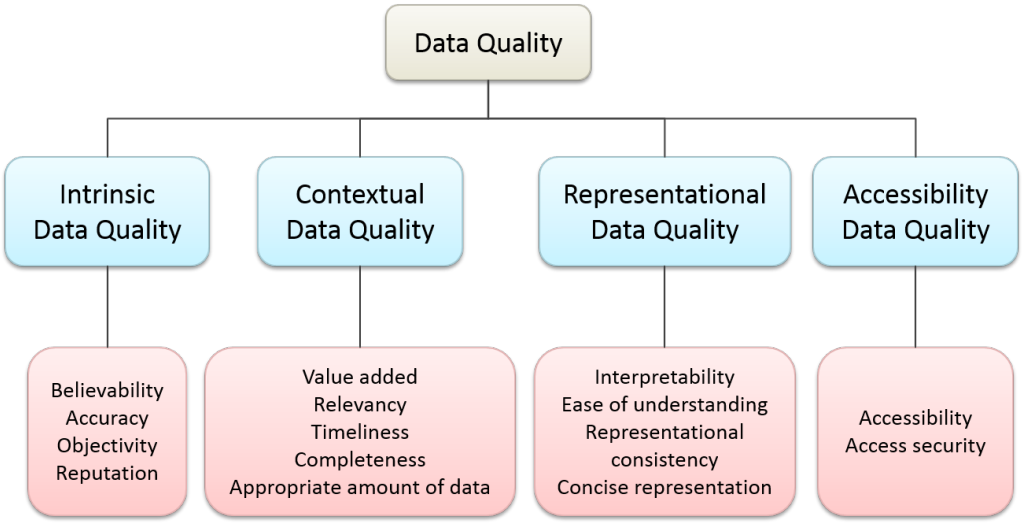


Figure 2.1: A conceptual framework of data quality [15]

A Conceptual Framework for Developing Quality Measures for Information Systems

The author of [5] focuses on the quality of information systems, which, as he argues, is highly dependent on the quality of the data that the systems use and manipulate. This results from the fact that in most cases, the data and the information that this data represents can be considered as the actual product of the system. He also notes that data quality measures should be implemented within the context that data is created and used by the data consumers. Therefore, similarly to [15], he proposes a hierarchical conceptual framework for developing quality measures for information systems. This framework includes three of the data quality categories encountered in [15] (contextual, representational and accessibility) and introduces two

additional categories (ergonomic and transactional), resulting to the hierarchical framework summarized by figure 2.2.

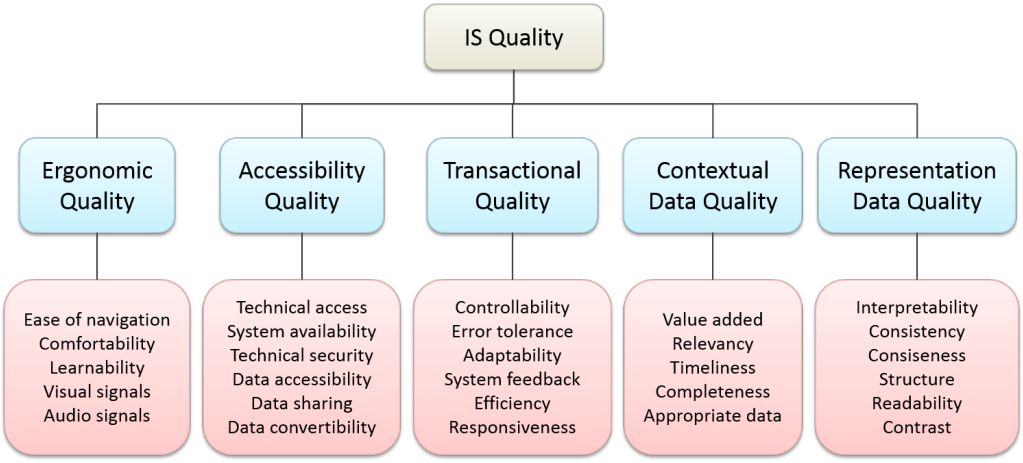


Figure 2.2: A hierarchical framework for information system quality [5]

2.2.2 Industry Guidelines and Implemented Frameworks

In this subsection we discuss some currently existing data quality management frameworks and guidelines. The purpose of this subsection is not to provide an exhaustive list of such frameworks, but rather a small selection of examples.

GS1 Data Quality Framework

The non-profit international organization GS1[1] provides an *industry-developed and endorsed data quality framework package* intended to educate businesses about the meaning and importance of data quality and offer a best-practices guide to controlling and improving data quality. This framework aims to provide a global data quality solution and consists of three sections: a list of requirements for a good data quality management system, a product inspection procedure and a self-assessment procedure.

International Monetary Fund Data Quality Assessment Framework

The Statistics Department of the International Monetary Fund[2] (IMF) developed a *data quality assessment framework* (DQAF) which identifies quality-related features of governance of statistical systems, statistical processes, and statistical products. This framework provides a structure for evaluating existing practices against a set of best practices, which include internationally accepted methodologies. It is mainly used to guide IMF staff on the use of data in

[1] http://www.gs1.org/
[2] http://www.imf.org/external/

several tasks, to assist country efforts, such as self-assessments, and to enable data users in evaluating data for policy analysis, forecasts and economic performance [8].

Australian Bureau of Statistics Data Quality Framework

The Australian Bureau of Statistics (ABS) defines a Data Quality Framework (DQF) that is aimed at different users in different settings, including government agencies, statistical agencies and independent research agencies. This framework is comprised of seven dimensions of quality and is designed to be used in evaluating statistical collections and products. To use the framework, ABS recommends the development of a *Data Quality Statement* (DQS), which is a presentation of information about the quality of a statistical collection or product using the DQF. To assist in the development of a DQS, ABS provides the *data quality statement tool* [1].

2.3 FoodCASE

This section focuses on FoodCASE, which is the information system within which we implemented and evaluated our data quality management framework. We start by presenting some general information about FoodCASE and continue with an overview of its architecture.

2.3.1 General Information

FoodCASE (Food Composition And System Environment) is a food science data management system that was developed at the ETH Zurich. Initially, FoodCASE was mainly a food composition management system used for the administration and generation of nutrient values. It is used to manage the data in the *Swiss Food Composition Database* (SFCDB), which was developed by the ETH Zurich and the Swiss Federal Office of Public Health. The SFCDB includes information about over 700 Swiss foods. Its records indicate the energy and nutrient content of the food in standardized form and constitute the basis for calculating the composition and nutritional value of food.

Recently, FoodCASE was extended to include the management of contaminants in order to fulfill the requirements for performing *Total Diet Studies* (TDS). FoodCASE is now part of the European project Total Diet Study Exposure (TDS-Exposure), which is a research project with 26 partners from 19 countries with the goal of harmonizing TDS within Europe.

2.3.2 Architecture

The FoodCASE back-end is implemented with the use of EJB session beans running on a JBoss application server. The GUI is implemented using Java Swing, while a PostgreSQL

database serves as the persistence data storage. The high-level architecture of the FoodCASE system is depicted in figure 2.3.

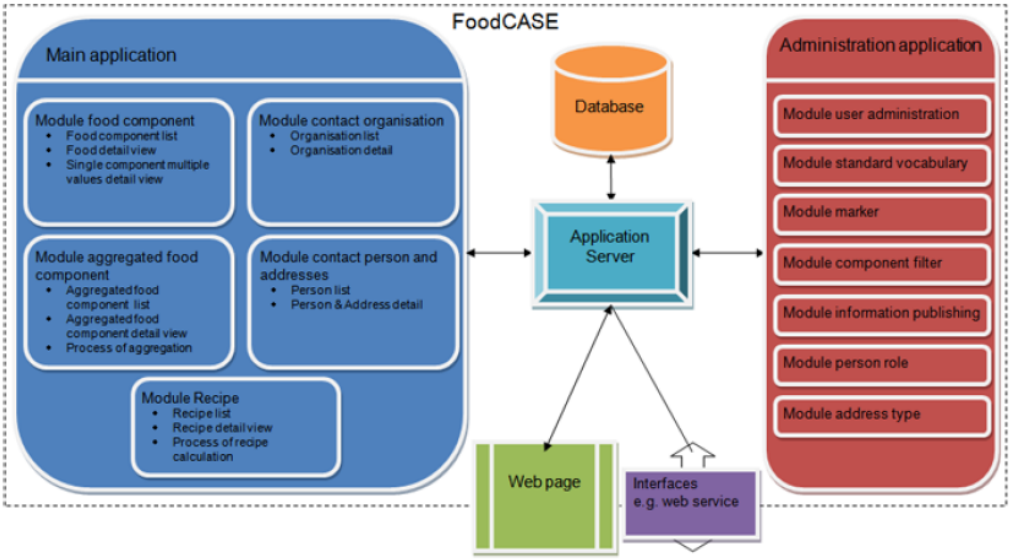


Figure 2.3: The high-level architecture of the FoodCASE system. Each module is depicted with the use of a different color.

As seen in figure 2.3, the FoodCASE architecture consists of the following six modules [10]:

- A PostgreSQL *Database* which stores all the business data of the application.
- A JBoss *Application Server* which runs EJB session beans. The session beans provide services which can be used by the client modules. The main responsibility of the EJB layer is to take care of the business logic and the persistence, and make them transparent to the clients, so that they can work directly with the business objects.
- A *Content Management System* (CMS) allowing the food compilers to manage the food composition data in the system.
- An *Administration Module* for the system administrators.
- A *Web Interface* which allows the public to query information about the composition of the foods available in Switzerland.
- A *Web Service* to export a single food item or the whole database as a EuroFIR Food Data Transport Package (FDTP).

2.4 Issues to be Addressed

Our background investigation revealed some of the data-quality-related issues that have not yet been adequately addressed by the research community. Many of the existing data quality

management frameworks that we encountered are designed as conceptual guidelines and do not offer any concrete implementation details. On the other hand, there are DQMF implementations that are too system-specific and cannot be easily generalized and used in different information systems. For this reason, we aim to introduce a DQMF concept that is general enough to be applicable to any information system, but also detailed enough and well-defined in order to facilitate the implementation of every aspect it entails.

Our work aspires to investigate the extend to which data quality can be modeled and managed by the use of constraints. Therefore, the DQMF that we propose is completely based on constraints, which we consider capable of accurately capturing most data quality requirements.

We also try to address two of the most important issues that are commonly associated with the use of constraints in large information systems. The first issue is the distribution of constraints and their validators in different layers and levels of the system, which can result in maintenance and performance issues, as well as inconsistencies. To overcome this, we explore ways in which constraints can be managed in a unified way.

The second important constraint-related issue that we focus on is the fact that after the constraints have been defined and declared, the users of the system are usually unable to configure them in any way. Additionally, all constraints are regarded as equally important by the system, which semantically might not be the case. To that end, our proposed DQMF offers several layers of configuration options to the users, allowing them to decide exactly what they wish to validate or analyze as well as how important each constraint should be.

Last but not least, our research revealed that data quality is generally viewed as an abstract concept. However, we believe that data consumers can often benefit from a quantitative representation of data quality. To that end, we introduce a novel data quality measurement and calculation scheme, which is also based on constraints.

3 Approach

In this chapter, we will describe in detail the main contribution of this thesis, which is a concept for a highly configurable data input validation and data quality management and analysis framework. This concept was implemented within the FoodCASE food science data management system, which will be discussed in detail in chapter 4. Even though we sometimes make use of examples taken from our FoodCASE implementation in order to illustrate or clarify some of the ideas in our concept, it is important to note that this concept could be implemented for any data management information system that applies constraints of any type on its data.

3.1 Concept

As mentioned in sections 1.1 and 2.4, one of the main goals of this work is to explore new ways of managing and measuring data quality in information systems. More specifically, we want to investigate the extend to which data quality can be viewed as a *constraint problem*. For this purpose, we introduce a concept for a *data quality management framework* that is solely based on and modeled by the use of constraints as *data quality indicators*. This concept was developed as a consolidation and enhancement of previous related work [10], [13] done in the GlobIS[1] group at ETH Zurich with the introduction of several novel ideas.

In subsection 2.1.4 we mentioned that a drawback of using constraints extensively in a large information system is that they are often defined in several parts and layers of the system, which makes their maintenance problematic and can possibly increase the system's perform-

[1] https://globis.ethz.ch/

ance costs. For this reason, a central focus of this thesis was to create a framework in which *all the constraints could be managed in a unified way*. The results of the research conducted by Probst [13] demonstrate that the use of *Bean Validation* (BV) for the definition and validation of constraints, can achieve this goal. The reasoning behind the use of BV is further discussed in subsection 3.1.1, while the related implementation details are explained in subsection 4.1.2.

Another main focus in this thesis is to provide a framework that is *highly flexible and easily customizable* not only by the developers that define the constraints, but also by its end users even at runtime. For this purpose, we introduced several new concepts, such as the *contracts* (subsection 3.1.3), *validation groups* (subsection 3.1.5) and *payload weights* (subsection 3.1.6), in order to provide several layers of configuration options.

3.1.1 Bean Validation

Bean Validation[2] (JSR 303, JSR 349) is a Java specification that allows programmers to express constraints on object models with the use of annotations. It has been accepted as part of the Java EE 6 specification. We will discuss about the technical details behind the use of Bean Validation in our work in subsection 4.1.2. In this section, we will focus on the features of Bean Validation that led to its selection for our data quality management framework.

As mentioned before, the most important characteristic that made Bean Validation an attractive choice is the fact that it allows for the definition of all the DQ-related constraints in a single place, which in our framework is the entity beans corresponding to the entities being validated and analyzed. This simplifies the management and maintenance of the constraints, as they are always defined together with the entities they constrain.

Additionally, the use of Bean Validation allows for the validation of all constraints in a unified way. Since data validation is a process that concerns all the levels of an application, it is common that the same validation logic is implemented in every single layer, which can lead to errors and increased performance costs [7]. This is demonstrated in figure 3.1.

In contrast, by using the Bean Validation specification, the validation logic is expressed as part of the constraint annotation definition and centralized in the domain model, which allows for their validation by using a single validator, as illustrated in figure 3.2.

Another important feature of Bean Validation, is the fact that besides the basic constraints it provides by default (e.g. `@NotNull`, `@Min`, `@Max`, etc.), it also allows for the creation of virtually any type of custom constraint. This provides a high-degree of flexibility and enables the use of Bean Validation as the exclusive constraint modeling and definition mechanism of an information system.

Bean Validation also provides mechanisms for grouping and labeling constraints, which we adopted as additional configuration mechanisms for our framework in the form of *validation groups* (subsection 3.1.5) and *payload weights* (subsection 3.1.6).

[2] http://beanvalidation.org/

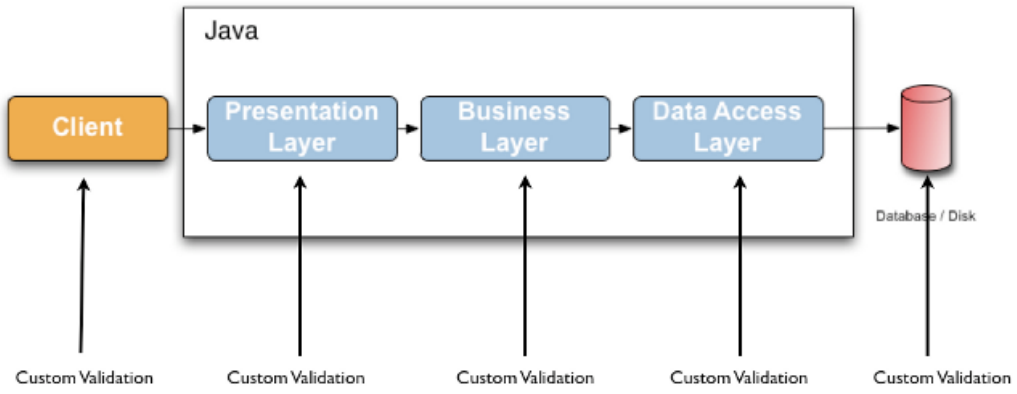


Figure 3.1: Implementing the same validation logic in each layer of the application [7]

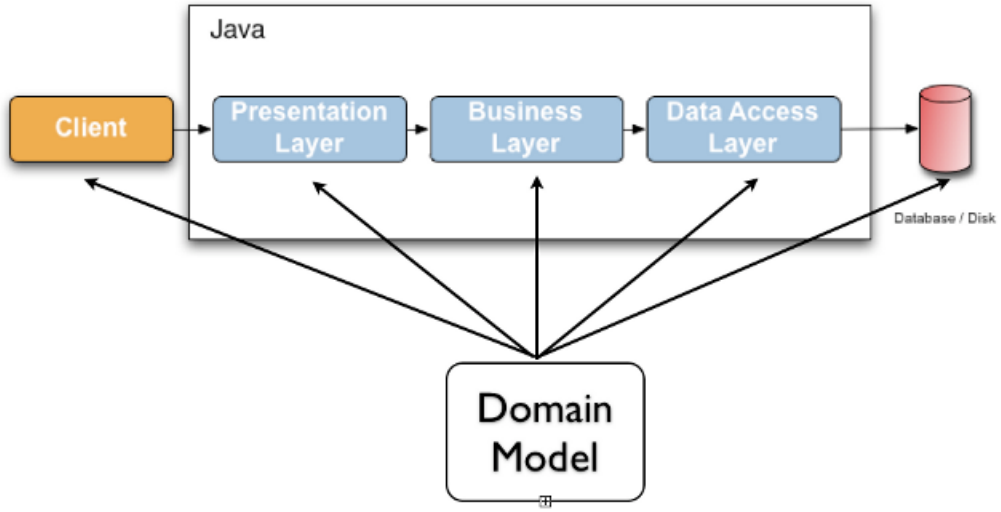


Figure 3.2: Bean Validation allows for the validation of all constraints in a unified way [7]

3.1.2 Data Quality Measurement and Calculation

Data quality is generally regarded as an abstract concept that cannot be easily measured and quantified in a unique and straightforward manner. In this thesis, we created our own data quality measurement and calculation scheme. This scheme results to an intuitive *data quality indicator* that corresponds to the degree that the data that is being validated or analyzed satisfies the defined constraints, and is measured in *(data) quality points*.

A key concept behind our quality measuring scheme is that every constraint is assigned a *constraint weight*. This weight can take an integer value between 0 and 100. The weight signifies the importance of the constraint for the user of the application, with a weight of 0 corresponding to an insignificant constraint and a weight of 100 corresponding to a constraint of highest importance.

According to our measuring scheme, the weights are used to calculate different levels of data quality indicators, using the notion of the *weighted arithmetic mean*. Each data quality indicator can take an integer value between 0-100 *quality points*. The different quality indicator levels belong to two categories, depending on whether they are being calculated during *data input validation* or during *data quality analysis*. The difference between these two categories is that in the case of *input validation quality indicators* we validate and analyze *only one entity object*, the entity object for which data is being entered. On the other hand, in the case of the *data quality analysis quality indicators*, we usually validate and analyze multiple entity objects, i.e. all entity objects of the type and characteristics that we are interested to analyze.

After making a distinction between the two categories of quality indicators, we will explain what the different possible quality indicator levels are for each category and how they are calculated. To help clarify the meaning of each indicator, we will use an example. Let's assume that the data that we are interested in consists of two data entities, entity A, which contains 100 entity objects, and entity B containing 200 entity objects. Let's also assume that there are three constraints declared for entity A: constraint a with weight 80, constraint b with weight 60 and constraint c with weight 90. Similarly, let's assume that the declared constraints for entity B are constraint d with weight 30 and constraint e with weight 100. The details of our example setup are summarized in table 3.1.

Entity	# of entity objects	constraint:weight
entity A	100	$a : 80$, $b : 60$, $c : 90$
entity B	200	$d : 30$, $e : 100$

Table 3.1: An example of two data entities, along with the number of entity object they contain and their declared constraints

Input Validation Quality Indicator ($IVQI$)

In the input validation case, there is only one quality indicator level, which represents the current quality of the data that has been so far entered for the object that is being modified. This *input validation quality indicator* ($IVQI$) is calculated using the equation 3.1, where n is the number of constraints that are defined for the current entity type, w_i is the weight of constraint i, while α_i is either 1, if constraint i is satisfied, or 0, if constraint i is violated.

$$IVQI = \|100 \cdot \frac{\sum_{i=1}^{n} \alpha_i w_i}{\sum_{i=1}^{n} w_i}\| \tag{3.1}$$

Using the example of table 3.1, let's assume that while entering data for an entity A object, the input validation reveals that while constraints a and c are satisfied, constraint b is violated. In this case, the $IVQI_A$ is calculated as described by equation 3.2 and is equal to 74 quality points out of 100.

$$IVQI_A = \|100 \cdot \frac{1 \cdot 80 + 0 \cdot 60 + 1 \cdot 90}{80 + 60 + 90}\| = \|100 \cdot \frac{170}{230}\| = \|73.91\| = 74 \tag{3.2}$$

Constraint-Level Analysis Quality Indicator ($CLAQI$)

In the data quality analysis case, we have three levels of quality indicators. On the first level, we encounter the *constraint-level analysis quality indicator* ($CLAQI$), which represents the quality of the particular constraint across all entity objects. This quality indicator is calculated using the equation 3.3. Please note that this quality indicator doesn't take any weights into account and also that each constraint corresponds to one and only one entity type at the time.

$$CLAQI = \|100 \cdot \frac{\text{no. of objects that satisfy current constraint}}{\text{total no. of objects (of current constraint entity type)}}\| \tag{3.3}$$

To continue with the example of table 3.1, let's assume that constraint a of entity A is satisfied in 80 out of the 100 entity A objects. Therefore, the $CLAQI_a$ for constraint a is equivalent to 80 quality points and is calculated by equation 3.4.

$$CLAQI_a = \|100 \cdot \frac{80}{100}\| = 80 \tag{3.4}$$

Entity-Level Analysis Quality Indicator ($ELAQI$)

The second data quality analysis quality indicator level corresponds to the *entity-level analysis quality indicator* ($ELAQI$), which takes into account all the objects of a particular entity type. This quality indicator is calculated by using the equation 3.5, where n is the number of constraints that are defined for the current entity type, w_i the weight of constraint i and $CLAQI_i$ the constraint-level analysis quality indicator for constraint i, which, as explained above, is calculated by equation 3.3.

$$ELAQI = \|\frac{\sum_{i=1}^{n} CLAQI_i w_i}{\sum_{i=1}^{n} w_i}\| \tag{3.5}$$

For example, let's assume that we want to calculate the $ELAQI_A$ of entity A of table 3.1. Similarly to $CLAQI_a$ for constraint a, which was calculated as 80 by equation 3.4, we calculate $CLAQI_b = 76$ for the constraint b and $CLAQI_c = 98$ for the constraint c. Therefore, using the above and the constraint weights defined in table 3.1, $ELAQI_A$ equals 86 quality points and is calculated by equation 3.6

$$ELAQI_A = \|\frac{80 \cdot 80 + 76 \cdot 60 + 98 \cdot 90}{80 + 60 + 90}\| = 86 \tag{3.6}$$

Overall Analysis Quality Indicator ($OAQI$)

The third-level analysis quality indicator is the *overall analysis quality indicator* ($OAQI$), which reflects the overall quality of the analyzed data. This quality indicator is calculated as the weighted arithmetic mean of the $ELAQI$ of all the analyzed entities, with the weights in this case being the number of objects per entity. The reasoning behind this choice is that we assume that more populous entities should affect the overall quality on a larger degree than less-populated entities. Of course, if for a particular system our assumption is incorrect and all entities must be considered equivalent for the quality indicator calculation, the use of a different calculation scheme is also possible, as discussed in section 7.1. In our implementation, the overall analysis quality indicator is calculated with the use of equation 3.7, in which n denotes the number of different analyzed entity types, $ELAQI_i$ the entity-level analysis quality indicator of entity type i and w_i the number of objects of entity type i.

$$OAQI = \| \frac{\sum_{i=1}^{n} ELAQI_i w_i}{\sum_{i=1}^{n} w_i} \| \tag{3.7}$$

To use the ongoing example, let's assume that, as calculated by equation 3.6, $ELAQI_A$ of entity A equals 86 quality points, while $ELAQI_B$ of entity B has been calculated as 70 quality points. Using the number of objects per entity that we defined in table 3.1, the $OAQI$ of the data is calculated by equation 3.8 and equals 75 quality points.

$$OAQI = \| \frac{86 \cdot 100 + 70 \cdot 200}{100 + 200} \| = \|75.33\| = 75 \tag{3.8}$$

3.1.3 Contracts

One of the central concepts that we introduce in this thesis is the notion of contracts. We use the term *contracts* to refer to text files in which we define the weight, i.e. importance, of each constraint in the system as well as whether the constraint is activated or not. Contracts can be used to define customized analysis and validation profiles for different users and applications of the information system. They can also be easily shared between users. Thus, it is possible for a user to have multiple contracts, while on the other hand the same contract can be shared by multiple users. Contracts can easily be imported, created and configured either via the administrative GUI, which will be further discussed in section 3.4, or by directly adding or editing an appropriate contract file with the use of any text editor.

If a user is not authorized or does not wish to specify a contract, a fallback (default) contract will be used. The *fallback contract* distinguishes between constraints of different importance based on the gathered constraint requirements. If the user selects a contract that does not include all constraints, a *default weight* that has been chosen by the user will be used for the excluded constraints.

More technical details about how the contracts are defined and what they contain and look

like will be discussed in section 4.2.2.

3.1.4 Warning to Error Threshold

According to our concept, the users are able to define a particular weight of their choice to serve as the *warning to error threshold* (*WtET*). As its name suggests, this special weight defines the distinction between two types of violations: the *warnings*, which are less severe, and the *errors*, 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 in section 3.3, as well as whether the user should be able to save their input when such a violation occurs.

3.1.5 Validation Groups

Each constraint can belong to one or more *validation groups*. Validation groups can be used to organize constraints into logical units. For instance, by using validation groups, we can categorize constraints according to their type. We can then use this categorization 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 only to the constraints that they are most interested in at any given time.

As will be further explained in subsection 4.2.3, new groups can only be defined by developers. Similarly, it is only possible to assign a constraint to a group during the declaration of the constraint in the related entity class code. However, the users of the framework can use contracts as an alternative way of creating groups at runtime, by activating only the constraints that they are interested in and deactivating the rest.

3.1.6 Payload Labels and Weights

The concept of *payloads* is part of the Bean Validation specification[3]. Payloads are typically used to associate metadata information with a given constraint declaration. In our framework, payloads have been repurposed as a way of 'labeling' specific constraints, in order to provide additional configuration and flexibility options to the user. To achieve that, we predefined six distinctive payload labels. Each of these labels corresponds to a different predefined weight, which we refer to as the *payload weight*. If desired, the users can choose to allow the payload weights to override the contract weights for all constraints for which a payload label has been declared.

Similarly to validation groups, we can only assign a payload label during the declaration of the constraint. Additionally, like groups, payload labels are completely optional. This implies

[3] http://beanvalidation.org/latest-draft/spec/

that for constraints that do not have an assigned payload label, the choice to allow payload weights override their contract weight will have no effect whatsoever and the chosen contract weight (or the fallback default) will be used in any case.

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. Or there might be a constraint that we regard as irrelevant to quality and would prefer to associate with a weight of 0. As contracts can be easily edited or changed, the payload mechanism allows us to assign constant weights in an easy-to-maintain manner. Of course, the option to allow payload weights to override contract weights is optional, and if not chosen, the payload labels solely serve their purpose of carrying metadata.

The payload labels that we defined along with their corresponding weights can be found in table 3.2.

Payload Label	Weight
Info	0
Suggestion	20
Recommendation	40
Warning	60
Error	80
SevereError	100

Table 3.2: The mapping between our predefined payload labels and their associated weights

3.2 Data Quality Analysis Module

After having described the main ideas behind our data quality management concept, we will continue with a description of the three modules of our proposed framework, starting with the *data quality analysis module*. As mentioned in the beginning of this chapter, all concepts and modules that we propose have been implemented and evaluated in the FoodCASE system. Most of the screenshots and examples that we use to describe our modules are taken from our FoodCASE implementation. However, our proposal could be applied to any information system and therefore any mention of FoodCASE is done purely for the sake of example.

The *data quality analysis module* is the central module of our proposed framework. It is part of the regular *client* of the information system and is accessible to any user that is authorized to use the client. It can be used for performing a custom quality analysis on the data portion that the user is interested in. The analysis is performed according to the contract that is chosen by the user. These settings can be changed in the analysis starting screen, an annotated version of which is depicted in figure 3.3.

In order to start a new analysis, the user needs to follow three steps (as depicted on figure 3.3): first, they should choose the desired analysis contract, by using the dialog box that appears

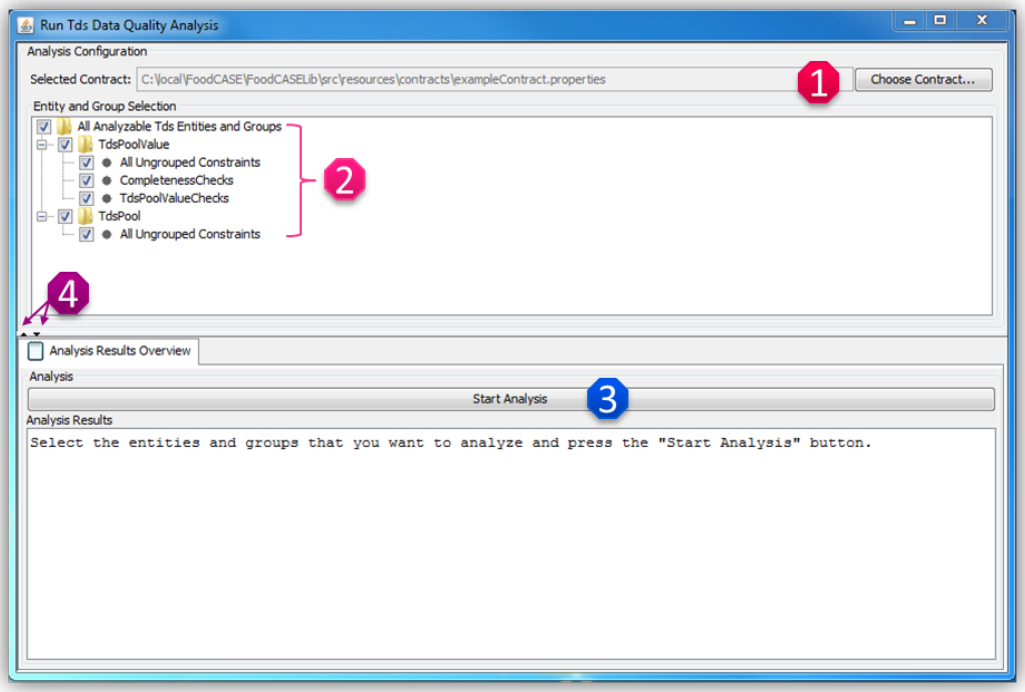


Figure 3.3: Users of the data quality analysis module can select the entities and groups that they want to analyze as well as the analysis contract

after pressing the associated button `(1)`. Next, they have to select the data portion they want to analyze, by using the entity and group selection tree `(2)`. This tree enables users to easily select either the entirety of all the analyzable entities and all of their groups, or a subset of them. Next, the user has to press the `Start Analysis` button `(3)`, in order to launch the analysis. The GUI also allows users to easily maximize the area (entity and group selection or analysis results) that they are interested in at each time point `(4)`.

After the analysis is performed, the analysis results appear in the `Analysis Results Overview` tab in textual form, as seen in figure 3.4. In addition, the three visualization tabs become visible as well `(1)`. The textual version of the analysis results consists of two parts. The first part `(2)` contains detailed statistics about each of the analyzed entities, including the number of validated objects, how many of them were violation-free, the total number of constraints that were validated and the number of violations that were found. The second part `(3)` presents the *entity-level quality indicator* ($ELAQI$) for each analyzed entity as well as the *overall analysis quality indicator* ($OAQI$) of the analyzed data.

3.2.1 Visualizations and Data Cleansing

As mentioned in the previous subsection, after the completion of the analysis, the three *analysis result visualization tabs* become visible. These visualizations provide the user with a

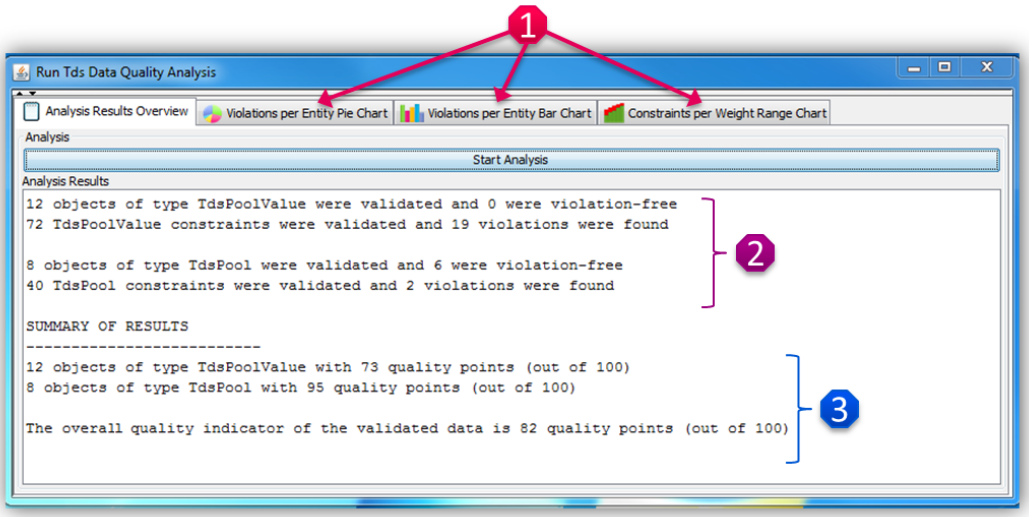


Figure 3.4: The `Analysis Results Overview` tab contains the results of the analysis in textual form

bird's eye view of the violations encountered. This overview is provided in two different *contexts*, violations per entity type and violations per constraint weight range, and three alternative *chart types*: a pie chart, a bar chart and a stacked bar chart. The visualizations allow the user to spot the most problematic areas easily and also drill down through their areas of interest and correct individual entries. We will now provide a description of each of the three different visualization options.

Violations per Entity Type Pie Chart

The *violations per entity type pie chart*, whose fully expanded version is depicted in figure 3.5, allows users to observe the number of violations per entity type. By clicking on the relevant tab, the users can first only see the pie chart in the panel's leftmost section `(1)`. The exact number of violations per entity can be encountered either as a tool tip by hovering the mouse over the corresponding pie chart slice, or as a text label `(4)` after clicking on the slice. Clicking on a pie chart slice also leads to the appearance of the entity-specific list of violated constraints `(2)`, sorted by their weight, which, according to the user's settings, can be either the contract or payload weight.

The user can drill down further by clicking on the type of constraint violations that they want to examine, which results in the appearance of the list of the actual entity objects that violate the corresponding constraint `(3)`. Finally, they are able to click on the problematic object entry that they are interested in correcting, which leads to a highlighted detail frame that lists all the issues with the corresponding object and its $IVQI$, which coincides with the input validation frame of figure 3.8.

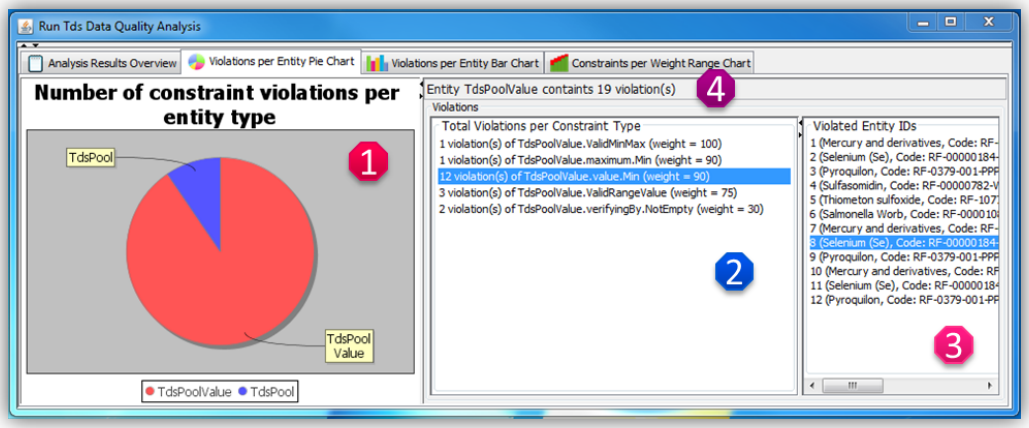


Figure 3.5: The *violations per entity type pie chart* depicts the total violations per entity type that were discovered during the analysis in the form of a pie chart

Violations per Entity Type Bar Chart

The *violations per entity type bar chart*, which can be seen in figure 3.6, presents the same information as the pie chart described before, i.e. the total discovered violations per entity type, but in the form of a bar chart. This alternative view was added to satisfy potential personal preferences of users towards a bar chart and to ensure that in the case where many different entity types are present, they can easily be distinguished between each other. Drilling down to individual problematic entries can be conducted following the same steps as before. In figure 3.6, it can be seen that some of the entries appear in *violet* text `(1)`. The *violet* color indicates that these entries have been edited *after* the completion of the analysis and it is possible that the associated violation sources have already been corrected.

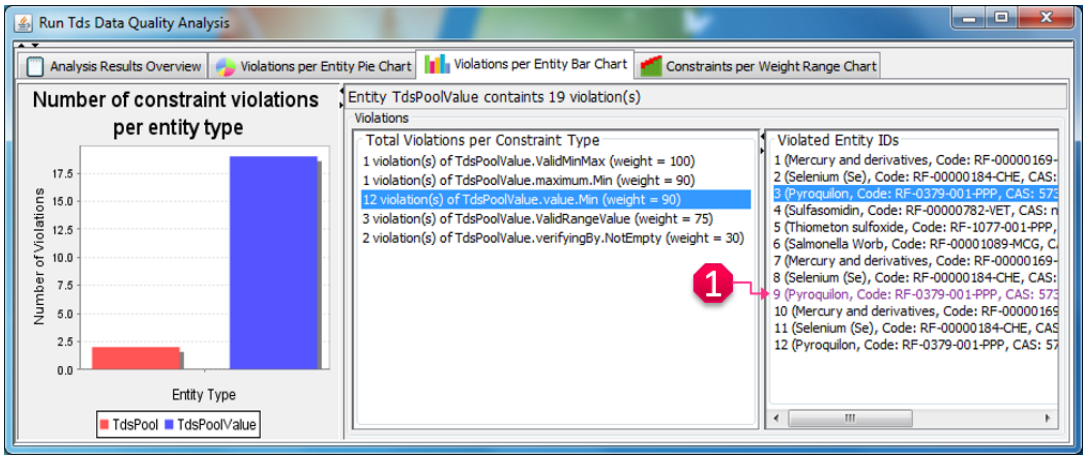


Figure 3.6: The *violations per entity type bar chart* depicts the total violations per entity type that were discovered during the analysis in the form of a bar chart

Constraints per Weight Range Stacked Bar Chart

The third visualization type, depicted in figure 3.7, presents the existing constraints in the system categorized by the numerical range in which their weight belongs to. For this categorization, we define five weight ranges: 0-20, 21-40, 41-60, 61-80 and 81-100. This visualization differs from the previous ones in two ways. First, it doesn't categorize the constraints based on their entity type, but on their weight. Secondly, it not only depicts the violated constraints (in *red*), but the satisfied ones as well (in *green*). However, a user can only drill down through the violations, which means that in the stacked bar chart on the leftmost part of figure 3.7, only the *red* part `(1)` is clickable. The detail panels `(2)`, `(3)` are similar to what has been described before, with the only difference being that in the violated constraint list `(2)` the violations are sorted primarily by *entity type name* and then by weight.

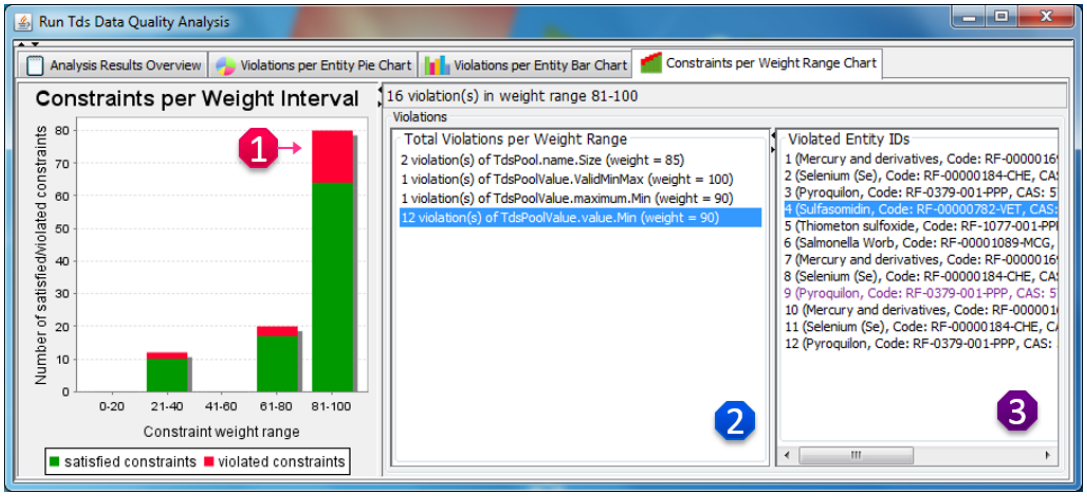


Figure 3.7: The *constraints per weight range stacked bar chart* depicts the total number of constraints (satisfied and violated) encountered in each of the five weight ranges

3.3 Data Input Validation Module

The *Data Input Validation Module* encompasses all the necessary tools for assisting the users in avoiding the introduction of data-quality-related issues, i.e. constraint violations, while entering or editing data in the system via the system's presentation layer (GUI).

We will now describe the different parts of the data input validation module that we propose, using as an example an instance from our FoodCASE implementation, which is depicted in figure 3.8.

As seen in figure 3.8, when the user enters an incorrect value, i.e. a value that violates one or more constraints, the corresponding entry field background becomes *red* `(1)` in order to indicate *errors*, i.e. violations of constraints with higher weight than the *warning to error threshold* (*WtET*, discussed in subsection 3.1.4), or *orange* `(2)` to indicate *warnings*, i.e.

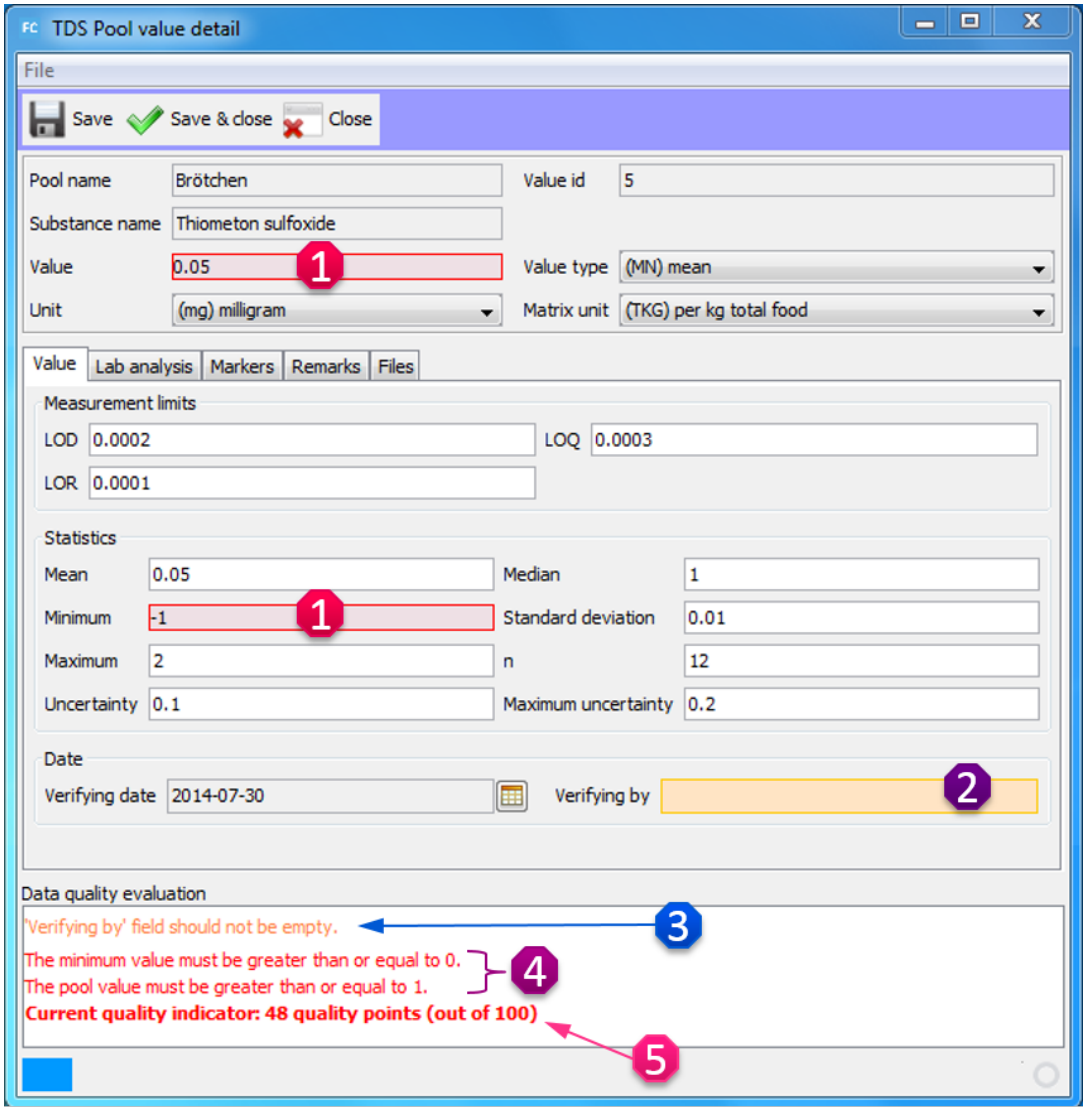


Figure 3.8: An annotated data input frame displaying various types of input validation feedback

violations of constraints with weight lower than the WtET. In addition, more detailed information about the discovered violations appears in the *data quality evaluation panel* at the bottom of the frame. This information consists of several parts, which can be distinguished by their different font types and colors, which will be explained below with the help of figure 3.8.

The plain font represents the list of violations that were discovered and might be colored either *red* for errors `(4)`, *orange* for warnings `(3)`, or *green* if no violations were found. Similarly, the bold font `(5)` is used to show the current object's *input validation quality indicator* ($IVQI$) and, as before, might have one of three colors: *red*, if there is at least one error, *orange* if all discovered violations are warnings, or *green* if no violations were found.

3.4 Administrative Tools Module

As discussed, our proposed framework offers several configuration options to allow users to tune the data quality management tools according to their wishes. Therefore, it is important to provide an *administrative tools module* that is only accessible to *'power' users* and allows them to easily change these configuration settings. Our proposed administrative tools module consists of three parts, which we will now describe in detail.

3.4.1 Contract Editing and Creation

The first part of our proposed administrative tools module can be used by a power user in order to create new contracts or edit already existing ones. An example of it in use is depicted in figure 3.9. To start with the editing process the user can select one of the existing contracts, by using the `Choose Contract` button `(1)`. The various parts of the selected contract are then displayed in the form of a 6-column table. Each row in this table represents a constraint that exists in the contract. The first (leftmost) four columns of the table display some of the actual characteristics of the constraint and are not editable. In more detail, the first three columns `(2)` represent the unique constraint identifier, which consists of the name of the entity in which the constraint is defined, the entity field to which the constraint applies, when applicable, and the name of the constraint. The fourth column `(3)` informs the user about whether the contract weight can be overridden by a payload weight, i.e. whether a payload label has been defined for this constraint. The rightmost two columns are the only ones that can be edited by the user. As their labels suggest, the fifth column `(4)` allows the user to edit the constraint contract weight while the sixth and last column `(5)` allows them to activate or deactivate the constraint for the analysis and input validation. The user can then save `(8)` or discard `(6)` their changes.

To start the process of creating a new contract, the user should press the corresponding button `(7)` and enter the name of the new contract in the dialog box that appears. As a result, all the existing constraints appear in the table with blank weights and activated. The user can then enter the weights of constraints and possibly deactivate some of them as desired.

3.4.2 General Validation and Analysis Settings

The second part of the framework's administrative tools allows the power user to edit some general settings that apply to both the analysis and input validation modules. An example of this panel can be found in figure 3.10. First, the user can decide whether the payload weights should override the corresponding contract weights `(1)`. Then, the user can choose two important weights, i.e. the *default weight* `(2)` that is used for a constraint if a corresponding contract or payload weight is not found as well as the *warning to error threshold* `(3)`, which was described in subsection 3.1.4. To help new users understand the quality and calculation scheme that the framework uses, the *general validation and analysis settings panel* also contains a brief overview of the main ideas behind it `(4)`.

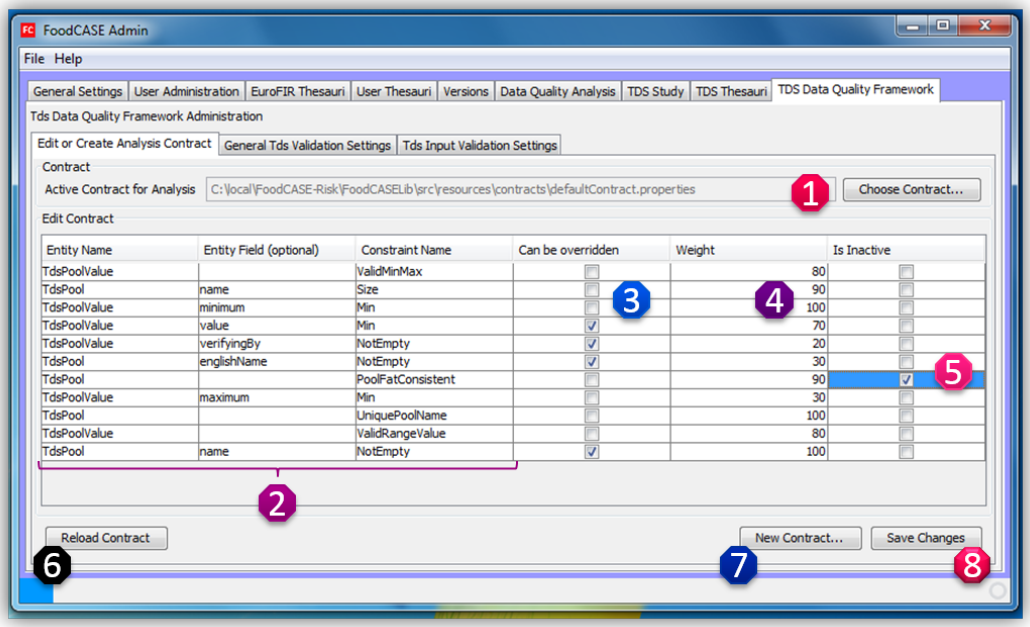


Figure 3.9: The *contract editing and creation* panel allows users to edit existing contracts or create new ones

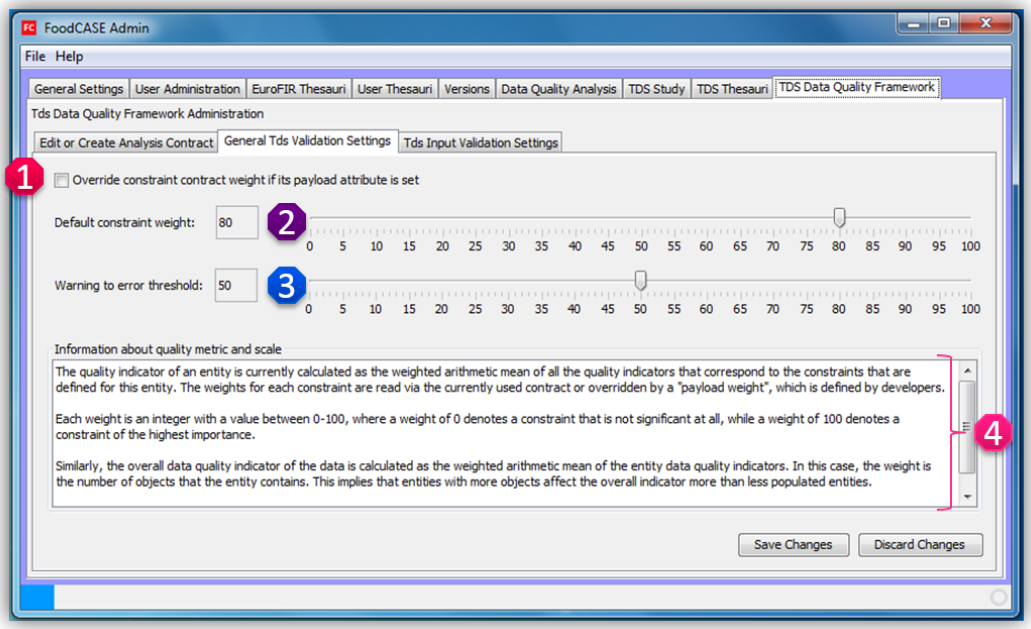


Figure 3.10: The *general validation and analysis settings* panel allows users to configure general settings for the validation and analysis

3.4.3 Input Validation Settings

The third and final part of our proposed administrative tools module enables the power user to configure the input validation module. This is depicted in figure 3.11. To this end, the user can select the contract `(1)` that they wish to use during input validation as well as the entities and groups `(2)` that they want to have validated during data input. Finally, they can choose whether they want warnings (see subsection 3.1.4) to be displayed along with errors when violations are discovered `(3)`.

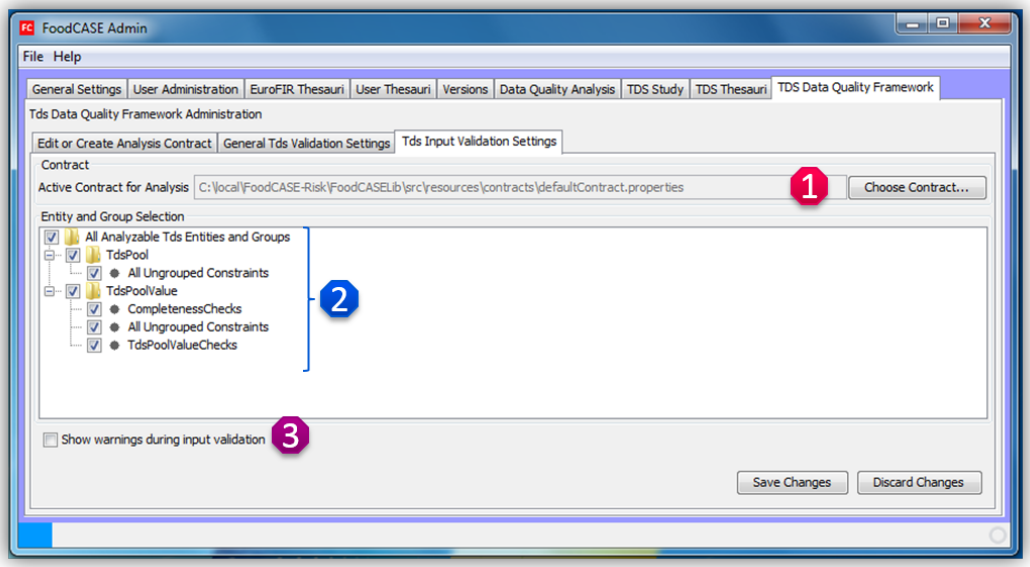


Figure 3.11: The *input validation and analysis settings* panel allows users to configure settings for the input validation and analysis

4 Implementation

This chapter focuses on the implementation details of the data quality management framework that we propose. After presenting an overview of some of the technologies used, we describe in depth how the framework was implemented within the FoodCASE system.

4.1 Technologies

In this section, we will present some of the main technologies that we used in our implementation and shortly discuss the reasoning behind their choice.

4.1.1 Enterprise JavaBeans

Enterprise JavaBeans (EJB) is a server-side component architecture for the Java Platform Enterprise Edition[1]. The EJB model enables the encapsulation of the business logic of the application. As FoodCASE is designed based on the EJB model, the business logic of our data quality management framework is implemented as a *Stateless Session EJB*.

4.1.2 Hibernate Validator

In subsection 3.1.1 we discussed the reasoning behind choosing the *Bean Validation (BV)* specification in order to express and validate the data quality constraints of our framework.

[1] http://www.oracle.com/technetwork/java/javaee/ejb/index.html

There are numerous existing bean and object validation frameworks for the Java language. Some of the most prominent ones that we encountered during our initial research include *Hibernate Validator*[2], whose 5.x version is the reference implementation of Bean Validation 1.1, *Apache BVal*[3] and *OVal*[4]. After a short examination of their features, we decided to focus our investigation on *Hibernate Validator* (*HV*) and *OVal* and performed an extensive comparison between the two, which can be found in Appendix B.

In the end, we decided to use *Hibernate Validator (HV) 5.x*, as it is currently better documented, more actively supported and highly integrated with *Java EE*. In addition, *HV* supports the use of *validation groups* and *payloads*, which allowed us to enhance our framework with additional features. As our comparison demonstrated, even though *OVal* does offer some additional attractive features, such as the supported use of scripted expressions for pre- and post-conditions, it lacks other more important ones, such as the ability to group constraints and define payloads. Thus, *HV* was deemed as the natural choice for our implementation.

4.1.3 Java Swing

Swing is a GUI widget toolkit for the Java language. It is part of the *Oracle Java Foundation Classes* (JFC), which include a set of features for building GUIs and adding rich graphics functionality and interactivity to Java applications[5]. Swing provides a set of GUI components that are written entirely in Java and are therefore platform-independent. It is currently in the process of being replaced by *JavaFX* as the standard GUI library for Java SE, even though both will be part of the Java SE specification and included in the JRE for the foreseeable future.

As the FoodCASE GUI is currently built using the Swing API, our framework's analysis and administrative GUI modules were also created using Swing components. More details about the implementation of these modules will be discussed in later parts of this chapter.

4.1.4 JFreeChart

JFreeChart[6] is a free chart library that is written entirely in Java. It provides an API that supports a wide range of chart types, including pie charts, bar charts, histograms, line charts and XY plots. JFreeChart is one of the most widely used chart libraries for Java. We chose it for our data quality visualizations because it provides a rich range of features, it is very well documented, and highly compatible with Swing, as it supports output to several Swing component. More about the implementation of our visualizations will be discussed in subsection 4.5.4.

[2] http://hibernate.org/validator/
[3] http://bval.apache.org/
[4] http://oval.sourceforge.net/
[5] http://docs.oracle.com/javase/tutorial/uiswing/start/about.html
[6] http://www.jfree.org/jfreechart/

4.2 Main Concept Implementation Details

In this section, we will describe the details of how the main concepts of our framework were implemented. We will also explain in which of the FoodCASE modules we implemented the corresponding classes and also provide some code samples for illustrative purposes.

4.2.1 Constraint Declaration and Definition

In our framework, all the constraints related to data quality are declared in the *entity beans* which correspond to the FoodCASE entities that the constraints apply to. In FoodCASE, all the entity beans can be found in the `FoodCASE_Lib` module.

Hibernate Validator (HV) allows three levels of bean constraints: *field constraints*, *property constraints* and *class constraints*. *Field-level constraints* can be expressed by annotating the fields of a class. *Property-level constraints* can be used for classes that adhere to the *JavaBeans* standards and are expressed by annotating the properties of a bean class instead of its fields. This is achieved by annotating the property's *getter* method, and should be used as an alternative to field-level constraints. Finally, *class-level constraints* can be used to designate the whole object as the subject of the validation. This is useful when the validation depends on a correlation between several properties of an object.

As of *HV 5.x*, which is the reference implementation of Bean Validation 1.1, constraints can also be applied to the parameters and return values of the *methods* and *constructors* of any Java type [7]. Currently, we have only defined, implemented and used bean constraints, however expanding the framework to also support method constraints is also possible and will be discussed in section 7.3.

Hibernate Validator provides a set of commonly-used field-level built-in constraints. These are the constraints defined by the Bean Validation specification, with the inclusion of several additional ones. Furthermore, the user can define virtually any type of custom constraint that meets their needs. We will further discuss these built-in and custom-defined constraints in the next parts of this subsection.

All the constraints in our framework are expressed as *Java annotations*. To declare a constraint, the user has to *annotate* the field, property or class they wish to constrain. Additionally, they might need or wish to define the corresponding constraint-specific attributes.

An example of a constraint declaration can be seen in listing 4.1. In this example the annotation of the declared constraint is `@Min` and there are 4 attributes specified: `value`, which is mandatory for this constraint, `message`, `groups` and `payload`. The last three attributes are always part of any constraint's *definition*, but can be omitted from its *declaration*.

It is important to note that not all constraints can be applied to any level. For instance, none of the default constraints can be placed at class level. In other words, class-level constraints are always custom constraints [7].

Listing 4.1: An example of a simple field-level constraint annotation declaration

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

Built-in Constraints

As mentioned before, Hibernate Validator provides a basic set of commonly used constraints. Most of these built-in constraints are the constraints specified in the Bean Validation 1.1 API. A list of these constraint annotations with their associated attributes in parentheses and a short description of their function can be seen below [7]:

- `@AssertFalse`: Checks whether the annotated element is false.
- `@AssertTrue`: Checks whether the annotated element is true.
- `@DecimalMax(value=, inclusive=)`: Checks whether the annotated value is less than (when `inclusive=false`) or less than or equal to (when `inclusive=true`) the specified maximum `value`.
- `@DecimalMin(value=, inclusive=)`: Checks whether the annotated value is larger than (when `inclusive=false`) or larger than or equal to (when `inclusive=true`) the specified minimum `value`.
- `@Digits(integer=, fraction=)`: Checks whether the annotated value is a number having up to `integer` digits and `fraction` fractional digits.
- `@Future`: Checks whether the annotated date is in the future.
- `@Max(value=)`: Checks whether the annotated value is less than or equal to the specified maximum `value`.
- `@Min(value=)`: Checks whether the annotated value is higher than or equal to the specified minimum `value`.
- `@NotNull`: Checks that the annotated value is not null.
- `@Null`: Checks that the annotated value is null.
- `@Past`: Checks whether the annotated date is in the past.
- `@Pattern(regex=, flag=)`: Checks if the annotated string matches the regular expression `regex` considering the given `flag` match.
- `@Size(min=, max=)`: Checks if the annotated element's size is between `min` and `max` (inclusive).
- `@Valid`: Validates the associated object recursively.

Besides the above *BV*-specified constraints, Hibernate Validator also provides some additional built-in constraints. These are the following:

- `@CreditCardNumber(ignoreNonDigitCharacters=)`: Checks whether the annotated character sequence passes the Luhn checksum test. `ignoreNonDigitCharacters` allows to ignore non digit characters (the default is false).
- `@EAN`: Checks whether the annotated character sequence is a valid EAN[7] barcode.
- `@Email`: Checks whether the specified character sequence is a valid email address.
- `@Length(min=, max=)`: Checks whether the annotated character sequence is between `min` and `max` (inclusive).
- `@LuhnCheck(startIndex=, endIndex=, checkDigitIndex=, ignoreNonDigitCharacters=)`: Checks whether the digits within the annotated character sequence pass the *Luhn checksum* algorithm. `startIndex` and `endIndex` allow to only run the algorithm on the specified substring. `checkDigitIndex` allows to use an arbitrary digit within the character sequence as the check digit. If not specified it is assumed that the check digit is part of the specified range. `ignoreNonDigitCharacters` allows to ignore non digit characters.
- `@Mod10Check(multiplier=, weight=, startIndex=, endIndex=, checkDigitIndex=, ignoreNonDigitCharacters=)`: Checks whether the digits within the annotated character sequence pass the generic *mod 10 checksum* algorithm. `multiplier` determines the multiplier for odd numbers (defaults to 3), `weight` the weight for even numbers (defaults to 1). `startIndex` and `endIndex` allow to only run the algorithm on the specified substring. `checkDigitIndex` allows to use an arbitrary digit within the character sequence as the check digit. If not specified it is assumed that the check digit is part of the specified range. `ignoreNonDigitCharacters` allows to ignore non digit characters.
- `@Mod11Check(threshold=, startIndex=, endIndex=, checkDigitIndex=, ignoreNonDigitCharacters=, treatCheck10As=, treatCheck11As=)`: Checks that the digits within the annotated character sequence pass the *mod 11 checksum* algorithm. `threshold` specifies the threshold for the *mod11 multiplier growth*; if no value is specified the multiplier will grow indefinitely. `treatCheck10As` and `treatCheck11As` specify the check digits to be used when the *mod 11 checksum* equals 10 or 11, respectively. Default to X and 0, respectively. `startIndex`, `endIndex`, `checkDigitIndex` and `ignoreNonDigitCharacters` carry the same semantics as in `@Mod10Check`.
- `@NotEmpty`: Checks whether the annotated element is not null nor empty.
- `@NotBlank`: Checks that the annotated character sequence is not null and the trimmed length is greater than 0. The difference to `@NotEmpty` is that this constraint can only be applied on strings and that trailing whitespaces are ignored.
- `@Range(min=, max=)`: Checks whether the annotated value lies between the spe-

[7] EAN: International Article Number (formerly known as European Article Number)

cified `min` and `max` (inclusive).

- `@SafeHtml(whitelistType=, additionalTags=, additionalTagsWithAttributes=)`: Checks whether the annotated value contains potentially malicious fragments. With the `whitelistType` attribute a predefined whitelist type can be chosen which can be refined via `additionalTags` or `additionalTagsWithAttributes`.
- `@ScriptAssert(lang=, script=, alias=)`: Checks whether the given `script` can successfully be evaluated against the annotated element.
- `@URL(protocol=, host=, port= regexp=, flags=)`: Checks if the annotated character sequence is a valid URL according to RFC2396. If any of the optional parameters `protocol`, `host` or `port` are specified, the corresponding URL fragments must match the specified values. The optional parameters `regexp` and `flags` allow to specify an additional regular expression (including regular expression flags) which the URL must match.

A more detailed description about the use and supported data types of these constraints can be found in the Hibernate Validator documentation [7].

Custom Constraints

Besides using the Hibernate Validator built-in constraints, we also defined our own custom constraints. These custom constraints can range from relatively simple class-level 'value-in-range' 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. For instance, a class-level 'value-in-range' would allow to check whether the annotated value is between the currently defined `min` and `max` fields of the object, which might be dynamically defined and not have a fixed value. This would not be possible with the built-in `@Range` annotation, as it accepts only constant `min` and `max` attributes and is not able to perform any cross-field checks.

Defining a new custom constraint involves the following steps. The first step is the creation of the corresponding *constraint annotation*. The second step is to implement an appropriate *validator* that includes the conditions and checks that need to be satisfied in order to validate the constraint. Finally, the third and last step includes the definition of a default *error message* that should be displayed when the constraint is violated.

In our data quality framework, all the custom constraints are defined in the `FoodCASE_Lib` module, as the annotations need to be available to the entity beans. The corresponding validators are defined in the same package as the constraints and by convention are named using the constraint name followed by the word '`Validator`'. For instance, if the class that defines the constraint is named `Unique.class`, the corresponding validator will be named `UniqueValidator.class`.

As with any Java annotation type, the constraint annotation is defined by using the

`@interface` keyword. As demanded by the Bean Validation specification, a constraint annotation must define three attributes: `message`, `groups` and `payload`. Besides these three mandatory attributes, it is possible to define additional constraint-specific attributes, such as the commonly used `value` attribute. Additionally, the definition of the constraint annotation contains several meta-annotations, which specify important constraint characteristics, including the supported target element types (`@Target`) and the associated validator (`@Constraint`). An example of the definition of such a constraint annotation can be seen in listing 4.2.

Listing 4.2: An example of a simple class-level constraint annotation. The element type `TYPE` is used to define a class-level constraint, while the element type `ANNOTATION_TYPE` allows for the creation of composed constraints

```
@Target({TYPE, ANNOTATION_TYPE})
@Retention(RUNTIME)
@Constraint(validatedBy = {ValidMinMaxValidator.class})
@Documented
public @interface ValidMinMax {

   String message() default "The minimum value can't be greater
      than the maximum value";

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

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

The definition of the custom constraint annotation is always followed by the implementation of a validator that can validate any element that has been annotated with the constraint annotation. This validator is created as a class that follows the naming convention described above, i.e. {`AnnotationName`}`Validator`, and implements the `ConstraintValidator` interface. This interface defines two parameters that are set in the implementation, namely the annotation to be validated and the element type that can be validated. An example of such a validator can be found in listing 4.3.

4.2.2 Contracts

As already mentioned in subsection 3.1.3, in our framework we use the term *contracts* to refer to files in which we define the weight (importance) of each constraint and state whether the constraint is currently active or not. These contracts are created as `.properties` files, in which each property represents a constraint in the format `key=value`, where `key` is a unique constraint identifier and `value` the associated weight, optionally followed by comma and the word `active` or `inactive`, indicating the status of the constraint. If the status is omitted, the constraint is considered active.

Listing 4.3: An example of validator for the ValidMinMax annotation

```
public class ValidMinMaxValidator implements
    ConstraintValidator<ValidMinMax, TdsPoolValue>{

    @Override
    public void initialize(ValidMinMax a) {
    }

    @Override
    public boolean isValid(TdsPoolValue t,
        ConstraintValidatorContext cvc) {
      boolean isValid;
        // code that determines whether the constraint is satisfied
            (isValid = true) or violated (isValid = false)
        return isValid;
    }
}
```

The *unique constraint identifier* that is used as the property's key, is defined as follows:

{entityName}.[{fieldName}].{constraintName}

In the above definition, the fieldName is only included in the case of field-level constraints.

In our framework, contracts can be created and edited either via the administrative GUI, which was described in subsection 3.4.1, or by directly creating or editing the contract files using a text editor. Contracts are currently saved in a resource folder in the FoodCASE_Lib module, but could also be saved elsewhere if desired. A small example of the content of a contract file can be seen in listing 4.4.

Listing 4.4: Examples of entries in a contract .properties file

```
TdsPoolValue.ValidMinMax=80     //class-level constraint, active
TdsPool.name.Size=90,active     //field-level constraint, active
TdsPoolValue.minimum.Min=100,inactive  //field-level constraint,
    inactive
```

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 easily shared and exchanged between users and applications of the system. This decision was based on the fact that contracts are not just meant to be representations of the personal preferences of a user, but rather interchangeable input validation and analysis profiles, that can be easily made available to all interested users and applications.

4.2.3 Validation Groups

As explained in subsection 3.1.5, the use of *validation groups* enables the organization of constraints into logical units, which in turn allows the user to restrict the analysis or input validation to the constraints that they are interested in.

Validation groups are defined as simple *tagging interfaces*, i.e. interfaces without any methods. Hibernate Validator provides the default group `javax.validation.groups.Default`, which contains all the constraints that do not belong to any other group and is assumed for the validation if no other group is specified. This implies that if we wish to validate all the constraints in an entity, we need to specify as validation arguments all the groups that contain at least one constraint in the entity as well as the `Default` group. However, this process can be bothersome and error prone. Therefore, we created an extra tagging interface called `AllChecks`, which extends all the defined validation group interfaces as well as the `Default` one. By specifying the `AllChecks` group as the group to be validated, we can be sure that all the constraints that are declared in an entity will be validated. Additionally, we defined another tagging interface, `ValidationGroup`, that all custom validation group interfaces should extend. The `ValidationGroup` interface was defined to ensure that only appropriate interfaces that represent validation groups can be used as validation arguments. The resulting UML diagram depicting the group interface structure in our framework can be found in figure 4.1.

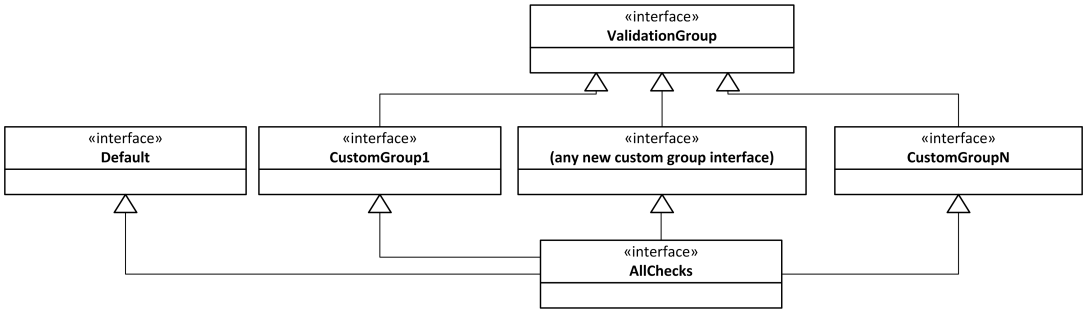


Figure 4.1: A UML diagram demonstrating the structure of the validation group interfaces in our framework. New validation groups should extend the `ValidationGroup` interface and should be extended by the `AllChecks` interface

All the above mentioned tagging interfaces, along with the custom validation groups that we have defined, are currently located in the `FoodCASE_Lib` module. To create a new custom validation group that is well integrated in our framework a user should follow the two steps described below:

1. Create a tagging interface that extends `ValidationGroup`.
2. Add the newly created interface in the list of interfaces that `AllChecks` extends.

Any new group that is defined this way can be included in the `groups` attribute of a constraint declaration (see listing 4.1) and thus will appear in the entity and group selection tree

(see figures 3.3, 3.11) during the analysis or input validation configuration.

4.2.4 Payload Weights

The use of *payload weights* in our framework has been described in subsection 3.1.6. This payload concept was implemented in the form of the `Severity` class, which contains the definition of all the *payload interfaces*, i.e. the *payload labels*, that were presented in table 3.2, such as `Info`, `Suggestion`, `Recommendation`, etc. These payload interfaces extend the Hibernate Validator `Payload` tagging interface. In addition, the `Severity` class includes a `Map`, which is used to map each of the payload interfaces (labels) to the corresponding payload weight, as seen in table 3.2.

The payload interfaces defined in the `Severity` class can be used on any constraint annotation, as seen in the example of listing 4.1. The `TDS Data Quality Analysis Bean`, which will be described in section 4.3, is able to identify which of the constraints have a specified payload interface and, if desired by the user, override the constraint contract weight with its payload weight during the analysis or input validation.

4.3 TDS Data Quality Analysis Bean

The *TDS Data Quality Analysis Bean* is an *EJB session bean* that contains all the methods that constitute the business logic of our framework. Its actual name is `TdsQualityAnalysisBean` and, like all the session beans in the FoodCASE system, it is located in the `FoodCASE_Server` module. This EJB is responsible for all the fundamental functionality of our framework, including constraint and constraint group validation, quality analysis and calculation of all types of quality indicators. Despite its name, it is not only used by the data quality analysis module, but also by the input validation and the administrative tools modules. Following the existing FoodCASE structure, this functionality can be invoked by other project modules, namely `FoodCASE_Client` and `FoodCASE_Admin`, via the `BeanBag` singleton.

Some of the main methods that this EJB bean provides include the following (the method parameters and return values are excluded):

- `calculateObjectQuality`: Calculates the quality of an entity object during input validation and returns its $IVQI$.
- `calculateQuality`: Calculates the $OAQI$ of the analyzed data during a data quality analysis and creates a textual summary of the analysis results.
- `constraintHasPayload`: Takes a constraint identifier String as an argument and checks the constraint's declaration to see if a payload has been defined for it.
- `createNewContractProperties`: 'Scans' all analyzable entities, i.e. the entities whose analysis is supported by our framework, for declared constraints and creates

a new *contract* `.properties` file with no weights defined.

- `getAllEntityObjects`: Returns a list of all objects of a specified entity type in order to be used for validation and analysis.
- `getAnalyzableEntities`: Returns a set of all the analyzable entities (i.e. entities whose analysis is supported by our framework).
- `getEntityValidationResults`: Validates the constraints belonging to the specified validation groups for all the objects of a specific entity type and creates a `ConstraintMap` (see subsection 4.4.2) to be used for further quality analysis and calculations.
- `getGroupsPerEntity`: 'Scans' the specified entity in order to discover the set of all validation groups that constraints in the entity belong to.
- `initializeWithProperWeight`: Only takes effect if the option to replace the constraints' contract weights with their payload weights is selected. If it is, it determines whether the specified constraint has declared a payload. If it has, it replaces the contract weight of the constraint with the declared payload weight in the `ConstraintMap` that the constraint belongs to.
- `isError`: Used during input validation, in order to determine whether a violation is an error based on the user-selected input validation settings.

In the remainder of this chapter, we will describe the main helper classes that are used by the data quality analysis EJB (section 4.4) and explain how the methods mentioned above are used in each context, namely during data quality analysis (section 4.5), input validation (section 4.6) and using the administrative tools to configure the framework's settings (section 4.7). Where needed, we will also provide the relevant implementation details. For the sake of brevity, from now on we will refer to the *TDS Data Quality Analysis Bean* as simply the 'analysis bean' or just the 'bean'.

4.4 Main Helper Classes

The data quality analysis EJB relies on the use of two main helper classes, which basically serve as the data structures that organize the data related to all the constraints, their weights and violation count for each entity. These helper classes are called `ConstraintValues` and `ConstraintMap` and are located in the `FoodCASE_Lib` module. We will now describe these two classes in detail.

4.4.1 ConstraintValues

The `ConstraintValues` class is used to organize data about important constraint-related values. Each individual constraint corresponds to one `ConstraintValues` object. The constraint-related values that are handled by this class are the constraint's weight, the number of constraint violations that were found in all entity objects, and information about whether

the constraint has an associated payload or not. It also holds the IDs of the entity objects that violate the constraint, which are used for the *data cleaning* function of the analysis visualizations.

4.4.2 ConstraintMap

The `ConstraintMap` helper class is used to organize all the constraint-related data that corresponds to one particular analyzable entity. Each `ConstraintMap` object represents the constraints of one entity, or more precisely, the constraints that belong to the selected validation groups for this entity. In detail, the main function of this class is to build and maintain a `HashMap`, which is used to map each eligible constraint to a `ConstraintValues` object, which represents important values associated with the particular constraint.

Despite its name, the `ConstraintMap` class is more than a simple map data structure. Apart from mapping constraint identifiers to their values, `ConstraintMap` is also responsible for loading the selected contract and reading its values. Additionally, it calculates the *entity-level analysis quality indicator* ($ELAQI$) and the *input validation quality indicator* ($IVQI$), and creates the string that contains the results of each entity-level analysis.

4.5 Data Quality Analysis Module

The function of our framework's *data quality analysis module* has already been described in section 3.2. We will now discuss the implementation details concerning this module.

4.5.1 General Information

The data quality analysis module takes the form of a Swing JFrame which is called `TdsQualityAssessmentFrame` and is located in `FoodCASE_Client`. This frame is depicted in figure 3.3. It consists of two parts: the `Analysis Configuration` part, which allows users to choose their desired analysis settings, and the `Analysis Results` part, which displays the results of the analysis in textual and visual form. From now on, we will refer to the `TdsQualityAssessmentFrame` as the 'analysis client' or, in the context of the current section, as just the 'client'.

4.5.2 Entity and Group Selection Tree Creation

The most interesting part about the `Analysis Configuration` section of the analysis module, is the *entity and group selection tree* (`(2)` in figure 3.3). This is a three-level *checkbox tree* that provides users with an overview of all the analyzable entities and their associated validation groups, and allows them to easily select the entities and groups that they want to

analyze. The choice of the check-box tree format for the representation of the analyzable entities and groups was based on the fact that this format allows for an easy and intuitive element overview and selection, and scales well when a large number of elements is involved. It is implemented as a *JIDE Common Layer*[8] `CheckBoxTree`, which is an open source component that provides the functionality we need.

To create this tree, we start with a root node representing all the analyzable entities and groups. Then, the analysis client calls the `getAnalyzableEntities()` method from the analysis bean, which returns a set containing all the entities whose analysis is supported by our framework. For each of these entities, a corresponding entity node is added to the tree. Then, for each entity, the client invokes the bean's `getGroupsPerEntity()` method and receives a set of groups in return. For each of these groups, a group node is added at the corresponding entity node, which concludes the creation of the tree.

4.5.3 Quality Analysis and Calculation

The analysis process begins as soon as the `Start Analysis` button of the analysis client frame (`(3)` in figure 3.3) is pressed. An overview of all the steps that are involved in the analysis is illustrated in figure 4.2.

As seen in this figure, after the analysis is initiated, the analysis client checks whether at least one entity as well as a valid contract is selected. If not, the user is notified and the client resumes its stand-by state. If the selection conditions are fulfilled, the client parses the entity and group tree in order to find out which entities and which of their groups should be analyzed. For each of the selected entities, the analysis bean's `getEntityValidationResults()` method is invoked. This method is responsible for all parts of the entity's analysis. After retrieving all the objects that belong to the entity, by invoking the bean's `getAllEntityObjects()` method, it creates an up-to-date `ConstraintMap` that consists of only the constraints that belong to the selected validation groups of the entity. It also makes sure that the weight of each constraint has the correct value, either according to its payload or the selected contract, by invoking the bean's `initializeWithProperWeight()` method. Then it validates all the objects of the entity and updates the `ConstraintValues` of each constraint according to the number of discovered violations. After the validation is completed, the results are analyzed and the *entity-level analysis quality indicator* ($ELAQI$) is calculated. Finally, it builds the entity-related analysis results message that will appear in the client. The method returns the created `ConstraintMap` object, which at this point contains all the information about the analysis of the entity. This `ConstraintMap` object is added to a `ConstraintMapList`, which holds the `ConstraintMap` objects of all the selected entities.

After this process is completed for each selected entity, the client invokes the bean's `calculateQuality()` method. This method, combines the analysis results represented by all the selected entities' `ConstraintMap` objects and calculates the *overall analysis quality indicator* ($OAQI$). It also builds the overall analysis results message which is dis-

[8] http://www.jidesoft.com/products/oss.htm

played on the client, along with the entity-specific analysis results messages.

After the analysis is completed, the client creates the visualizations and makes the previously hidden visualization tabs visible.

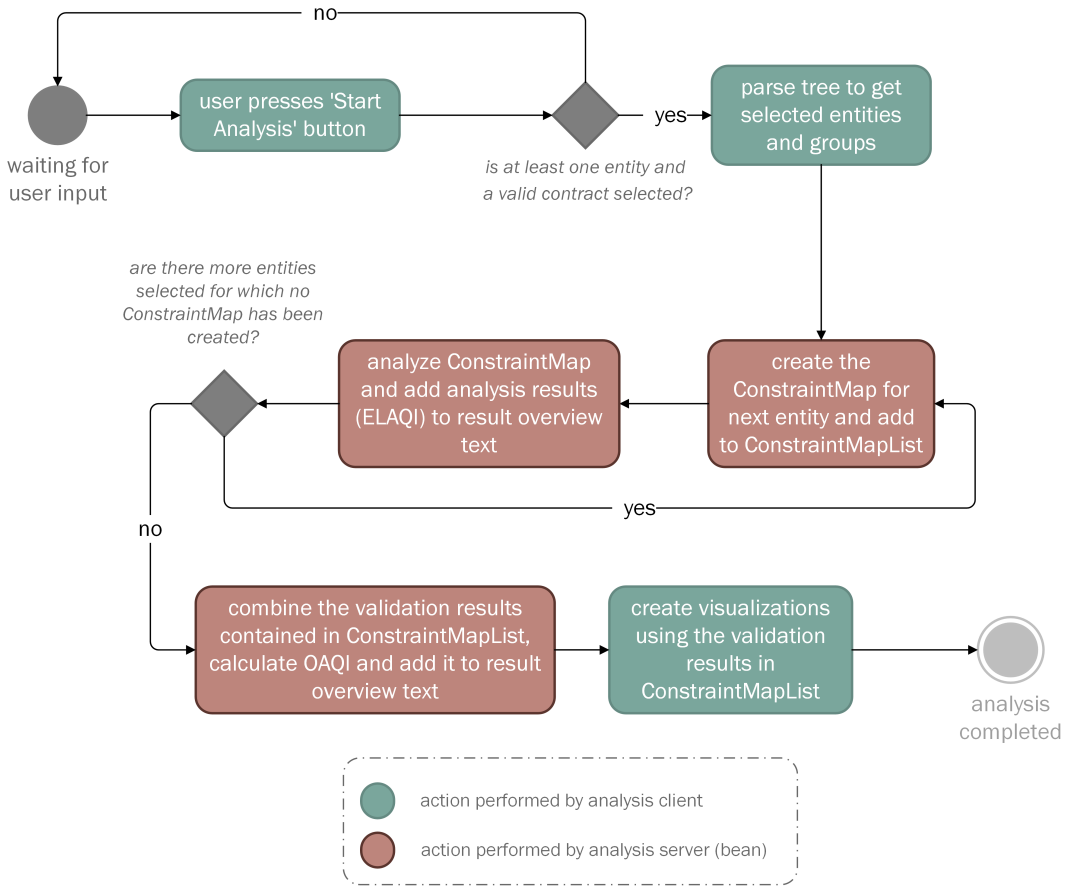


Figure 4.2: A flowchart demonstrating the steps that are involved in the analysis process

4.5.4 Visualizations

As mentioned in the previous subsection, the visualizations are created by the analysis client after the analysis bean has completed the analysis for the selected entities and validation groups, by using the information contained in the `ConstraintMap` objects. All three visualizations are created in a similar manner, utilizing a `ChartCreator` class. This class first creates the chart's *dataset* based on the type of the chart and the analysis information from the `ConstraintMapList`. Then, with this dataset, it creates the chart using the corresponding JFreeChart method (see subsection 4.1.4) and adds a listener that waits for the user to click in one of the graph's sections.

As soon as the user clicks in one of the violation-representing graph sections, the

`ChartCreator` creates a `ViolationInfoPanel` object, which in turn creates and displays the list of all the violated constraints that exist in the data represented by the clicked graph area. An example of this list is displayed in annotated section `(2)` of figure 3.5. This list is sorted by the graph-corresponding `Comparator`, either by the constraint weight or by entity name (and then weight). Selecting a violated constraint from the list leads to the creation and display of the list of entity object IDs that violate this constraint (`(3)` in figure 3.5), which is achieved with the help of the `ConstraintValues` object of the selected constraint, which creates this list during the data analysis. Double-clicking on one of these entity object IDs opens the corresponding entity object detail frame (figure 3.8), which is achieved using the existing FoodCASE infrastructure.

As explained above, the visualizations are created using the information that was gathered during the analysis. This implies that the information presented in the graphs might become 'outdated' if the user corrects some of the violation sources by editing an entity detail frame. To make the user aware of which of the entities have been edited, every time that a user opens an entity object detail frame via the analysis visualizations, the object's entity type and ID are added to a static 'list' (it is actually a more complicated data structure, handled by the `EditedEntityObjectIds` class, but we refer to it as a list for the sake of simplicity). This list is then used by a custom renderer (`EntityIdRenderer`), in order to display the entity object IDs contained in this list in *violet* color (`(1)` in figure 3.6).

4.6 Data Input Validation Module

The use of our *data input validation module* has been described in section 3.3. In the current section we will discuss the implementation of this module within FoodCASE. The input validation module is implemented as part of `FoodCASE_Client`, as the data that we want to validate is entered via the client.

4.6.1 General Information

To ensure optimal integration with FoodCASE, our implementation is based on the already existing input validation infrastructure. Figure 4.3 depicts a UML diagram of the main classes and interfaces involved in the input validation process, distinguishing between the previously existing types (in *blue*) and the new class we created for our framework (`BeanValidator`, in *orange*). In addition to these classes, another previously existing component that we adopted is the `ValidationPanel`, which is added at the bottom part of any analyzable entity detail frame and displays the results of the input data validation and evaluation.

We will now briefly describe the already existing FoodCASE input validation infrastructure. The `Validator` interface outlines the behavior that every validator should implement, including methods to determine whether the validated object has errors or warnings and methods to retrieve the associated error or warning messages. The abstract class `AbstractValidator` offers property change support for validators, while

`AbstractComponentValidator` registers the appropriate listener for each type of validated Swing component.

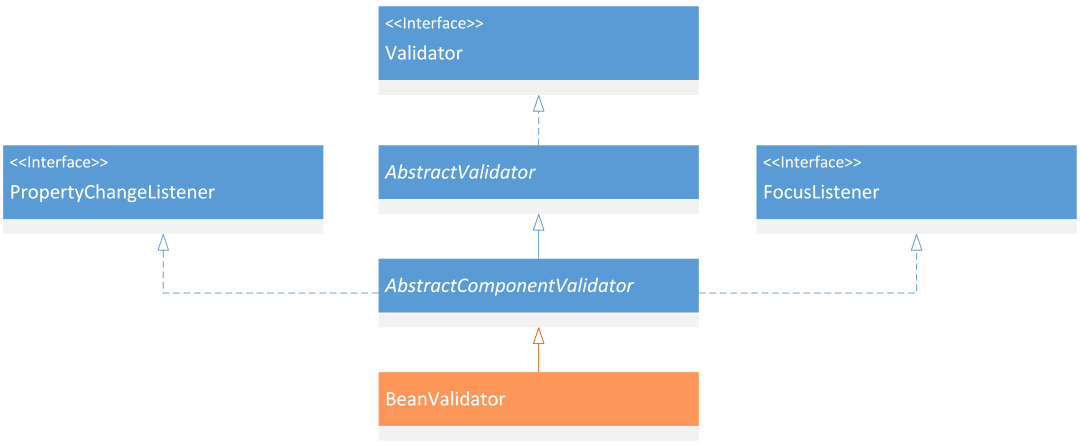


Figure 4.3: A UML diagram demonstrating the structure of the main input validation-related classes and interfaces in our framework. The only class that was created by us is depicted in *orange* color (`BeanValidator`) while the rest of the classes and interfaces (in *blue* color) were already part of the previously existing input validation framework

4.6.2 Input Validation and Quality Evaluation

The `BeanValidator` class is responsible for validating the entity constraints during data input, performing an evaluation and analysis of the input data and showing the results of the evaluation to the user, in the form of textual information and visual feedback. To enable input validation on an entity, we have to create and add a `BeanValidator` object to the `ValidationPanel` of the entity's detail frame. The `BeanValidator` constructor takes as arguments the entity frame's validated fields, i.e. the fields whose input needs to be validated, as well as the type of visual feedback that we want the user to receive when a violation is encountered, which will be discussed in more detail in subsection 4.6.3.

Every time that the focus is shifted from one of the validated fields, the `BeanValidator` object uses the *HV* `validate()` method to validate the input based on the user-defined settings. The result of this validation is a set of all the violations encountered. If this set is not empty, the `BeanValidator` object performs an evaluation of the input data in a manner similar to the data quality analysis process, which was described in subsection 4.5.3. This evaluation determines the visual feedback that the user receives on the validated entity fields (`(1)` and `(2)` in figure 3.8), as well as the violation messages that are displayed in the `ValidationPanel` at the bottom of the detail frame in plain font (`(3)` and `(4)` in figure 3.8). It also calculates the $IVQI$, which is displayed at the `ValidationPanel` in bold font (`(5)` in figure 3.8).

In detail, the `BeanValidator` object creates a `ConstraintMap` representing all the constraints that the user wishes to validate according to the currently selected settings. This is

achieved by invoking the analysis bean's `getNewOrResetConstraintMap()` method. Then, for each discovered violation, it invokes the bean's `isError()` method, which determines whether the violation is an error, i.e. the weight of the corresponding constraint is higher than the WtET, or a suggestion, and adds it to its maintained list of errors or warnings accordingly. Finally, it invokes the bean's `calculateObjectQuality()`, which returns the $IVQI$ of the entity object that is currently being edited.

4.6.3 Validation Feedback

Support of Different Visual Feedback Types

An extension that we added in the previously existing input validation infrastructure is the support of different types of visual feedback in case of violations, or *violation feedback*. In the already existing framework, fields that contain data that violates one or more constraints are colored in a *red* background, as seen in figure 4.4-1. In order to allow for more subtle violation feedback options but not interfere with the previously existing framework, we created an enumeration of four different violation feedback types (`BACKGROUND NONE`, `OUTLINE_BACKGROUND`, `OUTLINE`) that correspond to the four violation feedback versions illustrated in figure 4.4. When an input validator object is created (such as our `BeanValidator` described in the previous subsection), one of the enumeration types can be optionally given as an argument, which results in the desired type of violation feedback.

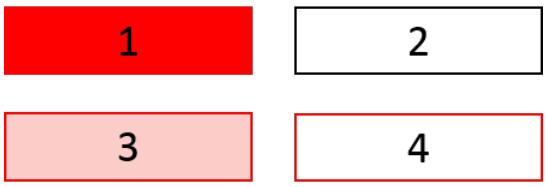


Figure 4.4: The different input validation visual feedback types supported by our framework: 1: red background, 2: no feedback, 3: light red background with red outline, 4: red outline

Additionally, we added support for *orange* highlighting of fields that contain warnings.

A Drawback of Using a Single Validator

In the previously existing framework, every constraint that had to be validated during data input required the addition of the corresponding validator to the detail frame. In our framework, the validation of all the constraints can be handled by a single `BeanValidator` object. This feature not only facilitates the maintenance of the input validation module, but also enables its expansion with new constraints in an effortless manner. However, using a single validator implies that when one of the constraints is violated, the validator will fail to validate the entity. This failure to validate will 'propagate' to all the constraints that use this

validator, even if they are not violated. This leads to the occurrence of violation feedback in all the fields that are validated during input, even if the data that has been entered in them is correct.

A workaround to this issue is to provide a mapping between constraints and fields, and make sure that the violation feedback is displayed only on the fields that are mapped to the violated constraints. Even though this leads to correct violation feedback, it requires additional maintenance and extensibility effort, thus counteracting the benefits of using a single validator.

4.7 Administrative Tools Module

Our data quality management framework's *administrative tools module* was implemented as part of the `FoodCASE_Admin` module, as it is meant to be accessible only by administrators or 'power' users. We already described the use of the administrative tools module in section 3.4. In the current section we will discuss some details about its implementation.

4.7.1 General Information

As the rest of the FoodCASE GUI components, the GUI for our administrative tools was implemented using the *Java Swing* toolkit. In order to integrate our administrative tools to the `FoodCASE_Admin` module as seamlessly as possible, we used the same class structure as the already existing quality framework administrative tools. In detail, we implemented our administrative tools in the form of a Swing tabbed pane, the `TdsDataQualityAnalysisAdminTabbedPane`. Currently, this tabbed pane contains three tabs, which take the form of three Swing panels: `TdsContractEditingPanel`, `TdsGeneralSettingsPanel` and `TdsInputValidationSettingsPanel`. The function of each of these tabs was described in section 3.4 and their implementation will be discussed in the following three subsections.

4.7.2 Contract Editing and Creation Panel

The main part of the *contract editing and creation panel* (`TdsContractEditingPanel`), depicted in figure 3.9, is the contract-editing table, which allows users to overview the constraints that exist in the framework, edit their corresponding weights and activate or deactivate them for each existing contract. It also allows the users to create new contracts in a simple and effortless way.

To implement this table, we created a custom table model, the `ContractTableModel`, which extends the Swing-defined `AbstractTableModel`. When a user loads an existing contract for editing, our table model is responsible for parsing the contract file and 'translating' the information it contains into the corresponding table column field. Each element in the contract file corresponds to one row in the table. The only table column whose

information is not directly taken from the contract file is the 'Can be overridden' column, which basically indicates whether a payload label has been defined for the specific constraint. The value for this column is retrieved by using the analysis bean's `constraintHasPayload()` method.

To create a new contract file, the panel invokes the analysis bean's `createNewContractProperties()` method. This method parses all the analyzable entities in order to discover their constraints and returns a `Properties` object that has all the existing constraints with no weights defined. This object is saved as the new contract file and is then loaded to the table as described in the previous paragraph.

In both cases, the user changes are saved in the contract `.properties` file, using the `store()` method of the `Properties` class.

4.7.3 General Validation and Analysis Settings

The *general validation and analysis settings* panel (`TdsGeneralSettingsPanel`) is used to allow the user to configure general settings of the data quality management framework according to their wishes. Therefore, this panel must be able to save the user preferences and configurations and ensure that they are available to the modules that they refer to at any time. To achieve that, we used the already existing `storeSettings()` method of the FoodCASE `UserBean`, which saves the selected settings in an appropriate table in the FoodCASE database.

4.7.4 Input Validation Settings

The creation of the entity and group tree is performed dynamically, in the same way that we described in subsection 4.5.2. Saving the tree selections in the database could be a difficult and error-prone procedure, as we cannot know in advance how many entities and groups the user might select. Therefore, we decided that the simplest and most reliable alternative would be to serialize the tree object and then load it once when the `FoodCASE_Client` module is run. This is handled by a helper class (`ValidationEntitiesGroups`), which is located in the `FoodCASE_Lib` module. This class loads the serialized tree and then parses it into a static map structure, which is then used by the `BeanValidator` class in order to decide which constraints it should validate.

4.8 Open Implementation Issues

In this section, we will discuss some of the issues that we encountered during the implementation of our framework that remain unresolved.

4.8.1 Displaying Previously Selected User Configurations

As discussed in sections 3.4 and 4.7, the *administrative tools* module is used by authorized 'power' users in order to change important settings related to the data quality analysis and input validation. These settings are saved in the database, with the exception of the input validation entity and group tree, which is serialized.

Even though the various settings are loaded and used correctly during data quality analysis and input validation, the display of previously selected settings in the administrative tools GUI has been problematic in some cases. For instance, the various checkbox values are not always displayed correctly. Similarly, displaying the previously selected input validation entity and group tree choices was inconsistent and had to be disabled. Even though this issue does not affect how the setting values are loaded in the modules in which the settings are actually used, it can be a source of confusion for system administrators, as it prevents them from recognizing which settings have been used until that point.

4.8.2 Loading of the Database Settings by the TDS Quality Analysis Bean

As described in sections 4.5 and 4.6, both the data quality analysis and input validation modules rely on the business logic that is implemented in the analysis bean (`TdsQualityAnalysisBean`). When one of these modules invokes a method from the analysis bean, the bean needs to be aware of the various user-specified input validation and quality analysis settings that are relevant to the method being invoked. Such settings include the selected validation contract, the default weight and the weight-to-error threshold. To allow the bean to directly load these settings from the database, we tried to inject the `UserBean` into the `TdsQualityAnalysisBean`. However, this resulted into several deployment errors. Due to time constraints, we were unable to thoroughly investigate and correct the source of these errors. As a workaround, the required settings are currently loaded by the module that needs to use one of the analysis bean's methods, i.e. either the data quality analysis module or the input validation module, and then passed as an argument to the invoked method. Even though this workaround works as required, it is a rather inelegant solution and should be addressed in the future.

5

Comparison with FoodCASE Food Composition Data Quality Management Framework

As we have already mentioned in the course of this report, the framework that we implemented is not the only existing data quality management framework (DQMF) in the FoodCASE system. The previously existing DQMF, conceptualized and developed mainly by Presser [12] and Mock [10], is responsible for the same main tasks as our framework, namely data quality input validation and overall data quality analysis. Currently, both frameworks are part of the FoodCASE system, but each of them targets a different part of the data: while the previously existing framework manages the quality of the food composition data, our framework focuses on the data related to total diet studies (TDS).

In this chapter, we will present an overview on how the two frameworks compare to one another, by focusing on the additional features that our framework provides.

5.1 General Concept

Both frameworks view data quality as a constraint problem and provide options for assigning an importance weight to the constraints in the system. Furthermore, both use this weight to calculate a quality score for the data. However, our framework introduces the concept of *contracts*, which can be easily created, edited and shared between users. Besides assigning weights to constraints, contracts can also be used to completely deactivate (or reactivate)

constraints, something that is not possible in the previous framework. In addition to contracts, our framework allows the user to specify the entities and groups of constraints that they wish to validate. These configurations are not only applied to the data quality analysis, but to the input validation as well.

5.2 Constraints

In our framework, constraints are defined as Java annotations and validated in a unified manner. In contrast, in the previous framework, there is a distinction between constraints that are defined for the data quality analysis, which take the form of database SQL constraints, and constraints that are defined for the input validation, which take the form of Java classes. This implies that the same constraint might need to exist in two forms, which can lead to maintenance issues and inconsistencies. Additionally, the validation of each constraint in the previous framework requires the explicit invocation of each of the corresponding validator methods, something that is not required in our framework, as the validation is handled in a unified way.

Furthermore, the contract concept in our framework implies that constraints can be easily deactivated and reactivated both for the analysis and input validation. Also, the distinction between *warnings* and *errors* is dynamic, as it depends on the weight of the constraint and a user defined weight-to-error threshold. Additionally, the *payload* concept allows the definition of a payload weight that, if desired, can override the corresponding constraint contract weight. Finally, the user can define a default weight that should be used for new constraints or any constraint with no defined weight. All the features that were mentioned in this paragraph are not available in the food composition DQMF.

5.3 Input Validation

Even though both frameworks use the same underlying validation and violation feedback infrastructure, the concept and implementation of input validation is very different between the two frameworks. The first big difference is that while in the food composition DQMF the user cannot configure the input validation, in our framework it is just as configurable as data quality analysis. The user can select a contract for the input validation, which affects the constraint weights as well as whether they are activated or not. As with the analysis, the user can also choose particular entities and groups to be validated, something that is also not possible in the previous framework. Also, the user can select the warning-to-error weight threshold and whether they want to display the violations of constraints that have a lower weight than this threshold. Finally, a quality indicator ($IVQI$) is calculated and displayed during input validation as well, which can motivate the user into correcting as many input data issues as possible.

5.4 Data Quality Analysis

The two frameworks' differences regarding the underlying concept of the data quality analysis were discussed in section 5.1. Even though both frameworks provide the possibility to drill down through the data in order to discover and correct potentially existing issues, our DQMF allows the user to select the entities and groups they want to analyze *before* the analysis. This can speed up the analysis and help the user identify the issues they are interested in faster and without the need of many 'drill-down steps'. Also, the visualizations in our framework are simpler, focus on violations and are used only for the general overview of the encountered violations. More detailed constraint-specific information is provided in textual form, which we consider as a more appropriate way to represent information in this level of detail.

6
User Evaluation

6.1 TDS-Exposure Workshop

A prototype of this work was presented at the TDS-Exposure workshop that took place during the third TDS-Exposure General Assembly in Zurich between the 9th and 13th of February 2015. This workshop was attended by several food science experts from around Europe. The participants were of many different positions such as researchers, compilers, laboratory workers and investigators. The presentation consisted of an explanation of the main concepts and ideas behind our framework, followed by a demo of the three implemented modules (input validation, data quality analysis and administration module) in the FoodCASE system.

In order to evaluate how our DQMF concept and implementation is perceived by its target users and to understand ways in which it could be improved in order to be in accordance with the users' wishes, at the end of the presentation the participants were asked to complete a questionnaire about several areas of the framework. The questionnaire was answered by 15 of the participants. Even though this number might seem low, these participants constitute a big part of all the food science experts within the TDS-Exposure project in Europe.

In this chapter, we will present the most interesting results that are revealed by an initial analysis of the questionnaire answers. This analysis is currently done in the context of individual answers, which means that we have not yet analyzed, compared and correlated the relationships between groups of answers or answers of users of different positions. The full list of questions contained in our questionnaire is presented at the appendix C.

6.2 Questionnaire Findings

In this section, we will present our findings organized according to the general question category that they correspond to. In all the questions whose answers we present, the participants could choose 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.

6.2.1 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	-
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 6.1: Findings related to the concept of contracts

6.2.2 Input Validation

#	Finding	SA%	A%	N%	D%	SD%
1	100% of the participants find a quality indicator during input validation ($IVQI$) 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	-
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 6.2: Findings related to input validation

6.2.3 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 analyze the data quality according to certain constraint groups	6.67	66.67	20	6.67	-
3	60% would like to be able to only analyze 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 6.3: Finding related to data quality analysis

6.2.4 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 6.4: Findings related to user access rights

6.2.5 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 6.5: Findings related to constraints

6.3 Overview of Key Findings

We will now present a list of the most interesting findings that emerged from our initial analysis of the participants' answers.

1. The findings of table 6.1 indicate that the majority of the participants agree with most aspects of the contract concept as it is defined at the moment. In detail, they like the idea of assigning weights to constraints (finding #1), they find the 0-100 scale intuitive (#2) and would like to be able to share and exchange contracts with other users (#4).

2. According to the table 6.1, the participants are generally favorable towards the payload concept (finding #6), while the majority of them doesn't mind if the payload weight can only be defined by developers (#7).

3. Being able to deactivate and activate constraints at runtime has both supporters and opponents (table 6.1, finding #5).

4. All of the participants consider having a quality indicator during data quality analysis useful and the majority of them would like a visual representation of quality as well (table 6.2, findings #1, #2).

5. If no contract is selected, the participants prefer the use of a default contract rather than considering all constraints equivalent (table 6.2, findings #3, #4).

6. Table 6.2 indicates that during data input, the participants would generally like to be able to save the data only if no severe violations occur (findings #5, #6). When violations do occur, most of them would like the system to provide an example of appropriate data (#8), but they wouldn't like it to insert the most probable value by default (#9).

7. Table 6.3 indicates that most of the participants find being able to perform an overall data quality analysis useful (finding #1). They also like the idea of being able to select the entities (#3) and constraint groups (#2) that will be analyzed.

8. As far as the analysis results are concerned, while the participants like having both a textual and visual representation of the results, they indicate a higher preference towards the visual representation (table 6.3, findings #4, #5).

9. The participants would like the data quality analysis to be available to any user, while they believe that only authorized users should be able to change the warning-to-error threshold and disable the appearance of warnings during input validation (table 6.3).

10. According to table 6.5, most participants find it acceptable if new constraints can only be defined by developers (finding #1). While the majority of the participants think that the definition of constraints at runtime is not necessary, approximately one quarter of the participants disagree (#4).

11. Most participants like being able to organize constraints into groups and would like to be able to define customized constraint groups at runtime (table 6.5, findings #2, #5).

12. While a big part of the participants find it acceptable if constraints can only be assigned to groups by developers, there is a non-negligible part (20%) that believes that assigning constraints to groups should be available to users too (table 6.5, finding #3).

Overall, the results of the questionnaire are very encouraging for our framework, as most of the findings indicate that the participants of the workshop agree with the majority of the framework's aspects. The answers also indicate that some participants are skeptical about the degree of freedom that should be available for the analysis and input validation and would generally prefer to restrict the access to important settings to power users only. Finally, most of the participants are satisfied if constraints, payloads and validation groups can only be defined by developers.

7 Extensibility

In this chapter we will discuss some of the ways in which our framework could be extended. It is important to note that the focus of this chapter is not on planned or currently necessary extensions, but rather on additional or alternative functionality that could be easily implemented if new requirements arise. The future work and planned extensions for our framework will be discussed in section 8.3.

7.1 Use of Different Data Quality Calculation Schemes

To calculate the various types and levels of quality indicators, we currently use the measurement and calculation scheme that was described in subsection 3.1.2. Our current calculation scheme is mainly based on the use of the *weighted arithmetic mean*. According to this scheme, depending on the type of quality indicator that we want to calculate, we use a different type of weight. Thus, the weights used for calculating $IVQI$ and $ELAQI$ are the *constraint weights*, while the weights used for calculating *OLAQI* are the *number of entity objects*.

While we believe that the constraint weights should in any case be part of the quality calculations, we acknowledge that some users of the framework might be interested in applying more sophisticated quality calculation schemes that possibly take additional factors into account, such as time-related conditions or additional entity importance indicators. On the other hand, it is also possible that some users consider that the number of objects that belongs to each entity should not be a factor in the $OLAQI$ calculation (described by equation 3.7). In this case, the $OLAQI$ would be equivalent to the simple arithmetic mean of the different entities' $ELAQI$ and would be calculated by the equation 7.1, in which n is the number of different

entity types and $ELAQI_i$ the entity-level analysis quality indicator of entity type i.

$$OAQI = \|\frac{1}{n}\sum_{i=1}^{n} ELAQI_i\| \tag{7.1}$$

To continue with the example of subsection 3.1.2, if we consider the entities presented in table 3.1 and assume that $ELAQI_A = 86$ quality points, while $ELAQI_B = 70$ quality points, the $OLAQI$ in this case will be calculated by equation 7.2 to be 78 quality points, while by taking the number of objects per entity into account (as in the example equation 3.8), it was calculated to be 75 quality points.

$$OAQI = \|\frac{86 + 70}{2}\| = 78 \tag{7.2}$$

If deemed necessary by the users, extending our system to support alternative calculation schemes is possible without the need of extensive changes in the system. Depending on the type of scheme that is desired, the only methods that are involved in the current calculations and would need to be altered are the analysis bean's `calculateObjectQuality()` and `calculateQuality()`, as well as the `calculateEntityQualityWeightedMean()` contained in the `ConstraintMap` class.

7.2 Additional Data Quality Analysis Visualizations

Our data quality analysis module currently offers three options for the visualization of the analysis results. These visualizations were described in subsection 3.2.1. If desired by the users, the framework could be extended by adding additional visualization views of the analysis results. For instance, some users might wish to have an overview of the analysis results organized by the group that the violated constraints belong to, or they might wish to have a violations per entity view that is normalized by the number of objects in the entity.

The effort for adding a visualization view that has a similar structure to the existing views, would mainly depend on the effort required for acquiring the data set that is needed and tuning the details related to the desired visualization graph type. The rest is for the most part taken care of by the existing framework infrastructure and the functionality provided by the JFreeChart API.

Of course, it is possible that in order to add more complicated visualizations that are of a significantly different nature to the already existing ones, more effort and additional extensions are required.

7.3 Support for Method Constraints

As discussed in subsection 4.2.1, our framework currently supports only bean constraint validation, i.e. field, property and class constraints. If deemed necessary, the framework could be extended to support method constraints as well.

To achieve that, there are a few additions that should be performed in the analysis bean. More specifically, any method that contains some kind of scan through the analyzable entity constraints should be extended in order to scan for method constraints as well. Currently, the methods that contain such a scan are `initializeNewConstraintMap()`, `createNewContractProperties()`, `constraintHasPayload()` and `getGroupsPerEntity()`.

Besides the above, it might also be needed to alter the definition of the unique constraint identifier, mentioned in subsection 4.2.2, which would possible require appropriate adjustments to the contract table and its model in the administrative module.

Having completed the above extensions, adding a new method constraint would be similar to adding a bean constraint, as described in subsection 4.2.1 and would require adding the method constraint definition, validator and declaration.

8 Conclusion

This chapter concludes this thesis report by providing a summary of the work that was presented, followed by a recap of our main contributions and an outline of the future work that could be done in the scope of the introduced data quality management framework (DQMF).

8.1 Summary of Work

8.1.1 Concept

In this thesis, we introduce a novel concept for managing data quality in information systems. This concept is based on the notion that data quality could be regarded as a *constraint problem*. This means that we could quantify and measure data quality in an information system based on the degree that the data complies to certain *constraints*. These constraints are basically the rules that the system has to follow in order to be considered of high quality.

Our concept takes into account that not all constraints should be equivalent as indicators of quality. For this reason, we decided that each constraint should be assigned a *weight*, which is a representation of the constraint's importance. We also acknowledge that the importance of each constraint might vary for different users and different applications of the information system. For this reason, we introduce the concept of *contracts*, which are files that associate every constraint to its own weight. A user can create one or more contracts to be used for different applications. In addition, the same contract can be shared and exchanged between different users.

Our concept also defines its own quality measurement and calculation scheme and scale.

According to this scheme, constraint weights can take integer values from 0-100. Using these weights, we define different levels of quality indicators to represent the quality according to an individual constraint ($CLAQI$), or the quality of individual entity objects ($IVQI$), entities ($ELAQI$) or the entirety of the data that can be analyzed by our framework ($OAQI$). These quality indicators are calculated based on the concept of the *weighted arithmetic mean*, and similar to weights, can take an integer value of 0-100 data quality points.

To provide additional flexibility and configuration capabilities, our framework allows organizing constraints into *validation groups*. Additionally, it defines certain *payload weights*, which are weights that can optionally be assigned to constraints at the time of the constraint declaration, and override the contract weight for the constraints that declare them. Finally, the framework enables the user to set a specific *weight-to-error threshold* (WtET). The WtET can be used to distinguish between two types of violations, warnings and errors, and take a different action based on the type of violation that is encountered.

8.1.2 Data Quality Analysis Module

To take full advantage of the capabilities that are offered by our concept, we propose a *data quality analysis module* that is responsible for analyzing the part of the data that the user is interested in at the time of the analysis according to the configurations and settings chosen by the user. In detail, this module allows the user to choose the *entities* and *constraint groups* that they want to analyze.

The analysis results are then displayed in textual and visual form. The textual form of the results consists of simple statistics about the constraints that were validated as well as the $ELAQI$ and $OAQI$ quality indicators. The visual form of the results consists of three visualization views: the violations per entity pie chart view, the violations per entity bar chart view and the constraints by weight range stacked bar chart view. All three visualization views offer 'drill-down' functionality, which allows the user to easily find more information about the violations they are interested in and even edit the affected entity objects in order to correct the causes of the violations.

8.1.3 Data Input Validation Module

The *input validation module* that we propose as part of our framework informs the user about potential constraint violations that they might introduce during data input. Being aware of these violations can help the user to avoid them, which in turn can result into data of higher quality. The user is informed about the violations in two ways. First, the user receives visual violation feedback in the form of highlighting of the affected data input fields, in one of the ways illustrated in figure 4.4. Secondly, the user also receives textual feedback about the violations, which takes the form of a color-coded list of the encountered violations as well as the quality indicator ($IVQI$) of the currently validated object. The color of the text represents the severity of the violations and can be *red* for errors, *orange* for warnings and *green*, if no violations were encountered.

Our input validation module offers the same level of flexibility and configurability as our proposed data analysis module. In detail, the users can select which specific entities and constraint groups they want to have validated during data input, as well as the contract that should be used for the validation. They can also select the WtET that should be used to distinguish between errors and warnings, as well as whether they want the system to display warnings or just errors.

8.1.4 Administrative Tools Module

In order to allow the user to easily configure the framework according to their wishes, our framework proposes an *administrative tools module*. This module consists of three parts, or *panels*. The first panel enables the creation of new contracts or the editing of existing ones. By editing the contract files, the user is able to change the importance of each constraint and, if required, to deactivate or reactivate certain constraints. The user is also able to identify which of the constraints are declared with the use of a payload weight.

The second panel of the module allows the user to configure settings that are used by both the analysis and input validation modules. In detail, via this panel the user is able to define the default weight that should be used for constraints which do not have a corresponding contract weight, as well as the WtET that should be used for the distinction between warnings and errors.

Finally, the third panel can be used for the configuration of the input validation module. More specifically, it enables the user to select the contract that should be used during the input validation as well as the entities and constraint groups that should be validated. It also allows the user to define whether warnings should be displayed or omitted during input.

8.1.5 Implementation for the FoodCASE System

All of the features of the introduced concept along with the three corresponding proposed modules (data quality analysis module, data input validation module and administrative tools module) were implemented for the FoodCASE food science data management system. The implementation is written in the Java language and relies on the use of Bean Validation for the definition, declaration and validation of the constraints as well as for the implementation of validation groups and payloads. More specifically, we use the Hibernate Validator API (version 5.x), which is the reference implementation of Bean Validation 1.1. The GUIs are implemented using *Java Swing*, while for the data quality analysis results' visualization we use the free *JFreeChart* chart creation library. This implementation is currently fully embedded in FoodCASE.

8.1.6 User Evaluation

Our framework was presented at the TDS-Exposure workshop that took place during the third TDS-Exposure General Assembly in Zurich. After attending a presentation and demonstration of the framework, the participants, consisting of several food science experts from around Europe, were asked to complete a questionnaire related to the framework.

Our initial analysis of the questionnaire results was very encouraging, as they generally indicated that the participants were in accordance with most aspects of the framework.

8.2 Contribution

The main contributions of this thesis can be outlined as follows:

1. Creation of a novel *constraint-based data quality concept* which:
    - provides an intuitive quantified view of quality by using constraints as data quality indicators
    - allows users to decide how important for quality each constraint should be
    - allows grouping constraints into groups
2. Utilization of *different types and levels of constraints*:
    - currently only field-level and class-level bean constraints are used, but the use of method constraints is also possible
3. Introduction of the concept of *contracts* in order to allow the user to:
    - define different analysis and input validation profiles in the form of contract files
    - share and exchange contract files with other users
    - change the importance of constraints at runtime
    - deactivate and reactivate constraints at runtime
4. Introduction of a novel *data quality measurement and calculation scheme* which:
    - is based on an intuitive 0-100 data quality points scale
    - defines different types and levels of constraints as data quality indicators
    - uses the weights of the constraints in the quality calculation
5. Introduction of a *payload weight* concept which:
    - enables the declaration of non-changeable constraint payload weights that can override the corresponding constraint contract weights
    - allows the user to decide whether they wish the payload weights to override the constraint weights

6. Conception of a *flexible and highly configurable data quality management framework* (DQMF) which:
    - allows the user to restrict the quality analysis and input validation only to the entities and constraint groups of interest
    - allows the user to choose the contract that should be used for the analysis and input validation at any time
    - provides the ability to manage and validate constraints in a unified way, both for data analysis and input validation
7. Proposal of a *data analysis module* which:
    - helps the user configure the analysis as desired
    - provides both textual and visual representations of the analysis results
    - is equipped with 'drill-down' capabilities, which assist the user in deciding the data areas that need to be improved and allow them to easily discover and edit individual entity object violations
8. Proposal of an *input validation module* which:
    - is just as configurable as the data quality analysis module
    - provides input validation feedback in both visual and textual form
    - allows the users to disable or enable warning messages
9. Proposal of an *administrative tool* which allows the user to:
    - configure every characteristic of the DQMF via an easy-to-use GUI
    - create new contract files, or view and edit existing ones in a convenient manner
10. Implementation of all the above-mentioned concepts, features and modules and creation of the FoodCASE DQMF which:
    - was evaluated by FoodCASE users and received highly positive feedback

8.3 Future Work

We will conclude our report by discussing some of the ways in which our framework could be optimized and extended. Some of these points are already planned for the future, while others are currently regarded as interesting ideas that could be considered in order to enhance the framework. Of course, this list is not exhaustive and additional optimization and extension options might be consider in the future.

8.3.1 Inclusion of Additional Analyzable Entities and Constraints

The main goal during the development of our current framework implementation for the FoodCASE system was to provide the required infrastructure, functionality and user interfaces that are needed for our proposed DQMF and realize all the features of our concept.

The next required step for a complete DQMF is the inclusion of all the entities that should be analyzed as well all the possible constraints that are related to the data quality of the system. Adding all the entities that should be analyzed is rather straightforward, in the sense that the system defines a limited number of entities pertaining to TDS-Exposure, which is the part of data within FoodCASE that our DQMF targets. However, adding all the quality-related constraints to the system is a much more demanding task, which, as explained in subsection 2.1.4, requires consulting the domain experts.

To achieve including as many relevant constraints as possible, there are plans for conducting a survey that should be completed by as many food science experts as possible. The survey will list all the entities and entity fields that are related to TDS-Exposure and inquire the participants about the types of constraints that would make sense for each of the listed elements and their importance. The results of this survey can then be analyzed by the FoodCASE developers, in order to determine and implement the desired constraints for the system.

8.3.2 Additional Analysis of User Evaluation Results

In chapter 6 we presented our initial analysis of the results of the questionnaire that was answered by the participants of TDS-Exposure workshop that took place during the third TDS-Exposure General Assembly in Zurich. Our framework was recently presented to the participants of the Food Composition for Nutrient and Exposure Assessment Workshop that took place during the EuroFIR Food Forum in Brussels[1]. The participants of this workshop were also asked to complete the questionnaire that is described in appendix C.

Unfortunately, due to time constraints, we were not able to analyze the answers that we received during the workshop in Brussels so far. However, one of the planned next steps is to analyze the results of both questionnaire answers thoroughly and perform a comparison between the answers of the two groups of participants.

8.3.3 Performance Tests and Comparison with the Food Composition DQMF

Chapter 5 discusses some of the most important similarities and differences between our DQMF and the food composition DQMF. As soon as our framework is enriched with additional constraints, it would be particularly interesting to test its performance and examine how it compares in relation to the performance of the food composition DQMF for equivalent amount of data and constraints. It would also be interesting to conduct comparative usability tests for the two frameworks.

[1] http://www.eurofir.org/

8.3.4 Persistence of Data Quality Analysis Results

The data quality analysis results are currently not persistent, which means that as soon as the user exits the analysis frame, the results are lost. An interesting extension to our framework would be to make the results persistent by storing them in the FoodCASE database. This would allow for a historical analysis of the data quality, which could help ensure that the quality of the data only increases over time.

8.3.5 Contract Editing and Creation Panel Enhancement

One extension that could enhance the usability of the *contract editing and creation panel* would be the option to compare the contract that is being edited with the current list of constraints at the framework and inform the user about the encountered differences. In detail, the user could be informed about whether there are system constraints that are not included in the contract or constraints that exist in the contract but have been removed from the system. In each case, the system could then prompt the user to decide whether they want to add the newly discovered constraints to the contract or delete them from the contract respectively. This would help the users in maintaining the contracts and keeping them up-to-date, especially in systems in which new constraints are being added or removed frequently.

8.3.6 Improvement of Input Validation Feedback Mechanism

As explained it the last part of subsection 4.6.3, in our current input validation module, the validation of all the constraints can be handled by a single `BeanValidator` object. The use of a single validator for all the validated constraints implies that in order to avoid receiving violation feedback in every field associated with any of these constraints, a mapping of constraints to frame fields needs to be provided. This solution, however, results into additional maintenance effort, as the mapping needs to be updated every time that a new constraint is added. Determining a way that would allow us to maintain the use of a single validator and receive correct violation feedback without the need of additional effort remains one of the open issues in our framework.

A

Acronyms and Abbreviations

ABS	Australian Bureau of Statistics
API	Application Programming Interface
BV	Bean Validation
CLAQI	Constraint-Level Analysis Quality Indicator
CMS	Content Management System
DQ	Data Quality
DQA	Data Quality Analysis
DQAF	Data Quality Analysis/Assessment Framework
DQF	Data Quality Framework
DQI	Data Quality Indicator
DQMF	Data Quality Management Framework
DQS	Data Quality Statement
DIV	Data Input Validation
EJB	Enterprise Java Beans
ELAQI	Entity-Level Analysis Quality Indicator
FDTP	Food Data Transport Package
FoodCASE	Food Composition And System Environment
GUI	Graphical User Interface
HBV	Hibernate Bean Validation
IMF	International Monetary Fund
IS	Information System
IVQI	Input Validation Quality Indicator
Java SE	Java Platform, Standard Edition
JFC	Java Foundation Classes
JSR	Java Specification Requests

Continued on next page

Acronyms and Abbreviations – *Continued from previous page*

JRE	Java Runtime Environment
OAQI	Overall Analysis Quality Indicator
SFCDB	Swiss Food Composition Database
TDS	Total Diet Study
WtET	Warning to Error Threshold

B

Comparison between Hibernate Validator and OVal

Table B.1: Comparison between Hibernate Validator and OVal

	Hibernate Validator	OVal
validation target	Java Beans	all types of objects
constraint configuration	• annotations • XML	• annotations • XML • POJOs (latter not well documented)
what can it validate?	• class fields • method/constructor parameters • method return values • complete object	• class fields • method/constructor parameters • method return values
JSR303 compliant	reference implementation	no, but comes with configurer that can translate the JSR303 built-in standard constraints to equivalent OVal constraints (also able to translate EJB3 JPA annotations

Continued on next page

Table B.1 – *Continued from previous page*

	Hibernate Validator	OVal
custom annotation based constraint declaration	possible	possible
express constraints using scripts	possible	possible
using scripted expressions for pre-/post-conditions	not possible by default	possible
cross-parameter constraints	available (for some or all parameters of a method or constructor)	not by default (perhaps possible with scripted expressions)
possibility to apply constraints to method and constructor parameters as well as method return values	yes (as of 1.1)	yes (but need to convert project to AspectJ project)
programming by contract capabilities	as of version 1.1 provides the possibility to express pre- and post-conditions	yes, by using AspectJ (even though not full-blown programming by contract capabilities)
constraint composition	possible	possible
recursive validation	possible	possible
declaring activation rules for constraints	not directly possible (but could be done by using class-level custom constraints)	possible with when attribute
grouping constraints	possible	not possible
payload definition	possible	not possible
personalized messages	possible	possible
availability of "probe mode" (to simplify UI user input validation)	no	yes
CDI integration	yes (since 1.1)	no

Continued on next page

Table B.1 – *Continued from previous page*

	Hibernate Validator	OVal
latest release	5.2.Alpha (10.2014), stable: 5.1.Final	1.84 (11.2013)

C

Questions from the TDS-Exposure Workshop Questionnaire

In this chapter we provide the list of questions that composed the questionnaire that was completed by the participants of the *TDS-Exposure workshop 2015* in Zurich. The introductory questions focused on the participants' position and skills and are meant to provide some degree of context to their responses. For the questions in sections C.1 to C.5, the participants could choose from a five-level Likert-type scale if they 'Strongly Disagree', 'Disagree', are 'Neutral', 'Agree' or 'Strongly Agree'. The remainder of the questions (sections C.6 to C.8) could be answered openly, with the exception of questions C.7-1 and C.7-3, 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)

C.1 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

C.2 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

C.3 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 analyze 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 analyze 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

C.4 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

C.5 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/analyze 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

C.6 Constraint Grouping

1. Can you think of additional ways to group constraints that would be useful to you? If so, please describe.

C.7 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?

C.8 Additional comments or suggestions

1. Do you have any comments or suggestions related to the data quality framework? If so, please describe.

List of Figures

List of Tables

Acknowledgments

First and foremost, I would like to offer my deepest gratitude to my supervisor, David Weber, for his excellent guidance, continuous support and positive attitude during the course of this thesis. He was always available to discuss problems, answer questions and offer advice, while allowing me enough freedom to explore my ideas.

Furthermore, I wish to express my sincere thanks to Prof. Dr. Moira C. Norrie and the GlobIS group for giving me the opportunity to work on my master thesis in such an interesting topic. Special thanks is owed to Dr. Karl Presser for the interesting discussions and his valuable insight into FoodCASE.

I am forever indebted to my parents for their unconditional support and encouragement. A big thanks is owed to my sister, Maria, for temporarily moving to Zurich, partly to be close to me. I would also like to thank all my friends for always being there for me. Special thanks goes to Karolos for being my lunch buddy throughout my studies and always starting interesting discussions. Last but not least, I am extremely grateful to my boyfriend, Jan, for his tremendous support and patience and for taking care of me for the past six months.

Bibliography

[1] ABS Data Quality Framework. https://www.nss.gov.au/dataquality/aboutqualityframework.jsp. [Online; accessed 02-April-2015].

[2] C. Batini and M. Scannapieco. *Data quality: concepts, methodologies and techniques*. Springer, 2006.

[3] P. Beynon-Davies. *Information systems: An introduction to informatics in organisations*, volume 9. Palgrave Basingstoke, 2002.

[4] T. Dasu and T. Johnson. *Exploratory data mining and data cleaning*, volume 479. John Wiley & Sons, 2003.

[5] A. Dedeke. A conceptual framework for developing quality measures for information systems. In *IQ*, pages 126–128, 2000.

[6] GS1 Website. http://www.gs1.org/, November 2014. [Online; accessed 13-November-2014].

[7] Hibernate Validator Reference Guide. http://docs.jboss.org/hibernate/validator/5.1/reference/, October 2014. [Online; accessed 17-March-2015].

[8] IMF DQAF Factsheet. http://dsbb.imf.org/images/pdfs/dqrs_factsheet.pdf, July 2003. [Online; accessed 02-April-2015].

[9] J. Juran and A. B. Godfrey. Quality handbook. *Republished McGraw-Hill*, 1999.

[10] R. Mock. *Data Quality Analysis for Food Composition Databases*. PhD thesis, Eidgenössische Technische Hochschule Zürich, Department of Computer Science, Institute of Information Systems, Global Information Systems Group, 2011.

[11] L. L. Pipino, Y. W. Lee, and R. Y. Wang. Data quality assessment. *Commun. ACM*, 45(4):211–218, Apr. 2002.

[12] K. P. Presser. *A requirement-oriented data quality model and framework of a food composition database system*. PhD thesis, Diss., Eidgenössische Technische Hochschule ETH Zürich, Nr. 20770, 2012, 2012.

[13] O. Probst. *Investigating a Constraint-Based Approach to Data Quality in Information Systems*. PhD thesis, Master thesis ETH Zürich, 2013, 2013.

[14] T. C. Redman. *Data Quality for the Information Age*. Artech House, Inc., Norwood, MA, USA, 1st edition, 1997.

[15] R. Y. Wang and D. M. Strong. Beyond accuracy: What data quality means to data consumers. *Journal of management information systems*, pages 5–33, 1996.

Eidgenössische Technische Hochschule Zürich
Swiss Federal Institute of Technology Zurich

Declaration of originality

The signed declaration of originality is a component of every semester paper, Bachelor's thesis, Master's thesis and any other degree paper undertaken during the course of studies, including the respective electronic versions.

Lecturers may also require a declaration of originality for other written papers compiled for their courses.

I hereby confirm that I am the sole author of the written work here enclosed and that I have compiled it in my own words. Parts excepted are corrections of form and content by the supervisor.

Title of work (in block letters):

Data Quality Assurance and Analysis for Food Science Data

Authored by (in block letters):

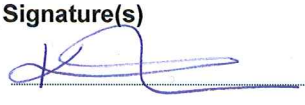
For papers written by groups the names of all authors are required.

Name(s):	**First name(s):**
Gemenetzi	Konstantina

With my signature I confirm that

- I have committed none of the forms of plagiarism described in the 'Citation etiquette' information sheet.
- I have documented all methods, data and processes truthfully.
- I have not manipulated any data.
- I have mentioned all persons who were significant facilitators of the work.

I am aware that the work may be screened electronically for plagiarism.

Place, date	**Signature(s)**
Zurich, 25.03.2015	

For papers written by groups the names of all authors are required. Their signatures collectively guarantee the entire content of the written paper.