Faculty of Sciences and Technology
of the University of Coimbra

Informatics Engineering

2015/2016

Dissertation (MSc)

Assessing the Behaviour of Service Applications in the Presence of Poor Quality Data

Seyma Nur Soydemir
seyma@student.dei.uc.pt

Supervisors:
Prof. Nuno Laranjeiro
Prof. Jorge Bernardino

2nd September 2016

Faculty of Sciences and Technology
of the University of Coimbra

Informatics Engineering

2015/2016

Dissertation (MSc)

Assessing the Behaviour of Service Applications in the Presence of Poor Quality Data

Seyma Nur Soydemir
seyma@student.dei.uc.pt

Supervisors:
Prof. Nuno Laranjeiro
Prof. Jorge Bernardino

Jury:
Main Opponent: Prof. Edmundo Monteiro
Opponent: Prof. Raul Barbosa

2nd September 2016

ACKNOWLEDGEMENT

I would like to express my gratitude to my supervisors, Prof. Nuno Laranjeiro and Prof. Jorge Bernardino, for the support provided during this master thesis, without which I would not have been able to reach this point. Furthermore, I would like to thank my family for the opportunity that they gave me to study abroad, and the support throughout these years.

Also, I would like to thank my friend Frederico Cerveira who helped me during my last years and was always available to give his opinion regarding my work. I wish also to thank all my friends that made my stay in Coimbra more pleasant, in particular to Ivano Alessandro Elia, Konstantia Barbatsalou and Carlos Cortinhas. This accomplishment could not have been possible without the support of many different people, too many to enumerate, to whom I wholeheartedly thank their assistance.

- *Seyma Nur Soydemir*

Resumo

As aplicações baseadas em serviços constituem a base de muitas organizações, devido à sua aptidão para suportar as mais variadas tarefas que contribuem para o funcionamento da organização. Por esta razão, a confiabilidade das aplicações baseadas em serviços afectam directamente o sucesso da organização, e podem causar danos financeiros e de reputação em caso de avaria.

As aplicações baseadas em serviços normalmente recorrem a sistemas de bases de dados para fornecer o seu serviço. Com o envelhecimento do sistema, erro humano e falhas ambientais, entre outros, os dados armazenados estão susceptíveis à perda de qualidade, e a partir daí ficando incorrectos. Um problema potencialmente grave consiste no uso dos dados pelas aplicações baseadas em serviços sem que estes tenham sido verificados, o que pode causar avarias capazes de afectar a própria organização.

Atualmente, os sistemas de gestão de bases e dados disponibilizam várias ferramentas que ajudam a garantir a qualidade dos dados, ao nível da base de dados, no entanto, uma aplicação baseada em serviços que seja bem desenvolvida deve ser resistente independentemente dos componentes de que faz uso.

Durante esta tese, uma abordagem que permite a avaliação do comportamento de uma aplicação baseada em serviços sob o efeito de dados com baixa qualidade foi desenvolvida. A abordagem consiste em interceptar os dados que vêm da base de dados para a aplicação, e modificar esses dados de acordo com uma mutação, que é escolhida a partir de uma lista de mutações que representam problemas de qualidade de dados reais. Uma ferramenta que implementa esta abordagem foi desenvolvida, e, de seguida, foram efectuadas experiências que resultaram na descoberta de vários *bugs* de *software* numa famosa aplicação de código livre, a par de uma classificação do seu comportamento.

Abstract

Service applications constitute the core of many organizations, due to their aptitude to support the most varied tasks that contribute to the operation of the organization. For this reason, the dependability of service applications directly affect the success of the organization, and can cause financial and reputation damages in case of failure.

Service applications often resort to database systems to fulfill their purpose. With the aging of the system, human errors and environmental faults, among others, the stored data is susceptible to the loss of quality, thereafter becoming incorrect. A potentially serious problem occurs when this data is used by the service applications without previous verification, which can cause business-damaging failures.

Nowadays, database management systems provide several tools that help ensure data quality, at the database-level, however, a well-designed service application must be robust independently of the quality of the data that it receives.

During this thesis, an approach has been developed to allow the evaluation of the behaviour of a service application under the presence of poor data quality. The approach consists in intercepting the data coming from the database to the service application, and modifying it according to a type-specific mutation, which is chosen from a list of mutations that represent real data quality problems. An usable tool that implements the proposed approach was developed, and experiments were conducted, which resulted in the discovery of several software bugs in a well-known open-source application, along with a classification of its behaviour.

Keywords. Service application, dependability, poor data quality, testing, DBMS

Contents

List of Figures

List of Tables

List of Acronyms

ANSI	**A**merican **N**ational **S**tandards **I**nstitute
API	**A**pplication **P**rogram **I**nterface
BSD	**B**erkley **S**oftware **D**istribution
CRM	**C**ustomer **R**elationship **M**anagement
CRUD	**C**reate **R**ead **U**pdate **D**elete
DB	**D**ata**B**ase
DBMS	**D**ata**B**ase **M**anagement **S**ystem
DOM	**D**ocument **O**bject **M**odel
ERP	**E**nterprise **R**esource **P**lanning
ID	**ID**entifier
IEEE	**I**nstitute of **E**lectrical and **E**lectronics **E**ngineers
ISO	**I**nternational **O**rganization for **S**tandardization
Java EE	Java **E**nterprise **E**dition
JDBC	**J**ava **D**ata**B**ase **C**onnectivity
JDO	**J**ava **D**ata **O**bjects
JPA	**J**ava **P**ersistence **A**PI
JSON	**J**ava**S**cript **O**bject **N**otation
JSP	**J**ava**S**erver **P**ages
LADC	**L**atin-**A**merican Symposium on **D**ependable **C**omputing
LOC	**L**ines **O**f **C**ode
MRP	**M**aterial **R**equirements **P**lanning
ODC	**O**rthogonal **D**efect **C**lassification
ORM	**O**bject **R**elational **M**apping
PDF	**P**ortable **D**ocument **F**ormat
PoC	**P**roof **o**f **C**oncept
PRDC	**P**acific **R**im International Symposium on **D**ependable **C**omputing
RDFS	**R**esource **D**escription **F**ramework **S**chema
SDC	**S**ilent **D**ata **C**orruption
SOAP	**S**imple **O**bject **A**ccess **P**rotocol
SQL	**S**tructured **Q**uery **L**anguage
WSDL	**W**eb **S**ervice **D**efinition **L**anguage
XML	E**X**tensible **M**arkup **L**anguage

1 Introduction

The increasing usage that service applications see in all kinds of organizations, driven by their affinity and success in the most varied roles, such as providing an easy-to-use, streamlined platform for client-to-company communication and internal use, makes them an important piece in the success of the business. The dependability of a service application now impacts the overall success of the company, in the sense that a failure can bring sizable losses both financially and in reputation.

Service applications, sometimes deployed as web applications, require testing to ensure their dependable behaviour. These applications usually interact with libraries, databases and the user to provide their service. Often the reason behind the existence of a software bug is that the software developers did not consider the possible problems that can arise when communicating with these components, be it a user providing incorrect content or a library not performing its intended function, both of which can cause the existence of poor data quality in the system.

The widespread usage of databases by service applications increases the probability of software bugs, in part because of an higher amount of lines of code (LOC) needed to implement this functionality [1, 2], and also because of a new external source of potentially unsafe and incorrect data. In fact, poor, invalid data inside a database is always a possibility, either because of a hardware fault that corrupts the data [3], a software fault in the myriad of code that lives in the path between the user and the database [4], or because the user has, intentionally or not, introduced incorrect data that is able to pass the validation checks that might have been implemented [5].

For all these reasons, a truly robust software application must not rely on the assumed reliability of other software components that it must interact with. If this is the case, then, whenever the expected level of reliability is not provided by the other components, the software application is able to continue its operation unaffected, or in the worst case, to detect the error and terminate in a clean mode.

Of the various components that can be harnessed by service applications, databases are the most common and a key part in storing and retrieving data that is essential to their function. Fortunately, database management systems (DBMSs) have improved in the last years, and now all of the big players feature tools that allow the user to establish rules and control data quality. Among these features we can count not-null and foreign key constraints, triggers and even stored procedures. Nevertheless, once again a well-designed and reliable service application must be independent of the external components that it depends on. The truth is that as with any other components, software, hardware and operator faults are an ever-present threat. Furthermore, a decoupling from the DBMS being used allows a wider range of choice, where criteria such as performance, ease-of-use, familiarity and dependability can be balanced according to the user's needs.

Functional and non-functional testing of service applications and other soft-

ware products is an already common practice in modern software development. Although this fast-paced cycle leads programmers to disregard non-functional testing in favor of functional testing, recently, interest has been shown in using fault injection to test performance, availability and recovery time under the presence of faults [6], or the security of the application [7]. However, little attention has been given to testing the robustness of such applications to invalid and incorrect input coming from the database, despite industry reports showing that developers often trust the correctness of the data being handled by their application, and when that assumption is wrong, big financial losses might occur due to the presence of poor quality data.

This thesis directly addresses the need for a way to study the robustness of service applications in the presence of poor quality data. We develop an approach capable of testing a service application under the presence of poor data quality, such as would happen if, for any reason (*e.g.*, hardware, software, incorrect content provided by the client), the data stored in the DBMS is of poor quality (*i.e.*, not fit for use). This approach works by mutating the content coming from the database to the service application (*e.g.*, via JDBC), according to a predefined set of rules, and then analyzing the behaviour of the application. To verify the effectiveness of this approach, a tool was developed and an initial experimental evaluation was undertaken using a well-known open-source service application. The fault model (*i.e.*, the mutation tables for each data type) used with the tool was designed after a throughout analysis of the state of the art in the area of data quality and data problems that occur regularly. As proved during the experiments, our approach is capable of uncovering programming bugs not only in the service application itself, but also in middleware libraries. Nevertheless, the main purpose of our approach is to classify the behaviour of the application according to a predefined scale (*e.g.*, CRASH scale).

During this thesis our work produced several contributions to the field, in particular:

- A comprehensive survey about data quality, including the importance of data quality and data quality dimensions and problems. It culminated in the study of data mutations that are representative of what occurs in the real world.
- Using an already existing basis, refined an approach for mutating data that comes from database to service application, and implemented it into a usable tool.
- Analysis of the behaviour of a well-known service application in the presence of mutated data, which also allowed to assess the effectiveness and usefulness of the proposed approach and tool in performing their goals.

Several outputs were created as the result of the thesis, in particular:

- A data quality survey that was submitted and presented in a top conference in the area of Dependable Computing (PRDC 2015).

- A practical experience report that was submitted to a conference, Latin-American Symposium on Dependable Computing (LADC 2016), and has since then been accepted (at the moment, a camera-ready version is being prepared for delivery).
- A freely available fault model containing mutations that are representative of data quality issues.
- A tool, including source-code, available online and capable of intercepting the JDBC calls between service application and database, and mutating its content according to the previous fault model. This tool rigorously implements the approach presented in this thesis, and was used to perform the experimental evaluation.

The rest of this thesis is structured as follows. In the following section, the state of the art of the topics that are related to the core theme of this thesis are presented. In Section 2, a review of the state of the art in the areas perpendicular to the topic of this work is given. In Section 3, the developed approach is presented and discussed. Section 4 has a more thorough description of the experimental setup that was used in our experimental evaluation, including details about the implementation of the tool and the test cases that were designed to test the service application. In Section 5 the results of the experimental evaluation are presented, including a listing of the software bugs that were uncovered, a classification of the application behaviour, and a set of guidelines to prevent the occurrence of the various programming bugs that were reported. Finally, Section 6 contains the conclusion of this thesis, and sets the plan for the future work.

2 State of the Art

In this section, the state of the art of the areas related with this work is presented. The literature review of this thesis is grouped in two main groups of research: Data Quality, particularly the impacts and problems caused by poor data quality, which provides the validity and justification for testing a service application using poor data, and Service Application Testing, where an introduction is given to the various kinds of testing of software applications, with a special emphasis to testing service applications. Furthermore, a presentation and comparison of automation tools is included, fueled by the need of such tool in our experimental evaluation and the overall usefulness that automation tools have in real world testing.

2.1 Data Quality

The interest and value that data has acquired in an array of areas, such as statistics, data mining, and big data, where it plays a key part supporting in their existence, increases the demand for data quality. In other words, organizations strive to possess data with the highest possible quality, in order to maximize the return from their processes. Incidentally, poor data quality can have disastrous consequences and must therefore be taken very seriously.

In this section, a detailed overview regarding the state of the art in Data Quality is provided, which includes the concepts, the impacts of poor data, data quality dimensions, data quality problems and, finally, as a link with the main topic of this thesis, data quality in service applications.

As a side note, the terms used by each author to identify the concepts of data quality (*e.g.*, data quality dimensions, data quality problems) by them proposed are also used in this section (and sub-sections), but are not modified (*i.e.*, the names of the concepts are used as they appear in the original papers where they were first presented), otherwise the original essence of their work would risk being damaged.

2.1.1 Concepts of Data Quality

Data quality can be defined in many diverse ways as is deeply described in [8]. In ISO 9000:2015 [9] it is defined as "the degree to which a set of characteristics of data fulfills requirements". Data quality takes into consideration whether data meets implicit or explicit expectations of data users [10], such as why data is needed and if it can be used as intended. Therefore, high quality data is "data that is fit for use by data consumers" [11]. Data quality dimensions guide the process of evaluating data quality. These dimensions are attributes that represent "a single aspect or construct of data quality" [12].

According to the above definitions, poor data quality can be defined as the the amount to which the properties of data do not fulfill their requirements. By not fulfilling all the requirements in order to be fit for use by data consumers, it

is prone to cause impacts that can affect the entities involved (*e.g.,* a company or a customer).

2.1.2 Impacts of Poor Data

To comprehend the importance of data quality, one must focus on the impacts of poor data. The literature presents different categorizations of data quality impacts. To understand and improve the data quality, Wang and Strong [12] focused on the meaning of data quality from the consumer's perspective and noted the substantial social and economical impacts of poor data. There is strong evidence that data quality problems affect many organisations [13]. It causes loss of profit, because profit is related with providing customers' needs and poor quality data hinders this service.

The awareness of poor data quality that enterprises should possess was taken into consideration when classifying the impacts that are presented in [14], which can be classified as *Operational*, *Tactical* or *Strategic* [14]. Enterprises provide many services to the customers and carry out internal processes that ensure the well-functioning of the organization. While executing these operations, poor quality data may cause unexpected problems, referred to as *Operational impacts*, which ultimately will affect the effectiveness of these operations. Furthermore, in the everyday life of a company, many decisions must be taken which guide the future of the organization. These decisions are strongly based on the data that is available to the decision-makers (*e.g.*, CEO). Therefore, if this data is of poor quality, the strategic decisions will suffer significantly, leading to severe consequences. These consequences, called as *Tactical Impacts*, can range from planned improvements that are impossible to be put into effect, to crippled services and, at a higher level, lower trust in the ability of the organization. Finally, companies need a well-defined strategy that defines their future plans and ensures the success of the organization. The existence of poor data leads to *Strategic Impacts*, which can be inaccurate decisions when defining the strategy and difficulties in the execution of these strategies.

A straightforward approach to analyse the degree to which poor data quality limits business success depends on the categorization of business impacts, as is presented in [15]. These negative impacts that affect businesses can be grouped into *Financial*, *Confidence and Satisfaction-based*, *Productivity* and *Risk and Compliance*. *Financial impacts* are the result of poor data in businesses at the financial level. Poor data causes unexpected cost in the operation of the enterprise and decreases the income, and may even cause extra charges. *Confidence and Satisfaction-based impacts* occur when poor data affects the main areas of the enterprise, such as decision making or reporting, and as an outcome the loss of organizational trust and satisfaction ensues. *Productivity impacts* occur when poor data directly affects the performance of the processes needed to ready a product, for example affecting the product quality or processing time. *Risk and Compliance impacts* results from poor data causing the enterprise to disobey certain rules (*e.g.*, laws, company policies, standards), or to mis-evaluate the risk taken when performing a certain investment or acquiring credit.

Even if all these impacts belong to different areas and are classified in different ways, they all inevitably lead to higher costs for organizations. Managing data quality comes with its own costs, which must be assessed and compared to the potential loss due to damaged data [16]. Data warehouses' projects failures are often related with poor quality data, such as not being able to relate data coming from different sources, having missing or wrong values, having non-standardized usage of data fields (*i.e.*, using different representations for dates, representing weights in different metrics), among others [17]. In order to prevent failures, it is better listen and answer data consumers' data quality needs [12]. To show its importance, a few examples of the failures caused by poor data quality in the real world are listed below:

- The advisory firm Gartner has estimated, in 2014, that the costs per year due to poor data on organizations to be on average $8.8 million dollars [18].
- In 2013, the US Postal Service reached the conclusion that around 6.8 billion mail letters could not have been successfully delivered due to poor data. In monetary terms, this lead to the postal service spending at least $1.5 billion in handling these mail letters [19].
- In 2011, a consulting firm announced that poor data costs $3.1 Trillion to the U.S [20].
- In 1978, U.S company *Prudential* lost almost $93 million due to a missing comma and zeros (the amount that *Prudential* claimed from a loan changed from $93 million to $93,000) [21].
- Rocket scientists at NASA had to abort the $80 million mission *"Mariner I"* because of a missing hyphen in a mathematical operation [4].
- In 1999, a data quality problem at a NASA mission caused a costly crash and made headlines all over the world, when the $125 million *"Mars Climate Orbiter"* flew off course and disintegrated because the engineers forgot to convert from lbf x s to N x s [4].

As a consequence of data quality impacts, the high cost caused by poor data quality is analyzed in several research with different classifications [16]. The United States' Department of Defense produced the data quality guidelines present in [16], where the cost of poor data quality is divided into two main groups: *Direct data quality cost* and *Indirect data quality cost*. Direct data quality cost is related with the impact of poor data in a process and the cost of improving and correcting the poor data. It includes **controllable costs**, that are due to the management of data quality such as prevention appraisal, **resultant costs**, which are costs caused by the impact of poor data quality, (*e.g.*, internal-error and external-error costs), and **equipment and training costs**, such as costs with acquiring specific hardware and software, and training on how to prevent, assess and fix data quality. Indirect data quality costs are related with the loss of credibility and customer satisfaction due to poor data quality.

Another research about data quality costs studies and categorizes potential costs related with low quality data under two major types [22]: *Improvement costs* and *costs due to low data quality*. Improvement costs are closely linked with the process of ensuring data quality, which includes prevention of poor data (*e.g.*, training and monitoring), detection of poor data, and repair. Costs due to poor quality data can be quantified according to how easy they are to measure and the impact that they cause, this yields two classes: direct and indirect costs. Direct costs are the cost that are more directly associated with poor data quality, and include the cost of verifying where the poor data is and re-entering it correctly. Indirect costs are caused by poor data quality but as a result of its effects on the organization, for example, costs due to a lower reputation, wrong decisions taken based on poor data, and lost investment opportunities.

As described in this section, the impacts of poor data (or in other words, when data does not meet expectations) have a strong effect on organizations and business. In the next section we present data quality dimensions extracted from diverse literature, and which can be used to measure data quality.

2.1.3 Data Quality Dimensions

In order to determine the value of information and how to improve it, measuring data quality is a key activity. However, before an evaluation can be done, an agreement on what (and how) should be measured has to be reached [23]. With the goal of identifying measurable aspects of data quality, data quality dimensions are identified and organized using very different measurements [24], as described in this section.

The data quality dimensions proposed in the work of Thomas Redman [25] are grouped under 3 main categories, which are *Conceptual View*, *Value*, and *Representation of Data*. The author listed the dimensions belonging to *Conceptual View of Data* under 6 subjects, which are "content, scope, level of detail, composition and view consistency" [25], all of them related to abstract notions. The *Value of Data* dimensions are, as implied in the name, dimensions that define quality in the value of data. They present 4 dimensions under this category, which are "accuracy, completeness, currency and value consistency" [25]. The third and final category is the *Representation of Data*, for which Redman proposes "appropriateness, interoperability, portability, format precision, format flexibility, ability to represent null values, efficient usage of recording media, and representation consistency" [25].

Data quality dimensions are a deeply researched topic in literature, given the importance of data quality as a study subject. However, the proposed dimensions often vary according to the research area or categories (*e.g.,* data model, value, domain, presentation and information policy). In Table 1, we present a comparison between the data dimensions used in different research papers.

Table 1: Overview of different classifications for data dimensions

Research	Quality Measurements	Quality Metrics
A Hierarchical Approach to Improving Data Quality [26]	4 Categories, 20 Dimensions	***Accuracy of data/Intrinsic:*** Accuracy, Objectivity, Believability, Reputation.
		Relevancy of data/Contextual: Value-Added, Relevancy, Timeliness, Completeness, Appropriate Amount of Data.
		Representation of data/Representational: Interpretability, Ease of Understanding, Representational Consistency, Concise Representation.
		Accessibility of data/Accessible: Accessibility, Access Security
A Data Quality Handbook for a Data Warehouse [27]	5 Dimensions	Correctness, Completeness, Consistency, Currency, Accessibility
DoD Guidelines on Data Quality Measurement [16]	6 Characteristics	Accuracy, Completeness, Consistency, Timeliness, Uniqueness, Validity
A New Method for Database Data Quality Evaluation at the Canadian Institute for Health Information (CIHI) [28]	5 Categories	Accuracy, Timeliness, Comparability, Usability, Relevance
	24 Characteristics	Over-coverage, Under-coverage, Simple response variance, Reliability, Correlated response variance, Collection and capture, Unit non-response, Item non-response, Edit and imputation, Processing, Estimation, Timeliness, Comprehensiveness, Integration, Standardization, Equivalency, Linkage-ability, Product comparability, Historical comparability, Accessibility, Documentation, Interpretability, Adaptability, Value.
Enterprise Knowledge Management [29]	5 Categories, 31 Characteristics	***Data Quality of Data Models:*** Clarity of definition, Comprehensiveness, Flexibility, Robustness, Essentialness Attribute Granularity, Precision of Domains, Homogeneity, Naturalness, Identifiability, Obtainability, Relevance, Simplicity, Semantic Consistency, Structural Consistency.
		Data Quality of Data Values: Accuracy,Null values, Completeness, Consistency, Currency/-timeliness.
		Data Quality of Data Domains: Enterprise agreement of usage, Stewardship, Ubiquity.
		Data Quality of Data Presentation: Appropriateness, Correct Interpretation, Flexibility, Format Precision, Portability, Representation Consistency, Representation of Null values, Use of storage.
		Data Quality of Information Policy: Accessibility, Metadata, Privacy, Redundancy, Security, Unit Cost.
The Practitioner's Guide to Data Quality Improvement [24]	3 Categories, 10 Characteristics	***Intrinsic Dimensions:*** Accuracy, Lineage, Semantic, Structure.
		Contextual Dimensions: Completeness, Consistency, Currency, Timeliness, Reasonableness and Identifiability.
		Qualitative Dimensions
Data quality assessment [30]	16 Dimensions	Accessibility, Appropriate Amount of Data, Believability, Completeness, Concise Representation, Consistent Representation, Ease of Manipulation, Free-of-Error, Interpretability, Objectivity, Relevancy, Reputation, Security, Timeliness, Understandability, Value-Added.
Data Quality: The Guide Field [31]	9 Categories, 51 Dimensions	***Accessibility/Delivery:*** Availability, Protocol, Security.
		Quality of Content Attribute Granularity, Comprehensiveness, Essentialness, Flexibility, Appropriate Use, Areas Covered, Homogeneity, Naturalness, Obtainability, Precision of Domains, Robustness, Semantic Consistency, Structural Consistency, Simplicity, Clear Definition, Identifiability, Source, Relevancy.
		Quality of Values: Accuracy, Completeness, Timeliness, Consistency.
		Presentation Quality: Appropriateness, Format Precision, Use of Storage.
		Flexibility: Portability, Representation Consistency, Null Values, Formats, Language, Ease of Interpretation.
		Improvement Feedback, Measurement, Track Record.
		Privacy: Consumer Privacy, Privacy of Others, Security.
		Commitment: Warning, Help, Special Requests, Commitment.
		Architecture: Library/Documentation, Logical Structure, Physical Structure, Naming, Rules, Redundancy, Unit Cost.

2.1.4 Classification of Data Quality Problems

In this section, we discuss research that deals with *Data Quality Problems*, *i.e.*, the specific poor data issues that affect services and organizations which, to some extent, rely on data for their operations.

Problems caused by poor data quality have been studied in previous research belonging to various different areas, due to the fact that these problems have effects in a vast range of information systems. The different research areas (*e.g.*, data cleaning, data quality tools, fault injection) led to the existence of many different classifications. For example, a research about data cleaning presented a classification of data quality problems [32], where the problems are classified as single-source and multi-source problems, both in schema level and in instance level. Multi-source problems occur when the data is integrated from multiple data sources, and commonly consist of different representations, overlaps and contradictions, which cause structural and naming conflicts [33, 34, 35]. Single-source problems depend on schema and integrity constraints. Schema-related problems occur due to data model and schema design defects, while instance level problems are related with errors and inconsistencies that cannot be prevented even when using the appropriate integrity constraints or a better schema.

Single-source problems can be sub-divided into Schema Level and Instance Level. Some examples of Schema Level problems are values that do not obey constraints or dependencies. Examples of Instance Level problems are values that are missing or contain small typos or the usage of abbreviations.

Multi-source problems can also be sub-divided into Schema Level and Instance Level. Examples of these Schema Level problems are when the same name is used among the different sources to refer to different entities, or, in the opposite side, when different names are used to refer to the same entity, and when the same entity is represented using a different structure (*e.g.*, different data types, different fields) in the various sources. Examples of Instance Level problems can include the same Instance Level problems of Single-source but are also extended by cases such as when data can refer to different points in time (*e.g.*, one source refers to 1 week ago, and the other two 1 month ago).

The research presented in [36] primarily classified poor data into 2 categories, which are "*missing data*" and "*not missing data*". Missing data can quite simply be divided according to whether a not-null constrained exists in the schema (*i.e.*, if the missing data is allowed to exist according to the schema, or if despite existing, the schema does not allow it). Not-missing data is grouped into "wrong data" and "not wrong but unusable" data. Wrong data has further sub-divisions depending on whether integrity constraint can be implemented or not. Not wrong but unusable data can occur because of problems during database integration or problems when receiving data from the user. For example, data may be inserted with different values for the same entity in different databases, or there is a lack of context that does not allow the available data to be correctly interpreted (*e.g.* homonyms, abbreviations, which unit was used, different representations of dates).

Another classification of data quality problems is presented in [37]. According to the authors, the data problems can be distinguished at fours levels, namely, level of attribute/tuple, level of a single relation, level of multiple relations and level of multiple data sources, which mimics the usual model of data found in organizations. Each level has a different granularity (*i.e.*, the amount of detail and context that is available), with the level of attribute/tuple being the one with the lowest granularity, and the level of multiple data sources having the highest granularity.

The problems at the level of attribute/tuple are the most specific and detailed, and include problems such as values that are missing or incorrect, values that are outside the expected domain, the existence of synonyms, homonyms and duplicated values, or the violation of a functional dependecy (*e.g.*, the age and birth year do not match). Each time that the level is increased (*i.e.*, we move to a level with a higher granularity), the problems become more general, high-level and abstract. For example, at the level of a single relation, a possible problem are duplicate tuples. At the level of multiple relations, problems can be incorrect references (*e.g.*, an entity points to a certain Postcode, but this Postcode is not present in the table that keeps all the postcodes of the Country), circular references (*e.g.*, Megane is a car from Renault, and Renault is a sub-model of Megane), or different representations for the same type of data (*e.g.*, different representation for dates). Finally, possible problems at the level of multiple data sources are the presence of homonyms and synonyms or different representations for the same type of data.

A classification of data anomalies is presented in [38], where the authors classify the anomalies into syntactical, semantic and coverage anomalies:

- **Syntactical anomalies**
  - Lexical errors: when the structure of the data does not match the expected format.
  - Domain format errors: when the data value is outside the expected domain.
  - Irregularities: when units and abbreviations are used in an inconsistent manner.
- **Semantic anomalies**
  - Integrity constraint violations: when a value or a group of values do not agree with a predefined integrity constraint (*e.g.*, the value of WEIGHT has to be non-negative).
  - Contradictions: when two values that have an implicit dependency on each other disagree (*e.g.*, when the city of birth disagrees with the country).
  - Duplicates: when there are two or more rows referring to the same entity, but one row would suffice.

  - Invalid tuples: when a group of values does not feature the previously mentioned anomalies, yet it still is not able to correctly represent reality.

- **Coverage anomalies**
  - Missing values: when values are missing in a row, usually due to deficiencies in the data collection process, which means that only an incomplete view of reality is available.
  - Missing tuples: when one or more entire rows are missing, therefore entities that exist in reality are not represented in the database.

In a survey of data quality tools [39], the authors present a classification of data quality problems similar to previous research [36], but following the clustering of [32], which divides data quality problems into schema level problems (that can be avoided by an improved schema design) and instance level problems (errors and inconsistencies of data that are not visible or avoidable by schema level, and cannot be prevented by a better schema definition). Additionally, schema level problems were grouped into two types: Avoided by DBMS, and not-avoided by DBMS. Instance level data quality problems were divided into two groups of problems, concerning single data records and multiple data records.

The data quality problems that are supported by DBMS in schema level are presented as "missing data, wrong data type, wrong data value, dangling data, exact duplicate data and domain constraint violation" [39]. These data quality problems can be avoided by using integrity constraints. For example, missing data means that a data field is empty, which can be avoided by using a not-null constraint. Wrong data type means that certain data is being represented using an incorrect data type, which can be fixed by defining and enforcing the exact type of each field. Wrong data values, in particular values that are outside the expected domain, can be prevented by defining a valid data range and applying it as a constraint. Dangling data occurs when multiple tables have references to each other that do not agree, this can be solved by using foreign keys. Duplicated data means that more than one entry exists for the same entity, and can be easily solved by defining unique or primary key constraints. Domain constraint violation can be solved by defining and enforcing constraints regarding the domain of data.

At the schema level, the data quality problems that cannot be avoided by relational databases are listed as "wrong categorical data, outdated temporal data, inconsistent spatial data, name conflicts, and structural conflicts" [39].

This research defines instance level problems as those that cannot be avoided at the schema level (*e.g.*, by using constraints or a certain schema design). This kind of problems was further divided into "single record" and "multiple record" problems. While in single record problems, their source is limited to that record itself, in multiple record problems, the cause is problems between multiple records. Examples of single record problems are dummy values used as

a way to represent missing data (when there is a not-null constraint) or wrong data (*e.g.*, because of typos). Examples of multiple record problems can be when there are duplicated entries for the same entity, or various entries that refer to the same entity but have different content (*i.e.*, they contradict themselves).

In the next subsection, we will discuss the various software testing approaches, paying particular focus to techniques that can be applied to service applications and that can be used to emulate data faults.

2.2 Software Testing

Testing can be defined as the execution of a program with the objective of finding errors (*e.g.*, software bugs) [40]. Although an apparently easy task, the design of test cases should be performed with the purpose to test both correct and incorrect scenarios. An unexperienced tester might choose to create test cases that have a high chance of passing (expected inputs), and shy away from creating useful cases with inputs that the program is not expecting, therefore failing to uncover the existing software bugs and accomplish the goal of testing.

A software bug is a concept that is harder to define than would be expected. However, its definition is an essential step before the term can be used unambiguously. In general, a software bug occurs when the implemented software disagrees with the product specification (*e.g.*, the software does not implement part of the specification, the software does something that the specification forbids, the software does something that is not defined in the specification) [41]. Furthermore, a software bug can also refer to whenever, in a somewhat subjective manner, the software is slow, difficult to use, or, in general, does not work as expected [41].

A possible classification of testing techniques groups them in two main methodologies, according to the access to internal structure (*e.g.*, source code) that is required and the aspects of the system that are studied: black-box and white-box.

Black-box testing is a type of testing where the internal structure of the program is ignored, instead focusing solely on finding occurrences where the program does not behave according to its specifications. An usage of the black-box approach to finding errors in the program is to perform a comprehensive test using all the possible (valid and invalid) combinations [40]. Some black-box testing methods are:

- **Functional testing** tests to ensure the functionality required by the program, by comparing the output during testing with the expected output.
- **System testing** tests the entire system to ensure its compliance and workability in various different environments of its non-functional and functional specifications.
- **Stress testing** tests the stability of the system when put under stress at and beyond the limits of what is expected.

- **Performance testing** tests the conformity of a system or component with the specified performance requirements.
- **Usability testing** is performed to evaluate the system regarding its ease of use by the client (*i.e.*, if it is easy-to-use or not).
- **Acceptance testing** is used to determine whether the finished product obeys the customer's requirements or specification.

White-box testing, also known as structural testing, studies the internal structure of the program. The test cases for white-box testing are created by analyzing the program logic [40], according to certain test design techniques (*e.g.*, control flow testing, data flow testing). The goal of this technique is to cover (*i.e.*, execute) all or a certain percentage of the instructions (or another metric) of the program. Some important white-box testing techniques extracted from literature are briefly described below:

- **Control flow testing** uses the control flow graph of the program to ensure a set of test cases that is capable of covering the whole code of the program (or a certain percentage) is created [42]. The control flow graph shows all the paths through the program, where each node represents a block of linear code (*i.e.*, without any jumps) and each vertex represents a jump instruction (*e.g*, if, for) [43].
- **Branch testing** is similar to control flow testing, but aims to execute each branch (*e.g.,* loops, if statements) at least once in the produced set of test cases.
- **Data flow testing** is a white-box testing method that also uses the control flow graph to create a set of test cases based on the initialization and access of data objects (*e.g.*, ensure that every data object has been initialized and used at least once) [44].
- **Loop testing** is used to validate the loops of a program, and consists in creating a set of test cases that ensure that every loop is not entered, entered once and entered multiple times.

2.2.1 Testing Service Applications

Given the effects that data quality has in service applications, fault injection started to be widely used to introduce data quality faults. Fault injection is a technique used to study the dependability of systems under the presence of a fault [45]. One of the definitions of *Dependability* is "the ability to deliver service that can justifiably be trusted" [46]. This definition is based strongly on the definition of *trust*. An alternate definition is "the ability to avoid service failures that are more frequent and more severe than is acceptable" [46]. This definition shows that the concept dependability depends on the requirements of each system. The concept of dependability is composed by many attributes, in particular, *Availability, Reliability, Safety, Integrity and Maintainability* [46].

Fault injection techniques can be implemented through hardware or software based techniques. Fault injection implemented by software, as opposed to implemented by hardware, can be further classified in two main groups: *compile-time* and *run-time injection*. While compile-time injection requires the modification of the executable before execution, as to introduce the faults, the latter works by dynamically injecting the faults during program execution. Because of the different approaches, run-time injection carries a higher overhead associated with the mechanisms needed to inject the fault, while compile-time injection does not present this run-time overhead, but it requires more time during preparation phase to instrument the executable or source-code.

Another popular way of testing service applications is to perform robustness testing. Robustness testing is the process of assessing the robustness of a system, by trying to activate faults or vulnerabilities (*e.g.*, software bugs) that the system already possesses, and that lead to incorrect behaviour (so called, robustness failures) [47].

Robustness is a secondary specialized attribute of dependability [46]. It is defined, according to ANSI/IEEE [48], as "the degree to which a system or component can function correctly in the presence of invalid inputs or stressful environmental conditions". Avizienis *et al.* define it as "dependability with respect to external faults, which characterizes a system reaction to a specific class of faults" [46].

The term *robustness testing* was first coined by Koopman *et al.*, for the Ballista project, and has since then been used by numerous other researchers. In the Ballista project [49], a tool was developed that is capable of modifying the parameters of system calls, in order to study the response of the operating system. This tool was then applied to 15 different operating systems, with surprising results, including a few system crashes. Given the pioneering nature of this work, the same authors also presented their own classification schema to be used when performing robustness testing (particularly, robustness testing of operating systems), the CRASH scale [50]. This scale has also seen significant use from other researchers, and, in fact, as will be explained later in this thesis, it is also the schema adopted for the classification of our results. As an example of another robustness testing tool, MAFALDA [51] can be used to assess the failure modes of microkernels under the presence of injected faults (internal and external), in particular, faults in the parameters of the system calls. Furthermore, it can also be used to evaluate the effectiveness of wrappers designed to improve these failure modes.

Fault injection can be used to implement robustness testing, by emulating the effects of hardware-related faults. In this scenario, fault injection can be used to test the ability of a certain component to handle faults in the interaction with other components. Another approach to conduct robustness testing is to feed the inputs of the system under study with random data (fuzzy testing), or by having predefined valid and invalid inputs, which can vary according to the data type, and feeding combinations of them as input. In fact, the boundary between fault injection and robustness testing can sometimes be difficult to draw, given their similarities in purposes and implementations.

The same can be said of mutation testing and fault injection of software faults, which follow a very similar approach, but study different topics. Mutation testing is a white-box testing technique which is used to obtain a score that represents the effectiveness of a certain test set in detecting software bugs. It works by adding specific software bugs to the code, according to mutation operators, thereby creating mutant programs, and then running the predefined tests and verifying if the tests are able to detect the presence of the injected software bug [52]. It can be considered a similar technique to the injection of software faults, since both consist in the modification of the source code of programs by adding faults, but whereas software fault injection is used to assess the resiliency of a system and its fault handling mechanisms, mutation testing is aimed at the assessing the effectiveness of test cases rather than focusing on the code. One example where mutation testing is used for testing web services is [53], where mutation operators are presented for the WSDL specification of web services.

Below we will provide an introduction to some practical examples of the above techniques applied to service applications. Web services, a way to deploy service applications, are programs that implement a certain functionality and offer their service over the Internet to other programs, such as other web services or web applications [54]. With the usage of Web services applications to extract and integrate data in heterogeneous business systems, data quality started to be an important issue regarding on how to provide accurate, complete, consistent, and correct information in Web services [55]. With the aim of providing information quality in web services, recent research worked on possible data quality problems caused by web services [56, 57, 58, 59].

Rychlý and Zouzelka [60] present fault injection as a means of performing robustness testing of web services, by injecting faults that mutate the communication between services. Examples of the proposed faults are replacing parts of a string, replacing with empty message, replacing with null, adding new or non-printable characters, and delaying messages.

The research paper in [61] explores the usage of Perturbation-Based testing applied to web services. The proposed approach consists on the application of perturbation operators to SOAP messages, in order to assess how the web services handle the presence of incorrect data in these messages. The operators build upon other works, and represent common faults that can happen in real systems. Examples of these operators are "set to null", "delete a node from the message" and "add a whitespace". The authors note that they are restricting their approach to SOAP messages, but the WSDL specification of the web services can be another place where to inject faults.

A tool for robustness testing of Web services, WebSob, is presented in [62]. This tool automatically generates and executes web-service requests, given a service provider's Web Service Description Language (WSDL) specification, and is capable of revealing robustness problems without the need of any previous insight into the underlying web service implementation.

Another tool, WS-FIT [63], harnesses fault injection of faults in the network to perform robustness testing of SOAP-based web services, by modifying these

messages.

Laranjeiro *et al.* [64] address robustness testing in web services, by proposing a robustness testing approach for web services, along with the respective tool and experimental evaluation of various web services. This approach is based on wrong call parameters, which consists of exceptional and valid values after a predefined set of rules is applied according to the data type of each parameter. The mutation operators focus on creating limit cases, such as null values, lower or higher than acceptable values, values with special characters, and malicious values (*e.g.*, SQL injection). To classify the behaviour of the web service, the authors adapt the well-known CRASH scale for usage in web services, by defining solely 3 possible classifications: Correct, Crash and Error. As a proof to their concept, the authors performed robustness testing of several web services, and prove that their approach and tool is capable of finding valuable software bugs.

The research that serves as the basis for this thesis is [59], which presents a general approach to testing the behaviour of service applications in the presence of poor data quality, along with a basic fault model (*i.e.*, mutation table) for the String data type. The proposed approach can be considered somewhat similar to robustness testing, in the sense that it attempts to study the robustness of the service application by trying to activate faults or vulnerabilities that it might have, however it differs from most uses of robustness testing because instead of mutating the values passed as inputs to calls of the system, it mutates the data that comes from calls to database and is then going to be used by the application. At the same time, the authors justify their interest in this subject by providing a justification for the validity and importance of this valuable, yet rarely studied topic, which given the widespread presence of service applications in nowadays organizations, and their abundant use of databases to store and retrieve business-critical data, deserves a more in-depth study.

This thesis answers that call and looks to build upon the proposed and abstract approach by further refining it, extending the fault model with more data types and data mutations that are obtained from the study of data quality and real-world data quality problems, implementing a tool and evaluating the behaviour of an open-source service application in the presence of poor data quality.

2.2.2 Automation Tools for Testing

Most of the testing methods (*e.g.*, regression, acceptance) require interaction between the human testers and the GUI, in order to verify the GUI's visual behaviour. These testing methods require the multiple repetition of a sequence of steps. Considering the huge size of many software projects, testing a project using traditional (*i.e.*, manual) approaches incurs a great cost, in part due to the long time and high human effort required.

Another problem that comes with manual testing is that possible human errors reduce the efficiency of testing [65], as well as having high time consumption because of repeating the same test cases over and over.

Automated testing was developed to overcome these limitations, and to allow shorter development cycles and an overall quicker, easier and more effective testing process. Many automated testing tools are available, some are commercial, while others are open-source, some require user experience in testing software applications, others are more user-friendly [66, 67].

Each automation testing tool has different properties, depending on their mode of operation. According to this criteria, the three main types of automation testing tools are [68]:

- **Recorder:** During the test case creation, a recorder tracks the mouse and keyboard actions of the tester. After the record is complete, the tester is able to playback, and manually edit the existing test case.
- **GUI aware:** This kind of automation tools record and play programmatically according to GUI elements. They use the internal structures, GUI elements and their features, to guide their operation. It requires knowledge and a close connection with the application being tested.
- **Visually:** These tools snapshot the screen areas during the test case creation, and then try to match these snaps to the contents currently onscreen. In this case, there is no need for application-specific knowledge.

It should be noted that sometimes automation tools can operate in more than one category at the same time (*e.g.*, being mostly "Visually" but also having "Recorder" features).

We studied some of these automation tools, in order to choose the tool that best fits our needs (during the experimental phase of this thesis). The automation tools studied are the following:

- **Selenium** [69] is a package of tools to perform or aid software testing and test automation. At the time of its conception, Selenium was a pioneer of test automation and presented a novel approach to solve the problem of performing repetitive actions, such as is common in the setting of software testing. The tools included in the package are Selenium (the engine itself), Selenium IDE and Selenium Grid. Selenium 2, also known as Selenium WebDriver, is the newest version of the test automation engine and was developed to provide a better support for dynamic web pages that are able to change the elements without reloading web page. The previous version of Selenium (the engine) is Selenium 1, or Selenium RC. At the moment Selenium 1 is considered obsolete, hence Selenium 2 became the default choice. One big downside of Selenium is the precarious support for dynamically created webpages, which can be a strong deterrent to its usage with certain websites. It can be considered to be a "GUI-aware" automation tool, due to its need to understand the internal structure (*i.e.*, DOM) of the webpage.
- **SikuliX** [70] is a visual automation tool capable of executing on Windows, Mac and some Linux/Unix-based operating systems. It makes use of image recognition, aided by the well-known library OpenCV, to compare

screenshots inside a test case with the contents currently being displayed on screen. When it finds a match it performs the desired action (*e.g.*, click, mouse over) over a specific point of the screen. The advantage inherent to this approach (*i.e.*, visually matching the contents) is the decoupling from the GUI's internals and the source code of the application. This property makes SikuliX a visual automation tool that is easily adaptable to different projects and GUIs. SikuliX supports test cases written in various scripting languages, namely in Python 2.7 (using Jython), Ruby 1.9 and 2.0 (using JRuby), Java, and in general, any other language that is Java-aware (*e.g.*, Scala, Clojure). Other functionalities consist of support for text recognition (OCR), which is provided by Tesseract and support for multi-monitor environments and remote systems.

- **Watir** [71] is an open-source (BSD license) automation tool for web browsers, which allows the development of tests that are easy to maintain and read. This tool emulates the same actions as a human, such as, clicking links, pressing buttons or filling forms. According to the previous classification, it is a "GUI aware" tool because it depends on the knowledge about the structure of the web page under test. It consists of a set of Ruby libraries, but can be used with service applications developed in any language. Currently, Watir, in itself, has limited support in Windows, where it only works with Internet Explorer. However, Watir-Webdriver extends the support to multiple browser, namely, Chrome, Firefox, Internet Explorer, Opera and HTMLUnit. The tests are written in Ruby, which in one hand makes it difficult for a beginner if he is not familiar with the language or programming, but on the other hand, provides a very high degree of freedom and functionality. For example, the choice of Ruby allows information stored in files or databases to be used in the test cases themselves.

- **Sahi** [72] is a paid automation tool targeted at service applications, which is available for any browser and operating system. The Sahi Controller, a part of the package, provides a visual interface to writing tests through recording the user actions in a browser, and playing back already created tests, among other less important features. Another component, the "Smart Accessor Identification", is used to identify these elements in a dependable manner, even in applications that make heavy use of dynamic Javascript and variable IDs. It features a scripting language, Sahi Script, which is heavily based on Javascript and is able to interact with the filesystem, CSV and Excel files, and databases. It also supports the calling of Java code to implement more advanced functionalities. It is a "GUI aware" automation tool with "Recording" features.

- **HP QuickTest Professional (QTP)** [73] is an automation tool, which is commonly used to aid in automated regression testing that is employed to diagnose software errors by comparing the actual results against the expected results. QTP uses Visual Basic Scripting (VBScript) as the

scripting language for the test cases. However, it provides two ways of creating these test scripts, one directed at beginners, which involves modifying fields of a table, and another aimed at experts, which allows the direct modification of the test cases' VBScript. It can be classified as a "GUI aware" automation tool. A disadvantage in comparison to some of its competition is that it is limited to the Windows operating system and, a proof of its age, still relies in obsolete technologies (*e.g.*, ActiveX and VBScript). Furthermore, not every browser is supported, as is the case with Opera. High licensing costs also make it prohibitive to use (a single seat licensing is around $7500 and does not include various extras) and even higher than other commercial options.

- **Ranorex** [74] is a proprietary application that provides a very user-friendly interface and easy approach to creating tests. It is a "Recorder" automation tool, which allows the user to perform various actions in his computer while Ranorex records them for posterior playback. For the more advanced user, it features an API for VB.NET and C#, instead of having its own standalone scripting language. There is no restriction in the type of application that Ranorex can record and test, and works in desktop, web and mobile platforms. The authors promise the most robust and reliable object recognition technology available (the so called RanoreXPath). However, a problem when using the recording functionality is that the system will suffer a significant slowdown while Ranorex attempts to identify the actions of the user.

Table 2 shows a comparison between the above mentioned automation tools, with a view to choosing the most adequate tool to use during this thesis.

Table 2: Comparison of Automation Tools

Name	Selenium	SikuliX	Watir	Sahi	QTP	Ranorex
License	Open-Source (Apache 2.0)	Open-Source (MIT)	Open-Source (BSD)	Open-Source & Commercial	Commercial	Commercial
Type	GUI-Aware + Recording	Visual	GUI-Aware	GUI-Aware + Recording	GUI-Aware	Recording + GUI-Aware
Target Applications	Web	General	Web	Web	General	General
Language	Selenese	Python + Ruby,Java,...	Ruby	Sahi Script	VBScript	C#, VB.NET
Ease of Use	4/5	5/5	2/5	4/5	3/5	5/5
Functionalities	3/5	2/5	4/5	3/5	4/5	5/5
Support for different systems	5/5	5/5	4/5	4/5	1/5	3/5

The classifications of "Ease of Use", "Functionalities" and "Support for different systems", are subjective and given by us, according to the obtained information and a brief hands-on experience with each tool.

Ranorex received a 3 out of 5 classification in the "Support for different systems" component, because, despite working with any type of application (web-based, desktop, mobile), it is only available in Windows.

Watir received a 2 out of 5 classification in "Ease of Use", because it requires the user to know how to operate and program in Ruby. A 4 out of 5 rating was given in "Support for different systems", solely because Watir, in itself, only supports Internet Explorer in Windows, however, Watir-Webdriver compensates for this weakness.

The 4 out of 5 classification given to Watir in the "Functionalities" component is due to the wide range of actions that it implements, and the fact it can harness the functionalities of Ruby (*e.g.*, use data stored in files or database).

QTP received a 1 out of 5 classification in "Support for different systems" because it lags behind in supporting new browsers, and its usage of obsolete technologies creates a strong dependency with Windows.

The choice for the automation tool to be used during this thesis fell over SikuliX, despite not being as complete as other tools, it has all the required functionalities, is very easy to use and the test cases can be created and modified quickly, requires no need to know the scripting language before hand, is open-source and does not carry significant performance overhead. Another good option would have been Selenium, however its shortcomings in dealing with very dynamic webpages were deemed too grave.

3 Approach

In this section, the approach for mutating data coming from database developed during this thesis is explained as generally as possible. This section presents the general approach, compares it with other viable approaches, provides the justification for our choice and states the key implementation details, such as the fault model that was used and how it was obtained. Other details of the approach, such as the methodology used to intercept the data, are also presented.

3.1 General Approach

The main aim of this thesis is to study how a service application that makes use of a database responds to the existence of poor data quality, and thereby to understand the impact and influence of poor data quality.

This study, in a general way, can be accomplished by following the steps shown in Figure 1.

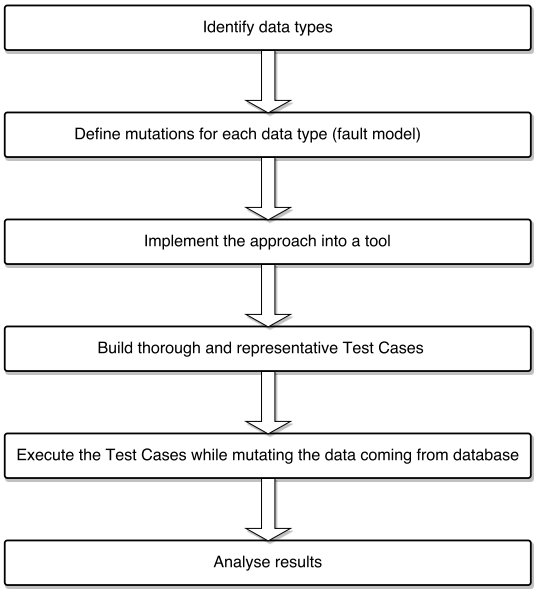

Figure 1: General workflow to study service applications using poor quality data

Using the diagram to guide our work, an approach was developed, which will be detailed below, that allows data to be mutated between the database and the service application.

The knowledge of the structure of a typical service application is essential in the development of this thesis. Once again it is helpful to remember that the objective is to be able to feed the service application with poor quality data, as if it had come directly from the data tier (database). A usual service application

follows a three-tier architecture with a separation between the presentation, logic and data tiers. A client's request usually follows a path from the moment that the client performs its request, for example, by interacting with a browser, until a response arrives. First the client uses the graphical user interface of the web application, shown in a browser, to request an action. The browser will then send the corresponding request over the network until it reaches the logic tier in the server. At this point, the request is processed by the logic tier, and if needed this tier can communicate with the data tier in order to fulfill the client's demand. When the logic tier has finished processing, it sends the response back to the client over the network, and the browser displays it to the user. Figure 2 has a representation of this process.

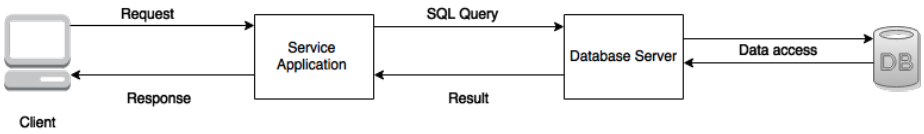

Figure 2: Usual Scenario of a Service Application

It is not difficult to understand that for this purpose the most straightforward solution is to directly change the data that we desire in the database itself. While this is a valid solution, it presents a few disadvantages that severely limit its usefulness. Firstly, it would require experiments that would take a long time because of having to change the desired value in the database, and, after the experiment is finished, restore it to the original value. Secondly, it would require the precise and hard to obtain knowledge that a certain value in the database would be used by our test cases. Without this information we risk that a mutation would in reality never be used by the web application, simply because, for that test case, the web application never reads it from the database. Thirdly, while modern DBMS have a set of tools (*e.g.*, foreign keys, triggers, not null constraints) capable of preventing some integrity issues, the web application should not assume that this functionality will always be available. The web application, if it wishes to be robust independently of which DBMS it works with, it must implement its own basic data quality checks. Therefore, by tying our approach to a specific DBMS and its tools, we would be limiting the mutations that we can be performed without modifying the database schema (*e.g.*, if the DBMS has not-null constraints, and the schema of the service application uses it, we cannot perform a mutation that changes a certain value to *null*, because the DBMS will refuse performing that action, as it would violate the constraint). Fourthly, this approach calls for access with write permissions on database, which cannot be attained in certain scenarios. Finally, once again we are unnecessarily associating our experiments with a DBMS and thereby becoming less portable.

A more suitable approach, and which was the one chosen for this thesis, is to mutate the value while in transit from the database. This allows for a higher

abstraction and independence from the DBMS (*e.g.*, MySQL, PostgreSQL, Oracle) that is being used, and also foregoes the need for having any access to the database. It is also much quicker and easier to perform mutations that would otherwise conflict with the in place DBMS' constraints. Furthermore, we can be sure that the mutated value will be fed to the application (*i.e.*, we only mutate data that the application requests from the database), and may later be used. However there is a downside to this approach: the implementation of permanent data mutations (*i.e.*, values in database that will keep the same mutated value constantly) is very hard and slow to do. Instead this approach injects non-persistent data mutations. Depending on the configuration, two database reads to the same location will return different results because of different mutations (the same cell suffered two mutations). While this is not the most desirable behaviour, it is enough to test the web application's handling mechanisms, and only occurs when a value is read more than once.

To control the approach, a mechanism capable of starting and stopping mutation of the data access calls at request of the user must be developed. Our proposal is to use a simple file in the system storage, which holds a binary value (*i.e.*, 0 or 1) according to whether mutations should be performed or not. The user, or any other program, is then able to change this value at will, which an intercepted data access call must check before proceeding with a mutation.

To allow the study of a service application under the influence of poor data quality, another mechanism must be implemented, which allows certain mutations to be skipped, according to certain factors, such as the location of the instruction of the service application that is calling the database. This mechanism is an essential part of this approach because it allows operation after the service application has crashed or blocked (*i.e.*, could not handle poor data quality), by skipping the mutations that cause the problem (after an analysis has been carried out).

When looking at each experiment run of our approach, the flow inside each run can be grouped into 3 consecutive phases, represented in Figure 3, which have a well-defined purpose.

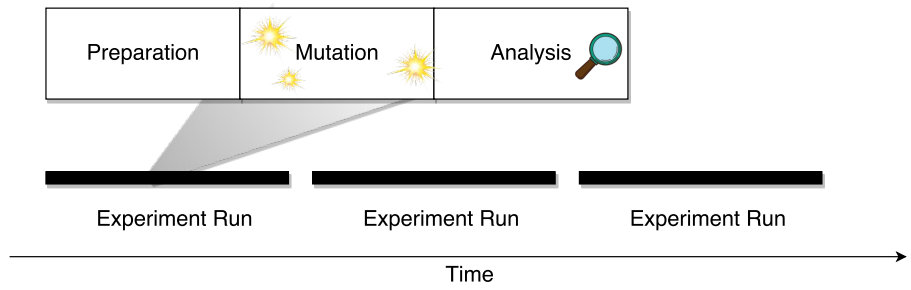

Figure 3: The three phases that make up an experiment run

An experiment run starts with the **"Preparation"** phase, where the application is warmed-up for the next steps. This phase commences by starting the

web server and sending requests (*e.g.*, obtain home page, perform login) to the service application, in order to prepare the system for the following steps.

The next phase is **"Mutation"**, where the tool is allowed to mutate the data coming from database (by changing a file in the system storage), at the same time that a workload is being executed. At the moment, this phase continues until the application is not capable of responding to the actions of the user. However we admit that different termination conditions could exist, such as for example, stopping after a certain percentage of database accesses have been mutated. This phase may vary in time depending on the number of database accesses and user requests, or if the application fails to work prematurely. It can also be considered the most important phase of the approach, because it is here where the application is really put under test.

The last phase is **"Analysis"**, which is where the results obtained during the **"Mutation"** phase are analyzed, in order to prepare the next experiment run. It starts with exporting and manually analyzing the log file and extracting information such as in which part of the code the mutation happened, what was the original value, and what is the new mutated value. The main objective of this phase is to be able to identify the reason behind the application failure, and to identify which mutation must be skipped in the next run, so that the application can execute further. When the application's code is being analyzed, if a software bug is detected, it should be noted.

As shown in Figure 4, the Data Mutation Tool is located between the Service Application and the Database Server, and mutates the query's results that are sent from the Database Server to the Service Application.

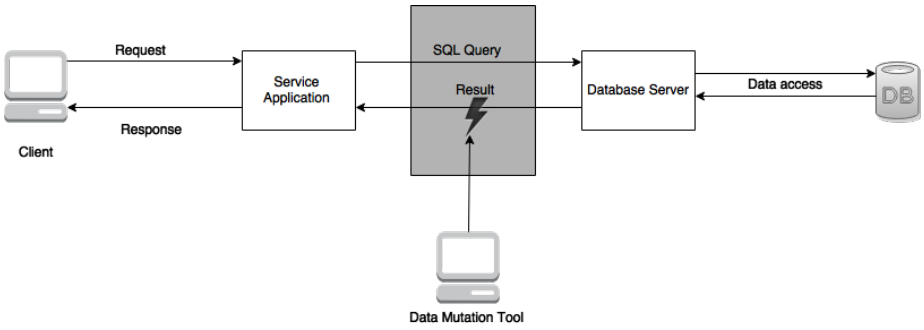

Figure 4: Scenario with a Service Application and our Data Mutation Tool

The algorithm inside the Data Mutation Tool, which executes whenever a data access is made during the **"Mutation"** phase, is described in Figure 5.

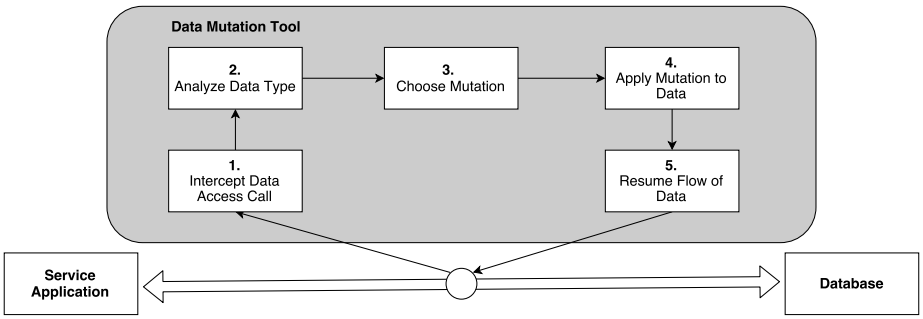

Figure 5: Basic algorithm of the Data Mutation Tool

Our concept is generic, therefore the approach can be applied to a wide range of programming languages and data access drivers. In this thesis, we focused in applying our approach to a Java environment, where one of the available data access drivers is JDBC (Java Database Connectivity). We opted to implement our approach by instrumenting this driver to mutate the data coming from the database.

3.2 Implementation of a Poor Data Injector

To put into practice the theoretical approach, delineated in Section 3.1, we proceeded to develop our own implementation that could suit our needs for future experiments. Two ways of tackling this problem were considered, modifying the exchanged network packets between the service application and the database to modify the responses, and, instrument the JDBC driver that provides the connection between them. The first adversity that must be handled is that service application and database are forced to communicate over the network, even if they reside in the same physical machine, the possibility of local communication not allowed. This fact introduces an artificial restriction into the setup, and leads to lower performance and higher latency without adding any benefit. The other problem is that capturing network packets is difficult to do if we consider that we must know which packets are coming from the database to the service application, which of them are responses (because at the moment SQL queries are not of our interest), where the data is located inside the packet and what data type it is.

Another approach is to modify the responses of JDBC calls, which is a more suitable and effortless way to apply mutations into the data. This approach supports both local and non-local communication between service application and database. For this purpose the JDBC driver which provides the connection between application and database can be instrumented. In our case we opted to use AspectJ [75] and define an Around annotation around JDBC's getString(), getInt(), and similar functions that can be used to read the results from database. An AspectJ Around annotation allows the interception of every function call that obeys a certain rule. When a function call is intercepted,

the code that has the annotation is called instead of the original function. The annotated code is then free to perform the actions that it desires, which includes calling the intercepted function and using or changing its return value. For our specific case, in this annotation we added code that would receive the value sent by the database, and then according to certain variables, perform a data mutation and return this value instead of the one originally sent by the database.

In Listing 1, an example is provided of how this instrumentation could be done using AspectJ.

Listing 1: Example of the use of AspectJ to instrument the JDBC driver

```
1  @Around("execution(* java.sql.ResultSet.getDouble(..))")
2  public synchronized Object logAroundDouble(ProceedingJoinPoint
3      joinpoint) throws Throwable{
4
5      Object result = null;
6      result = joinpoint.proceed();
7      Double resultDouble = (Double) result;
8      ( ..... )
9      return resultDouble;
10 }
```

The first line of the code listing contains the AspectJ annotation that states that whenever a function that matches the supplied rule is called (in our case, whenever *java.sql.ResultSet.getDouble* is called, independently of the passed parameters), the function below must be called before. Inside the *logAroundDouble* function, we can see that it receives a *ProceedingJoinPoint*. By calling the *proceed* function of this object, we allow the execution flow to continue, and receive the resulting output. Armed with the result of *java.sql.ResultSet.getDouble*, we can later modify its value and return the modified value to the service application.

The use of AspectJ brings many advantages over manually modifying and recompiling the JDBC driver. For instance, it can easily be applied to any JDBC driver (*e.g.*, driver for MySQL, Oracle, PostgreSQL), because it depends only on the JDBC specification that must be implemented by all drivers.

At the same time, our proposed approach logs various information that is essential for later analysis in two distinct places, which are the data mutation tool itself and the Servlet Filter (which will be explained later). The data mutation tool logs all the mutations that have been performed, along with details about their time, original and modified value, mutation that was applied and the stack-trace of the code that received the mutated value.

In Listing 2 we present a sample of how an entry in the log of the performed mutations looks like. The log starts by presenting the time when the mutation occurred and then presents the stack-trace containing the hierarchical tree of functions called until the mutation occurred. The log also keeps the *input value* which corresponds to the original data value before mutation, the mutation

(which includes the data type) and the *output value*, which is the result of mutation after it is concluded. In this particular example a *moveSubstring* mutation was performed over a string (571).

Listing 2: Sample of a log file with mutations

```
Time: 2016-03-31 02:24:19.362
Stack-Trace of Mutation caller
java.lang.Exception: Stack-Trace
( ........ )
 at org.hibernate.loader.Loader.doList(Loader.java:2533)
 at org.hibernate.loader.Loader.listIgnoreQueryCache(Loader.java:2)
 at org.hibernate.loader.Loader.list(Loader.java:2271)
 at org.hibernate.loader.hql.QueryLoader.list(QueryLoader.java:452)
( ........ )
Input (before mutation): 571
Used Mutation: String_m11_moveSubstring
Output (after mutation): 517
```

We developed a Servlet Filter that resides at the entry of the service application and logs the requests that the client makes, corresponding to the second and final location in our approach where a log is created. While this component is not an essential part of the proposed approach, and therefore is not described in Section 3.1, it allows an higher degree of control over the requests that enter the system, which will prove useful in the experiments that will be performed. We consider each request to be an User Action, and we enforce a policy that only allows one concurrent request at any time (by using the *synchronized* keyword in the filter), in order to easen the analysis of the results when attempting to obtain all the mutations that occurred during the window of a certain request. This filter has a data logging function but can also be used to start and stop the data mutation process, according to certain criteria (*e.g.*, one data mutation per User Action).

The previous explanation of the individual components is complemented, in Figure 6, with the flow inside the system and the interactions between the components,

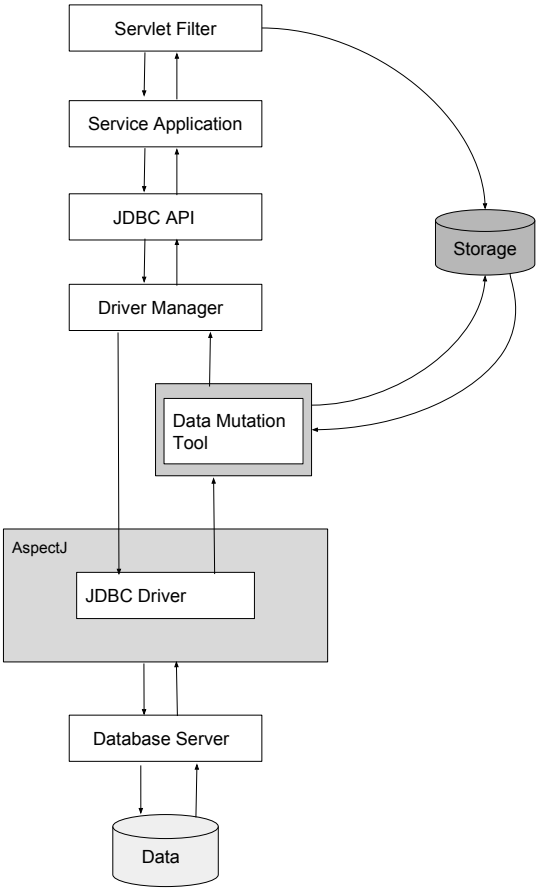

Figure 6: How the Data Mutation Tool integrates into the JDBC flow

The Servlet Filter is located between the Service Application (running in the Server) and the Browser (of the Client), and intercepts the network requests that the Browser performs. These requests are then stored in the Storage. In some occasions the Servlet Filter is also capable of communicating with the Mutation Tool to start and stop the mutation process.

The Mutation Tool stays between the Service Application and the Database (DBMS) and is capable of writing logging information to the Storage, and reading files that act as an interface to control the operation of the Data Mutation Tool.

To mutate the data coming from database in a representative manner it is necessary to know which data quality issues can occur in the real world. This knowledge is however not readily available in the literature. To produce these essential assets, a study of data quality was performed in order to extract as

much information as possible to aid the process of creating these mutations.

3.3 Identifying Data Quality Problems

In the process of implementing our approach into a tool, the first step was the creation of the available mutations for each of the various data types (*i.e.*, fault model). Because of having opted by instrumenting JDBC as a way to intercept all data access calls, we have become restricted to the internal data types of JDBC. For this reason, a comparison between the JDBC [76], SQL and Java native types was required before equivalence classes that serve as the data types of our fault model could be formed. The mapping between the data types is presented in Table 3:

Table 3: Mapping between our self-defined groups, JDBC types and Java types

Equivalence Class	**JDBC Type**	**Java Type**
String	CHAR	String
	VARCHAR	
	LONGVARCHAR	
	CLOB	Clob
	DATALINK	java.net.URL
Decimal	NUMERIC	java.math.BigDecimal
	DECIMAL	
Boolean	BIT	boolean
	BOOLEAN	
Integer	TINYINT	byte
	SMALLINT	short
	INTEGER	int
	BIGINT	long
Double	REAL	float
	FLOAT	double
	DOUBLE	double
Binary	BINARY	byte[]
	VARBINARY	
	LONGVARBINARY	
	BLOB	Blob
Date	DATE	java.sql.Date
Time	TIME	java.sql.Time
Timestamp	TIMESTAMP	java.sql.Timestamp
Object	DISTINCT	(Depends on Object)
	JAVA OBJECT	
Reference	REF	Ref

Overall, there are 11 different groups of data types and each group can aggregate more than 1 JDBC or Java data type. Sometimes different JDBC types translate to the same type in Java. For each of these 11 groups, a set of

applicable mutations was built, and then implemented in Java code for usage in our tool. It should be noted that the fault model used in our tool includes only the mutations that can be implemented in Java. Mutations that despite being possibly valid but due to limitations of the programming language or of the data type cannot be done are not presented here (*e.g.*, it is impossible to have an empty Java *int* variable, but the user is free to leave the input empty in the user interface). In practice, this fact should not limit the effectiveness of the proposed fault model, because the service application, if written in Java, would suffer from the exact same limitations in the presence of such mutations, and would have to somehow deal with them (*e.g.*, using the previous example, a possible approach would be for the user interface to complain and not proceeded with an empty field, or for a default value to be assigned to the variable).

As the basis of our work, we reused the mutations for Strings proposed in [59], but then proceeded to carry out an extensive survey about data quality, with particular attention given to the data quality problems that occur in practice, so that we could justify the mutations proposed by Ivaki et al [59] and complement the original model with more mutations, including mutation operators for all the other data types.

As presented in the survey paper that resulted from our study of data quality, across the literature there is a common and recurrent group of data quality problems. Table 4, which was extracted from our survey paper [77], presents these data quality problems, grouped according to source and level [32], and associates each problem with the data quality dimensions that it can affect.

Table 4: Data Quality Problems mapped into Dimensions

Problem types		Data quality problems	Accessibility	Accuracy	Completeness	Consistency	Currency
Source	**Level**						
Single	**Instance**	Missing data		•	•		
		Incorrect data		•			
		Misspellings		•			
		Ambiguous data	•	•			
		Extraneous data	•			•	
		Outdated temporal data		•			•
		Misfielded values	•	•	•	•	
		Incorrect references		•			
		Duplicates	•				
	Schema	Domain violation		•			
		Violation of functional dependency		•			
		Wrong data type	•			•	
		Referential integrity violation	•	•	•	•	
		Uniqueness violation		•			
Multiple	**Instance**	Structural conflicts	•			•	
		Different word orderings	•			•	
		Different aggregation levels	•	•		•	
		Temporal mismatch		•		•	•
		Different units	•			•	
		Different representations	•			•	
	Schema	Use of synonyms	•				
		Use of homonyms	•				
		Use of special characters	•				
		Different encoding formats	•			•	

The data quality problems stated above are mostly general and not tied to any data type. However, our research showed some of these data quality problems can have different representations according to the data type being considered (*e.g.*, although *Incorrect Data* is a general problem that can affect a multitude of data types, it occurs differently in a String type than in an Integer). For this reason, a study about these particularities ensued.

When focusing on the String data type, the possible reasons (spelling errors) for strings are discussed in a wealth of papers [78, 79]. The widely agreed

classification of spelling errors defines four main groups, which are omission, insertion, substitution and transposition. Errors of omission, where a letter is left out, constitute 30 - 40% of the errors; errors of insertion, where an extra letter is added, constitute 25-35% of the errors; errors of substitution, where an incorrect letter is substituted for the correct letter, constitute 15-20% of the errors; errors of transposition, where two adjacent letters are interchanged, constitute 10-15% of the errors. Estimations about the occurrence of these four error types are presented in [80], where the authors note that these four type of errors constitute 80% of all misspelling errors, and in [81], where this value rises to about 95%. In addition to these four types, another paper[82] proposes the use of two more types: Added space and Dropped space. This addition is justified with the non negligible percentage of occurrences of these two errors (7.12%).

Literature regarding other data types is less prevalent, but nevertheless can be found. This lack of information is minimized by the fact that plenty of data formats (*e.g.*, integer, date) can be represented as a String, and share the same entry mode (*e.g.*, keypad), therefore making it possible to share some string-specific mutations among data types. This observation is based in plenty of anecdotal evidence [83, 84].

Errors in Dates are mentioned in [85], where the authors conclude that a notorious error often found in their dataset was the incorrect and abusive use of the default value.

When talking about integer and decimal data, Thimbleby et al [86] introduce what they call a standard class of error: the miskeying of a decimal point or zero. The paper also presents other common problems in a very practical approach, by stating an incorrect value or invalid key combination and presenting the final output in a variety of systems, ranging from spreadsheets, pumps, mobile phone apps, search engines, handheld calculators to office software. The authors also highlight the importance that errors such as dropping a decimal separator, ignoring the integer part of a decimal number or miscounting the number of digits (with special emphasis to the digit '0') in a very large number can have.

The data mutations that were created and used during this thesis are displayed in Appendix A. As an example, the mutations for the String data type are presented in Table 5, where the string *"John Smith"* was used as the example input.

Table 5: Mutations for String data types

No	Fault Description	Argument	Configuration	Example
1	Replace by null value	-	-	null
2	Replace by empty string	-	-	“”
3	Replace by same size string	-	-	Aks9DLM34q
4	Add random nonprintable characters to string	Position	Prefix	\u0010John Smith
			Middle (Random position)	John\u0010 Smith
			Suffix	John Smith\u0010
		Length	Random between 1 and string's original size	-
5	Replace by same size alphanumeric string	Number of characters	Random quantity of characters between 1 and the total number of alphabetic characters minus 1. Randomly distributed among the available position	Jo3n Smixh
6	Add random numeric characters to string	Position	Prefix	0158John Smith
			Middle (Random position)	John01 S4mi5th
			Suffix	John Smith544
		Length	Random between 1 and string's original size - string's maximum size	-
7	Convert alphabetic characters to opposite case	Position	First character	john Smith
			Last character	John SmitH
			Random position	John sMITH
			First character of each word	john smith
			Last character of each word	JohN SmitH
			Random position in each word	JOHn sMith
		Length	Random between 1 and string or word original size minus Position	-
8	Insert whitespace	Position	Leading	John Smith
			Trailing	John Smith
			Random position between words	John Smith
		Quantity	The amount of consecutive white space characters to insert	-
9	Add characters to string	Position	Prefix	ooJohn Smith
			Middle (Random position)	John Shhhmith
			Suffix	John Smithii
		Quantity	Random between 1 and the string's original size. The new characters to add must be present in original string.	-
10	Add substring to string	Begin	From 0 to the string's original size minus 1.	-
		Length	From 1 to the string's original size minus Begin.	-
		New Position	Prefix	ohnJohn Smith
			Middle (Random position)	Johnohn Smith
			Suffix	John Smithohn
11	Move substring	Begin	From 0 to the string's original size minus 1.	-
		Length	From 1 to the string's original size minus Begin.	-
		New Position	Prefix	ohnJ Smith
			Middle (Random position)	J Sohnmith
			Suffix	J Smithohn
12	Remove substring	Begin	From 0 to the string's original size minus 1.	J Smith
		Length	From 1 to the string's original size minus Begin.	
13	Remove whitespace	Position	Leading	John Smith
			Trailing	John Smith
			Random position between words	JohnSmith
14	Delimited reverse	Begin	From 0 to the string's original size minus 2.	John Shtim
		Length	From 2 to the string's original size minus Begin	
15	Replace by predefined SQL string	-	Dictionary based: replace the string by a keyword or symbol such as SELECT, FROM, INSERT, DELETE, UPDATE, ', ",	SELECT
16	Replace with symbols	-	The string is replaced by a random symbol	$
17	Misspelling	-	Replace one word by a misspelled word (Dictionary-based) or in case of no match being found, use a random single edit operation (insertion; deletion; substution of a single character or transposition of two adjacent characters) over a randomly selected word	John Smtih
18	Replace with a limit size	Size	Java's UTF maximum size	-
			Java's UTF maximum size +1	-
19	Replace with an imprecise value	-	Dictionary based: Chooses a single random word and replaces it by the respective acronym or abbreviation	J. Smith
20	Replace by specific type string	-	Replace value with another data types (e.g., numeric, date, time, timestamp, boolean mutations, to be defined)	30
21	Replace whitespaces by	-	Replaces all the white spaces by symbols (_, . , %, $, €)	John_Smith, John%Smith
22	Add extraneous data	Position	Before	PhD John Smith
			After	John Smith CEO
23	Replace by homonyms	-	Replaces one word by its homonym, when possible.	allowed -> aloud
24	Replace by synonyms	-	Replaces one word by its synonym, when possible.	happy -> cheerful

3.4 Failure Classification

In order to make sense of the information obtained during our experiments, a way of classifying it must be used. Only then the content of interest contained in the results can be obtained.

One of the objectives of performing data mutation tests is to study the behaviour of the application in the presence of bad quality data. The behaviour of the application can differ because of the occurrence of a failure in the application, and this can perceptible from the user's point of view or from the server's internal view. To classify the different behaviours that a service application can have, a classification scale had to be defined.

The CRASH scale [50], which is a classification originally designed for the robustness of operating systems, soon appeared to be a good choice. However there were some limitations because the scale was not aimed at the study of service applications, but rather operating systems.

Since the experiments done during this thesis focus on service applications, the CRASH scale had to be slightly modified to fit our needs, but without modifying the original rationale. Below is the minimally modified CRASH scale that we used for each classification.

- **Catastrophic:** A catastrophic failure occurs whenever a failure is not contained within the user action where it occurred and affects the entire system, causing the service application to stop working completely and all current and newer user actions to fail. It requires the restart of the service application or webserver to recover from it.

- **Restart:** This classification is given whenever an user action hangs, *i.e.*, it becomes unresponsive to inputs and does not provide any visual indication of progress. This failure does not propagate to the system and can usually be recovered from by reattempting the same user action. To detect its occurrence a timeout mechanism is implemented that detects if a certain user action has stopped providing any response since a certain amount of time.

- **Abort:** This failure mode occurs whenever an user action interrupts its execution abruptly, usually accompanied by some sort of error message that indicates to the user that a problem has occurred.

- **Silent:** This failure mode occurs whenever the service application fails to behave according to what would be expected from it, yet does not display any visual message or error log that can indicate this situation.

- **Hindering:** The provided error message or exception is incorrect and hinders the correct diagnosis of problem.

We use this version of the CRASH scale to classify the behaviour of the service application under test in the Results section. When comparing the original CRASH scale with our proposal, the differences amount only to an

adaption from the reality of an operating system to what is present in a service application. For example, the original scale uses the return codes of the system calls as a means to classify the failures, however this feature is not available in a service application, instead being replaced with visual messages and error logs.

4 Experimental Evaluation

This section deals with the specific characteristics of the experiments that were carried out during this thesis to: verify the applicability, effectiveness and usefulness of the approach detailed in Section 3, and obtain a study of a real-world service application in the presence of poor quality data. This section is subdivided into three parts: *Experimental Setup* (Section 4.1), which has the details about the hardware and software where the experiments were carried out; *Test Cases* (Section 4.2), which describes the test cases that were used and why; *Description of Experiments*(Section 4.3), which has the details of the experimental process.

4.1 Experimental Setup

To test the effectiveness of our approach a setup needed to be found that was representative of the real-world usage of service applications and that made use of a database system. Many open-source service applications were considered for taking a part in the experimental setup, among which were OpenCMS and DotCMS, two of the leading CMS solutions. In the end the choice was to use Openbravo© 3.0, because of the high number of supported DBMS, size of the code base and wide variety of functionalities.

Openbravo is a commercial open source ERP business solution for Small, Medium and Large enterprises. It allows companies to manage their entire business solution, and supports the following processes: Sales, Manufacturing, Procurement, Projects, MRP, and Finance. Furthermore, it has functionalities like creating and exporting reports in several formats, including Microsoft Excel and PDF [87]. With its all-purpose ideology, Openbravo reached high market penetration, holds a very high place in Open Source solutions of this area [88], and has a series of publicized success histories [89].

Openbravo has built-in compatibility with five different database systems: Oracle, MySQL, PostgreSQL, Apache Derby and HSQLDB. In the experimental setting, we opted to pair Openbravo with PostgreSQL 9.3. PostgresSQL [90] is a powerful open-source object-relational database system with plenty of features and a significant share in various real-world scenarios [91, 92].

In order to run the chosen service application (Openbravo), a web server that supports the Java EE technologies (*e.g.*, Java Servlet, JavaServer Pages) must be used. Nowadays, there is a good number of open-source projects available that can fulfill this role. For the experimental setup, we opted for Apache Tomcat 7.0.68, in part because of the familiarity that had already been acquired with it, but also because of the decent market share that Apache Tomcat has [93].

All of our experiments took place in the same physical machine, which meant that the server and the client resided in the same machine. The setup of the client tried to emulate one of the many possible combinations that can be seen in real-world.

Openbravo is a trademark of Openbravo S.L.U. and is used under license.

For the browser we opted to use Mozilla Firefox 45.0.2, a well-known name with a significant share of the browser market [94] and therefore likely to be found in the wild. The browser is used in our experiments to provide the interface with the service application.

Because of the need for automating our testing process as much as possible, we used SikuliX 1.1.0 [70], a versatile visual automation tool which can reproduce clicks and other actions that a human user can do.

The experiments were executed in a ASUS N55SL laptop, equipped with a Intel Core i7-2670QM processor running at up to 3.10GHz, 8GB of memory and a 1TB hard drive. The Operating System was Windows 7 64-bits.

The experimental setup can be seen in a condensed format in Table 6.

Table 6: Hardware and Software characteristics of the Experimental Setup

Hardware		*CPU*	Intel i7-2670QM @3.10GHz
		Memory	8 GB
		Hard Drive	1 TB
		Operating System	Windows 7 (64 bits)
Software	**Server**	*Service Application*	Openbravo 3.0
		Web Server	Apache Tomcat 7.0.68
		DBMS	Postgres 9.3
	Client	*Web Browser*	Mozilla Firefox 45.0.2
		Visual Automation	SikuliX 1.1.0

The above configuration, in terms of software choices, is representative of a normal commercial installation that can be found in the real world. In terms of hardware, we were limited by the available options, and therefore instead of a more professional system, a personal laptop had to be used.

4.2 Test Cases

Various test cases were developed to exercise the multiple components that make up Openbravo. Since Openbravo is a complex application and very time consuming to thoroughly test, just a limited number of test cases, able of being performed inside the timeframe of this thesis, were designed. The test cases took into consideration the CRUD model [95] which represents the four basic operations that can be performed in a persistent storage (*e.g.*, database): CREATE, READ, UPDATE and DELETE. These four operations can easily be mapped to their corresponding SQL statement: INSERT, SELECT, UPDATE, DELETE. By using this model when creating the test cases it becomes easy to ensure that all basic operations are represented by the tests. A brief list of all test cases accompanied by the CRUD dimensions that each test features is presented in the Table 7.

Table 7: Test Cases and how they fit in the CRUD model

Test Case No	Test Case Name	CRUD
Test Case 1	Login	READ
Test Case 2	Create Organization	CREATE
Test Case 3	Create a new User	CREATE
Test Case 4	Create a new Role	CREATE
Test Case 5	Create Product	CREATE
Test Case 6	Delete Product	DELETE
Test Case 7	Update Product	UPDATE
Test Case 8	Export Product Categories to Speadsheet	READ

All the chosen test cases are relatively complex and also perform read operations (*e.g.*, visualize a detailed list of the content that already exists). However the classification was given according to the main purpose of each test case.

For each test case, the corresponding activity diagram was drawn. Figure 7 shows the activity diagram for the "Create Product" test case (Test Case 5). The remaining diagrams can be seen in Appendix B. The test case starts with the steps that must be performed to browse until the window that allows the creation of a new product, which starts in "Click in 'Application'" and finishes in "Click in 'New Product' icon". At this point the user must perform certain steps, which consist of data entry, without the need of following a specific order. When all these steps are done, the user can then finish the test case by saving the new product and exiting from that window.

4.3 Description of Experiments

Each experiment exercises different areas of the service application, yet all experiments share the same flow, only differing in the test case that will be executed. A visual representation of this flow is shown in Figure 8, and a connection is made with the three phases of our approach (Preparation, Mutation, Analysis) by assigning a certain color to each step.

The purpose of the *Preparation* phase is to execute all the necessary steps needed to prepare the system to execute the remaining phases. At this point no data coming from database has been mutated. The first step of the experimental process is to fire up Apache Tomcat and wait for it to fully load. After the web server has started, a new browser instance is launched and used to browse to Openbravo's login page. At this point, two different paths can be taken, depending on the test case that is being executed. In most test cases, the flow is to log-in into Openbravo using the "administrator" account, and waiting for the main page to load. However there is one test case (Test Case 1 - Login), aimed specifically at testing the login mechanism, where the *Preparation* phase ends right after Openbravo's main page has loaded.

After the *Preparation* phase has finished, we commence the *Mutation* phase

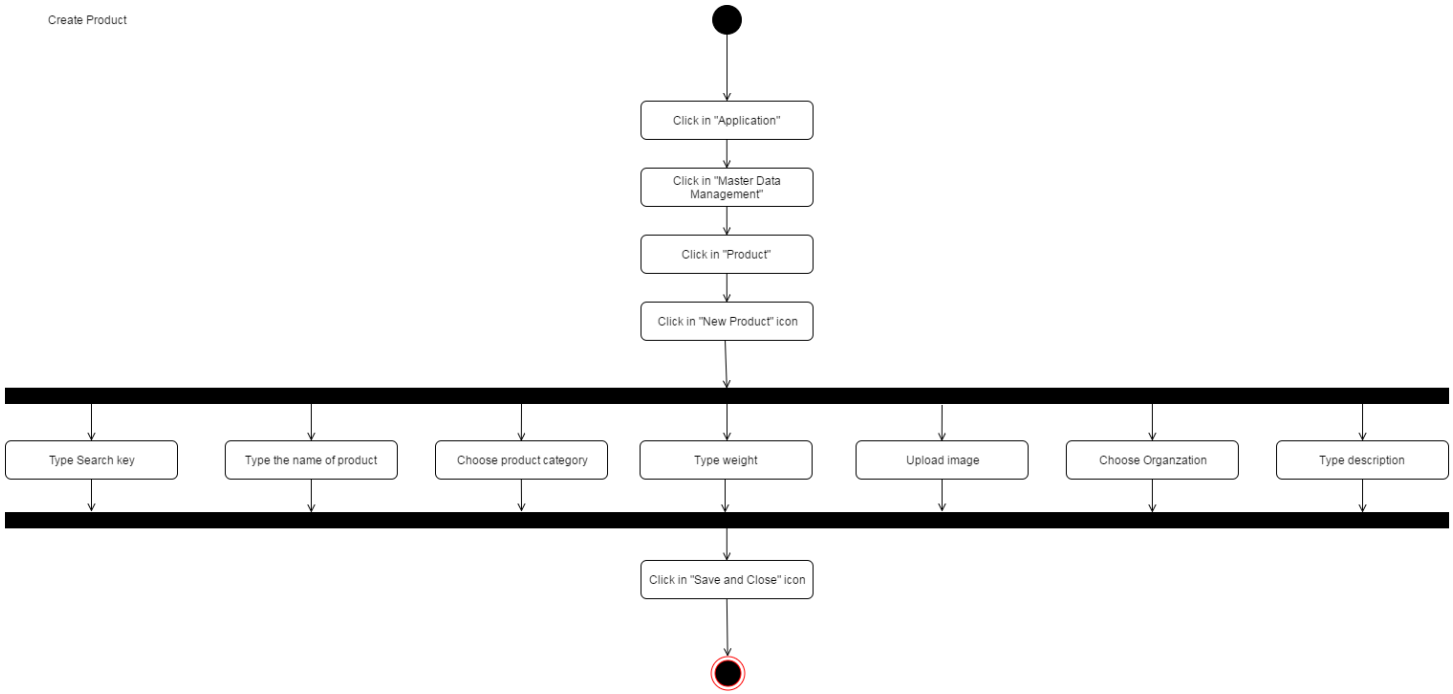

Figure 7: Activity Diagram for Test Case 5 - Create Product

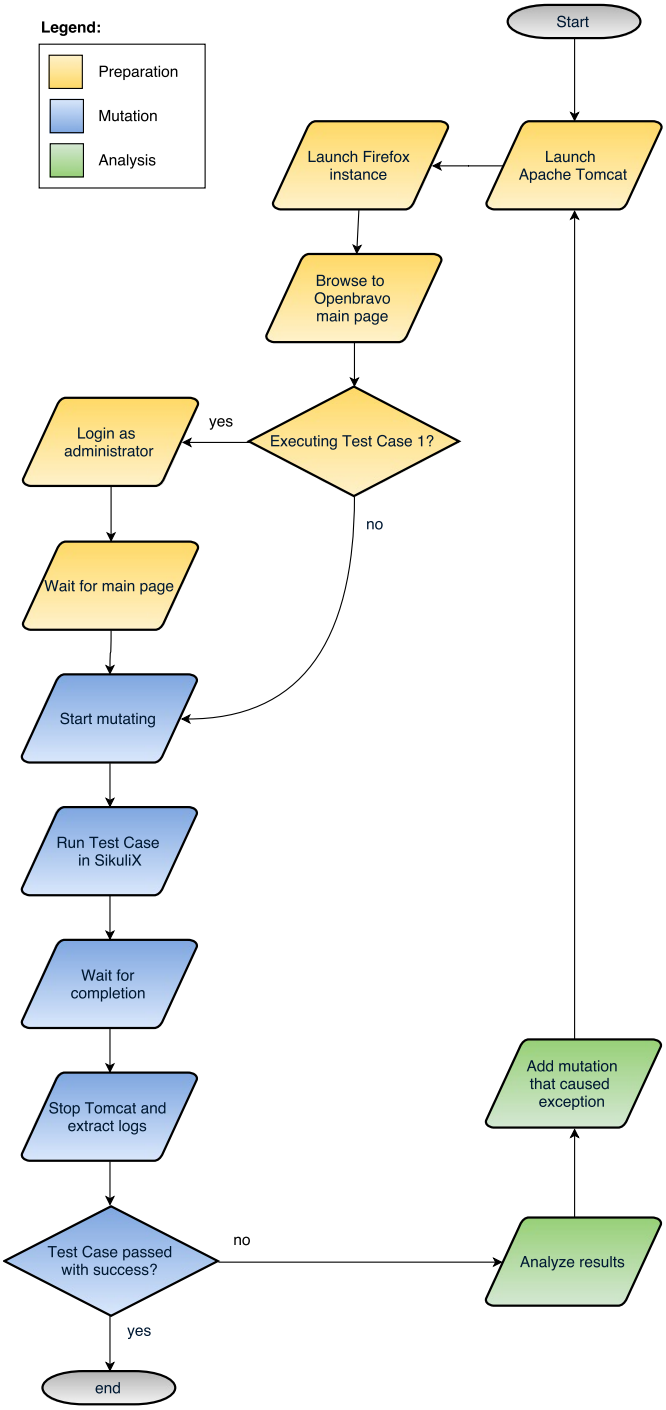

Figure 8: Flow of an experiment

by signaling the tool that it can start mutating the data coming from database, through the modification of a binary value stored in a pre-defined file in the system's hard drive. With the mutation tool enabled, SikuliX is used to accurately reproduce the steps of the test case, which can take some minutes. Along the way we take note of the behaviour that Openbravo exhibits by taking screenshots. When the test case has finished executing, Tomcat is stopped and the various logs produced by Openbravo, Apache Tomcat and our mutation tool are obtained.

At this point we enter the *Analysis* phase. Armed with the information obtained until this point, it is possible to understand how the application's behaviour was affected by data with poor quality. If the test case has completed successfully (the application showed the expected behaviour), this experiment has finished. However if the test case failed (application had an unexpected behaviour) the flow returns to the first step, but only after analyzing the log files to understand which mutation caused it to fail, and then adding this mutation to a file that keeps track of all. When a mutation is put in this file, it will not occur anymore, because next time the data mutation tool will detect and skip it. By doing this we are sure that in the next run the same mutation cannot cause that exception (thereby the same exception will not happen again, unless it is caused by another different mutation). This last step allows us to proceed with our tests, while finding problems inside Openbravo that can later be fixed by proposing changes to the source code. The experiment stops only when the test case passes successfully, until then we repeat the above procedure.

5 Results and Analysis

The results presented in this section include a classification of the behaviour of Openbravo under the presence of poor quality data, as well as various software defects in Openbravo and in two libraries (Hibernate and PostgreSQL JDBC driver). In second plan, an analysis of the exceptions that occurred during the experiments and statistics relating to the experiments is included, with a view to better characterize the nature of our experiments. At the end of this section, the obtained knowledge is summarized and simple programming guidelines that can prevent the witnessed programming flaws are presented.

5.1 Overview of results

The analysis of the results provided by the experimental evaluation lead to various observations. In this introductory subsection, we present a brief overview of the key points that will help the reader digest the following subsections.

One of the main goals behind the experiments was to characterize the behaviour of the service application under poor quality data, according to a predetermined scale (in our case, an adapted CRASH scale). This analysis returned the failure modes that happened in each test case, and a posterior analysis was made to retrieve the reason behind them.

The most common failure mode was *Abort*, which means that the application was unable to finish executing a certain task. One particularly interesting example of this failure mode is when a ClassNotFoundException occurs, which means that Openbravo has failed to dynamically load a certain class, and, therefore, the functions of that class have become unavailable. To the user this means that a part of the functionality is not present. This specific problem can only be solved by logging out and then logging in again, to force Openbravo into reloading the failed class. Otherwise the functionalities will always stay disabled for the current session.

Another noteworthy occasion, which is also the only *Catastrophic* occurrence witnessed during the experiments, occurred because a certain mutated value was stored in the browser's session, which lead to a service terminating exception appearing every time an attempt was made to access Openbravo, until a different browser or private session was used, or until Apache Tomcat was restarted.

Furthermore, a few occurrences of the *Restart* failure mode were also detected, which, after analysis, were attributed to two different causes. The first cause is the fact that, occasionally, the user interface did not reflect the unexpected termination of a task (*e.g.*, by showing an error message), and transmitted an everlasting loading screen instead. The other situation occurs when the user is forced to log out (*e.g.*, Openbravo detects that the session has expired), but this action takes an excessive amount of time, and the timeout limit used in the experiments (3 minutes) is exceeded by far, hence a *Restart* classification being attributed.

Finally, it should be noted that we witnessed occurrences of the *Silent* failure mode, which were caused by a mutated value that did not raise any exception,

went by unnoticed, but still was able to affect the logic flow of the application.

An important conclusion reached during these experiments was the confirmation that the approach and respective tool are capable of discovering software faults in different components of the system, such as the service application itself and the middleware libraries that it uses. Several software bugs were discovered, some of which share big similarities regarding their cause, which allowed their aggregation into a group (*"Null Pointer bugs"*).

The most common software bugs belong to the so-called *"Null Pointer bugs"* group, which occur whenever Openbravo receives an unexpected *null* value from the database and fails to detect its presence. Later, when Openbravo continues its execution, it will attempt to call a function from this *null* variable (*i.e.* null.function()) and thereby cause a NullPointerException to be thrown.

Particularly interesting software bugs were also uncovered outside of the service application, in the code of Hibernate and the PostgreSQL JDBC driver.

Furthermore, A potential security vulnerability, second-order SQL injection, due to the dynamic creation of SQL queries using inputs received from database and without any validation, was discovered. A would-be attacker could possibly exploit this vulnerability by modifying the specific cells in database, or by intercepting the connection between database and service application (as is done by our tool). Less noteworthy flaws were also found in the design of the user interface when displaying error messages.

Finally, from the knowledge obtained in the results, we design and present programming guidelines aimed at helping developers avoid the same mistakes that we detected.

5.2 Service application behaviour

The foremost result to extract from this study is the characterization of the application behaviour under poor data quality, which reflects the ability of the application to withstand mutated data and asserts its possible failure modes. As explained in Section 3.4, to perform this evaluation we used a slightly adapted CRASH scale. We present in Table 8 the failure modes that were witnessed in each test case (each capital letter refers to one of the failure modes in the CRASH acronym).

Table 8: Behavioural analysis of Openbravo according to adapted CRASH scale

Test case	Crash
Login	C, A,R
Create Organization	A
Create a New User	A, R, S
Create a new Role	A, R
Create Product	A, R, S
Delete Product	A, R
Update Product	A, R
Read Export Product Categories to Spreadsheet	A, R, S

The first observation that can be taken is that the most prevalent failure mode was *Abort*. When analyzing the information contained in the logs we inferred that this failure mode can be caused by most mutations and exceptions, and, because of that, we are unable to find a specific well-defined pattern. In fact, as long as the failure is severe enough to stop the execution of the current task, this failure mode will very likely appear (later in this section we provide an example where the stoppage of a task leads to a *Restart* failure mode).

However, it must not go without notice that a particular occurrence had very interesting contours. This interesting example of an *Abort* failure is described below and causes a *ClassNotFoundException* to be thrown. From studying Openbravo's functioning we have already inferred that moderate usage of dynamic class loading is done to load classes that implement certain functionalities at run-time. A *ClassNotFoundException* occurs precisely because a mutation has changed the class name and Openbravo fails to find and load that class. If a class is not loaded, then the functions that it provides cannot be used. We studied a specific run where the above scenario occurred, and discovered that a failure at loading a class means that its functionality is disabled during a time period, *i.e.*, we found that a failure of this kind implies that the class cannot be loaded until a log-out and log-in has been done. After a log-out/log-in is done, then Openbravo will attempt to load the class again, and if successful the functionality is restored.

In contrast to the prevalence of the *Abort* failure mode, the *Catastrophic* failure mode had only one occurrence. The analysis of this occurrence revealed that the culprit was a mutation (in this particular case, a mutation that adds whitespaces) that affected the "userId" value (*i.e.*, the ID in the database of the user that we are currently logged in as). This mutation changed the correct value of "100" (which corresponds to the created-by-default Openbravo account) with "100 ". This incorrect value would cause a *NullPointerException* in JSP code of the index page (index.jsp), because a *SELECT* query to the database would return *null* since it could not find any user by this ID. The JSP code lacked any protection against this possibility, and, immediately after, attempted to call a function from this variable. The obvious outcome is the raising of a *NullPointerException* and the immediate interruption of the task. However, the interesting part, and the reason that separates it from a regular *Abort*, is that the mutated and incorrect "userId" value is stored in the browser session after the first use, and therefore, will permanently be used each time that an attempt is made to access Openbravo, and without fail lead to an exception. The only solution is to use a new browser (or open a Private Browsing session), or to restart Openbravo and Apache Tomcat.

The JSP code that tries to use the "userId" to get an item from database, and then throws the *NullPointerException* is shown in Listing 3.

Listing 3: Excerpt of code where Catastrophic failure mode can occur

```
1  final VariablesSecureApp vars1 = new VariablesSecureApp(request, false);
2  OBContext.setAdminMode();
3  try{
4    String roleId=vars1.getRole();
5    Role role;
6    if(roleId==null || roleId.equals("")){
7      role=OBDal.getInstance().get(User.class, userId).getDefaultRole();
8    }else{
9      role = OBDal.getInstance().get(Role.class, vars1.getRole());
10   }
```

The database access is done in line 7, by using *userId* as a parameter. The database will return *null*, which will then be used to call *"getDefaultRole"*, which throws the exception.

As a reminder, a **Restart** classification is given when the application hangs or takes more time than expected to execute the test case (timeout). In the experiments, we consider the consecutive amount of time that the browser shows no signs of operation to the user (*i.e.*, visually there are no significant changes) as the way to measure the timeout, and choose a timeout value of 3 minutes. Two different causes for a **Restart** failure mode were detected during the analysis of the results. One of the cases took place in a portion of code which is called periodically to check if the user's session is valid (in DB), and therefore, if the user is logged in. What occurred was that a mutation was performed to the results coming from database (in this case, a replace-by-null mutation), which replaced a numeric value ("1", but represented as a String) with a *null* value. This *null* value would later have an attempt to be accessed, which caused a *NullPointerException*. Because of this exception, the value of the variable *"loggedOK"* is never changed, and keeps the default value (*false*). The function will eventually return *false*, which will indicate Openbravo that the user should not be logged in. Openbravo will then proceed to redirect the user to the login page. However, from a user perspective all of these steps give the appearance that the application has hanged, partly because of no visual feedback (*e.g.* error message or other kind of information) and also because of the fact that until the user is successfully redirected to the login page, more than 5 minutes (value used solely as an example and limited to our specific setup) are spent. Due to the timeout limit (in our case, we opted to use 3 minutes) being exceeded before any visible indicator of action is displayed, we classified this occurrence as a **Restart**, despite the fact that if enough time is allowed, Openbravo will eventually redirect the user to the login page. The most important parts of the code of the *"loggedOk"* function is shown in Listing 4.

Listing 4: Excerpt of code where Restart failure mode can occur

```
1  public static boolean loggedOK(ConnectionProvider connectionProvider,
2      String adSessionId) throws ServletException {
3      String strSql = "";
4      strSql = strSql + "SELECT COUNT(*) AS TOTAL " +
5              "FROM AD_SESSION WHERE AD_SESSION_ID = ? " +
6              "AND SESSION_ACTIVE = 'Y' " +
7              "AND ISACTIVE = 'Y'";
8      boolean boolReturn = false;
9      PreparedStatement st = null;
10     int iParameter = 0;
11     try {
12
13         ( .... )
14
15         ResultSet result = st.executeQuery();
16         if (result.next()) {
17             boolReturn = !UtilSql.getValue((ResultSet)result,
18                             (String)"total").equals("0");
19         }
20         result.close();
21         }
22     catch (SQLException e) {
23         log4j.error((Object)("SQL error in query: " + strSql +
24                             "Exception:" + e));
25         throw new ServletException("@CODE=" +
26                                    Integer.toString(e.getErrorCode()) +
27                                    "@" + e.getMessage());
28     }
29     catch (Exception ex) {
30         log4j.error((Object)("Exception in query: " + strSql +
31                             "Exception:" + ex));
32         throw new ServletException("@CODE=@" + ex.getMessage());
33     }
34     finally {
35         try {
36             connectionProvider.releasePreparedStatement(st);
37         }
38         catch (Exception ignore) {
39             ignore.printStackTrace();
40         }
41     }
42     return boolReturn;
43 }
```

The parts of interest in the above listing are lines 17 and 18 , where the *NullPointerException* happens, and lines 8 and 42, where the variable *"boolReturn"* is initialized to *false*, and where the function returns, respectively. If we imagine that a *NullPointerException* occurs in line 18, it is easy to understand that the function will return the default value of *"boolReturn"*, which is *false* and means that the user is logged out.

The other **Restart** case that was analyzed and had a different origin happened when a new page is loading, and a **NullPointerException** occurs (once again because of a replace-by-null mutation), which forces the task to stop. At the same time, there is no error message being shown to the user. In fact, the only thing that the user can see is a "loading" bar that keeps moving, and thereby giving a false sense of work. Once again this case was classified as a *Restart* instead of an *Abort*, because of the false impression of constant loading that is provided to the user, and which will ultimately make the user wait until its patience has been exceeded.

The **Silent** failure mode is assigned when the application does not show any visible error message neither in the server log files nor in the client's browser, but it is not able to perform the necessary functionality or does not allow the user to execute the required steps to completes his desired functions.

A example for this case, which occurred in more than one Test Case, is where a specific button was grayed out and disabled (*i.e.*, impossible to be clicked in order to access its functionality). Specifically, the disabled button corresponded to the "Create New Item" button. When analyzing the reason behind this odd behaviour, we found that a certain mutation (in particular, the mutation that adds a random numeric character) was being called in a function named *"hasReadOnlyAccess"*. As the name suggests, the objective of this function is to distinguish between a user that has Read-Only access to a certain area of the application, and one that has Read-Write access. The original, un-mutated value would assign us with Read-Write privileges, but the mutation changed the value in a way that Openbravo believes that we have Read-Only privileges (the original value was "0" and the mutated value became "03"). An excerpt of the code for the function that calls *"hasReadOnlyAccess"* is shown in Listing 5, while a partial Java code corresponding to the important parts of the *"hasReadOnlyAccess"* is shown in Listing 6.

Listing 5: Excerpt of code where Silent failure mode can occur (1)

```
1  for (Tab tab : window.getADTabList()) {
2      final boolean readOnlyAccess = org.openbravo.erpCommon.utility
3          .WindowAccessData.hasReadOnlyAccess(dalConnectionProvider,
4                                              roleId, tab.getId());
5      String uiPattern = readOnlyAccess ? "RO" : tab.getUIPattern();
6      // window should be read only when is assigned with
7      // a table defined as a view
8      if (!"RO".equals(uiPattern) && ("T".equals(windowType) ||
9          "M".equals(windowType)) && tab.getTable().isView()) {
```

Listing 6: Excerpt of code where Silent failure mode can occur (2)

```
1  public static boolean hasReadOnlyAccess(ConnectionProvider
2      connectionProvider, String adRoleId, String adTabId)
3      throws ServletException {
4
5      boolean boolReturn = false;
6
7      ( .... )
8
9      PreparedStatement st = null;
10     int iParameter = 0;
11     try {
12         st = connectionProvider.getPreparedStatement(strSql);
13         QueryTimeOutUtil.getInstance().setQueryTimeOut((Statement)st,
14         SessionInfo.getQueryProfile());
15
16         ( .... )
17
18         ResultSet result = st.executeQuery();
19         if (result.next()) {
20             boolReturn = !UtilSql.getValue((ResultSet)result,
21             (String)"total").equals("0");
22         }
23
24     ( .... )
25
26     return boolReturn;
```

The lines that should be paid attention to are 20 and 21 of Listing 6, where it can be seen that *"hasReadOnlyAccess"* will return *true* (*i.e.*, the user has read-only access) if the result coming from database is different than "0".

The graying-out of the button, shown in Figure 9, is solely the visual feedback

of Openbravo to show that we do not have permissions.

Figure 9: Example of a disabled button, during a Silent failure mode

Because this mutation is not enough to cause an exception or other major problem, it goes by unnoticed in the system and does not raise any problem possible of being logged. It can be compared to a Silent Data Corruption (*i.e.*, data corruption that is not detected by the existing mechanisms).

Finally, it must be stated that, to the best of our knowledge, there was no occasion where *Hindering* occurred. However its assessment is difficult due to the fact that an assessment must be made regarding the truth of a certain error log.

5.3 Software defects uncovered during experiments

During our experiments, we were able to classify the behaviour of a service application under the effect of poor data quality, as presented in Section 5.2. However, the mutation of data being sent to the service application can also uncover software defects (*i.e.*, a deficiency in the product that causes it to deviate from the specification, can be a software bug) that could normally go unnoticed in "traditional" testing approaches. During our experiments, a good amount of software bugs were discovered both in Openbravo and in two libraries that provide support for database access, Hibernate and PostgreSQL JDBC driver. Tables 9 and 10 provide a detailed description of all the bugs uncovered, and include a brief explanation of why they occur, a possible solution, and information relative to the mutation, input and output that triggered their occurrence in our experiments.

The most common experienced software bug was recurrent enough to have its own group, *"Null Pointer bugs"*, which refers to bugs that end in *java.lang.NullPointerException* and are always caused by the lack of verification after data is read from database, which allows *null* values to stay unnoticed. When variables that have these *null* values try to be accessed, the operation fails by throwing a *java.lang.NullPointerException*. In general, the solution consists in verifying the values obtained from database, to at least ensure that they are not *null*.

Entry 19 is connected to one of the peculiarities of Openbravo, its ability to dynamically create SQL queries from other smaller SQL parts that are stored in database. The problem in this approach resides in the fact that if there is no appropriate validation of the content obtained from database, as is the case, the occurrence of SQL injection is a possibility. For example, a malicious user with access to the tables in database where these values are stored, or capable of

Table 9: Software bugs found in the service application

No	Exception	Mutation	Original Value	Mutated Value	Description	Suggested Fix
1	java.lang.StringIndexOutOfBoundsException	String_m2_replaceByEmptyString or String_m12_removeSubstring or String_m17_misspeling	"Y" or "N"	(Empty String)	Hibernate tries to access the first character (position 0) of a String, but that string might be empty, and Hibernate does not expect this. If the String is empty an unexpected exception is thrown.	Hibernate should check if the String is empty or not, in the same way that it already checks for a null value
2	java.lang.ClassNotFoundException	String_m2_replaceByEmptyString	org.openbravo.base.model.domaintype.BigDecimalDomainType$Quantity	(Empty String)	The name of dynamic loaded classes is kept in database. If a mutation occurs in one of these names, an exception will be thrown because Openbravo will fail to dynamically load that class.	Change the architecture to not save the name of dynamically loaded classes in the DB., but rather have it in a static format.
3	java.lang.NullPointerException	String_m1_replaceByNull	{"org.openbravo.scheduling.ProcessContext":{"user":100,"role":"42D0EEB1C66F497A90DD526DC597E6F0","language":"en_US","theme":"ltr\/org.openbravo.userinterface.skin.250to300Comp\/250to300Comp","client":"23C59575B9CF467C9620760EB255B389","organization":0,"warehouse":"","command":"DEFAULT","userClient":"","userOrganization":"","dbSessionID":"","javaDateFormat":"","jsDateFormat":"","sqlDateFormat":"","accessLevel":"","roleSecurity":true}}	null	Variable "value" has no checks to ensure that is not null before it is used.	Check if variable "value" is not null before using it.
4	java.lang.NullPointerException	String_m1_replaceByNull	By Descending Amount	null	Variable "textData[i].text" is null, but there is no previous check before using it.	Check if variable "textData[i].text" is not null before using.
5a	java.lang.NullPointerException	String_m1_replaceByNull	3	null	Same as 5b. The exception occurs in the same place. but the mutation happens somewhere else.	Check if variable "accessLevel" is not null before using.
5b	java.lang.NullPointerException	Any mutation which modifies the input, except making it completely empty	X, 800018, 0	Any except empty	One of the variables "classInfo.id" and "classinfo.type" can suffer a mutation, which will later cause "accessLevel" to become null. This occurs because the value of "accessLevel" is obtained from DB by passing the other two variables as parameters.	Check if variable "accessLevel" is not null before using.
6	java.lang.NullPointerException	String_m1_replaceByNull	DL_STARTING	null	A mutation causes the value of "data[i]" or "data[i].value" to be null, and Openbravo fails to verify it.	Check that data[i] and data[i].value are not null before using them.
7	java.lang.NullPointerException	String_m7_convertOppositeCase	Legal without accounting	lEGAL WITHoUT ACCOUNTING	The mutation of the test of the Organization will make a future DB query fail because it cannot retrieve the ID of the Organization that (precisely) matches that name. That query will return null which will later cause the exception. (note: this name is passed by the client to Openbravo at the moment of insertion, and therefore the client is able of changing its value on purpose)	Check if the obtained Organization is not null.
8	java.lang.NullPointerException	String_m15_replaceBySQLString	0	WHERE	The ID of an Organization is mutated when displaying webpage to client. This value comes from the client to the server (as in No. 7), where Openbravo tries to match the ID with an object, but it fails and returns null. This null is passed to another function and is used without verification.	Check if "parentOrg" is not null before using.

Table 10: Software bugs found in the service application (Cont)

No	Exception	Mutation	Original Value	Mutated Value	Description	Suggested Fix
9	java.lang.NullPointer Exception	String_m8_insertWhitespace	"10"	" 10 "	Variable "referenceID" is mutated and when Openbravo tries to find a class with that ID it fails and returns null. This null is passed on without verification.	Check if "ref" is not null before using.
10	java.lang.NullPointer Exception	String_m7_convertOppositeCase	23C59575B9CF467C9620760EB255B389	3c59575b9cF467C9620760eb255b389	A wrong value for "referenceID" causes an access to DB to fail and return null. This null is propagated without being checked.	Check if "rowClientId" is not null before using.
11	java.lang.NullPointer Exception	String_m7_convertOppositeCase	/org/openbravo/client/application/templates/ob-view-field.js.ftl	/ORG/openbravo/client/application/templates/ob-viEW-FIELD.JS.FTL	Variable "path" is mutated and Openbravo fails to find the file, which causes "url" to become null. Openbravo fails to handle the null correctly.	Check if "url" and "path" are null or not.
12	java.lang.NullPointer Exception	String_m1_replaceByNull	208	null	Variable "table" is used without a verification of its value.	Check if variable "table" is not null before use.
13	java.lang.NullPointer Exception	String_m22_addExtraneous	2013-07-04 23:45:43.694	PhD2013-07-04 23:45:43.694	In this case the string of the date is mutated which causes a failure when converting from string to timestamp (and the value becomes null). Openbravo has a function to compare two dates. However it does not verify if the dates it receives are null or not (and it is also not verified in other places that call this function). This causes a NullPointerException when one of the dates is null.	Dates "d1" and "d2" should be checked before usage.
14	java.lang.NullPointer Exception	String_m1_replaceByNull	151D205AB49443F9990449F10FC218BE	null	"alertRule" is used without previous verification.	Check if "alertRule" is not null before using.
15	java.lang.NullPointer Exception	Integer_m9_subtract1	1	0	"targetEntity" becomes null because the value passed to getEntityByTableId() is not correct.	Check if "targetEntity" is not null before using.
16	java.lang.NullPointer Exception	String_m1_replaceByNull	C5A8FBD40E6148C0A475C2D83FDAFD32	null	"mainTableTree" becomes null because of a replaceByNull mutation. This value is then used by Openbravo, which causes the exception.	Check if "mainTableTree" is not null before using.
17	java.lang.NullPointer Exception	String_m7_convertOppositeCase	DC206C91AA6A4897B44DA897936E0EC3	dc206c91aa6a4897b44da897936e0ec3	Variable "organization" is null, because strOrgId has been previously mutated. "strOrgId" is used to obtain the value of organization, which fails and returns NULL. Openbravo then proceeds to use this value and causes the exception.	Check if "organization" is not null before using.
18	java.lang.ArrayIndex OutOfBoundsExcepti on	String_m9_addCharactersToString	Issummary='N'	Issummary='N'slm'	The JDBC Driver of PostgreSQL has a programming bug that occurs when a query with an unterminated string is present. Despite the query being badly formed, the driver fails to handle it correctly, for example by giving an error message. This exception occurs because of an off-by-one bug triggered by the function that tries to find the closing single quote. When this function fails to find the closing single quote, it returns the position of the last character in the query as the end of the string. However another part of code in the driver, has a programming oversight that does not expect this behaviour, and the end result is an attempt to access a position that is one place after the end.	The function that finds the end of the string should return an adequate value, indicating that it failed to find an end. And the remaining code should be changed to take into account this possible scenario.
19	org.postgresql.util.PS QLException	Any mutation	-	-	Openbravo stores data in DB that is retrieved for later building other SQL queries. The problem resides in the lack of protection that these values stored in database have. It is possible for a malicious attacker (or, purely simply due to a data quality problem), to modify these values and build another SQL query that will run with Openbravo's privileges and can potentially undermine the security of the system. (e.g., the attacker can perform SQL injection to obtain sensitive information)	One way to close this vulnerability is to validate and sanitize all the data obtained from database. By doing this, it is possible to limit which characters and values are acceptable to be used. Another approach would be to move away from a system where queries can be dynamically created from data that is read from database. However we understand that this last approach might not always be feasible to implement.

tampering with the connection between database and service application (as we did), is capable of modifying these values to perform actions that he should not normally be able to (*e.g.*, create a new account with high privileges, drop a table, read privileged content). This is a severe flaw that can possibly allow an attacker even with limited local access to the system to perform privilege escalation and to perform actions that can compromise the security of the system. The solution to this issue consists again in implementing and applying a strict validation of all inputs coming from database. However, an even better approach would be to forgo this architectural approach entirely and to move away as much as possible from dynamically creating SQL queries.

The defects found during our experiments also extend to the user interface of the application. The interface of a service application is where the end-user communicates with the application. By nature, the end-user should not be presented with information that he does not need to know, and which will only confuse and disturb his experience. This observation also applies to error messages, where the widely accepted rule is that an error message presented to the user should be human-readable, easily understandable and helpful to solve the problem [96]. Unfortunately this is not what happens with Openbravo, where the error messages contain the exact same copy of the exception message, or other unhelpful content. By applying the Orthogonal Defect Classification (ODC) [97], and analyzing all the error messages obtained during the experimental evaluation, the defects of the user interface can be assigned to two different impact categories. One of them is *Usability*, which is one of the 9 impact categories proposed by IBM for usage with ODC [98], the other one is *Security*, which was employed because no other category was adequate. A more detailed explanation follows:

- **Usability:** the error messages are unhelpful and unintelligible to the user. Figure 10 shows one example of such an error message, which consists solely of a standard text ("Error occurred:") followed by the name of the Java exception.

- **Security:** sometimes the error messages inadvertently leak information to a possible attacker about the internal functioning of the system. Whenever a direct copy of the exception message is displayed to the user, there is the risk that database schema details, stack traces, path information, file names and database content are displayed. This configures an **information disclosure** vulnerability, and can help a possible attacker to break into the system. Figure 11 shows an error message obtained during our experiments, where the schema and contents of a database row are displayed.

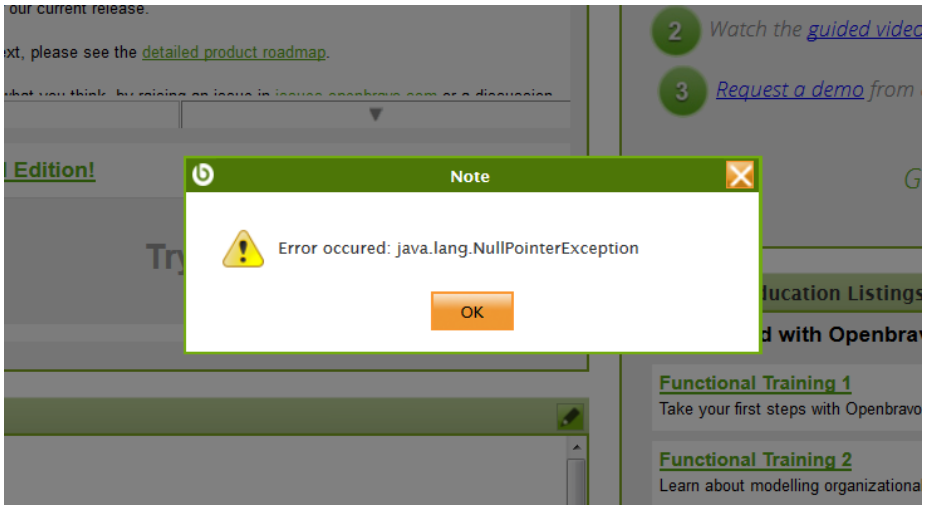

Figure 10: Error message with shortcomings at the Design level

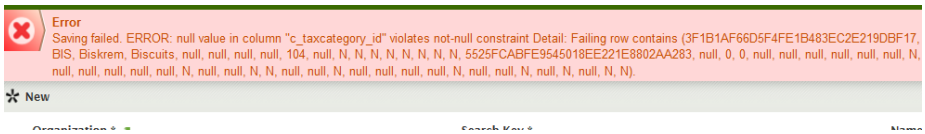

Figure 11: Error message with shortcomings at the Security level

Of particular importance are entries 1 and 18, which represent software bugs that are not in Openbravo's code, but rather in the code of libraries. The discovery of these issues is the perfect proof that our data mutation tool is capable not only of discovering problems in the service application being tested but also in the software of the entire system (*e.g.*, libraries).

The first entry of the table refers to a bug in **Hibernate**, a very well-known and widely-used library that acts as an intermediary and helper for database access, using an approach that is commonly known as object-relational mapping (ORM). The Hibernate version used during the experiments was 3.6.3-Final, but even at the moment of writing the bug here uncovered had yet to be fixed in the latest version (Hibernate 4.2.21).

One of the functionalities of Hibernate is the conversion of values stored in multiple formats (*e.g.*, string, integer, character) into the corresponding boolean value. This functionality is used whenever a boolean value is stored in database as another format. One of the supported formats that can be converted into a boolean is a String, where Openbravo defines 'Y' and 'N' to represent *True* and *False*, respectively. However, a programming flaw exists in this code, which can be seen in Listing 7. Although Openbravo checks (in line 2) if the input value is *null*, before proceeding, it does not take into account the possibility of an empty

string. The bug occurs in line 17, when an attempt is made to access position 0 of the string. If this string is empty, a *java.lang.StringIndexOutOfBoundsException* exception is raised, and thereby the process interrupted. The proposed solution is to check if the string is empty, as is already done for the *null* value.

Listing 7: Excerpt of the defective Hibernate function

```
1  public <X> Boolean wrap(X value, WrapperOptions options) {
2      if ( value == null ) {
3         return null;
4     }
5     if ( Boolean.class.isInstance( value ) ) {
6        return (Boolean) value;
7     }
8     if ( Number.class.isInstance( value ) ) {
9        final int intValue = ( (Number) value ).intValue();
10       return intValue == 0 ? FALSE : TRUE;
11    }
12    if ( Character.class.isInstance( value ) ) {
13       return isTrue( ( (Character) value ).charValue() )
14                                        ? TRUE : FALSE;
15    }
16    if ( String.class.isInstance( value ) ) {
17       return isTrue( ( (String) value ).charAt( 0 ) )
18                                        ? TRUE : FALSE;
19    }
20    throw unknownWrap( value.getClass() );
21 }
```

Entry 18 refers to a software bug found in the **JDBC driver of PostgreSQL**, which provides the connection between the Java application and the database. Unlike what happened in Hibernate's bug, this software fault had already been found and fixed prior to our discovery [99]. However this does not invalidate the ability of our tool for finding bugs that extend over the service application under test.

This software bug occurs in the parser of the JDBC driver that is going to receive the SQL query and translate it from a human-readable format to a format which can be internally used in the driver. More specifically, the parser fails to correctly handle queries that contain unterminated strings (in other words, odd number of single quotes or double quotes). The programming flaw is spread across two functions. An excerpt of the first function is shown in Listing 8, and the second function is shown in Listing 9.

The first function iterates over every character of the query and looks for predefined characters that it must escape (among which is the single quote). When it finds a single quote (line 12), it will call the second function (line 13) to find and return the location of the next single-quote. The position of the first and second single quotes is then used to calculate the parameters for a

copy (line 15). The software bug occurs when the query has an unterminated string (*i.e.*, one of the single quotes has no match), which leads the second function to return the length of the entire query (line 31 of the second function). This value is incorrect, and when used to calculate the copy operation (line 15) it will cause the JDBC driver to attempt to access a position that is one place after the end of the character array (*aChars*), thereby causing a *java.lang.ArrayIndexOutOfBoundsException* exception.

This software bug can be fixed in a few different ways, but we advise an improvement of how both functions interact between them. For example, the first function should be able to understand when the second function fails to find a single quote and then act accordingly. The second function could also return a value that more clearly indicates that something has failed (*e.g.*, -1), instead of just returning the query's length, which can be overlooked by the programmers as a normal, acceptable value and indicative of a successful operation, when in reality it causes a problem and occurs when the function fails to fulfill its purpose.

Listing 8: Excerpt of the defective PostgreSQL JDBC code (1)

```
1  public static String unmarkDoubleQuestion(String query,
2      boolean standardConformingStrings) {
3
4      if (query == null) return query;
5
6      char[] aChars = query.toCharArray();
7      StringBuilder buf = new StringBuilder(aChars.length);
8      for(int i=0, j=-1; i< aChars.length; i++)
9      {
10         switch (aChars[i])
11         {
12             case '\'': // single-quotes
13                 j = Parser.parseSingleQuotes(aChars, i,
14                                 standardConformingStrings);
15                 buf.append(aChars, i, j-i+1);
16                 i = j;
17                 break;
```

Listing 9: Excerpt of the defective PostgreSQL JDBC code (2)

```
1  public static int parseSingleQuotes(final char[] query,
2                int offset, boolean standardConformingStrings) {
3      // check for escape string syntax (E'')
4      if (standardConformingStrings
5              && offset >= 2
6              && (query[offset-1] == 'e' || query[offset-1] == 'E')
7              && charTerminatesIdentifier(query[offset-2]))
8      {
9          standardConformingStrings = false;
10     }
11
12     if (standardConformingStrings)
13     {
14         // do NOT treat backslashes as escape characters
15         while (++offset < query.length)
16         {
17             switch (query[offset])
18             {
19             case '\'':
20                 return offset;
21             default:
22                 break;
23             }
24         }
25     }
26     else
27     {
28        // ( ..... )
29     }
30
31     return query.length;
32 }
```

Having obtained several programming bugs, we decided to use a standard classification for software anomalies, designed by IEEE [100], to classify each of the software defects according to Severity, Type, Effect and Probability. Severity can vary between *Inconsequential*, *Minor*, *Major* and *Critical*, according to how disruptive the bug is, and how important are the functions that it affects. Type can be *Data*, *Interface*, *Logic*, *Description*, *Syntax*, *Standards* or *Other*, according to the root cause of the defect. Effect can be *Functionality*, *Usability*, *Security*, *Performance* or *Serviceability*. Probability can be *High*, *Medium* or *Low*, according to how likely the defect is to be triggered. To fill theses columns the knowledge obtained during the experiments was used. The resulting classi-

fication is presented in Table 11.

Table 11: Classification of Software Defects

Software Bug	Severity	Type	Effect	Probability
Hibernate Bug	Minor	Logic	Functionality	High
PostgreSQL JDBC Driver Bug	Minor	Logic	Functionality	Low
Class Not Found Bugs	Major	Other	Functionality	Medium
Null Pointer Bugs	Minor	Logic	Functionality	Low
SQL Injection Bugs	Critical	Logic	Security	Low

SQL injection was the only software bug to obtain a *Critical* classification, due to the security impact that it is capable of causing in the system. It was also the only one to have an *Effect* in *Security*, unlike the remaining bugs which all had an impact in the *Functionality*. However, it was assigned a *Low* probability of occurrence, based on the results of our experiments. The Class Not Found bug was given a rating of *Major* regarding its *Severity*, because the failure to load a class translates to a part of the functionality being inaccessible, which, as we have explained in Section 5.2, can only be recovered from after a log-out and log-in. Because this bug is the product of an architectural decision, it was not considered to be a *Logic* bug like all the remaining bugs, but rather it was given the classification of *Other*.

In our experiments, the bug that occurred more often was, by far, the Hibernate bug, hence the *High* probability that it was attributed. However its severity was considered to be *Minor*, because it usually translates to a failure of the current task to finish execution, but does not affect other tasks and does not have a persistent effect. For a similar reason the *Minor* classification was also attributed to the Null Pointer bugs. In our analysis, often a Null Pointer bug caused a failure that was confined to the task under execution. Sometimes, if the exception occurred during a loading phase, we witnessed Openbravo appear to infinitely try to load, despite in reality the task having already stopped. Because of this behaviour, the user is given the perception of an hang (infinite loading).

5.4 Distribution of exceptions

During the experimental process, whenever mutated data is injected, there is a strong possibility that the application finishes the run with the presence of exceptions in its logs. According to the type of exception that is displayed, it is possible to study their frequency and ascertain which exceptions are more common, what they mean and which mutations cause them. The exceptions and their number of occurrences during all our experiments are presented in

Table 12* .

Table 12: Exceptions that Openbravo threw during the experiments

No	Exception Types	Number of Occurrences	Percentage
1	org.openbravo.base.validation.ValidationException	676	28,34
2	java.lang.StringIndexOutOfBoundsException	449	18,83
3	java.lang.NullPointerException	416	17,44
4	org.hibernate.ObjectNotFoundException	202	8,47
5	org.postgresql.util.PSQLException	196	8,22
6	org.openbravo.client.kernel.OBUserException	96	4,03
7	javax.servlet.ServletException	83	3,48
8	java.lang.Exception	79	3,31
9	java.lang.ClassNotFoundException	64	2,68
10	java.lang.IllegalArgumentException	43	1,8
11	org.openbravo.base.exception.OBException	23	0,96
12	org.openbravo.base.exception.OBSecurityException	21	0,88
13	org.hibernate.StaleStateException	12	0,5
14	org.codehaus.jettison.json.JSONException	9	0,38
15	org.hibernate.LazyInitializationException	7	0,29
16	org.openbravo.base.util.CheckException	2	0,08
17	org.openbravo.service.json.OBStaleObjectException	2	0,08
18	com.thoughtworks.xstream.mapper.CannotResolveClassException	1	0,04
19	org.openbravo.erpCommon.utility.PropertyNotFoundException	1	0,04
20	java.lang.ArrayIndexOutOfBoundsException	1	0,04
21	java.lang.IndexOutOfBoundsException	1	0,04
22	java.lang.IllegalStateException	1	0,04

While some exceptions are native to Java, others are created by Openbravo itself to represent specific unhandled situations that cannot be expressed using the existing exceptions, and others belong to external libraries that Openbravo uses to provide its service. In more detail, the exceptions are analyzed according to their meaning and reason of occurrence:

- **org.openbravo.base.validation.ValidationException:** Openbravo makes use of input validation functions to ensure basic constraints (*e.g.*, field has to have certain size, field cannot be null or empty, ...) whenever some fields of an object are set. If the check fails then this exception is thrown. As a result of the occurrence of this exception, the possible

* The sum of all percentages is different from 100% due to rounding.

behaviours that were witnessed (from the client perspective) were: application fails during page loading and hangs in this phase; a popup error message is shown to end user with the exception message directly obtained from Java (not good to be seen by end user); the JavaScript stack-trace is displayed to end user in a popup message (once again unacceptable to be shown to the end user).

- **java.lang.StringIndexOutOfBoundsException:** This exception is characteristic of the bug found in Hibernate, where an empty string is not being well-handled in a certain function.

- **java.lang.NullPointerException:** NullPointerExceptions very often indicate a software bug in Openbravo's code that occurs because of the deficient verification of the variables value's after it is read from DB and before it is used. What often occurs is that the value being read from DB is mutated to *null* and Openbravo fails to account for this possibility and simply decides to continue execution. In other cases, the scenario is a bit more complex: the data coming from DB is mutated to an incorrect value, which causes a posterior call to database to return *null* (*e.g.*, a search by ID returns null when the ID has been mutated to some non-existent value and it cannot find the desired entity).

- **org.hibernate.ObjectNotFoundException:** Occurs when no row with the given identifier exists and Hibernate is unable to find the corresponding information in database. This occurs when the identifier is mutated to an incorrect value.

- **org.postgresql.util.PSQLException:** This kind of exceptions are closely linked with PostgreSQL, which is the DBMS used in the experiments. This exception occurs whenever PostgreSQL is unable to successfully complete a query. In our experiments, this was mainly due to 3 reasons: the column name being mutated and, thereby, becoming incorrect; a previous exception causing the abortion of an active transaction, and all the next queries failing (being ignored) for the sole reason that the transaction is aborted; invalid characters for the encoding being used are present in the query. Usually this kind of exception has a high severity and will cause the current activity to fail.

- **org.openbravo.client.kernel.OBUserException:** This exception is thrown only in a place of Openbravo (in the DataSourceServlet). It signals that the current logged in user does not have enough permissions to view a certain resource. In our experiments it occurs because of a previous exception or mutation that causes the permission check to fail. Its impact is limited in the sense that either a previous and more critical exception has already crashed the application, or that the worst possible result is that some data (such as a listing of products) is not displayed.

- **javax.servlet.ServletException:** This exception represents a problem during the execution of a Servlet, which can be due to a variety of reasons. Therefore, it is a very general exception.

- **java.lang.Exception:** This is the basic exception in Java from where every other exception inherits from. In Openbravo it usually appears accompanied by an error message of a specific exception.

- **java.lang.ClassNotFoundException:** This exception occurs because Openbravo makes occasional use of dynamic class loading (*e.g.*, to load classes relative to visual themes), and decides to store the information relative to the classes that it wants to load (which in itself is relatively static content) in the database. A mutation in this information causes the dynamic loading to fail, simply because the name of the class becomes wrong. When the loading of the class fails, whatever would be the job of the class cannot be completed.

- **java.lang.IllegalArgumentException:** In our experiments, this exception was thrown always from the same place, when an attempt at loading a FreeMarker template failed because of a mutation to the name of the language.

- **org.openbravo.base.exception.OBException:** OBException is an exception type created by Openbravo that has a quite general functionality and does not represent any particular problem. In our experiments, it can represent a variety of problems, such as not being able to update an entry in database, not being able to compute data to show to the user, among others. While the problem that occurred can easily be understood, the root cause behind it is not very explicit.

- **org.openbravo.base.exception.OBSecurityException:** This is a general exception of Openbravo, which occurs when it detects security violation (*i.e.*, according to the logic of Openbravo a certain action cannot be performed, otherwise it would break the basic security principles [*e.g.*, confidentiality, integrity]). For example, this exception is thrown when a client tries to access data belonging to another client, or when it tries to read data to which he does not have permission.

- **org.hibernate.StaleStateException:** This exception occurs in update or delete operations whenever the ID of the target object can no longer be found. In practice, this usually occurs when there is an attempt to update an object after it has been deleted (therefore, the object does not exist anymore). However, the exception is not limited to this specific scenario.

- **org.codehaus.jettison.json.JSONException:** This exception occurs whenever there is a problem parsing a JSON message exchanged between the client's browser and the service application. In our experiments, this occurred either because of the existence of non-printable characters (*e.g.*,

new-line character) which interfere with the parsing of the message, or because when Openbravo attempts to read the value associated with a certain key, that key cannot be found in the JSON content.

- **org.hibernate.LazyInitializationException:** This exception occurs whenever there is unfetched data that must be accessed but the session has already been closed. Usually Hibernate takes a lazy approach and delays fetching data, such as arrays and complex objects, from database until it is really needed. This increases the performance of Hibernate, but can cause this exception to be thrown.

- **org.openbravo.base.util.CheckException:** Openbravo is equipped with a set of functions to check the names of the columns in database, and from these names infer which data types they are holding (*e.g.*, string, integer). When these functions fail because of an unexpected column name, this exception is thrown.

- **org.openbravo.service.json.OBStaleObjectException:** This exception is thrown whenever the JSON converter implemented in Openbravo tries to update a certain object, but the values that are inside the object are different from the values in the JSON message. The object is assumed to be stale (not up to date).

- **com.thoughtworks.xstream.mapper.CannotResolveClassException:** This exception is thrown when XStream cannot find the class to be dynamically loaded.

- **org.openbravo.erpCommon.utility.PropertyNotFoundException:** This exception occurs when a certain property is not defined.

- **java.lang.ArrayIndexOutOfBoundsException:** This exception is thrown whenever an attempt is made to access an illegal position (*i.e.*, negative position or higher than the size) of an array.

- **java.lang.IndexOutOfBoundsException:** Similar to *java.lang.ArrayIndexOutOfBoundsException*, but can indicate an access to an illegal position in any kind of objects (*e.g.*, array, string, vector), and not only in an array.

- **java.lang.IllegalStateException:** This exception signals that a certain operation cannot be performed at the current time, because the application or environment is not in the correct state to perform it.

Extracting these exceptions from the application's logs is done mostly in an automated manner, by using a specially designed script, however in some cases manual intervention is required to fix a small number of cases where the script fails.

Table 13 presents the relationships between an exception, its description (the same exception is able of being caused by more than one reason, and therefore can have different descriptions) and its origin.

The origin of a failure can be attributed to one of three different components that were identified when looking at the results: the DBMS (in our case, PostgreSQL), the Middleware (*e.g.*, Hibernate, Freemarker) and the Service Application (in our case, Openbravo).

When looking at the table, it is possible to see that most of the exceptions are associated with the Service Application, but there is also a significant of DBMS and Middleware exceptions. Most of these exceptions only have a possible description. However, *org.postgresql.util.PSQLException* has nine possible descriptions.

5.5 Statistics regarding the experiments

Another venue of analysis of the results, which reflects the workload and mutations, is to look at statistics regarding the data mutation process. For example, Table 14 shows the absolute amount and percentage of mutations grouped according to their data type. This analysis provides a better understanding of how homogeneous is the distribution of mutations performed during the experiments.

Table 14: Amount of mutations grouped per data type

Data Type	Number of Mutations	Percentage (%)
Long Integer	42145	2,426
String	1659584	95,536
Decimal	1872	0,108
Timestamp	13416	0,772
Int Integer	19	0,001
Date	106	0,006
Boolean	19966	1,149
Object	35	0,002
Total	1737143	100

It is easy to see that despite the significant amount of different data types, the high majority of mutations belonged to the String data type. followed from very far away by the Long data type.

Table 15 shows how many runs were need to successfully complete each test case (*i.e.*, test case must finish with success).

Table 13: Relation between Exceptions and other attributes

Origin	Exception	Exception Message
DBMS	org.postgresql.util.PSQLException	column <mutated_output> does not exist
		syntax error at or near "<mutated_output>"
		relation "<mutated_output>" does not exist
		current transaction is aborted, commands ignored until end of transaction block
		Key (<key>)=(<mutated_output>) is not present in table "<table>"
		value too long for type character varying(32)
		invalid byte sequence for encoding "UTF8": 0x00
		null value in column "<column>" violates not-null constraint
		missing FROM-clause entry for table <mutated_output>
	org.hibernate.StaleStateException	Batch update returned unexpected row count from update [0]; actual row count: 0; expected: 1
	org.hibernate.ObjectNotFoundException	No row with the given identifier exists
	org.hibernate.LazyInitializationException	could not initialize proxy - no Session
		illegal access to loading collection
Middleware	com.thoughtworks.xstream.mapper.CannotResolveClassException	<mutated_output>
	org.codehaus.jettison.json.JSONException	Unterminated string at character <position> of <mutated_json>
		JSONObject["<key>"] not found
	java.lang.IllegalArgumentException	Template processor for language <mutated_output> not found
	java.lang.StringIndexOutOfBoundsException	String index out of range: 0
Service Application	org.openbravo.base.validation.ValidationException	<mutated_output> is too long, it has length <length_mutated>, the maximum allowed length is 32
	java.lang.NullPointerException	-
	org.openbravo.client.kernel.OBUserException	AccessTableNoView
	java.lang.ClassNotFoundException	<mutated_output>
	org.openbravo.base.exception.OBSecurityException	Client (<mutated_output1>) of object (Organization(^L0) (name: *)) is not present in ClientList <original_output_2>
		Entity ADWindow may only have instances with client 0
	org.openbravo.erpCommon.utility.PropertyNotFoundException	-
	org.openbravo.service.json.OBStaleObjectException	@OBJSON_StaleDate@
	org.openbravo.base.exception.OBException	Error while determining the node deletion policy
		Exception when getting message for key: AccessTableNoView
		Exception when getting personalization settings for window ADWindow(<ID>) (name: <name>)
		Exception when getting personalization settings for tab ADTab(<ID>) (name: <name>, window: <window_ID>)
		Error while exporting a CSV file
		Error while computing combo data
	org.openbravo.base.util.CheckException	Mapping name: null not found in runtime model
	java.lang.IndexOutOfBoundsException	Index: 0, Size: 0
	java.lang.IllegalStateException	Singleton is not set
	java.lang.Exception	String index out of range: 0
		Error while trying to locate the tab
		column <mutated_output> does not exist
		No row with the given identifier exists
	javax.servlet.ServletException	@CODE=@Entity ADWindow may only have instances with client 0
		@CODE=@null
		current transaction is aborted, commands ignored until end of transaction block

Table 15: Number of runs for each test case

Test Case No	Test Case Name	Number of Runs
Test Case 1	Login	96
Test Case 2	Create Organization	60
Test Case 3	Create a new User	80
Test Case 4	Create a new Role	76
Test Case 5	Create Product	78
Test Case 6	Delete Product	75
Test Case 7	Update Product	50
Test Case 8	Export Product Categories to Spreadsheet	64

The amount of mutations performed for each Test Case was also studied and the results can be seen in Table 16.

Table 16: Amount of mutations performed in each Test Case

Test case	Number of Mutations	Number of Runs
Test Case 1	132317	96
Test Case 2	427243	60
Test Case 3	292468	80
Test Case 4	164063	76
Test Case 5	120089	78
Test Case 6	123507	75
Test Case 7	183237	50
Test Case 8	329273	122

There is some variation in the number of mutations, with the peak being Test Case 8 with 329273 mutations, which is also the test case with the higher amount of runs. However if we ignore this specific test case, there does not appear to exist a strong correlation between the number of mutations and the number of runs.

5.6 Guidelines for robustness against poor data quality

Taking into consideration the results obtained from our experiments and after a careful analysis of the various uncovered programming bugs, it has become possible to define a few guidelines that can be used as best practices to minimize the occurrence of programming bugs similar to those that were witnessed during the experimental evaluation. First and foremost, we propose that every input coming from an untrusted source (in our case, the database) must be validated

as thoroughly as possible. In Openbravo, the contents read from database, almost always, lacked any kind of verification that could ensure even the lowest expectations (*e.g.*, assure that it is not null, assure that it obeys basic criteria for the expected value). If a rule that enforced the mandatory verification of contents coming from database had been strictly applied in Openbravo's code, the majority of the bugs would not have been present.

Middleware developers also play a role in ensuring a bug-free system, as is proven by the two bugs that were discovered in libraries. The advice for middleware developers is to never trust any of the content coming from the database (and also the application), and once again, verify it thoroughly and gracefully fail if a problem that prevents execution has been found. The ***Hibernate*** bug occurred because of an oversight in these validation checks, which allowed an unexpected value (empty string) to cause an unforeseen condition. The ***PostgreSQL JDBC*** bug was made possible because once again the programmers did not take into consideration how their code would handle an unexpected input (unterminated quote), and their approach ended with an off-by-one bug that attempted to access a non-existing array position. In these two cases, the validations were in place, however their implementation was flawed, due to certain scenarios having not been taken into account.

Finally, it is a good idea to think about how the architecture of the system is designed from a robustness point, and, at the design phase, to avoid situations and programming solutions that unnecessarily compromise this aspect. An example of this, is the abundant use that Openbravo makes of storing data in database that will later be used to create new queries or to dynamically load classes. Once again, this architecture, paired with the lack of input validation, has opened the service application to the occurrence of second order SQL injection.

From a user interface design point of view, we advise that exceptions should be shown to the user in a streamlined manner, and providing a human-readable explanation and an easy way for the user to identify which error occurred (*e.g.*, error code). In no occasion should the user be presented with the complete exception message or stack-trace, which is not only of no use to him, as it will certainly also disturb his browsing experience. Openbravo was particularly deficient in this aspect, where exceptions were displayed to the end-user without any treatment, and in the most varied manners (pop-up messages, small error boxes). Furthermore, the exception message can include sensitive information (*e.g.*, database schema, file paths, parts of SQL queries) which can help an attacker.

6 Conclusions and Future Work

In this section, an overview of the work performed during this thesis is provided, followed by a reflection about what can be the future work after the end of this Master thesis.

6.1 Overall view

One of the first steps performed in this thesis, still during the first semester, was an extensive survey about data quality, which included data quality dimensions and data quality problems. From this study, a fault model (*i.e.*, our data mutation tables) that closely follows real-world data quality problems was developed.

Then, after careful consideration, and fueled by the lack of previous research in this area, an approach was devised to inject mutated data (*i.e.*, poor quality data) into the results that come from a DBMS to a service application, with the objective of testing the behaviour of any service application in the presence of poor quality data. This approach and the fault model were converted from an abstract concept into a usable tool.

Finally, this tool, was employed to conduct experiments against a well-known service application (Openbravo), and thereby classify its behaviour (using an adapted CRASH scale for that purpose). Furthermore, these experiments yielded several software bugs present both in the service application and in the middleware libraries (namely, Hibernate and the PostgreSQL JDBC driver). After analyzing the results, guidelines were developed for programmers so that they can avoid the mistakes that were identified.

By the end of this thesis, the developed tool and fault model had been distributed online, for the entire community to be able to use and assess their service applications. Furthermore, two conference papers had been written and submitted to renowned peer-reviewed international conferences of the area. Namely, a survey paper about data quality had been submitted and accepted to PRDC 2015 (Appendix C), and another paper, describing our approach, tool and results, had been submitted and accepted to LADC 2016 (Appendix D).

Overall, we consider the outcome of this thesis as very positive, with an array of contributions to the area, and believe that the goals that were proposed have been attained.

6.2 Future Work

Given the fact that this work, to the best of our knowledge, is a first in this particular area, it is understandable that plenty of improvements can still be undertaken. The topics that we consider most important and feasible in a short to medium term will be introduced in this subsection.

During the experimental phase of this thesis, only a single experimental setup was tested. In the future, we desire to apply our approach to different service applications, in particular applications that use different components,

such as ORMs (*e.g.*, instead of using Hibernate, it can use EclipseLink JPA or OpenJPA, or even it might not use any ORM, and instead use a different approach, such as JDBC or JDO).

Another interesting analysis that can be done is to compare the same system when only one component is modified, and thereby understand the impact that a specific component has, and compare the various available competitors. For example, taking the experimental setup described in this thesis as a basis, we could easily and without any code modification change the DBMS from PostgreSQL to MySQL or Oracle. Or we could use a different web server, and instead of Apache Tomcat, we could use WildFly or GlassFish.

During our study of the results, the completion of each test case was defined as the point where the application is no longer capable of behaving correctly under the mutated data. However, the last experiment run of each Test Case worked correctly, and the Test Case would finish successfully. Despite the apparent success of the application in completing the Test Case, the reality is that the application is full of poor data, which can have a silent effect in its behaviour. As a future work, we desire to continue the study of the application after the Test Case has finished, so that we can detect the possible existence of Silent Data Corruption, or other problems that are not being manifested when they should.

A situation that may occur in larger, more extensive experiments is that the performance overhead added by our tool can become noticeable and intrusive. Although this was not the case in the experiments performed during this thesis, it is a possibility which, as a future work, can be avoided by using optimized algorithms that can reduce this overhead.

Another aspect deserving of our attention is the installation process of our tool. At the moment this process, despite straightforward, is manual. The automation of the installation (*e.g.*, executable installer, virtual machines) is beneficial in attracting potential users, since a simple installation and operation reduces the difficulties faced by interested users. This aspect should not be disregarded, as it is a key factor in ensuring that the tool (and approach) attain a wide reach. In fact, several technologies that are commonly used in research groups face difficulties when trying to gain widespread use in the business world (*e.g.*, the usage of fault injection to assess the dependability of systems), mainly due to their difficult usage. The most time-consuming and difficult part of the entire experimental process is definitely the analysis of results in order to extract valuable information, such as failure modes or software bugs in the code, which right now is completely manual. For this reason, improvements in this area will benefit the most. Our ultimate goal as a future work is to have a completely automated way of analyzing the results, right after each experiment run, and easily obtain information such as uncovered software bugs, which mutation caused a certain exception and a classification regarding the failure modes. We believe that if this goal is reached then our approach will become a very powerful tool in studying the dependability of service applications.

References

[1] Nachiappan Nagappan and Thomas Ball. Use of relative code churn measures to predict system defect density. In *Proceedings. 27th International Conference on Software Engineering, 2005. ICSE 2005.*, pages 284–292. IEEE, 2005.

[2] T. Capers Jones. Measuring programming quality and productivity. *IBM Systems Journal*, 17(1):39–63, 1978.

[3] Lakshmi N Bairavasundaram, Andrea C Arpaci-Dusseau, Remzi H Arpaci-Dusseau, Garth R Goodson, and Bianca Schroeder. An analysis of data corruption in the storage stack. *ACM Transactions on Storage (TOS)*, 4(3):8, 2008.

[4] The typo that destroyed a NASA rocket. 2015-06-25.

[5] Maggie Urry. Book errors figure in Salamon $770m pretax loss. *The Financial Times*, 1995.

[6] Marco Vieira and Nuno Laranjeiro. Comparing web services performance and recovery in the presence of faults. In *IEEE International Conference on Web Services (ICWS 2007)*, pages 623–630. IEEE, 2007.

[7] Yao-Wen Huang, Shih-Kun Huang, Tsung-Po Lin, and Chung-Hung Tsai. Web application security assessment by fault injection and behavior monitoring. In *Proceedings of the 12th International Conference on World Wide Web*, WWW '03, pages 148–159, New York, NY, USA, 2003. ACM.

[8] K Roebuck. Data quality: High-impact strategies-what you need to know: Definitions, adoptions, impact. *Benefits, Maturity, Vendors*, 2011.

[9] ISO. Quality management systems — fundamentals and vocabulary. ISO 9000:20015, International Organization for Standardization, Geneva, Switzerland, 2015.

[10] Laura Sebastian-Coleman. *Measuring data quality for ongoing improvement: a data quality assessment framework*. Newnes, 2012.

[11] Diane M Strong, Yang W Lee, and Richard Y Wang. Data quality in context. *Communications of the ACM*, 40(5):103–110, 1997.

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

[13] Anthony Giannoccaro, Graeme G. Shanks, and Peta Darke. Stakeholder perceptions of data quality in a data warehouse environment. *Australian Computer Journal*, 31(4):110–116, 1999.

[14] Thomas C Redman. The impact of poor data quality on the typical enterprise. *Communications of the ACM*, 41(2):79–82, 1998.

[15] Evaluating the Business Impacts of Poor Data Quality. The data quality benchmark report. Technical report, Knowledge Integrity Incorporated, Business Intelligence Solutions, 1 2011.

[16] Phillip Cykana, Alta Paul, and Miranda Stern. DoD guidelines on data quality management. In *IQ*, pages 154–171, 1996.

[17] P Larry. English, improving data warehouse and business information quality: methods for reducing costs and increasing profits, 1999.

[18] Gartner. Magic Quadrant for Data Quality Tools. `https://www.gartner.com/doc/reprints?id=1-259U63Q&ct=141126&st=sb`, November 2015.

[19] Rob Karel. The Äll InC̈osts of Poor Data Quality. `http://www.computerworld.com/article/2949323/data-analytics/the-all-in-costs-of-poor-data-quality.html`, July 2015.

[20] newswire.com. Dirty Data Costs the US Economy $3.1 Trillion Yearly. `https://www.newswire.com/dirty-data-costs-the-us-economy/128732`, September 2011.

[21] Time. Blunders: An $11 million typo. 2015-06-25.

[22] M Eppler and Markus Helfert. A classification and analysis of data quality costs. In *International Conference on Information Quality*, 2004.

[23] Mónica Bobrowski, Martina Marré, and Daniel Yankelevich. Measuring data quality. *Universidad de Buenos Aires. Report*, pages 99–002, 1999.

[24] David Loshin. *The practitioner's guide to data quality improvement*. Elsevier, 2010.

[25] Thomas C Redman and A Blanton. *Data quality for the information age*. Artech House, Inc., 1997.

[26] Marcey L Abate, Kathleen V Diegert, and Heather W Allen. A hierarchical approach to improving data quality. *Data Quality Journal*, 4(1):365–369, 1998.

[27] Eva Gardyn. A data quality handbook for a data warehouse. In *IQ*, pages 267–290, 1997.

[28] Jennifer Long and Craig Seko. A new method for database data quality evaluation at the canadian institute for health information (CIHI). In *IQ*, pages 238–250, 2002.

[29] David Loshin. *Enterprise knowledge management: The data quality approach*. Morgan Kaufmann, 2001.

[30] Leo L Pipino, Yang W Lee, and Richard Y Wang. Data quality assessment. *Communications of the ACM*, 45(4):211–218, 2002.

[31] Thomas C Redman. *Data quality: the field guide*. Digital Press, 2001.

[32] Erhard Rahm and Hong Hai Do. Data cleaning: Problems and current approaches. *IEEE Data Eng. Bull.*, 23(4):3–13, 2000.

[33] Christine Parent and Stefano Spaccapietra. Issues and approaches of database integration. *Communications of the ACM*, 41(5es):166–178, 1998.

[34] William W Cohen. Integration of heterogeneous databases without common domains using queries based on textual similarity. In *ACM SIGMOD Record*, volume 27, pages 201–212. ACM, 1998.

[35] Vipul Kashyap and Amit Sheth. Semantic and schematic similarities between database objects: a context-based approach. *The VLDB Journal—The International Journal on Very Large Data Bases*, 5(4):276–304, 1996.

[36] Won Kim, Byoung-Ju Choi, Eui-Kyeong Hong, Soo-Kyung Kim, and Doheon Lee. "A Taxonomy of Dirty Data". *Data mining and knowledge discovery*, 7(1):81–99, 2003.

[37] Paulo Oliveira, Fátima Rodrigues, and Pedro Rangel Henriques. "A Formal Definition of Data Quality Problems". In *IQ*, 2005.

[38] Heiko Müller and Johann-Christph Freytag. *Problems, methods, and challenges in comprehensive data cleansing*. Professoren des Inst. Für Informatik, 2005.

[39] José Barateiro and Helena Galhardas. A survey of data quality tools. *Datenbank-Spektrum*, 14(15-21):48, 2005.

[40] Glenford J Myers, Corey Sandler, and Tom Badgett. *The art of software testing*. John Wiley & Sons, 2011.

[41] Ron Patton. *Software testing*. Sams, 2001.

[42] Thomas Ostrand. White-box testing. *Encyclopedia of Software Engineering*, 2002.

[43] Frances E Allen. Control flow analysis. In *ACM Sigplan Notices*, volume 5, pages 1–19. ACM, 1970.

[44] Janvi Badlaney Rohit Ghatol Romit Jadhwani. An introduction to dataflow testing. *NCSU CSC TR-2006-22*, 2006.

[45] Raul Barbosa, Johan Karlsson, Henrique Madeira, and Marco Vieira. Fault injection. In *Resilience Assessment and Evaluation of Computing Systems*, pages 263–281. Springer, 2012.

[46] Algirdas Avizienis, J-C Laprie, Brian Randell, and Carl Landwehr. Basic concepts and taxonomy of dependable and secure computing. *IEEE transactions on dependable and secure computing*, 1(1):11–33, 2004.

[47] Henrique Madeira et al. *Assessing, Measuring, and Benchmarking Resilience (AMBER) - State of the Art*. 2009.

[48] Ieee standard glossary of software engineering terminology. *IEEE Std 610.12-1990*, pages 1–84, 12 1990.

[49] Philip Koopman and John DeVale. Comparing the robustness of posix operating systems. In *Fault-Tolerant Computing, 1999. Digest of Papers. Twenty-Ninth Annual International Symposium on*, pages 30–37. IEEE, 1999.

[50] Nathan P Kropp, Philip J Koopman, and Daniel P Siewiorek. Automated robustness testing of off-the-shelf software components. In *Fault-Tolerant Computing, 1998. Digest of Papers. Twenty-Eighth Annual International Symposium on*, pages 230–239. IEEE, 1998.

[51] Manuel Rodríguez, Frédéric Salles, Jean-Charles Fabre, and Jean Arlat. MAFALDA: Microkernel assessment by fault injection and design aid. *Lecture notes in computer science*, pages 143–160, 1999.

[52] Yue Jia and Mark Harman. An analysis and survey of the development of mutation testing. *Software Engineering, IEEE Transactions on*, 37(5):649–678, 2011.

[53] Reda Siblini and Nashat Mansour. Testing web services. In *Computer Systems and Applications, 2005. The 3rd ACS/IEEE International Conference on*, page 135. IEEE, 2005.

[54] Wuzhi Xu, Jeff Offutt, and Juan Luo. Testing web services by XML perturbation. In *Software Reliability Engineering, 2005. ISSRE 2005. 16th IEEE International Symposium on*, pages 10–pp. IEEE, 2005.

[55] Mark Hansen, Stuart Madnick, and Michael Siegel. *Data integration using web services*. Springer, 2003.

[56] G Shankaranarayanan and Yu Cai. "A Web Services Application for the Data Quality Management in the B2B Networked Environment". In *System Sciences, 2005. HICSS'05. Proceedings of the 38th Annual Hawaii International Conference on*, pages 166c–166c. IEEE, 2005.

[57] Maria Grazia Fugini, Barbara Pernici, and Filippo Ramoni. "Quality analysis of composed services through fault injection". *Information Systems Frontiers*, 11(3):227–239, 2009.

[58] Xitong Li, Stuart Madnick, Hongwei Zhu, and Yushun Fan. "Improving Data Quality for Web Services Composition". In *Proceedings of the VLDB Quality in Databases (QDB) Workshop, Lyon, France*, 2009.

[59] Naghmeh Ivaki, Nuno Laranjeiro, and Marco Vieira. Towards evaluating the impact of data quality on service applications. In *Dependable Systems and Networks Workshop (DSN-W), 2013 43rd Annual IEEE/IFIP Conference on*, pages 1–7. IEEE, 2013.

[60] Marek Rychlỳ and Martin Zouzelka. Fault injection for web-services. In *ICEIS (2)*, pages 377–383, 2012.

[61] Lourival F de Almeida and Silvia R Vergilio. Exploring perturbation based testing for web services. In *Web Services, 2006. ICWS'06. International Conference on*, pages 717–726. IEEE, 2006.

[62] Evan Martin, Suranjana Basu, and Tao Xie. Websob: A tool for robustness testing of web services. In *Companion to the proceedings of the 29th International Conference on Software Engineering*, pages 65–66. IEEE Computer Society, 2007.

[63] Munro Malcolm Looker, Nik and Jie Xu. Ws-fit: A tool for dependability analysis of web services. 2004.

[64] Nuno Laranjeiro, Marco Vieira, and Henrique Madeira. Improving web services robustness. In *Web Services, 2009. ICWS 2009. IEEE International Conference on*, pages 397–404. IEEE, 2009.

[65] Inderjeet Singh and Bindia Tarika. Comparative analysis of open source automated software testing tools: Selenium, sikuli and watir. *International Journal of Information and Computation Technology*, 4:15, 2014.

[66] Gerald D Everett; Raymond McLeod; Wiley InterScience (Online service). *Software testing : testing across the entire software development life cycle*. Wiley, 2007.

[67] Gurock Software. Test automation tools - popular automated testing tools and software, April 2016. `http://www.testingtools.com/test-automation/`.

[68] Raimund Hocke. *SikuliX Documentation, Release 1.1.0-Beta1*. 2016.

[69] Selenium Project. Seleniumhq browser automation - test automation for web applications. `http://docs.seleniumhq.org/docs/01_introducing_selenium.jsp#test-automation-for-web-applications`, April 2016.

[70] SikuliX. SikuliX powered by raiman. `http://www.sikulix.com/`.

[71] Watir. Watir.com | web application testing in ruby. `https://watir.com/`.

[72] Sahi Pro. Automation testing tool for web applications | free - sahi. http://sahipro.com/.

[73] Hewlett Packard Enterprise Development LP. Automated testing, unified functional testing, uft | hewlett packard enterprise. http://www8.hp.com/us/en/software-solutions/unified-functional-automated-testing/.

[74] Ranorex GmbH. Test automation for GUI testing | ranorex. http://www.ranorex.com/.

[75] AspectJ Team. The AspectJ programming guide, 2003.

[76] Thomas Mahler and Armin Waibel. JDBC types, December 2012. https://db.apache.org/ojb/docu/guides/jdbc-types.html#mapping-table.

[77] Nuno Laranjeiro, Seyma Nur Soydemir, and Jorge Bernardino. A survey on data quality: Classifying poor data. In *Dependable Computing (PRDC), 2015 IEEE 21st Pacific Rim International Symposium on*, pages 179–188. IEEE, 2015.

[78] Sylvia A Gardner. Spelling errors in online databases: what the technical communicator should know. *Technical Communication*, pages 50–53, 1992.

[79] Barbara Nichols Randall. Spelling errors in the database: Shadow or substance? *Library Resources & Technical Services*, 43(3):161–169, 2011.

[80] Fred J Damerau. A technique for computer detection and correction of spelling errors. *Communications of the ACM*, 7(3):171–176, 1964.

[81] Joseph J Pollock and Antonio Zamora. Collection and characterization of spelling errors in scientific and scholarly text. *Journal of the American Society for Information Science*, 34(1):51–58, 1983.

[82] Terry Ballard and Arthur Lifshin. Prediction of OPAC spelling errors through a keyword inventory. *Information Technology and Libraries*, 11(2):139, 1992.

[83] Kai A Olsen. The $100,000 keying error. *Computer*, (4):108–106, 2008.

[84] Tara Bernard. An $18 million lesson in handling credit report errors. *The New York Times*, 2013.

[85] Jonathan I Maletic and Andrian Marcus. Data cleansing: Beyond integrity analysis. In *IQ*, pages 200–209. Citeseer, 2000.

[86] Harold Thimbleby and Paul Cairns. Reducing number entry errors: solving a widespread, serious problem. *Journal of the Royal Society Interface*, 7(51):1429–1439, 2010.

[87] Openbravo. Openbravo commerce suite. http://www.openbravo.com/about/company/.

[88] Dennis Howlett. Can Openbravo challenge incumbent ERP with open source? http://www.zdnet.com/article/can-openbravo-challenge-incumbent-erp-with-open-source/, March 2011.

[89] Openbravo. Customer Successes | Other Industries | Openbravo. http://www.openbravo.com/other-industries/customers-successes/.

[90] Michael Stonebraker and Greg Kemnitz. The POSTGRES next generation database management system. *Communications of the ACM*, 34(10):78–92, 1991.

[91] Ken Kitchen. Gartner on the state of PostgreSQL and open-source DBMS. https://www.linkedin.com/pulse/gartner-state-postgresql-open-source-dbms-ken-kitchen, July 2015.

[92] Donald Feinberg, Merv Adrian, Nick Heudecker, Adam M. Ronthal, and Terilyn Palanca. Magic quadrant for operational database management systems. *Gartner*, October 2015.

[93] Nikita Salnikov-Tarnovski. Most popular Java EE containers: 2015 edition. https://plumbr.eu/blog/java/most-popular-java-ee-containers-2015-edition, April 2015.

[94] Craig Buckler. Browser Trends January 2016: 12 month review. http://www.sitepoint.com/browser-trends-january-2016-12-month-review/, January 2016.

[95] James Martin. *Managing the Data Base Environment*. Prentice Hall PTR, Upper Saddle River, NJ, USA, 1st edition, 1983.

[96] Rolf Molich and Jakob Nielsen. Improving a human-computer dialogue. *Communications of the ACM*, 33(3):338–348, 1990.

[97] Ram Chillarege. Orthogonal defect classification. *Handbook of Software Reliability Engineering*, pages 359–399, 1996.

[98] Stephen H Kan. Secrets of software quality: 40 innovations from IBM. *IBM Systems Journal*, 35(1):116, 1996.

[99] Arrayindexoutofboundsexception - issue #369 - pgjdbc/pgjdbc - github. https://github.com/pgjdbc/pgjdbc/issues/369, September 2015.

[100] IEEE standard classification for software anomalies. *IEEE Std 1044-2009 (Revision of IEEE Std 1044-1993)*, pages 1–23, Jan 2010.

Appendices

A Data Mutation Tables

Table A.1 shows the mutations defined for the integer data types. The original input used for the "Example" column was *123*.

Table A.1: Mutations for Integer data types

No	Fault Description	Argument	Configuration	Example
1	Replace by random same size	-	-	145
2	Remove one random numeric character from data	-	-	13
3	Duplicate one of the numeric characters	Position	From 0 to the numeric value's original size. Repeat digit will be put in front of Position	1223
4	Invert two consecutive digits	Begin	From 0 to the numeric value's original size minus 1	132
		Length	From 2 to the numeric value´s original size minus Begin	
5	Add one numeric character	Position	Random	1123
6	Flip sign	-	Positive numbers become negative and vice-versa	-123
7	Replace one random numeric character	Position	Random.	153
8	Add 1	-	-	124
9	Substract 1	-	-	122

Table A.2 presents the mutation for the Time data types. The original input used for the "Example" column was *02:10:08*. It should be noted that values higher or lower than the limit of the fields (hours, minutes, seconds) are considered valid in these mutations.

Table A.2: Mutations for Time data types

No	Fault Description	Argument	Configuration	Example
1	Replace by null value	-	-	null
2	Replace by empty dummy time	-	-	0:00:00
3	Replace by random time	Field	Hours field	7:10:08
			Minutes field	2:52:08
			Seconds field	2:10:25
		Value	Uniformly random between the maximum and minimum allowed value for the field	-
4	Reverse field	Field	Hours field	20:10:08
			Minutes field	02:01:08
			Seconds field	02:10:80
5	Add one extra digit	Position	Before hours value	102:10:08
			After hours value	021:10:08
			Before minutes value	02:110:08
			After minutes value	02:101:08
			Before second value	02:10:108
			After second value	02:10:081
6	Duplicate one digit	Position	Before hours value	002:10:08
			After hours value	022:10:08
			Before minutes value	02:110:08
			After minutes value	02:100:08
			Before second value	02:10:008
			After second value	02:10:088
7	Remove one digit	Field	Hours field	0:10:08
			Minutes field	2:01:08
			Seconds field	2:10:00
		Position	Random character	-

Table A.3 presents the mutation for the Date data types. The original input used for the "Example" column was *2020-05-10*. As happened with the Time data types, values higher or lower than the limit of the fields (year, month, day) are considered valid in these mutations.

Table A.3: Mutations for Date data types

No	Fault Description	Argument	Configuration	Example
1	Replace by null value	-	-	null
2	Replace by empty dummy date	-	-	0000-00-00
3	Replace by random date	Field	Year field	2120-05-10
			Month field	2020-08-10
			Day field	2020-05-12
		Value	Uniformly random between the maximum and minimum allowed value for the field	-
4	Add one extra digit	Position	Before year value	12020-05-10
			After year value	20201-05-10
			Before month value	2020-105-10
			After month value	2020-051-10
			Before day value	2020-05-110
			After day value	2020-05-101
5	Remove one digit	Field	Year field	2000-05-10
			Month field	2020-5-10
			Day field	2020-05-01
6	Replace one digit with a random digit	Field	Year field	2320-05-10
			Month field	2020-08-10
			Day field	2020-05-14
7	Invert digits in a field	Field	Year field	2002-05-10
			Month field	2020-50-10
			Day field	2020-05-01
8	Swap values among fields	Source	Year field	05-2020-01
			Month field	2020-10-05
			Day field	10-05-2020
		Target	Year field	10-05-2020
			Month field	2020-10-05
			Day field	10-05-2020
9	Replace with the default value	Default date	-	1970-01-01
10	Remove first 2 digits of year field	-	-	0020-05-10
11	Add 100 years	-	-	2120-05-10
12	Subtract 100 years	-	-	1920-05-10
13	Duplicate digit	Field	Year field	20200-05-10
			Month field	2020-055-10
			Day field	2020-05-110

Table A.4 presents the mutation for the Timestamp data types. The original input used for the "Example" column was *2020-05-10 11:11:11.121212121*. Once again, values higher or lower than the limit of the fields are considered valid in these mutations.

Table A.4: Mutations for Timestamp data types

No	Fault Description	Argument	Configuration	Example
1	Replace by null timestamp	-	-	null
2	Replace by empty dummy timestamp	-	-	0000-00-00 00:00:00.000000
3	Replace by random timestamp	Position	Year field	2012-05-10 11:11:11.121212
			Month	2020-11-10 11:11:11.121212
			Day field	2020-05-29 11:11:11.121212
			Hour field	2020-05-10 02:11:11.121212
			Minute field	2020-05-10 11:54:11.121212
			Second field	2020-05-10 11:11:23.121212
			Nanosecond field	2020-05-10 11:11:11.565654
		Value	Uniformly random between the maximum and minimum allowed value for the field	-
4	Reverse field	Field	Year field	0202-05-10 19:45:31.143684
			Month	2020-50-10 19:45:31.143684
			Day field	2020-05-01 19:45:31.143684
			Hour field	2020-05-10 91:45:31.143684
			Minute field	2020-05-10 19:54:31.143684
			Second field	2020-05-10 19:45:13.143684
			Nanosecond field	2020-05-10 19:45:31.486341
5	Add one extra character to timestamp	Field	Before year value	12020-05-10 11:11:11.121212
			After year value	20209-05-10 11:11:11.121212
			Before month value	2020-205-10 11:11:11.121212
			After month value	2020-051-10 11:11:11.121212
			Before day value	2020-05-310 11:11:11.121212
			After day value	2020-05-105 11:11:11.121212
			Before hour value	2020-05-10 611:11:11.121212
			After hour value	2020-05-10 114:11:11.121212
			Before minute value	2020-05-10 11:711:11.121212
			After minute value	2020-05-10 11:113:11.121212
			Before second value	2020-05-10 11:11:411.121212
			After second value	2020-05-10 11:11:119.121212
			Before nanosecond value	2020-05-10 11:11:11.9121212
			After nanosecond value	2020-05-10 11:11:11.1212128
6	Duplicate one character on timestamp value	Position	Before year value	22020-05-10 11:11:11.121212
			After year value	20202-05-10 11:11:11.121212
			Before month value	2020-505-10 11:11:11.121212
			After month value	2020-055-10 11:11:11.121212
			Before day value	2020-05-110 11:11:11.121212
			After day value	2020-05-101 11:11:11.121212
			Before hour value	2020-05-10 111:11:11.121212
			After hour value	2020-05-10 111:11:11.121212
			Before minute value	2020-05-10 11:111:11.121212
			After minute value	2020-05-10 11:111:11.121212
			Before second value	2020-05-10 11:11:111.121212
			After second value	2020-05-10 11:11:111.121212
			Before nanosecond value	2020-05-10 11:11:11.2121212
			After nanosecond value	2020-05-10 11:11:11.1212122
7	Add 100 years to the timestamp	-	-	2120-05-10 11:11:11.121212
8	Subtract 100 years from the timestamp	-	-	1920-05-10 11:11:11.121212
9	Remove 2 digits of the year field	-	-	20-05-10 11:11:11.121212121

Table A.5 presents the mutation for the Boolean data types. The original input used for the "Example" column was *True*.

Table A.5: Mutations for Boolean data types

No	Fault Description	Argument	Configuration	Example
1	Replace by null value	-	-	null
2	Flip value	-	True becomes False, and vice-versa	FALSE
3	Set to True	-	-	TRUE
4	Set to False	-	-	FALSE

Table A.6 presents the mutation for the Decimal data types. The original input used for the "Example" column was *123,13*.

Table A.6: Mutations for Decimal data types

No	Fault Description	Argument	Configuration	Example
1	Replace by null value	-	-	null
2	Replace by random same-size decimal values	-	-	456,12
3	Add 1	-	-	124,13
4	Substract 1	-	-	122,13
5	Remove one random numeric character from data	-	-	13,13
6	Duplicate one of the numeric characters	Position	Random from 0 to the numeric value's original size. Repeated digit will be put in front of original digit	1223,13
7	Reverse two consecutive characters	Position	Random from 0 to the numeric value's original size minus 1	132,13
8	Add numeric character	Position	Random from 0 to the numeric value's original size	4123,13
9	Flip sign	-	Positive numbers become negative and vice-versa	-123,13
10	Zero out the integer part	-	-	0,13
11	Zero out the fractional part	-	-	123,0
12	Remove the integer part	-	-	13,0
13	Remove the fractional part	-	-	123,0
14	Remove the decimal separator	-	-	12313,0
15	Add 10^-X	X	Random, from 1 to the limit of the type	123,14
16	Subtract 10^-X	X	Random, from 1 to the limit of the type	123,12
17	Shift decimal separator	Direction	Left	12,313
			Right	1231,3
		Amount	One position, as long as there at is at least 1 digit in both sides	-
18	Add 1 zero next to decimal separator	Position	After decimal separator	123,013
			Before decimal separator	1230,13
19	Swap fractional and integer parts	-	-	13,123
20	Reverse fractional part	-	-	123,31
21	Reverse integer part	-	-	321,13
22	Skipping decimal point near zero	-	-	12313,0

Table A.7 presents the mutation for the Double data types. The original input used for the "Example" column was *123,13*.

Table A.7: Mutations for Double data types

No	Fault Description	Argument	Configuration	Example
1	Replace by random same-size double values	-	-	456,12
2	Add 1	-	-	124,13
3	Substract 1	-	-	122,13
4	Remove one random numeric character from data	-	-	13,13
5	Duplicate one of the numeric characters	Position	From 0 to the numeric value's original size. Repeat digit will be put in front of Position	1223,13
6	Reverse two consecutive numeric characters	Begin	From 0 to the numeric value's original size minus 1	132,13
7	Add numeric character	Position	Random position	1223,13
8	Flip sign	-	Positive numbers become negative and vice-versa	-123,13
9	Zero out the integer part	-	-	0,13
10	Zero out the fractional part	-	-	123,0
11	Remove the integer part	-	-	13,0
12	Remove the fractional part	-	-	123,0
13	Remove the decimal separator	-	-	12313,0
14	Add 10^-X	X	Random, from 1 to the limit of the type	123,14
15	Subtract 10^-X	X	Random, from 1 to the limit of the type	123,12
16	Shift decimal separator	Direction	Left	12,313
			Right	1231,3
		Amount	One position, as long as there at is at least 1 digit in both sides	-
17	Add a zero next to decimal separator	Position	After decimal separator	123,013
			Before decimal separator	1230,13
18	Swap fractional and integer parts	-	-	13,123
19	Reverse fractional part	-	-	321,13
20	Reverse integer part	-	-	123,31
21	Skipping decimal point near zero	-	-	12313

Table A.8 presents the mutation for the Binary data types. The original input used for the "Example" column was *[10, 4, 5, 1, 149]*, where each position inside the array represent the value of a byte (from 0 to 255).

Table A.8: Mutations for Binary data types

No	Fault Description	Argument	Configuration	Example
1	Replace by null value	-	-	null
2	Replace by empty	-	-	[]
3	Add byte	Position	Prefix	[1, 10, 4, 5, 149]
			Middle (Random)	[10, 4, 5, 1, 149]
			Suffix	[10, 4, 5, 149, 1]
		Value	Random from 0 to 255	-
4	Remove bytes	Position	Prefix	[5, 149]
			Middle (Random)	[10, 149]
			Suffix	[10, 4]
		Quantity	Random from 1 to size of Binary - Position	-
5	Duplicate byte	Position	Prefix	[10, 10, 4, 5, 149]
			Middle (Random)	[10, 4, 5, 5, 149]
			Suffix	[10, 4, 5, 149, 149]
6	Replace byte	Position	Prefix	[6, 4, 5, 149]
			Middle (Random)	[10, 4, 6, 149]
			Suffix	[10, 4, 5, 6]
		Value	Random from 0 to 255	-
7	Drop X most significant bits	X	From 1 to 8 bits that make a byte	[0, 0, 0, 16]
8	Drop X least significant bits	X	From 1 to 8 bits that make a byte	[2, 4, 5, 5]
9	Flip sign bit of a byte	Position	Prefix	[138, 4, 5, 149]
			Middle (Random)	[10, 132, 5, 149]
			Suffix	[10, 4, 5, 21]

Table A.9 presents the mutation for the Object data types. Due to the abstract definition of a Java object, and how difficult it would be to represent in text, the example column may be left empty.

Table A.9: Mutations for Object data types

No	Fault Description	Argument	Configuration	Example
1	Replace by null object	-	-	null
2	Replace by empty object	-	-	-
3	Replace by object of different type	-	-	-

Table A.10 presents the mutation for the Reference data types. Due to the abstract definition of a Java reference, and how difficult it would be to represent in text, the example column may be left empty.

Table A.10: Mutations for Reference data types

No	Fault Description	Argument	Configuration	Example
1	Replace by null	-	-	null
2	Replace by reference to object of different type	-	-	-

B Activity Diagrams of Test Cases

Figure B.1 shows the activity diagram for Test Case 2 - Create Organization.

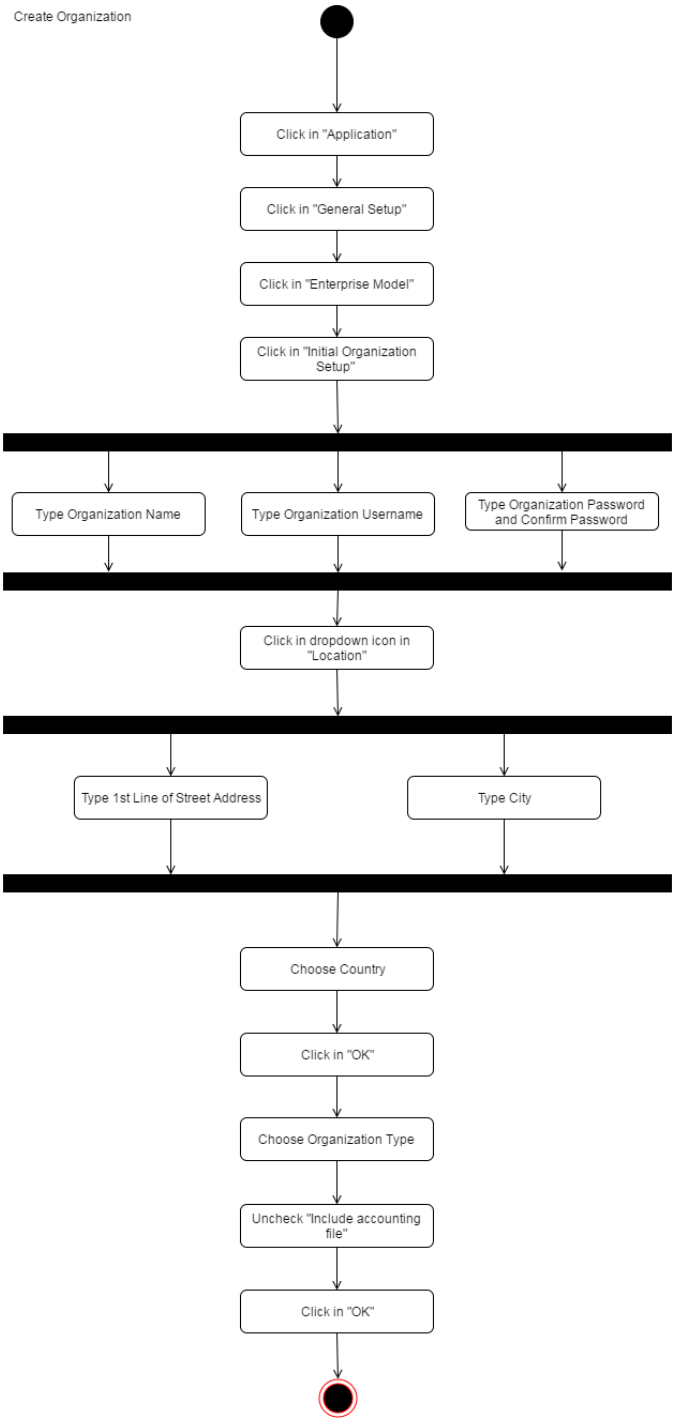

Figure B.1: Activity Diagram for Test Case 2 - Create Organization

Figure B.2 shows the activity diagram for Test Case 3 - Create User.

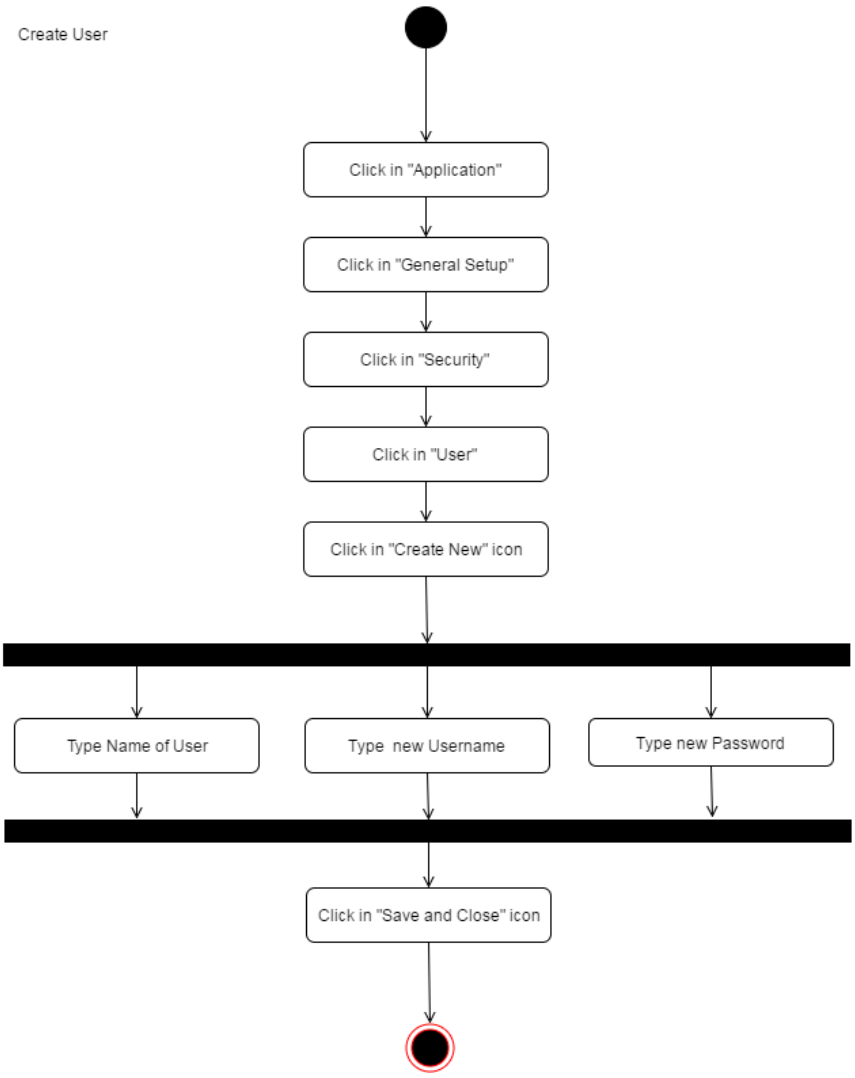

Figure B.2: Activity Diagram for Test Case 3 - Create User

Figure B.3 shows the activity diagram for Test Case 4 - Create Role.

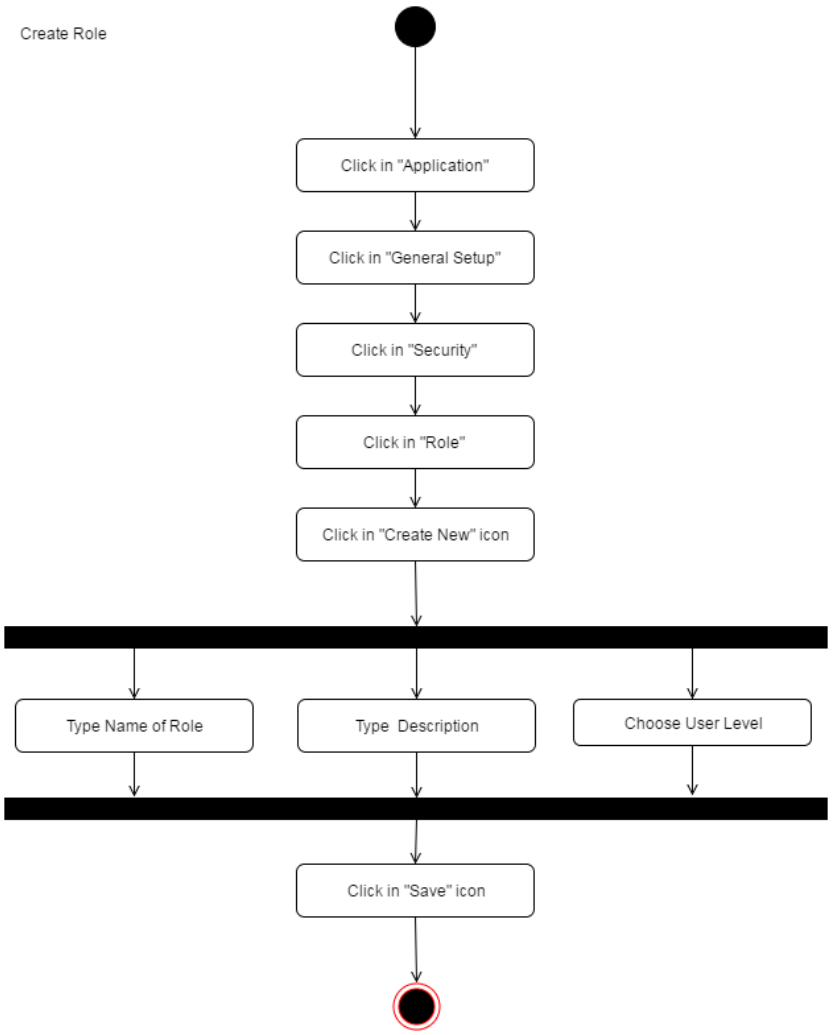

Figure B.3: Activity Diagram for Test Case 4 - Create Role

Figure B.4 shows the activity diagram for Test Case 5 - Create Product.
Figure B.5 shows the activity diagram for Test Case 6 - Delete Product.

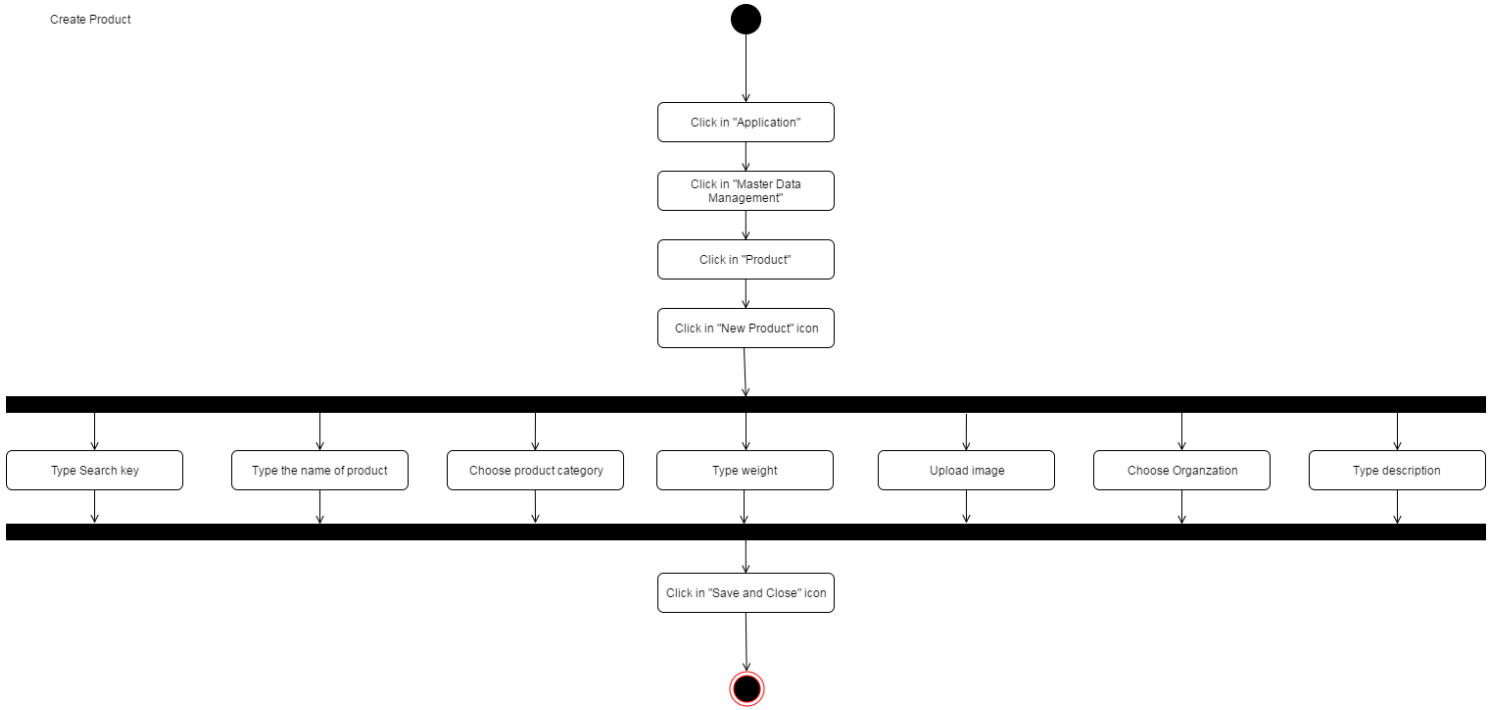

Figure B.4: Activity Diagram for Test Case 5 - Create Product

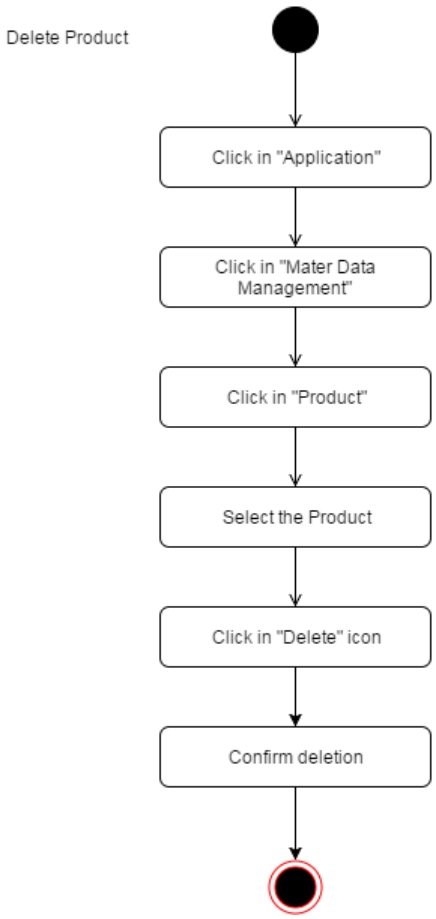

Figure B.5: Activity Diagram for Test Case 6 - Delete Product

Figure B.6 shows the activity diagram for Test Case 7 - Update Product.

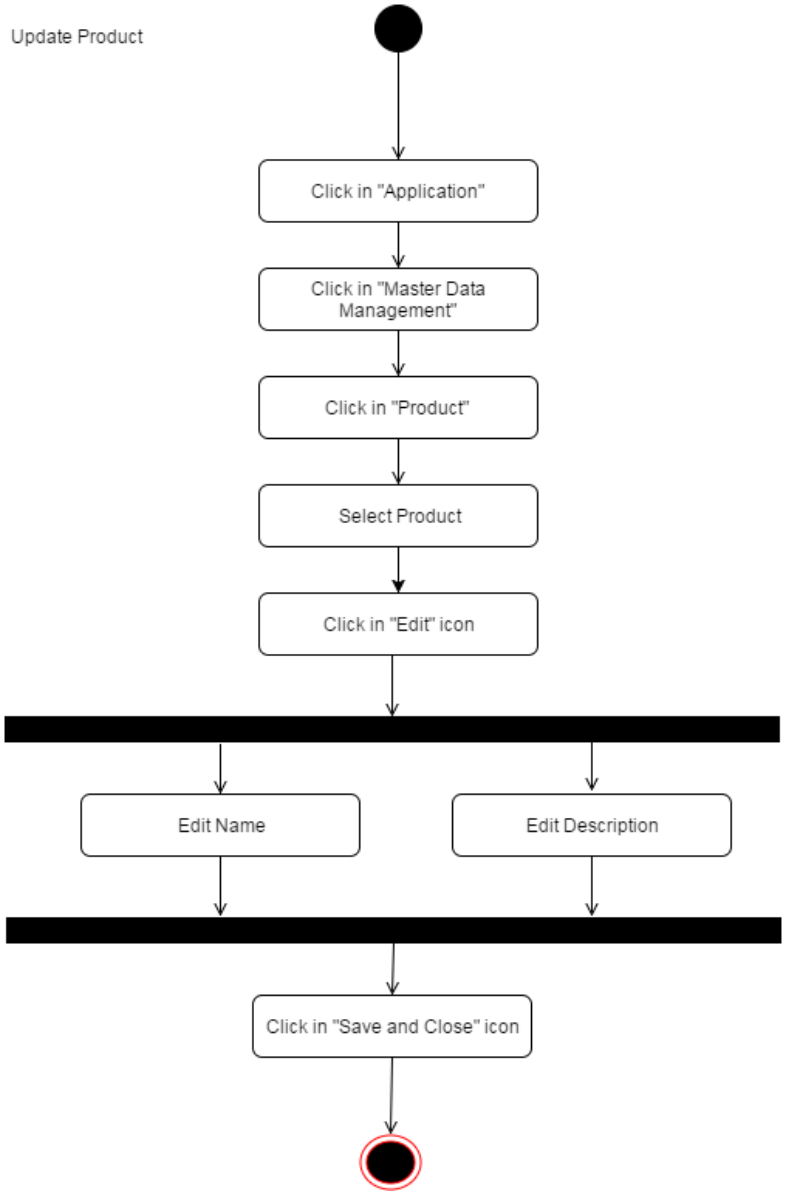

Figure B.6: Activity Diagram for Test Case 7 - Update Product

Figure B.7 shows the activity diagram for Test Case 8 - Export Product Categories to Spreadsheet.

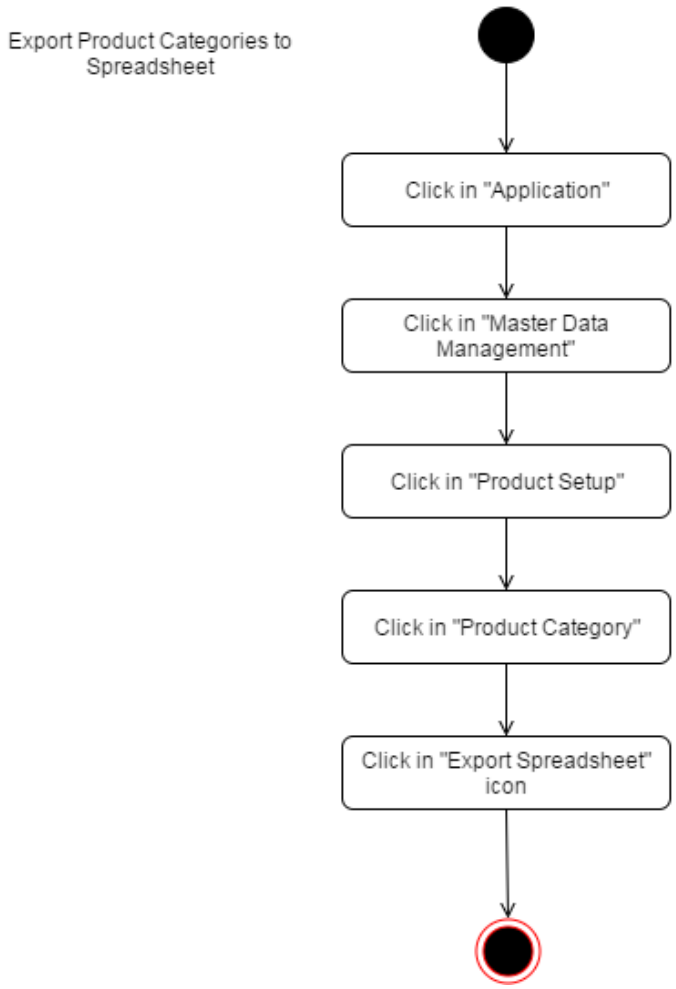

Figure B.7: Activity Diagram for Test Case 8 - Export Product Categories to Spreadsheet

C Survey paper on Data Quality

Below is provided the conference paper that was presented in *The 21st IEEE Pacific Rim International Symposium on Dependable Computing* (PRDC 2015).

A Survey on Data Quality: Classifying Poor Data

Nuno Laranjeiro*, Seyma Nur Soydemir*, and Jorge Bernardino*†
*CISUC, Department of Informatics Engineering
University of Coimbra, Portugal
seyma@student.dei.uc.pt, cnl@dei.uc.pt
†Polytechnic of Coimbra, Portugal
jorge@isec.pt

***Abstract*—Data is part of our everyday life and an essential asset in numerous businesses and organizations. The quality of the data, i.e., the degree to which the data characteristics fulfill requirements, can have a tremendous impact on the businesses themselves, the companies, or even in human lives. In fact, research and industry reports show that huge amounts of capital are spent to improve the quality of the data being used in many systems, sometimes even only to understand the quality of the information in use. Considering the variety of dimensions, characteristics, business views, or simply the specificities of the systems being evaluated, understanding how to measure data quality can be an extremely difficult task. In this paper we survey the state of the art in classification of poor data, including the definition of dimensions and specific data problems, we identify frequently used dimensions and map data quality problems to the identified dimensions. The huge variety of terms and definitions found suggests that further standardization efforts are required. Also, data quality research on Big Data appears to be in its initial steps, leaving open space for further research.**

Keywords*—*Poor data quality; dirty data; poor data classification; data quality problems

I. Introduction

Nowadays data (i.e., any information, especially facts or numbers, regarding some object [1]) supports many enterprise-level businesses, forming one of their most valuable assets. Its use can have short-term impact on the businesses (e.g., the reception of purchase orders) but also medium or long-term implications. The way data is being currently used in data analysis or mining processes to support decision making, including higher level decisions such as operational (e.g., at a department, or local store level) or tactical (i.e., from the organization point-of-view) is a clear example of such implications [2]. Thus, the need for organizations to possess high quality data is essential, as it can increase the opportunity for the organizations to deliver services that clients can rely on [3].

Data quality, sometimes referred to as information quality [4], has been defined in many diverse ways in the literature [5]. High quality data can refer to whether data meets the expectations of the users [6], which can be human users or systems, or can be defined as "data that is fit for use by data consumers" [7]. In the ISO/IEC 25012 standard [8] it is defined as "the degree to which a set of characteristics of data fulfills requirements". Examples of such characteristics include completeness, accuracy or consistency of the data [9], whereas the requirements express the needs and constraints that contribute to the solution of some problem [10].

According to the above definitions, poor data quality, can be defined as the degree to which a set of characteristics of data does not fulfill the requirements. By not fulfilling all the requirements it is not fit for use by data consumers, and thus it is prone to cause impact on the entities involved (e.g., a company, a customer). The impact can be of three types: operational (i.e., causing customer and employee dissatisfaction and increased costs); tactical (i.e., affecting decision making and causing mistrust), and strategic impacts (i.e., affecting the overall organization's strategy) [11]. Overall, any system or enterprise that heavily relies on data is prone to experience problems if the data being handled does not possess the expected quality attributes.

Previous research and industry reports are clear in indicating the severe damage caused by the presence of poor quality data in diverse contexts and at many different levels [12], [4], [13], [14]. In addition to the severe impact on business performance, at the same time investment is being made in data quality using unsophisticated approaches [14], frequently without proper knowledge on data quality. In fact, understanding what is good data, how it can be measured, and also improved, is a quite difficult problem due to the variety of definitions, its multiple dimensions and applicability to different contexts [5]. In this paper we analyze the state of the art in data quality, particularly considering the perspective of data quality classification.

The quality of data can be analyzed from multiple dimensions. A dimension is a measurable data quality property that represents some aspect of the data (e.g., accuracy, consistency) that can be used to guide the process of understanding quality [15], [16], [9]. Thus, some particular data can be described as of being high quality, according to one or more dimensions. Since the use of data is pervasive, it is very common to find different terms referring to the same dimension (e.g., currency is many times used as timeliness) in the literature [5]. For instance, dimensions are sometimes referred to as attributes, or characteristics [17]. *Data quality problems*, often named dirty data [18], are the specific problem instances that we can find within a dimension (e.g., under the accuracy dimension, we can find format problems) and that prevent data from being regarded as high quality [19]. The terms used for data quality problems also differ [19], [20], [21], and also the mapping between each different problem to each dimension (depending on the definitions used).

Performing a superficial identification of data quality issues is relatively straightforward. However, performing a thorough identification of issues (e.g., to be used as means for quan-

titatively measuring data quality, or to select data cleaning procedures), which is obviously guided by the dimensions that characterize the quality of data, is a quite difficult task. This is especially true, considering the multitude of categories and sub-categories that different authors propose, many times to designate the same aspects [16].

In this paper we survey data quality research, especially in the field of data quality classification, with focus on data quality dimensions and data quality problems. We describe the different classification structures and dimensions identified by researchers, present the different data quality problems found, and perspectives used to group these problems. Finally, we map the problems into a subset of dimensions most frequently cited in the literature. The analysis suggests the need for further standardization efforts and open space for research in data quality for Big Data, which appears to be in an initial stage.

This paper is organized as follows. Section II discusses the impact of poor data quality and Section III describes typical dimensions used to describe data quality. Section IV identifies data quality problems and maps them to a set of frequently cited data quality dimensions. Finally, Section V concludes this paper.

II. The impact of poor data

There are clear evidences that data quality problems affect many organizations [22], [14] and result in different types of impact, including substantial social and economical impacts [11], [13]. Understanding these impacts is relevant to understand the overall importance of the presence of poor data and its far-reaching implications, but also to create awareness in practitioners and researchers regarding this topic. In this section we briefly overview the different types of impact caused by poor data, with emphasis on financial cost, which can be the most immediate impact of a data quality problem.

The need for awareness of poor data in enterprises and a classification of its impact is presented in [11]. At a conceptual level and from the organization point-of-view, the following main impact types can occur:

- Operational: Employee and customer dissatisfaction, and also increased cost of operations (e.g., resources are used to correct errors).
- Tactical: Effects on decision making, more difficult reengineering, mistrust within the organization.
- Strategic: Increased difficulty to define and execute organizational strategies, contributes to issues regarding data ownership, and diverts management attention from crucial aspects (e.g., customers or competition).

The main types of data quality impact are described from a business perspective in [13]. The four impact areas identified are: 1) Confidence and satisfaction-based impact: refers to the satisfaction of actors, such as customers, employees, or suppliers, which can result in lowered organizational trust, incoherent management and operational reporting, and incorrect decisions. 2) Productivity: higher processing time, lower throughput and, as a consequence, increased workloads, ultimately reflecting on the product quality. 3) Risk and compliance: refers to possible risks (e.g., investment, competitive, capital fraud) and to regulations and policies, such as those imposed by governments, industry, or the company itself. 4) Financial: lower profit, decreased cash flow and higher operational costs. Penalties tend to increase and opportunities are lost.

Despite of the multiple perspectives on the impact of data quality, in the end, all have the potential to bring in financial costs for organizations. The following examples illustrate real cases, where poor data brought significant costs to the entities involved: 1) Gartner recently found that the annual cost brought in by poor data in a set of surveyed organizations in 2014 was on average of $13.3 million dollars [23]. 2) The US Postal Service calculated in 2013 that more than 6 billion pieces of mail could not be delivered due to wrong bad postal data. Processing that email costed over $1.5 billion, which is just the direct fraction of the wasted costs [24]. 3) Poor data is known to cost over 3 trillion per year to US economy [25].

The potentially huge financial cost caused by poor data is an obvious aspect that has been the focus of several research works, which present different views on this matter [26], [27], [28]. Besides the obvious costs brought in by poor data, on the other hand managing data quality also comes with its own costs, and there is always a trade-off between any data quality management strategy or procedure and the potential loss caused by poor data. This further reflects the importance of understanding the costs of poor data.

The US Department of Defense produced a set of data quality guidelines [26], where the cost of poor data quality is distinguished into two main groups: i) direct costs; and ii) indirect costs. The direct data quality costs are related with the impact of poor data in a process and with the cost of improving and correcting the poor data. This group includes controllable costs, due to the management of data quality (e.g., prevention, appraisal, correction costs), resultant costs, related with the impact of poor data quality (e.g., internal and external error costs), and equipment and training costs (e.g., costs of hardware and software for basic operational support, and training required to prevent, assess, and correct data). Indirect data quality costs are essentially related with the loss of creditability and customer satisfaction.

Authors in [29] review the costs associated with low quality data and categorizes them in two types: i) improvement costs; and ii) costs due to low data quality. The former are related with the information quality process such as prevention (training, monitoring, standard development and deployment), detection (analysis and detection), and repair (planning and implementation). The latter costs are categorized in direct or indirect. Direct costs refer to verification, re-entry, and compensation costs; whereas indirect costs refer to lower reputation, wrong decisions or actions and sunk investment. In [28] authors have reused the taxonomy defined in [29] as a basis for further analysis on data problems.

Data quality costs are classified under three main groups in [27]: a) data entry; b) data processing; and c) data use. Data entry quality costs may either be caused by the low quality of data (e.g., cost of correcting), or preventive costs (e.g., training, defect prevention). Data processing quality costs are also organized in two subgroups: costs of re-processing dirty data (e.g., re-work, rolling back) and process improvement costs (e.g., costs of detecting, analyzing, and reporting dirty

data). Finally, in data use costs, the authors consider a division between direct costs (e.g., lost revenue and opportunities, compensation costs) and indirect costs (e.g., liability costs, lower reputation).

Although we can find different views for the classification of the impact of data quality, in general the key areas focused are costs associated with poor data and costs regarding improvement of data quality. While some authors group costs according to the phase in data lifecycle where the action is taken (e.g., entry, processing, use), others use different criteria, such as the moment when the action takes place (i.e., before or after poor data caused some impact). The categorization of costs in direct and indirect is also very much common, but while some research takes into account the cost in association with ensuring data quality, other authors are simply interested in the more direct costs related with poor data quality. Despite these different views, understanding the cost of poor data is a crucial tool for researchers or practitioners to take the right actions, concerning their specific goals.

III. The Dimensions of Data Quality

In this section, we review the literature and systematize different definitions of data quality dimensions (i.e., the measurable data quality properties that represent some aspect of the data, such as accuracy, or completeness). We build on this knowledge to identify the most frequently cited dimensions. The detailed information regarding the dimensions identified in all analyzed works is summarized by the end of this section, in Table II. The literature review is organized in the following two major sets and in chronological order within each set (with exception of the seminal work in [15], which is analyzed first):

1) **Set I:** Research or industry work that **defines** data quality dimensions.
2) **Set II:** Research or industry work that **selects, ranks, or suggests a subset** of data quality dimensions.

In order to determine the quality of data and how it can be improved, measuring data quality is a key activity in any data-centric organization. However, it is impossible to evaluate without agreeing on what should be measured [30]. With the goal of identifying measurable aspects of data quality (i.e., dimensions), different perspectives and strategies are used to perform this identification and to organize the dimensions [17].

A seminal work in [15] provides a definition of data quality dimensions, gathered from data consumers, and organizes these dimensions in categories. Hundreds of dimensions were collected and then reduced into 20 under 4 categories, and further reduced and repositioned into 15 dimensions under the following 4 categories. The **intrinsic category** includes dimensions which express the natural quality of the data; the dimensions under the **contextual category** express the fact that data quality must be considered within a specific context; the **representational category** refers to dimensions that are related with the format and meaning of the data; and finally, the **accessibility category**, refers to dimensions that express how accessible data is to users. The final set of dimensions under these four categories is presented in Table I.

TABLE I. Data quality dimensions, adapted from [15].

Category	Dimension	Description
Intrisic	Accuracy	Data is is correct (error-free) and reliable.
	Believability	Degree to which data is seen as credible and true.
	Objectivity	How impartial the data is.
	Reputation	Data contents or source are kept in high consideration.
Contextual	Appropriate amount	How suitable is the quantity of the data.
	Completeness	Refers to the scope of the information in the data.
	Relevancy	How usable, applicable, or interesting the data is.
	Value-added	Data provides a competitive advantage.
	Timeliness	The age of the data.
Representational	Concise representation	Data is compactely represented.
	Ease of understanding	How clear, readable, or understandable the data is.
	Interpretability	The extent to which the data meaning is explained.
	Consistency	Data continuously presented in the same format.
Accessibility	Access security	Access is secure or can be restricted.
	Accessibility	The degree to which the data is retrievable.

The research grouped in ***Set I***, which this paragraph begins, ***defines data quality dimensions*** and usually provides some hierarchical view of the dimensions. An example is [31] where dimensions are categorized according to an external view, which is related with the use and the effect of a given system, and to an internal view, which refers to the construction and internal operations needed to achieve some functionality. The dimensions are further separated into data-related and system-related. As an example of dimensions fitting in the external view, conciseness is data-related dimension, whereas efficiency is a system-related dimension. Some dimensions fit on both views, such as timeliness, or on both the data and the system category.

In [7] the authors use the same definitions presented in [15], discuss a set of data quality projects, and provide recommendations for professionals to improve data quality from the data consumer perspective. Data quality is put in perspective of dynamic organizations that many times downsize or merge in [26]. The dimensions used in this work are accuracy, completeness, consistency, timeliness, uniqueness and validity. Authors discuss that data quality characteristics and conformance measures used in data quality management in the US Department of Defense are similar to those used to measure data quality in most information systems, despite that in this case the size of the potential issues can be quite large, as it involves thousands of systems worldwide.

A modeling perspective is taken in [32] to define data quality dimensions. In that particular context, a data item is considered to be a triple (a *value*, from a *domain*, within an *entity*). The three categories defined by the author are associated with: i) the data model (which mostly refers to the structure of the data), ii) the data values (the raw data itself), and with the rules for iii) the data representation (the set of rules that define how to keep data).

A broad view on several data quality topics is provided in [33]. On the information quality assessment topic, the author discusses several characteristics of data quality, which are grouped in two categories: i) Inherent quality which refers to how accurate the data describes the world being modeled (i.e., the static quality characteristics of the data, such as completeness of the data, or conformance to business rules); and ii) Pragmatic quality which refers to how the data allows users to achieve their goals and how understandable it is in a particular format (e.g., how clear, how usable it is).

Information quality criteria (e.g., reputation, completeness, accuracy) are discussed in [34] and organized in three classes: 1) Subject, which comprises criteria that can only be scored by

individual users, based on their knowledge; 2) Object, which includes criteria that are determined by the analysis of the information itself; and 3) Process, which holds criteria that are determined by querying, which means that the procedure of querying is the source of the scores.

Principles that can be used by organizations towards the definition of usable data quality metrics are described in [12]. The authors discuss that, in the enterprise domain and in areas such as as healthcare, finance, or consumer product, companies must consider not only the objective metrics of the data, but also the subjective metrics which reflect the experience of the individuals that are involved with the data. In this work, the data quality dimensions are quite similar to the ones presented in [15] and [7] and have been extracted from [35]. The main difference is that there are new references to 'Ease of Manipulation' and 'Free-of-Error'. There are also slight differences in names, such as 'Security', or 'Understandability'. In [35], the authors aim to provide a methodology for assessing the performance of organizations in developing and delivering information products and services to consumers.

An enterprise view of data quality is presented in [36], where 5 categories are used to group dimensions. Data quality is defined in terms of the data model (e.g., how clear and robust); the data values (e.g., how accurate the values are, or how complete); the information domains, which mostly refers to the presence and enforcement of standards and usage agreements at the enterprise level; the data presentation (e.g., how portable or appropriate); and the information policy (e.g., presence and quality of metadata, privacy). In [37] the authors mention 24 characteristics which can be integrated in 5 dimensions (accuracy, timeliness, comparability, usability, relevance). No explicit mapping between dimensions and characteristics is presented in this work. The AIMQ model is based on the PS/IQ model and defines four quadrants for data quality dimensions [38]. 'Sound' and 'Dependable' are two quadrants that refer to information that complies with specifications, the former is associated with product quality, the latter with service quality. The remaining two quadrants are 'Useful' and 'Usable' and refer to data that meets or exceeds the consumer's expectations. Again they match product and service quality, respectively.

A conceptual framework and belief function to assess information quality is discussed in [39]. In what concerns classification, 4 criteria are used: Accessibility, the capability to retrieve information; Interpretability, how understandable and meaningful the data is to the user; Relevance, which is the applicability of the data to the user's goals; and Credibility, the degree of belief on the information, based on its accuracy, completeness, consistency, and non-fictitiousness.

A general framework for assessment of information quality is proposed in [40]. It considers a large amount of data quality problems, and defines a taxonomy of dimensions divided in three categories: i) Intrinsic, characteristics that can be measured with respect to some reference; ii) Relational or contextual, which refers to aspects that concern relationships or some specific usage context; and iii) Reputation, which refers to the position of the object with respect to some activity or cultural structure.

The ISO/IEC 25012 standard [8] defines a data quality model that comprises 15 characteristics within two points of view: inherent, which includes all characteristics that intrinsically have the potential to comply with the explicit and implicit need under specified conditions; and system dependent, which refers to how data quality is achieved and kept in a computer system under specified conditions. Some of the characteristics share both points of view.

Data quality dimensions are discussed from a practical point-of-view in [17]. The author's rationale is that, although we can find many proposals for dimensions in the literature, practitioners usually tend to use dimensions that can be effectively measured. Thus, these tend to assume a greater importance in such contexts. The dimensions discussed in [17] are divided in three main groups: intrinsic, contextual, and qualitative. The two former groups hold a set of dimensions, with each dimension being associated with a set a characteristics, in turn associated with metrics that allow measuring that particular aspect of the data. The last group essentially reflects the dimensions for which the measuring aspects are less clear.

The dimensions of data quality considering the needs of the data consumers are presented in [41]. The dimensions are grouped under 9 categories: 1) Accessibility/Delivery, how reachable the data is; 2) Quality of Content, e.g., how clear, appropriate, or relevant; 3) Quality of values, e.g., accurate, complete; 4) Presentation Quality (e.g., format precision); 5) Flexibility, how portable or interpretable; 6) Improvement, e.g., how measurable; 7) Privacy; 8) Commitment, (is help available help or warnings regarding the data usage); and 9) Architecture, which mostly refers to the structural aspects of the data (e.g., logical or physical structure). This work also initiates the discussion of the research belonging to ***Set II***, as it also ***selects a subset of data quality dimensions***, by presenting the following as the most frequently cited by data consumers: availability, security; comprehensiveness, appropriate use, clear definition, source, relevancy, accuracy; ease of interpretation; measurement; early warning, help; library, documentation, naming, unit cost. Identifiability is a critical dimension that is usually forgotten by data customers but still holds a critical importance for them.

An overview of data quality, which considers its multi-dimensional nature is presented in [16]. The authors analyze several works on data quality, match the dimensions found in the literature with the works, and compare the definitions of dimensions used by the different authors. Conclusions include that accuracy, completeness and time-related dimensions (currency, timeliness, and volatility) are applicable to many contexts (i.e., they are more generic) and that additional dimensions can be used to enrich this set.

In [42] the author selected the following five dimensions that are quite popular among researchers and that have particular relevance to data warehouses: correctness, completeness, consistency, currency, and accessibility. The author emphasizes that other dimensions may also be worth considering the context of systems that handle large amounts of data. The data consumer perspective is used in [43] to select the most relevant data quality characteristics for a web portal. PDQM and ISO/IEC 25012 have been used as basis for defining the characteristics and the model defines two points of view: inherent and system dependent; four categories: intrinsic, operational, contextual, and representational. Within each category a set of characteristics and subcharacteristics are also identified.

Authors in [44] survey an ample set of areas in data quality and also analyze work that defines data quality dimensions, despite not being the the main focus of the work. Four dimensions (accuracy, completeness, consistency, and timeliness) are presented as being the most commonly discussed in the literature. On the other hand, in [45] the scope is limited to the social web. The authors have identified 5 categories, which include 42 criteria: 1) Process pragmatics, the degree to which data can be accessed by a user; 2) Information pragmatics, refers to how useful, applicable, and understandable the information is within some user context; 3) Semantics, which indicates how well the data represents the external world; 4) User pragmatics, which refers to credibility and trustworthiness from a user point of view; and 5) Syntactics, which expresses how well the data conforms to other information, such as stored metadata.

Data quality is a vital aspect in genome annotation, which has the key goal of marking the key features of a genome linking them to the related literature. As a complex collaborative task, any error can cause serious problems. The work in [46] creates 5 constructs (accuracy, accessibility, usefulness, relevance, security) that were generated and selected from a set of data quality dimensions. In the context of medical informatics, authors in [47] have analyzed 245 papers in the data quality domain, and in which 13 defined data quality. Most of these papers see this topic as a multidimensional concept and, despite the specific domain of the paper, the authors found out that the most used dimensions are completeness, accuracy, correctness, consistency and timeliness.

In [5] a broad survey on information quality is also presented. The authors discuss the virtues and limitations of the classification presented in ISO/IEC 25012 [8] and also present diverse definitions found in the literature for each of the most frequently mentioned dimensions (accuracy, completeness, consistency, timeliness, and currency).

Research on the quality of **Big Data** appears to be in its initial steps. The role of big data in the modernization of statistical production is discussed in [48], with particular focus in data quality. Three hyper-dimensions are used (source, metadata, data) to group dimensions and sub-dimensions in three different phases (some dimensions apply to more than one phase), which are close to the stages defined by the general statistical business process model [48]. In [49] authors discuss the fact that data quality is more relevant when big datasets, from heterogeneous sources, integrated at different velocities are used. Thus the authors emphasize the importance of the consistency dimension, in three views: Contextual (the extent to which different datasets can be used within the same domain); Temporal (the time slot consistency required for producing, analyzing, and understanding the data); and Operational (the degree to which it is technically possible to analyze a dataset). The authors map the ISO/IEC 25012 [8] characteristics into these three views and into big data's three Vs: Velocity, Volume, and Variety.

Table II summarizes the research work on data quality analyzed in this section. The table is organized in chronological order, and focuses on the categorization structure and terms used in each work to classify data quality, which are referred under the 'Structure' and 'Terms used' columns, respectively.

According to the research analyzed here and summarized in Table II, we find the subset of data dimensions presented in the next paragraph to be the most frequently cited among the analyzed works. The definitions are the ones used in the ISO/IEC 25012 standard [8]; however, accessibility has been complemented with the definition in [45] for clarity, and consistency has been adapted also from [45] for the same reason:

- **Accessibility:** The degree to which data can be accessed in a specific context of use, which includes suitability of representation.
- **Accuracy:** Degree to which data's attributes correctly represent the true value of the intended object.
- **Completeness:** The degree to which an entity has values for all expected attributes and related entity instances.
- **Consistency:** The degree to which an information object is presented in the same format, being compatible with other similar information objects.
- **Currency:** The extent to which data holds attributes of the right age.

The research analyzed in this section shows clear differences in how data quality is perceived. Although the 'fitness for use' concept is widely accepted, the heterogeneity in the structure and naming and definition of dimensions is very clear. This has to do sometimes with the domain of the work (e.g., web, manufacturing), but even when the context is different, there are many similarities and we observe the same dimensions being repeatedly used by different authors. Although there have been a standardization efforts [8], the definitions found in ISO/IEC 25012 are also quite generic, leaving space for the context of use and opening space for further research. Finally, the research in big data appears to be in its initial steps; considering the scale of these systems and the potentially higher impact of poor data issues, further research seems to be of utmost importance.

IV. Mapping Data Quality Problems

Problems regarding data quality, (i.e., dirty data [18]), have been analyzed in previous research and are present in different domains, including healthcare [50], data science [51], or Cyber-Physical Systems [52], just to name a few. In this section, we discuss research that deals with data quality problems, i.e., the specific poor data issues that affect services and organizations, which rely on data for their operations. This section is organized in chronological order, with exception of the seminal work in [19], which initiates the discussion. In Table III, which is presented by the end of this section, we associate the data quality problems identified in the literature with their corresponding dimensions (as identified in Section III).

The literature shows data quality problems from a few different perspectives, with authors many times using some form of hierarchical organization. Some authors classify problems based on their origin (single/multiple) and application level (e.g., instance, schema) [19]; others based on their absence or presence (and, in this latter case, if it is incorrect or correct

TABLE II. DATA QUALITY CATEGORIZATION IN THE LITERATURE.

Research work	Structure	Terms used	
Beyond accuracy: What data quality means to data consumers [15]	4 categories, 15 dimensions	***Intrinsic:*** accuracy, objectivity, believability, reputation	
		Contextual: value-added, relevancy, timeliness, completeness, appropriate amount of data	
		Representational: interpretability, ease of understanding, representational consistency, concise representation	
		Accessible: accessibility, access security	
Anchoring data quality dimensions in ontological foundations [31]	2 views, 2 categories, 26 dimensions	**Internal view**	***Data-related:*** accuracy, reliability, timeliness, completeness, currency, consistency, precision
			System-related: reliability
		External view	***Data-related:*** timeliness, relevance, content, importance, sufficiency, usableness, usefulness, clarity, conciseness, freedom from bias, informativeness, level of detail, quantitativeness, scope, interpretability, understandability
			System-related: timeliness, flexibility, format, efficiency
DoD Guidelines... [26]	6 characteristics	Accuracy, completeness, consistency, timeliness, uniqueness, validity	
Data Quality Handbook.. [42]	5 dimensions	Correctness, completeness, consistency, currency, accessibility	
Data Quality for the Information Age [32]	3 categories, 27 dimensions (15 characteristics)	***Data model:*** content (data *relevance, obtainability of values, clarity of definition*), scope (*comprehensiveness, essentialness*), level of detail (*attribute granularity, domain precision*), composition (*naturalness, identifiability, homogeneity, minimal unnecessary redundancy*), model consistency (*structural, semantic consistency*), reaction to change (flexibility, robustness)	
		Data values: accuracy, completeness, currency, and consistency	
		Data representation: appropriateness, interpretability, portability, format precision, format flexibility, ability to represent null values, efficient usage of recording media, representation consistency	
Improving data warehouse and business information quality [33]	2 categories, 15 dimensions	***Inherent:*** definition conformance, value completeness, validity/business rule conformance, accuracy to surrogate source, accuracy to reality, precision, non-duplication, equivalence/concurrency of redundant or distributed data	
		Pragmatic: accessibility, timeliness, contextual clarity, derivation integrity, usability, rightness or fact completeness	
Assessment methods for information quality criteria [34]	3 classes, 22 criteria	***Subject:*** believability, concise representation, interpretability, relevancy, reputation, understandability, value-added	
		Object: completeness, customer support, documentation, objectivity, price, reliability, security, timeliness, verifiability	
		Process: accuracy, amount of data, availability, consistent representation, latency, response time	
Data Quality: The Guide Field [41]	9 categories, 51 dimensions	***Accessibility/Delivery:*** availability, protocol, security	
		Quality of content: attribute granularity, comprehensiveness, essentialness, flexibility, appropriate use, areas covered, homogeneity, naturalness, obtainability, precision of domains, robustness, semantic consistency, structural consistency, simplicity, clear definition, identifiability, source, relevancy	
		Quality of values: accuracy, completeness, timeliness, consistency	
		Presentation quality: appropriateness, format precision, use of storage	
		Flexibility: portability, representation consistency, null values, formats, language, ease of interpretation	
		Improvement: feedback, measurement, track record	
		Privacy: consumer privacy, privacy of others, security	
		Commitment: warning, help, special requests, commitment	
		Architecture: library/documentation, logical structure, physical structure, naming, rules, redundancy, unit cost	
Enterprise Knowledge Management: The Data Quality Approach [36]	5 categories, 31 characteristics	***Data models:*** clarity, comprehensiveness, flexibility, robustness, essentialness attribute granularity, domain precision, homogeneity, naturalness, identifiability, obtainability, relevance, simplicity, semantic/structural consistency	
		Data values: accuracy, null values, completeness, consistency, currency/timeliness	
		Information domains: enterprise agreement of usage, stewardship, ubiquity	
		Data presentation: appropriateness, correct Interpretation, flexibility, format precision, portability, representation consistency, representation of null values, use of storage	
		Information policy: accessibility, metadata, privacy, redundancy, security, unit cost	
Data quality assessment [12]	16 dimensions	Accessibility, appropriate amount of data, believability, completeness, concise/consistent representation, ease of manipulation, free-of-error, interpretability, objectivity, relevancy, reputation, security, timeliness, understandability, value-added	
A New Method for Database Data Quality Evaluation at the Canadian Institute... [37]	5 categories	Accuracy, timeliness, comparability, usability, relevance	
	24 characteristics	Over-coverage, under-coverage, simple response variance, reliability, correlated response variance, collection and capture, unit and item non-response, edit and imputation, processing, estimation, timeliness, comprehensiveness, integration, standardization, equivalency, linkage-ability, product comparability, historical comparability, accessibility, documentation, interpretability, adaptability, value	
AIMQ: a methodology for information quality assessment [38]	4 quadrants (2 views plus 2 views)	**Product quality**	**Conforms to specifications** (***Sound***): free-of-error, concise representation, completeness, consistent representation
			Meets or exceeds consumer expectations *(Useful):* Appropriate amount, relevancy, understandability, interpretability, objectivity
		Service Quality	**Conforms to specifications *(Dependable):*** Timeliness, security
			Meets or exceeds consumer expectations *(Usable):* Believability, accessibility, ease of operation, reputation
A conceptual framework and belief-function approach to assessing overall information quality [39]	4 attributes, 8 subattributes	***Accessibility***	
		Interpretability: Intelligible, meaningful	
		Relevance: user-specified criteria, timely (considers age and volatility)	
		Credibility: accuracy, completeness, consistency, and non-fictitiousness	
A Framework for Information Quality Assessment [40]	3 categories, 22 dimensions	***Intrinsic:*** Accuracy/validity, cohesiveness, complexity, semantic consistency, structural consistency, currency, informativeness / redundancy, naturalness, precision / completeness	
		Relational or contextual: Accuracy, accessibility, complexity, naturalness, informativeness / redundancy, relevance (aboutness), precision / completeness, security, semantic consistency, structural consistency, verifiability, volatility	
		Reputation: *authority*	
ISO/IEC 25012 [8]	2 points of view and 1 mixed view, 15 characteristics	***Inherent:*** accuracy, completeness, consistency, credibility, currentness; accessibility, compliance, confidentiality, efficiency, precision, traceability, understandability	
		System dependent: availability, portability, recoverability	
		Inherent and system dependent: accessibility, compliance, confidentiality, efficiency, precision, traceability, understandability	
SPDQM: SQuaRE-aligned portal data quality model [43]	2 points of view, 4 categories, 27 characteristics, 15 subcharacteristics	**Inherent**	***Intrinsic:*** accuracy, credibility (objectivity, reputation), traceability, currentness, expiration, completeness, consistency, accessibility (interactive, operation ease, cust. support), compliance, confidentiality, efficiency, precision, understandability
			Operational: availability, accessibility, verifiability, confidentiality, portability, recoverability
		System dependent	***Contextual:*** validity (reliability, scope), value-added (applicability, flexibility, novelty), relevancy (novelty, timeliness), specialization, usefulness, efficiency, effectiveness, traceability, compliance, precision
			Representational: concise representation, consistent representation, understandability, (interpretability, amount of data, documentation, organization), attractiveness, readability
The Practitioner's Guide to Data Quality Improvement [17]	3 categories, 10 Characteristics	***Intrinsic:*** Accuracy, Lineage, Semantic, Structure.	
		Contextual: completeness, consistency, currency, timeliness, reasonableness and identifiability	
		Qualitative dimensions	

(continued on next page)

TABLE II. DATA QUALITY CATEGORIZATION IN THE LITERATURE. (CONTINUED FROM PREVIOUS PAGE)

Research work	Structure	Terms used	
Information Quality Dimensions for the Social Web [45]	5 categories, 42 criteria	***Process pragmatics:*** ease of operation, ease of navigation, interactivity, flexibility of representation, suitability of representation, access security, accessibility, latency, response time, availability	
		Syntactics: consistency, semantic consistency, structural consistency, conformability, naturalness, integrity	
		User pragmatics: believability, verifiability, amount empirical evidence, reliability, reputation, user-conformability, enjoyability	
		Semantics: informativeness, conciseness, accuracy, objectivity, currency, completeness, cohesiveness, degree of context, maintainability, unambiguous	
		Information pragmatics: understandability, interpretability, usability, efficiency, value add, complexity, relevancy, timeliness, volatility	
Prioritization of data quality dimensions and skills requirements in genome annotation work [46]	5 constructs, 19 dimensions	***Accuracy:*** accuracy, unbiased, believability, traceability	
		Accessibility: accessibility, believability, appropriate amount of information	
		Usefulness: interpretability, understandability, ease of manipulation, consistent representation, value added	
		Relevance: relevant, concise representation, up-to-date, reputation, value added	
		Security: security, traceability	
Towards an ontology... [47]	5 dimensions	Completeness, accuracy, correctness, consistency, and timeliness	
A Suggested Framework for the Quality of Big Data [48]	3 hyperdimensions, 11 dimensions, 3	***Source:*** institutional/business environment, privacy and security	
		Metadata: complexity, accessibility clarity, completeness, usability, time factors, coherence(linkability and consistency), validity	
		Data: accuracy (selectivity), coherence (consistency and linkability), validity, time factors	
A Data Quality in Use Model for Big Data [49]	3 by 3 views, 15 characteristics	**Contextual consistency**	***Velocity:*** consistency, credibility, confidentiality ***Volume:*** completeness, credibility ***Variety:*** accuracy, consistency, understandability
		Temporal	***Velocity:*** consistency, credibility, currentness, availability ***Volume:*** availability ***Variety:*** consistency, currentness, compliance
		Operational consistency	***Velocity:*** completeness, accessibility, efficiency, traceability, availability, recoverability ***Volume:*** completeness, accessibility, efficiency, availability, recoverability ***Variety:*** accuracy, accessibility, compliance, efficiency, precision, traceability, availability,

but unusable) [20]; The work in [21] considers instance and relation levels with a distinction between problems related with single or multiple instances/relations; data quality problems are also viewed in terms of syntactics, semantics, and coverage (missing objects) [53]; the work in [54] merges the concepts defined in [19] and [20]; finally, problems are also viewed in respect to their relation to context [28].

The reference work in [19] discusses data cleaning approaches and classifies data quality problems with respect to the source of information: single or multiple. **Single-source** problems are related with the (wrong or absent) definition of integrity constraints. **Multi-source** problems relate with the integration of data from multiple sources, which, for instance, might hold different representations of the same values, or contradictions. Also, in general, all issues that refer to the instance level apply, but with the potential of producing greater impact. Each of these two classes of problems are further divided in [19] into **schema-level**, which are related with defects in the definition of the data model and schema, and **instance-level** which are problems that are not visible at the schema level and cannot be prevented by restrictions at the schema level (or by redesign).

As presented in [19], **single-source instance-level** problems include missing values, abbreviations, or misspellings. A **single-source schema-level** problem can be, for instance, the violation of a unique value, or referential integrity. **multi-source schema-level** problems are many times structural (different representations for the same objects) or naming conflicts (use of synonyms or homonyms). In **Multi-source instance-level** we may find issues related with different representation of values, or interpretation (e.g., different measurement units used), or values referring to different time periods.

A hierarchical taxonomy consisting of 33 primitive dirty data types, primarily distinguished between missing data and non-missing data is proposed in [20]. Further subcategories include whether the data is incorrect or correct but unusable and data problems are further distinguished in terms of whether they could have been prevented by techniques supported by relational databases or not.

Another classification of data quality problems is presented in [21]. In short, the authors organize data quality problems in a hierarchy of four levels: multiple data sources; multiple relations; single relation; and attribute/tuple, which is similar to the organization in [19]. Examples of problems at the level of attribute/tuple include missing values, misspellings, or syntax violations; at the single relation level problems include approximate and inconsistent duplicate tuples; at the multiple relation level problems include referential integrity violation, or incorrect reference; and at the level of multiple data sources we can find heterogeneity of syntaxes, measure units and representation, among others.

In [53] the authors classify data anomalies into syntactical, semantic and coverage. Syntactical anomalies include differences between object structure format, domain format errors, and general irregularities, such as the non-uniform use of values units, and abbreviations. Semantic anomalies include integrity constraint violations, contradictions (e.g., difference between age and birth), duplicates, and invalid tuples, which correspond to the remaining cases. Finally, coverage anomalies include missing values or tuples.

A classification of data quality problems is presented in [54]. The classification holds similarities to the work in [20], however the authors follow the clustering of [19] by dividing data quality problems into schema level (that can be avoided by an improved schema design) and instance level problems (that are not visible or avoided at the schema level). Schema level problems are further divided in: i) supported by relational database management systems (RDBMS), ii) not-supported by RDBMS. Instance level data quality problems are divided in: i) problems concerning single data records, and ii) problems concerning multiple data records.

Data quality problems are analysed as context-independent and also context-dependent in [28]. An extra axis is considered, which presents problems from a user perspective versus an information perspective. Thus, from the information perspective, a context-independent problem would be a spelling error, missing data, or duplicate data; whereas a context-dependent problem could be the violation of business rules or company regulations. From the user perspective, a context independent

problem can be the perception of inaccessible or insecure information; whereas a context dependent problem would be the irrelevance of the information to the work, its low credibility, or inconsistency.

The work in [55] references a set of data quality problems for character data and links the data quality domain to the issues in the robustness testing domain, where some of the issues used to test systems for robustness are similar to some data quality problems (e.g., null values, values out of range). In [56] the authors identify and map data quality problems into a selected set of dimensions, which also includes uniqueness, a dimension not considered for the mapping in this work. Finally, the work in [18] focuses on time-oriented dimensions and problems and summarizes the key research on data quality problems carried out by different authors, detailing the classification used by each author.

The way dirty data is classified varies from author to author, and this includes the approach used for performing classification. Despite this, the research analyzed shares similar findings and discuss common problems, which affect data quality according to the dimensions defined in Section III. In Table III we map each problem to the identified dimensions, using the structure proposed in [19], and thus showing the relationship between data quality problems and dimensions. The problems present in the table have been collected from the works analyzed in this paper, some have been merged (e.g., missing values) in a single line for clarity, other have been omitted as they only differ in the cause of the problem (e.g., integrity problems problems due to the lack of transaction isolation [20]), are too specific or defined in terms of its cause (e.g., mutually inconsistent data due the an action not being triggered [20]), or lack some consensus among researchers (e.g., self-relationship circularity presented only in [21]). Each filled circle indicates that the occurrence of this data quality problem affects the related dimension. Each data quality problem is discussed in the following paragraphs.

TABLE III. DATA QUALITY PROBLEMS MAPPED INTO DIMENSIONS.

Problem types		Data quality problems	Accessibility	Accuracy	Completeness	Consistency	Currency
Source	Level						
Single	Instance	Missing data		•	•		
		Incorrect data		•			
		Misspellings		•			
		Ambiguous data	•	•			
		Extraneous data	•			•	
		Outdated temporal data		•			•
		Misfielded values	•	•	•	•	
		Incorrect references		•			
		Duplicates	•				
	Schema	Domain violation		•			
		Violation of functional dependency		•			
		Wrong data type	•			•	
		Referential integrity violation	•	•	•	•	
		Uniqueness violation		•			
Multiple	Instance	Structural conflicts	•			•	
		Different word orderings	•			•	
		Different aggregation levels	•	•		•	
		Temporal mismatch		•		•	•
		Different units	•			•	
		Different representations	•			•	
	Schema	Use of synonyms	•				
		Use of homonyms	•				
		Use of special characters	•				
		Different encoding formats	•			•	

Considering a **single source** of information, at the **instance level**, *missing data* [20], [54], [53] is a quite common problem, although it can correspond to an empty value, or to a dummy value indicating that there is no data. This issue hinders the completeness dimension, since the data is not present. In addition, accuracy is impaired, because the value is not correct. *Incorrect data* [20] occurs when a value does not conform to the real entity (e.g., an age of 17 instead of 18). The data is wrong and thus not accurate (see Section III). A *misspelling* [19] (e.g., 'Adnrew' instead of 'Andrew') also represents an accuracy problem.

A data problem may occur when the data can be interpreted in more than one way, i.e. the *data is ambiguous* [20]. It can refer to cryptic values (e.g., a 'Level' attribute holding the 'A' value) [19], abbreviations (e.g., J. Locke can be interpreted as John or Joseph Locke) and incomplete context [20] (e.g., Coimbra can stand for the city in Brazil or the city in Portugal). It impairs accuracy, but it also adds as an accessibility problem, as it diminishes the degree to which the data can be accessed (i.e., it has to be interpreted) and it is a representation problem. *Extraneous data* is a data problem where additional data is represented, for instance the use of the title and name in a name field (e.g., John Smith, Director). This problem hinders the access to the required data, because unnecessary data is also present. In addition, it poses a clear format issue, thus affecting the consistency dimension.

Data may be valid for a time point or interval, but may also become obsolete (*outdated temporal data*) [20]. This problem leads to violation of the currency (i.e., the data is not of the right age) and accuracy (i.e., the data is not correct anymore) dimensions. *Misfielded values* [19] occur when the data values are stored in the wrong place (e.g., the first name of a person stored in the last name placeholder). Thus the problem affects accessibility (it is not in the right place), accuracy (the correct place for the data will be empty), completeness as the data is not present where expected, and consistency as the object may not be compatible with others.

Incorrect references occur when, for instance, an employee is associated with the wrong department, which damages accuracy as the data is no longer correct. *Duplicates* [19], [53] occur when the same data appears repeated one or more times (e.g., two entries for the same client) without violating uniqueness (e.g., the identifier in use is not repeated). This makes the access to the data more difficult, as navigation through the data will be required.

Domain violation is a typical **single source, schema level** problem [21], for which similar terms exist in the literature (e.g., illegal values [19], wrong categorical data [20]). *Violation of functional dependency* [53] is a different issue, which occurs when some functional dependency between objects is broken. For instance, *Age* and *Birthdate* are dependent values which can contradict themselves if one is wrong. We find accuracy problems in both cases (domain violation and violation of functional dependency), as the information is not correct.

A *Wrong data type* [20] (also found in the literature associated with syntax violation [21]) is the violation of a data type constraint. An example can be a date represented

as a timestamp (for which we do not have time information). It is an accessibility issue, as it is a representation problem, that makes the data access more difficult. Consistency is also impaired as the format differences can cause incompatibilities with other similar objects.

A *Uniqueness* [21] constraint specifies that a given object property is unique and not null (e.g., a passport number). Thus, when violated, accuracy is impaired, as the data is not correct. *Referential integrity violation* [20] occurs when one entity has no correspondence in another entity (e.g., a bank account which has no associated client), which clearly violates the accuracy dimension, but also impairs accessibility as we do not have access to the correct data; completeness, as data is missing from the storage; and consistency as the missing data may lead to incompatibilities with other objects.

The problems presented with single source origin are intensified when data comes from multiple sources, where the data in the sources may be represented differently, overlap or contradict. This results in a large degree of heterogeneity between data models, schema designs and the actual data [19]. At the **multi-source single-instance** level we can find *Structural conflicts* [19], which refer to different representations of the same object in different sources. This kind of problems have impact in the accessibility of the data, as the representational issue impairs the access to the data, and also in consistency as the different formats cause incompatibility problems.

Different word orderings [20] occur when the expected ordering of data is not met (e.g., name and title, instead of title and name). This data problem violates the consistency and accessibility of data, by breaking the pattern and increasing the effort to access the data. When data is retrieved from different sources and correspond to *different aggregation levels* (i.e., entries per week vs entries per day) also impair accessibility and consistency, but also accuracy as the data will be wrong.

Temporal mismatch [20] occur when the sources of data refer to different points in time. The data is not accurate (it is wrong in terms of time) and obviously damages the currency dimension. The *use of different units* (e.g., feet instead of meters) impairs accessibility as it is a representation issue and also consistency as the different format creates incompatibilities with other objects. The same happens with *different representations* of a value (e.g., gender represented as M/F or Male/Female).

The problems originating from **multiple sources at the schema level** include *homonyms* [21] that occur when the same name is used for different objects. The use of homonyms creates difficulties in accessing the data. The same happens with the use of *synonyms*, where different names are used for the same object, and also with the use of *special characters* (e.g., ą, š, o͡o, ...). *Different encoding formats* [19] also harm the accessibility of data, as the representation may not be suitable, but in this case consistency is impaired as the different format causes incompatibilities between objects.

As we have seen in Section III, the terms and definitions found concerning dirty data are also diverse, but not as heterogeneous as what we observed for the definition of dimensions. We observed a few different perspectives mainly related with the grouping of the problems, and not so much regarding the problems themselves. A different aspect is that the number of works found defining dirty data issues is much smaller, suggesting it might be a more restricted problem. The advent of Big Data opens space for future research on this topic.

V. Conclusion

Data quality management is an important area of research and investment in information technology, supported by the potentially high impact of poor data in organizations. To measure the quality of data, identifying dimensions is an essential step, as each data quality problem potentially affects different dimensions. In this paper we surveyed the research area of data quality classification, with particular emphasis on the definition of data quality dimensions and on the classification of poor data problems (i.e., dirty data). We observed a large diversity of structures and terms used in the literature, sometimes designating the same aspects, and for which this kind of knowledge base is of utmost importance. We identified a frequently cited set of dimensions, gathered data quality problems, and mapped problems to the top cited dimensions. Open research lines include the effective standardization of dimensions, problems and mapping (including their structure and relations) and data quality classification and dirty data definition for Big Data systems, in which research appears to be in its initial steps.

Acknowledgments

This work has been partially supported by the projects CErtification of CRItical Systems (CECRIS), Marie Curie Industry-Academia Partnerships and Pathways (IAPP) number 324334; and DEsign, Verification and VAlidation of large-scale, dynamic Service SystEmS (DEVASSES), Marie Curie International Research Staff Exchange Scheme (IRSES) number 612569, both within the context of the EU Seventh Framework Programme (FP7).

References

[1] "Data Definition in the Cambridge English Dictionary," 2015. [Online]. Available: http://dictionary.cambridge.org/dictionary/english/data

[2] H. Baldwin, "Drilling Into the Value of Data." [Online]. Available: http://www.forbes.com/sites/howardbaldwin/2015/03/23/drilling-into-the-value-of-data/

[3] C. W. Fisher and B. R. Kingma, "Criticality of data quality as exemplified in two disasters," *Information & Management*, vol. 39, no. 2, pp. 109–116, 2001.

[4] M. Ge and M. Helfert, "A review of information quality research—develop a research agenda," in *International Conference on Information Quality 2007*, 2007.

[5] C. Batini, M. Palmonari, and G. Viscusi, "Opening the closed world: A survey of information quality research in the wild," in *The Philosophy of Information Quality*. Springer International Publishing, 2014, pp. 43–73.

[6] L. Sebastian-Coleman, *Measuring data quality for ongoing improvement: a data quality assessment framework*. Newnes, 2012.

[7] D. M. Strong, Y. W. Lee, and R. Y. Wang, "Data quality in context," *Communications of the ACM*, vol. 40, no. 5, pp. 103–110, 1997.

[8] ISO/IEC, "Software engineering – Software product Quality Requirements and Evaluation (SQuaRE) – Data quality model," ISO/IEC, Tech. Rep. ISO/IEC 25012, 2008.

[9] C. Batini and M. Scannapieco, *Data Quality: Concepts, Methodologies and Techniques*, 1st ed. Springer Publishing Company, 2010.

[10] "Swebok v3 guide ieee computer society." [Online]. Available: http://www.computer.org/web/swebok/v3-guide

[11] T. C. Redman, "The impact of poor data quality on the typical enterprise," *Communications of the ACM*, vol. 41, no. 2, pp. 79–82, 1998.

[12] L. L. Pipino, Y. W. Lee, and R. Y. Wang, "Data quality assessment," *Communications of the ACM*, vol. 45, no. 4, pp. 211–218, 2002.

[13] D. Loshin, "Evaluating the business impacts of poor data quality," Knowledge Integrity Incorporated, Business Intelligence Solutions, Tech. Rep., January 2011.

[14] E. D. Quality, "The data quality benchmark report," Experian Data Quality, Tech. Rep., January 2015.

[15] R. Y. Wang and D. M. Strong, "Beyond accuracy: What data quality means to data consumers," *Journal of Management Information Systems*, pp. 5–33, 1996.

[16] M. Scannapieco, P. Missier, and C. Batini, "Data quality at a glance," *Datenbank-Spektrum*, vol. 14, pp. 6–14, 2005.

[17] D. Loshin, *The practitioner's guide to data quality improvement*. Elsevier, 2011.

[18] T. Gschwandtner, J. Gärtner, W. Aigner, and S. Miksch, "A Taxonomy of Dirty Time-Oriented Data," in *Multidisciplinary Research and Practice for Information Systems*, ser. Lecture Notes in Computer Science, G. Quirchmayr, J. Basl, I. You, L. Xu, and E. Weippl, Eds. Springer Berlin Heidelberg, 2012, no. 7465, pp. 58–72.

[19] E. Rahm and H. H. Do, "Data cleaning: Problems and current approaches," *IEEE Data Eng. Bull.*, vol. 23, no. 4, pp. 3–13, 2000.

[20] W. Kim, B.-J. Choi, E.-K. Hong, S.-K. Kim, and D. Lee, "A taxonomy of dirty data," *Data mining and knowledge discovery*, vol. 7, no. 1, pp. 81–99, 2003.

[21] P. Oliveira, F. Rodrigues, and P. R. Henriques, "A formal definition of data quality problems," in *IQ*, 2005.

[22] A. Giannoccaro, G. G. Shanks, and P. Darke, "Stakeholder perceptions of data quality in a data warehouse environment," *Australian Computer Journal*, vol. 31, no. 4, pp. 110–116, 1999.

[23] Gartner, "Magic Quadrant for Data Quality Tools," 2014. [Online]. Available: http://www.gartner.com/technology/reprints.do?id=1-259U63Q&ct=141126&st=sb

[24] R. Karel, "The 'All In' Costs of Poor Data Quality," Jul. 2015. [Online]. Available: http://www.computerworld.com/article/2949323/data-analytics/the-all-in-costs-of-poor-data-quality.html

[25] H. Tibbetts, "Fixing a $3 Trillion Dirty Data Problem with "Crowd Computing" - Integration on the Edge: Data Explosion & Next-Gen Integration." [Online]. Available: http://www.ebizq.net/blogs/integrationedge/2012/01/fixing-a-3-trillion-dirty-data-problem-with-crowd-computing.php

[26] P. Cykana, A. Paul, and M. Stern, "Dod guidelines on data quality management," in *Conference on Information Quality*, 1996, pp. 154–171.

[27] C. Batini, D. Barone, M. Mastrella, A. Maurino, and C. Ruffini, "A framework and a methodology for data quality assessment and monitoring," in *ICIQ*, 2007, pp. 333–346.

[28] M. Ge and M. Helfert, "Cost and Value Management for Data Quality," in *Handbook of Data Quality*, S. Sadiq, Ed. Springer Berlin Heidelberg, 2013, pp. 75–92.

[29] M. Eppler and M. Helfert, "A classification and analysis of data quality costs," in *International Conference on Information Quality*, 2004.

[30] M. Bobrowski, M. Marré, and D. Yankelevich, "Measuring data quality," *Universidad de Buenos Aires. Report*, pp. 99–002, 1999.

[31] Y. Wand and R. Y. Wang, "Anchoring data quality dimensions in ontological foundations," *Communications of the ACM*, vol. 39, no. 11, pp. 86–95, 1996.

[32] T. C. Redman and A. Blanton, *Data Quality for the Information Age*. Artech House, Inc., 1997.

[33] P. E. Larry, "Improving data warehouse and business information quality: methods for reducing costs and increasing profits," 1999.

[34] F. Naumann and C. Rolker, "Assessment methods for information quality criteria," in *Conference on Information Quality*, 2000.

[35] B. K. Kahn, D. M. Strong, and R. Y. Wang, "Information quality benchmarks: Product and service performance," *Communications of the ACM*, vol. 45, no. 4, pp. 184–192, 2002.

[36] D. Loshin, *Enterprise knowledge management: The data quality approach*. Morgan Kaufmann, 2001.

[37] J. Long and C. Seko, "A new method for database data quality evaluation at the canadian institute for health information (cihi)," in *Conference on Information Quality*, 2002, pp. 238–250.

[38] Y. W. Lee, D. M. Strong, B. K. Kahn, and R. Y. Wang, "AIMQ: a methodology for information quality assessment," *Information & management*, vol. 40, no. 2, pp. 133–146, 2002.

[39] M. Bovee, R. P. Srivastava, and B. Mak, "A conceptual framework and belief-function approach to assessing overall information quality," *International Journal of Intelligent Systems*, vol. 18, no. 1, pp. 51–74, Jan. 2003.

[40] B. Stvilia, L. Gasser, M. B. Twidale, and L. C. Smith, "A framework for information quality assessment," *Journal of the American Society for Information Science & Technology*, vol. 58, no. 12, pp. 1720–1733, Oct. 2007.

[41] T. C. Redman, *Data quality: the field guide*. Digital Press, 2001.

[42] E. Gardyn, "A data quality handbook for a data warehouse," in *Conference on Information Quality*, 1997, pp. 267–290.

[43] C. Moraga, M. Moraga, A. Caro, and C. Calero, "Spdqm: Square-aligned portal data quality model," in *Ninth International Conference on Quality Software, QSIC*, 2009, pp. 24–25.

[44] C. Batini, C. Cappiello, C. Francalanci, and A. Maurino, "Methodologies for Data Quality Assessment and Improvement," *ACM Comput. Surv.*, vol. 41, no. 3, pp. 16:1–16:52, Jul. 2009.

[45] M. Schaal, B. Smyth, R. M. Mueller, and R. MacLean, "Information Quality Dimensions for the Social Web," in *Proceedings of the International Conference on Management of Emergent Digital EcoSystems*, ser. MEDES '12. New York, NY, USA: ACM, 2012, pp. 53–58.

[46] H. Huang, B. Stvilia, C. Jörgensen, and H. W. Bass, "Prioritization of data quality dimensions and skills requirements in genome annotation work," *Journal of the American Society for Information Science and Technology*, vol. 63, no. 1, pp. 195–207, Jan. 2012.

[47] S. T. Liaw, A. Rahimi, P. Ray, J. Taggart, S. Dennis, S. de Lusignan, B. Jalaludin, A. E. T. Yeo, and A. Talaei-Khoei, "Towards an ontology for data quality in integrated chronic disease management: A realist review of the literature," *International Journal of Medical Informatics*, vol. 82, no. 1, pp. 10–24, Jan. 2013.

[48] UNECE/HLG, "A suggested Framework for the Quality of Big Data: Deliverables of the UNECE Big Data Quality Task Team," UNECE/HLG, Project Deliverable Big Data Quality Framework, 2014.

[49] I. Caballero, M. Serrano, and M. Piattini, "A Data Quality in Use Model for Big Data," in *Advances in Conceptual Modeling*, ser. Lecture Notes in Computer Science, M. Indulska and S. Purao, Eds. Springer International Publishing, Oct. 2014, no. 8823, pp. 65–74.

[50] R. S. Mans, W. M. P. v. d. Aalst, and R. J. B. Vanwersch, "Data Quality Issues," in *Process Mining in Healthcare*, ser. SpringerBriefs in Business Process Management. Springer International Publishing, 2015, pp. 79–88.

[51] B. T. Hazen, C. A. Boone, J. D. Ezell, and L. A. Jones-Farmer, "Data quality for data science, predictive analytics, and big data in supply chain management: An introduction to the problem and suggestions for research and applications," *International Journal of Production Economics*, vol. 154, pp. 72–80, Aug. 2014.

[52] K. Sha and S. Zeadally, "Data Quality Challenges in Cyber-Physical Systems," *J. Data and Information Quality*, vol. 6, no. 2-3, pp. 8:1–8:4, Jun. 2015.

[53] H. Müller and J.-C. Freytag, *Problems, methods, and challenges in comprehensive data cleansing*. Humboldt-Universitat, Berlin, Germany, 2005.

[54] J. Barateiro and H. Galhardas, "A survey of data quality tools," *Datenbank-Spektrum*, vol. 14, no. 15-21, p. 48, 2005.

[55] N. Ivaki, N. Laranjeiro, and M. Vieira, "Towards Evaluating the Impact of Data Quality on Service Applications," in *Workshop on Reliability and Security Data Analysis (RSDA 2013) co-located with The 43rd Annual IEEE/IFIP International Conference on Dependable Systems and Networks (DSN 2013)*. Budapest, Hungary: IEEE Computer Society, Jun. 2013.

[56] L. Li, T. Peng, and J. Kennedy, "A rule based taxonomy of dirty data," *GSTF International Journal on Computing*, vol. 1, no. 2, 2011.

D Conference paper

Below is provided the conference paper that was submitted and accepted in *Latin-American Symposium on Dependable Computing* (LADC 2016).

Testing Web Applications Using Poor Quality Data

Practical Experience Report

Nuno Laranjeiro*, Seyma Nur Soydemir*, Jorge Bernardino*†
*CISUC, Department of Informatics Engineering
University of Coimbra, Portugal
cnl@dei.uc.pt, seyma@student.dei.uc.pt
†Polytechnic of Coimbra, Portugal
jorge@isec.pt

***Abstract*—Web applications are nowadays being used to support enterprise-level business operations and usually rely on back-end databases to deliver service to clients. Research and industry reports indicate the huge impact the quality of the data can have on businesses, especially when applications are not prepared for handling low quality data. In fact, even in widely tested and used applications, the presence of poor data can sometimes result in severe failures and bring in disastrous consequences for clients and providers, including financial or reputation losses. In this paper, we present an approach based on the runtime injection of poor quality data on the database interface used by web applications, which allows understanding how vulnerable the application is to the presence of poor quality data. Results indicate that the approach can be easily used to disclose critical problems in web applications and supporting middleware, helping developers in building more reliable services.**

Keywords—testing; web applications; poor quality data; data quality problems; object-relational mapping; JDBC driver.

I. INTRODUCTION

Web applications are nowadays being used as the interface of many businesses to the outside world, providing service that is frequently supported by back-end databases. In this type of environments, a failure in the web application can impair the whole business potentially bringing in huge losses for providers. These losses can simply refer to lost business transactions, but can also refer to other types of financial losses (e.g., time to repair systems, human resources assisting system recovery), including reputation losses [1].

The current trend of fast-paced development of software leads developers to focus on functionality, and non-functional aspects, such as application robustness, are often disregarded [2]. Industry reports show that the way applications handle data is an aspect that is many times disregarded, i.e., developers many times assume that the data being handled by the application is correct, which is not always the case. In fact, severe system failures and/or huge financial losses caused by the presence of poor quality data have been widely reported in the industry [1]. This kind of problem is relatively well-known in the robustness testing domain, where tests using invalid inputs applied on public interfaces of many different systems have been used with great success [2]–[4]. However, there is still some missing link between the data quality community and the software engineering research and practitioner communities. This missing link prevents developers from creating applications that are prepared to resist to the presence of poor data. Although there are numerous tools for data cleaning and data quality assessment, to the best of our knowledge, there is currently no practical approach or tool that developers can use to test the behavior of applications in the presence of poor quality data.

In this paper, we propose an easy-to-use and low-intrusiveness approach that is based on the injection of poor quality data at the application-storage interface. In our approach we inject data on returning result sets from the database and the injected data is based on typical data quality problems, which were identified based on a large survey of the state of the art in dirty data [5]. In short, we replace valid data coming from the database with poor data, that should be properly handled by the application, and observe the application behavior. As we will see, tests can be performed automatically and, if needed, can also be tuned to fit the specificities of the application being tested. For the time being, the analysis of the tests results is carried out in a manual way.

The approach was used to test an open-source and widely used application for commerce and business, which is used worldwide and which we designate by ERPx. We were able to disclose several bugs not only in the code being tested; but also issues in the Object Relational Mapping (ORM) framework used by ERPx (a mainstream middleware framework used by applications to satisfy persistency requirements); and also in the JDBC driver used by the application. Results show that our approach and tool can be used to not only disclose problems related with poor data, at different software levels, but also as an informative means to correct and build more reliable software. The main contributions of this paper are the following:

- A test-based approach for understanding the behavior of web applications in presence of poor quality data.
- A free and open-source tool, named Poor Data Injector [6], that implements the approach and can be easily used by developers and providers to test their applications.

The outline of this paper is as follows. The next section presents the related work on poor quality data and testing. Section III presents the approach for testing web applications in the presence of poor data. Section IV presents a case study carried out using our tool and discusses the results. Section V concludes this paper.

II. Background and Related Work

Data quality, sometimes referred to as information quality [7], has been defined in diverse ways in the literature [8]. The ISO/IEC 25012 standard defines it as "the degree to which a set of characteristics of data fulfills requirements"[9]. Examples of such characteristics include completeness, accuracy, or consistency [5], whereas the requirements express the needs and constraints that contribute to the solution of some problem [10].

Industry reports and previous research indicate the severe damage caused by the presence of poor quality data in diverse contexts [7], [11]–[13], with the Gartner Group reporting bad data as the main cause of failure in CRM systems [14]. Data volume growth can also intensify its management complexity, which can in turn result in a higher probability of generating poor data. The fast-changing dynamics of the Web environment can easily lead to the degeneration of customer data (for instance due to the introduction of new software components holding bugs) and this has actually been observed in the field [14].

Although the analysis and improvement of data quality has gathered plenty of attention (e.g., to perform data cleaning operations) from researchers and practitioners [12], [15]–[19], and despite the well-known impact of poor data in business critical systems [20], understanding how well an application is prepared to handle the inevitable appearance of poor data has been largely overlooked. For this purpose, the identification of representative data quality problems and how they should be integrated in test cases is vital. In [5] we researched the state of the art in data quality classification and data quality problems precisely to support the creation of such test cases. Regarding the definition of the test approach, the closest studies for understanding application behavior in presence of incorrect inputs, although from a different perspective, come from the robustness testing area [2]–[4].

Robustness testing allows to understand the behavior of a system in the presence of invalid input or stressful conditions [3]. The goal of these tests is to stimulate the system being tested so that possible existing internal errors are triggered and this allows developers to solve or wrap the identified problems. This technique can be used to differentiate systems according to the number and severity of the problems uncovered and has been mostly applied to the public interface of several systems, from a black-box perspective [3], [4]. The interaction points between different independent systems has rarely been used in robustness testing research, and to the best of our knowledge typical poor data quality issues have not yet been considered.

Ballista [3] is a tool for testing robustness that combines acceptable and exceptional values on calls to kernel functions of operating systems. The values used in each call are randomly extracted from a specific set of predefined tests, that apply to the particular data type involved in the call. The results of the tests are used to classify each operating system in terms of its robustness, according to a predefined scale (the CRASH scale) that distinguishes several failure modes. MAFALDA [4] is another robustness testing tool that allows characterizing the behavior of microkernels in the presence of faults. Closer to the Web domain, in previous work we defined an approach to assess the behavior of web services in the presence of tampered SOAP messages [2]. It consists of a set of robustness tests based on invalid web services call parameters. The services are classified according to the failures observed during the execution of the tests and using an adaptation of the CRASH scale [3].

The impact of invalid data on the reliability of web services has been the object of research in [21]. The approach includes building an architecture view of the system being tested; measuring the data quality or validity with a tool; measuring the reliability of the data and software components; creating a state machine using the system architecture as basis; and computing the overall system reliability. The invalid types used in the study are limited to seven already present issues. There is no use of issues that might affect the system in the future, so the approach is limited to reliability estimation based on identified and already present issues. In our own previous work [22], we defined a preliminary view for a testing approach using dirty data. In this paper we define the complete approach using a comprehensive set of data quality problems [5] and now demonstrate the utility of the approach in a real-world scenario.

III. Testing Applications Using Poor Data Injection

In this section we explain our approach to test applications in the presence of dirty data. We first overview the core mechanism involved in our approach, then explain the main phases of the approach, and finally describe in detail the components used to implement the approach.

A. Injecting poor data

Our approach is based on the presence of an instrumented data access driver (e.g., a JDBC driver) that we place between the application, which we generally designate as a service application, and the data storage (e.g., a database management system), which has the exact same interface of a regular data access driver. We emphasize that no changes to the service code, database management system, or database are required. The core concept is that the driver is able to intercept all calls to the database management system and it simulates the presence of poor quality data, by replacing the original data coming from the database, with poor data. A simplified view of this scenario is described in Figure 1.

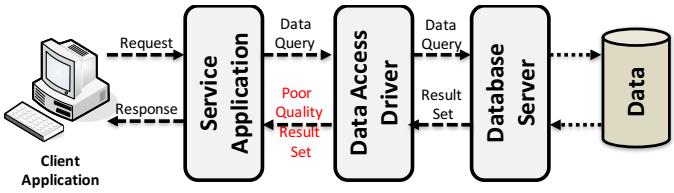

Fig. 1. The general setup required for testing using poor quality data.

Using the scenario depicted in Figure 1 as basis, the goal is to understand if the application code can handle the poor data coming from the database in a robust way, or if, on the other hand, the service is poorly built and cannot tolerate the presence of such data (e.g., it becomes unavailable or throws unexpected exceptions when handling the data). This core mechanism is used in a set of distinct phases in our approach, explained in the next section. Despite the multiple phases, using the mechanism is very easy, essentially requiring a simple replacement of the data access driver with our instrumented version.

Note that any application should be able to handle poor data and this is especially true for applications that are deployed in business-critical environments. The presence of poor data in a storage system is known to increase with the ageing of applications, and in the currently dynamic Web environment, where applications change very often, it is likely that residual bugs, user misuse, manipulation of data by other services, or even malicious accesses to data cause the appearance of such problems in the storage.

In practice, and in the case of our prototype testing tool, the driver involved is a regular JDBC driver, which is instrumented to perform all the necessary steps. Thus, we do not modify the driver code directly, we rely on the external API and on code instrumentation packages (i.e., AspectJ) to intercept calls to well-known methods that are used by the application to access the data [23], [24]. By purely relying on the API to perform the Aspect-Oriented instrumentation, we can apply our approach to any JDBC driver. An important aspect is that this specific setup is an application of the overall approach, which is generic as the concepts involved are present in other mainstream languages. Although we have used a Java-based scenario, a similar scenario could be used, for instance, with Python, or C# .NET.

B. Approach execution phases

Our approach involves the sequential execution of the following phases, which are explained in the next paragraphs:

1) **Warm-up:** Valid client requests are issued to the web application and the goal is simply to warm-up the application and infrastructure to resemble typical operational conditions;
2) **Injection:** Valid requests are sent to the web application and poor data inputs are injected (i.e., data mutations are applied) during operation execution, whenever a data access point is carried out by the application;
3) **Analysis:** The behavior of the service is analyzed (e.g., by examining the responses).

During the first two phases, we assume the presence of some workload generation client that places valid requests on the system. These requests are then used as basis to perform different functions according to the phase being executed. All requests sent to the service and their responses should be logged. The same happens with the operation of the data access driver, which should also log any data mutation applied (for debugging purposes). The intention is that, upon service failure, the user can understand which sequence of requests (and mutations) caused the failure and, if there is the option to restore the system state, replay requests and compare responses. Although what is really important is that the developer is able to understand the cause of the failure, this kind of option is advantageous for debugging activities, where a fix must be introduced and tests must be rerun, to verify the correct behavior of the system.

During the **warm-up phase** the instrumented data access driver performs like a regular driver. The instrumented data access driver intercepts all data access calls, but does not inject any mutated data during this phase, the goal is simply to let the system warm-up, to better resemble typical working conditions.

The **injection phase** is central to the approach. During this phase we replace genuine data coming from the database with data that, for the particular data type and value being accessed, represents a poor data quality problem. In order to define which types of problems should be included in our testing approach, in previous work we surveyed the state of the art in data quality classification and identified representative data quality problems (e.g., misspellings, abbreviations, imprecisions, extraneous data) associated with common data types (e.g., text, numbers, dates) [5]. The goal was precisely to support the idea brought in this paper: that we can design and use testing with poor quality data to effectively disclose software bugs, or at least bad programming practices. Thus, in the present paper we use such data quality problems, in particular the ones applicable to single values, to build a fault model that is used during testing. Specific combinations of multiple values to represent more complex issues is left for future work and out of the scope of this paper. As the total number of data quality problems is quite large, we made the model of the identified problems available at [6] and present a subset in Table I, for clarity.

The injection of mutated data can be done once per service operation execution (i.e., per each client call), as the goal is to understand the impact of the faulty data in the execution of that particular operation. However, there is also the option to inject any given number of faults during the execution of a service operation (which we have followed in our experiments). Even though this latter option may create difficulties in understanding the exact causes of failures (as multiple faults are involved), it is often the typical choice in the robustness testing domain due to its simplicity and ability to disclose problems. In fact, we do not consider any particular state of the application (either than the one led to by the user) and in this sense the tests resemble traditional robustness tests.

In general, all public operations should also be tested, but this depends on the goals of the tester (which may be simply interested in testing a few cases). For each operation to be tested, each of the data access points present in the code should also be tested in this phase. This naturally depends on the client workload providing enough coverage and dealing with aspect is something out of the scope of this paper, which we intend to target in future work. In some applications, data access points may be shared by different operations. Even in these cases, it is

TABLE I. PARTIAL EXAMPLE OF POOR DATA QUALITY FAULTS

Data type	Issue description
String	Replace by null
	Replace by empty
	Replace a word by a misspelled word (Dictionary-based) or, if no match, use a random single edit operation (insertion; deletion; substitution of a single character; or transposition of two adjacent characters) over a randomly selected word
	Add whitespace in a leading or trailing position, or between words (random choice)
	Add extraneous data in leading, trailing, or random position (random choice)
	…
Integer	Add one numeric character
	Set to zero
	Remove one random numeric character
	Flip sign
	…
…	…

desirable to exercise the different public operations, as we are passing in different areas of the code and might disclose different problems if bugs are present. Each data access point should be tested with all predefined poor data faults. The desirable execution profile of the injection phase, which we just described, is represented in Figure 2.

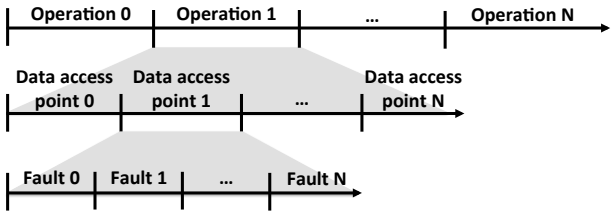

Fig. 2. Basic execution profile of the tests.

The **injection** phase could be automatically configured to stop when a given percentage of data access points has been covered by the tests (provided that such information is collected during the warm-up phase). In the case of this preliminary work, we manually determine that a test should stop when the user action as concluded (with a response being delivered to the user) or when a failure is detected.

The last phase is the **analysis** of the results of the tests, which involves classifying any observed failures (e.g., using a well-known failure mode scale, such as the CRASH scale) and also the number, location, and origin of the problems disclosed during the tests. This latter step requires access to the source code to understand exactly if the output shown by the tests is a software bug, what is the exact location in the code (as a mutated value injected at a specific point in the code may only be improperly used later in the code) and why it is a problem (so that it can be fixed). If the tester wants to classify the service behavior it is possible to use the CRASH scale, which classifies the severity of the failures observed in five levels, Catastrophic, Restart, Abort, Silent, or Hindering [3]. This step is not mandatory, and is only necessary if there is, for instance, the need for comparing systems or prioritize bug fixing.

C. *Components and Setup*

Figure 3 presents the key components involved in our approach and their interactions in a services environment. The elements involved in a typical services interaction are represented in gray boxes, in light brown we have the components that respect to our approach. All components in solid lines are mandatory, whereas the dashed components are optional (their use depends on the application being tested and on the tester requirements). As mentioned, in the case of our prototype, applying and deploying our mechanism requires no change to existing source code. The following paragraphs explain the components and their main functions.

The component that plays a key role in implementing our approach is named **AOP Wrapper** as it is essentially a data access driver (in our case, a JDBC driver) that has been wrapped to include our injection logic. Thus, the driver byte code is instrumented to be able to inject poor data inputs in returning calls to the storage. This procedure simply involves replacing the original value retrieved from the database by a poor data value, retrieved from our list of data quality problems (please refer to Table I).

The **HTTP Filter** is a component that is executed at two moments: 1) before processing each client request; 2) after the client has been processed (and immediately before the response is delivered to the client). Thus, in the case of a typical web application, this can simply be an HTTP Filter, which the major web serving platforms allow configuring (any Java EE platform allows), or any other component that is executed the two moments referred (e.g. an HTTP proxy). In the case of our prototype we used an HTTP Filter for this purpose.

The HTTP Filter also allows a fine-grain control over the tests. In particular, it marks the beginning and end of a client request (which in general maps to the execution of a particular user operation) and this allows us to understand that a particular data access point is being accessed as part of a given user request. It also allows the driver to understand if it has already mutated a value for this client request or not. Thus, to allow easier debugging we may execute a single poor data injection per client request, even if that request involves multiple data accesses. Overtime, we will eventually cover all data accesses, as long as a workload with proper coverage is provided. As a final note, this filter allows logging server responses in a centralized manner, so that we can later analyze the behavior of the application being tested in an easy manner.

In the case of our prototype [6], the server-side components are all assembled in a single unit (i.e., one jar file) that implements our approach, which will replace the original data access driver (e.g., a JDBC driver). This jar file includes the filter, which can also be used to initialize and load any specific configuration for the tests. To use our testing approach we simply need to change the original data access driver with our own and it will be ready to perform blind injection of poor data for any java web application that uses PostgreSQL as data access

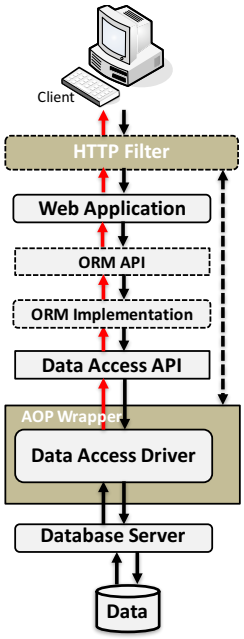

Fig. 3. Detailed view of the components involved in the tests.

driver. Changing our tool to use another driver (e.g., MariaDB, Oracle) is simply a matter of writing the names of the required dependencies in our maven project descriptor file and recompiling and packaging using *mvn package*, no further implementation or configuration is needed.

IV. Case Study

In this section we describe a case study carried out to illustrate the applicability of our approach. We explain the setup used, the tests executed, and discuss the results.

A. Experimental Setup

We selected a well-known and widely used commercial open source ERP business solution for enterprises. It allows companies to manage their entire business, and supports typical processes such as sales, manufacturing, or finance, just to name a few. As we cannot disclose the name of the tool, we designate it ERPx. ERPx requires a database which we chose to be PostgreSQL 9.3, and a server for deployment for which we chose the well-known Apache Tomcat 7.0.68. As we intended to repeat the tests at least once, besides a regular browser, we recorded and later replayed user actions on the browser (when operating ERPx) using SikuliX 1.1.0.

B. Selecting and Executing the Test Cases

ERPx is an application of huge dimensions and due to this, we selected a few test cases for testing. We considered the CRUD model [14] for performing this selection, so that we could have operations with different profiles: CREATE, READ, UPDATE and DELETE. Note that all test cases selected are quite complex and also perform read operations, but we classified them according to their main purpose. Thus, the test cases are mostly composed of read operations. The goal was to obtain a good mix between test cases that potentially have different data access patterns or are built in a different way. In practice, any test case will do as long as it accesses the database. Table II presents the operations selected for testing, their mapping to the CRUD model, and a reference to the failures uncovered in each operation during testing. These failures are discussed in the next section.

TABLE II. Results Overview

Operation		
Name	***Type (CRUD)***	***Failure Reference***
Login	R	A, B, C, D
Create Organization	C	A, C, D
Create a new User	C	A, B, C, D
Create a new Role	C	A, B, C, D
Create Product	C	A, B, C, D, E
Delete Product	D	A, B, C, D
Update Product	U	A, B, C, D
Export Product Categories	R	A, B, D

C. Results and Discussion

As shown in Table II, during the experiments we were able to uncover failures in all operations tested. As discussed in the next paragraphs, the disclosed issues were actually found at three main locations of the system being tested: i) the application itself; ii) in the very popular Object-Relational Mapping framework used by the system; and iii) in the also widely used PostgreSQL driver code.

Table III presents an excerpt of the issues disclosed during the tests. We were able to disclose further issues, but due to length restrictions in this paper we opted discuss only a set of problems that manifested in different forms and at different structural locations of the system (as noticed in Table III). Note also that all of these examples are problematic, even those where no message was shown to the user, as the tests eventually led the application to become unusable.

Failure A mostly occurred whenever the data involved was set to null, however, in the case of the example, a mutated variable value (variable *referenceID*) causes an access to the database to return null, and this null value is then used without being checked. This results in a *NullPointerException* being thrown (a check would need to be made to prevent the exception). In the end, this exception triggers a *TemplateModelException* that eventually is shown to the user in an alert box.

ERPx loads several classes dynamically, and *Failure B* occurs when one of those names is wrong (due to the application of a mutation). If the goal is to dynamically load the classes there are not many alternatives, as the names must reside outside the code. On the other hand, if these names are not likely to change, they can also be kept in compiled code. Although we do not have enough information to specify what should be the right design choice, disclosing this issue can help developers understanding if this is actually the right design decision or not and how the application handles this type of situation. Anyway, the user should be informed in case of error, especially if it is a problem that renders the application unusable.

Failure C is a critical case. It actually represents a second order SQL Injection problem, where malicious data in the database is unsafely used to build an SQL query. An attacker might be able to obtain sensitive information, as the information obtained from the database is currently not sanitized by the application. This shows that this testing technique, besides pointing out potential design or implementation flaws, also has potential to disclose security problems. In addition, and although the error messaging system of the application was correctly triggered, the actual error message shown to the user discloses the contents of an entire database table row, which should obviously not happen.

Failure D is a very interesting case, where the technique served to disclose a fragility in the implementation of the quite popular Object-Relational Mapping framework used by the application. In this case, the framework tries to access the first character of a string and fails as the string had become empty due to the mutation applied. Although the framework previously checks if the string is null, it does not check if it is empty and immediately accesses its first character. It then fails with a *StringIndexOutOfBoundsException*, this is an implementation flaw, very similar to the one described next (which has already received attention and a correction from the developer community).

Failure E occurs when adding characters that include a single quote to a string. This is a reported bug [25] in the driver being used in the experiments (PostgreSQL JDBC Driver 9.4-1201) and has been corrected in version 9.4-1204. Basically, the code fails to find the closing single quote and returns the position

TABLE III. SELECTED CASES FROM THE EXPERIMENTS

Ref	Root Exception Triggered	Location	Last Mutation	External behavior
A	NullPointerException	Application	changeToOppositeCase	*TemplateModelException* reported to the user.
B	ClassNotFoundException	Application	addExtraneous	No message displayed to the user.
C	PSQLException	Application	replaceBySQLString	Application error message disclosing table row contents
D	StringIndexOutOfBoundsException	JPA Middleware	replaceByEmptyString	Application error message stating *String index out of range*
E	ArrayIndexOutOfBoundsException	JDBC Driver	addCharactersToString	No message displayed to the user.

of the last character in the query as the end of the string. The problem is that in another part of the driver, the code does not expect this behavior, and the result is an attempt to access a position that is one place after the end.

During the tests we were expecting to find a few application-level problems, but it was interesting to see that this type of testing was able to actually find problems at the middleware level (in our case at two levels – ORM framework and JDBC driver). The fact that the middleware is widely used and tested emphasizes the importance of performing this kind of tests to disclose issues that might affect applications experiencing unexpected conditions. Furthermore, the ability to find problems beyond 'simple' exceptions (or inadequate messages presented to the user) and that represent security issues further emphasizes that this type of testing has the potential to produce results that are of high value for application architects and developers.

V. CONCLUSION

This paper presented a testing approach, based on the injection of poor quality data at the application–storage interface. Data quality problems are injected on returning result sets from the database and the application behavior is observed. The tests disclosed several different failures, including bugs at the application, JPA implementation, and also the JDBC driver used. In future work we intend to perform a larger scale evaluation, further explore different configurations of this approach, and explore the state of the system being tested.

ACKNOWLEDGMENTS

This work has been partially supported by the project DEsign, Verification and VAlidation of large-scale, dynamic Service SystEmS (DEVASSES), Marie Curie International Research Staff Exchange Scheme (IRSES) number 612569, within the context of the EU Seventh Framework Programme (FP7).

REFERENCES

[1] D. Loshin, "Evaluating Business Impacts of Poor Data Quality," *Information Quality Journal*, 2011.

[2] N. Laranjeiro, M. Vieira, and H. Madeira, "A Robustness Testing Approach for SOAP Web Services," *JISA*, vol. 3, no. 2, pp. 215–232, Sep. 2012.

[3] P. Koopman and J. DeVale, "Comparing the robustness of POSIX operating systems," in *Twenty-Ninth Annual International Symposium on Fault-Tolerant Computing*, 1999, pp. 30–37.

[4] M. Rodríguez, F. Salles, J.-C. Fabre, and J. Arlat, "MAFALDA: Microkernel Assessment by Fault Injection and Design Aid," in *The Third European Dependable Computing Conference on Dependable Computing*, 1999, pp. 143–160.

[5] N. Laranjeiro, S. Nur Soydemir, and J. Bernardino, "A Survey on Data Quality: Classifying Poor Data," in *The 21st IEEE Pacific Rim International Symposium on Dependable Computing (PRDC 2015)*, Zhangjiajie, China, 2015.

[6] N. Laranjeiro, N. S. Seyma, and B. Jorge, "Poor Data Injector Toolset," 2016. [Online]. Available: http://eden.dei.uc.pt/~cnl/papers/2016-ladc-zip. [Accessed: 25-May-2016].

[7] M. Ge and M. Helfert, "A Review of Information Quality Research - Develop a Research Agenda.," in *12th International Conference on Information Quality*, Cambridge, MA, USA, 2007, pp. 76–91.

[8] C. Batini, M. Palmonari, and G. Viscusi, "Opening the Closed World: A Survey of Information Quality Research in the Wild," in *The Philosophy of Information Quality*, Springer International Publishing, 2014, pp. 43–73.

[9] ISO/IEC, "Software engineering – Software product Quality Requirements and Evaluation (SQuaRE) – Data quality model," ISO/IEC, ISO/IEC 25012, 2008.

[10] *SWEBOK V3 Guide IEEE Computer Society*. .

[11] L. L. Pipino, Y. W. Lee, and R. Y. Wang, "Data quality assessment," *Communications of the ACM*, vol. 45, no. 4, pp. 211–218, 2002.

[12] D. Loshin, *The practitioner's guide to data quality improvement*. Morgan Kaufmann, 2010.

[13] E. D. Quality, "The data quality benchmark report," Experian Data Quality, Jan. 2015.

[14] R. Marsh, "Drowning in dirty data? It's time to sink or swim: A four-stage methodology for total data quality management," *Journal of Database Marketing & Customer Strategy Management*, vol. 12, no. 2, pp. 105–112, Jan. 2005.

[15] A. Caro, C. Calero, E. Mendes, and M. Piattini, "A Probabilistic Approach to Web Portal's Data Quality Evaluation," in *Quality of Information and Communications Technology, 2007. QUATIC 2007. 6th International Conference on the*, 2007, pp. 143 –153.

[16] B. Xiaojuan, N. Shurong, X. Zhaolin, and C. Peng, "Novel method for the evaluation of data quality based on fuzzy control," *Journal of Systems Engineering and Electronics*, vol. 19, no. 3, pp. 606 –610, Jun. 2008.

[17] M. Bergdahl, M. Ehling, E. Elvers, E. Földesi, T. Körner, A. Kron, P. Lohauß, K. Mag, V. Morais, and A. Nimmergut, *Handbook on Data Quality Assessment Methods and Tools*. Wiesbaden, 2007.

[18] O.-H. Choi, J.-E. Lim, H.-S. Na, K.-J. Seong, and D.-K. Baik, "An Efficient Method of Data Quality Evaluation Using Metadata Registry," in *Advanced Software Engineering and Its Applications, 2008. ASEA 2008*, 2008, pp. 9 –12.

[19] H. Galhardas, D. Florescu, and D. Shasha, "Declarative Data Cleaning: Language, Model, and Algorithms," in *In VLDB*, 2001, pp. 371–380.

[20] A. Haug, F. Zachariassen, and D. van Liempd, "The costs of poor data quality," *Journal of Industrial Engineering and Management*, vol. 4, no. 2, Jul. 2011.

[21] E. Musial and M.-H. Chen, "Effect of Data Validity on the Reliability of Data-centric Web Services," 2012, pp. 576–583.

[22] N. Ivaki, N. Laranjeiro, and M. Vieira, "Towards Evaluating the Impact of Data Quality on Service Applications," in *Workshop on Reliability and Security Data Analysis (RSDA 2013) co-located with The 43rd Annual IEEE/IFIP International Conference on Dependable Systems and Networks (DSN 2013)*, Budapest, Hungary, 2013.

[23] Eclipse Foundation, "The AspectJ Project," 2006. [Online]. Available: http://www.eclipse.org/aspectj/. [Accessed: 08-Apr-2016].

[24] G. Kiczales, J. Lamping, A. Mendhekar, C. Maeda, C. Lopes, J. M. Loingtier, and J. Irwin, "Aspect-Oriented Programming," in *11th European Conference on Object-oriented Programming*, Jyväskylä, Finland, 1997.

[25] "PostgreSQL JDBC Driver – GitHub," 17-Sep-2015. [Online]. Available: https://github.com/pgjdbc/pgjdbc/issues/369.