Laranjeiro et al. Journal of the Brazilian Computer Society (2017) 23:14
DOI 10.1186/s13173-017-0063-x

Journal of the
Brazilian Computer Society

RESEARCH Open Access

Testing data-centric services using poor quality data: from relational to NoSQL document databases

Nuno Laranjeiro[1*], Seyma Nur Soydemir[1], Naghmeh Ivaki[1] and Jorge Bernardino[1,2]

Abstract

Businesses are nowadays deploying their services online, reaching out to clients all around the world. Many times deployed as web applications or web services, these business-critical systems typically perform large amounts of database operations; thus, they are dependent on the quality of the data to provide correct service to clients. Research and practice have shown that the quality of the data in an enterprise system gradually decreases overtime, bringing in diverse reliability issues to the applications that are using the data to provide services. These issues range from simple incorrect operations to aborted operations or severe system failures. In this paper, we present an approach to test data-centric services in presence of poor quality data. The approach has been designed to consider relational and NoSQL database nodes used by the system under test and is based on the injection of poor quality data on the database–application interface. The results indicate the effectiveness of the approach in discovering issues, not only at the application-level, but also in the middleware being used, contributing to the development of more reliable services.

Keywords: Testing, Web applications, Poor quality data, Object-relational mapping, JDBC drivers, NoSQL drivers, NoSQL databases, Relational databases

Introduction

Web applications and services are nowadays being used as the interface of many businesses to the outside world. Independent of the type of interface provided to clients (e.g., HTML, REST, MOM), these business-critical systems are usually data-centric and rely on the access to a back-end database, which is responsible for handling the persistence requirements of the applications. This database is often a relational database, but recently, providers are increasingly using NoSQL databases, which are becoming quite popular, mostly due to their scalability properties [1]. Independent from the type of database (relational or NoSQL), these business-critical data-centric systems strongly depend on the quality of data to provide correct and reliable services [2]. In this kind of environment, a service failure can result in huge losses to the service provider, ranging from lost business transactions to customer dissatisfaction and irrecoverable reputation losses [3].

Research and industry have previously reported the severe harm caused by the presence of poor quality data in many contexts [4–7], with the Gartner Group highlighting bad data as the main cause of failure in CRM systems [8]. One of the common problems behind failures reported in the literature is that developers, many times, assume that the data being handled by the application is correct, which is not always the case. In fact, research has shown that the quality of the data in an information system tends to decrease with time, so it is somehow expectable that systems will eventually have to handle poor quality data. Yet, developers are, in general, little aware of this kind of problem.

Poor quality data enters systems in different ways, not only due to human error or lack of proper data validation mechanisms, but also due to the presence of residual bugs in the code [9]. The way systems are nowadays being developed, with frequent changes and hard time constraints, leads developers to focus on the

* Correspondence: cnl@dei.uc.pt
[1]CISUC, Department of Informatics Engineering, University of Coimbra, Coimbra, Portugal
Full list of author information is available at the end of the article

functional properties of systems, disregarding important non-functional properties such as robustness or security [10]. With little time to apply runtime verification techniques, developers end up by deploying systems that comply with basic functionality but that may fail in presence of unexpected conditions. A large part of the problem is that the most popular runtime testing techniques tend to test systems using their external interfaces, disregarding both the internal interfaces and the quality of the data being used [11, 12].

The ability to gracefully handle erroneous inputs is a well-known problem in the robustness testing domain, where tests using invalid inputs applied on external interfaces of many different systems have been used with great success [10–12]. In the case of the data quality community, the problem of poor quality data (also known as dirty data) is also very well known, with numerous tools and techniques for handling different problems involving dirty data. However, most of these tools and techniques either focus on assessing the quality of the data in a particular system [13] or on eliminating data quality problems (i.e., perform data cleaning) [14, 15]. Developers still lack practical approaches or tools for testing the behavior of services that make use of a persistent storage (relational or NoSQL), in the presence of poor quality data.

In a previous work, we presented a prototype for testing relational database applications in presence of poor quality data [16], which is essentially a tool for injecting poor quality data on relational database interfaces (namely on JDBC interfaces). In this work, we further generalize the approach and extend it to document NoSQL databases (e.g., MongoDB, Apache CouchDB), which are nowadays attracting a lot of attention from the industry and are increasingly being used in real system deployments. In short, our approach is based on intercepting database accesses and applying typical data quality mutations to the data being delivered to the application. During the tests, all software layers above the instrumented code are monitored for any obvious failures or suspicious behaviors. In this work, in addition to the variations in the operations considered for the NoSQL databases and further technical details behind the implementation of the approach, we give particular importance to consider the cases where complete objects are read from the database and the potential presence of poor data in those objects. In the case of relational databases, accesses are normally done over simple data types (e.g., strings, integers, dates); however, when NoSQL databases are used, storing and accessing full documents is a quite typical case.

We used our approach to test (i) an open-source and widely used relational database application for commerce and business, which we designate by AppR and (ii) an open-source NoSQL JSON-based web application, which we name AppN. During the tests, we were able to disclose several bugs in both applications, which highlight the usefulness of the approach. Moreover, the results also revealed the effectiveness of our approach in disclosing bugs at the middleware level. We detected a known bug in the PostgreSQL JDBC driver used by AppR [17], and we disclosed a new bug in Red Hat Hibernate 5.2.6 (i.e., a mainstream object-relational mapping middleware framework, used by applications to satisfy persistency requirements). We reported this new bug [18], which triggered a correction made by Hibernate developers in version 5.2.7. Results show the usefulness of the approach in disclosing bugs in application and middleware software used by millions of people. The main contributions of this paper are the following:

- An approach for testing relational and NoSQL database applications in presence of poor quality data;
- An open-source, zero-configuration, and free tool (available at [19]), which can be used as a data access replacement to test relational and NoSQL applications in presence of poor quality data;
- The identification of critical bugs, including security vulnerabilities, at the application and middleware levels and an analysis on how they could be avoided and corrected.

The remainder of this paper is organized as follows: The next section presents the related work on poor quality data and testing. The "Testing applications using poor data injection" section presents the approach for testing web applications in the presence of poor quality data. The "Case study" section presents a case study carried out using our approach and discusses the results. Finally, the "Conclusion" section concludes this paper.

Background and related work

Data quality, which is sometimes referred to as information quality [4], has been defined in many diverse ways in the literature [20]. A generally well-accepted definition is set by the ISO/IEC 25012 standard, which defines data quality as "the degree to which a set of characteristics of data fulfills requirements" [21]. These requirements refer to the needs and constraints that contribute to the solution of some problem [22], whereas the characteristics refer to properties of the data, such as its completeness, accuracy, or consistency [23].

The severe damage brought in by the presence of poor quality data in computer systems is well known in the industry and in the research communities [4–7]. As an example, the Gartner Group recently reported bad data as the main cause of failure in CRM systems, and several field studies show that the quality of the data, in a data-centric system (e.g., ERP or CRM), decreases over time

[8]. Nowadays, this problem is further aggravated, with the higher interconnectivity, complexity, and fast-changing dynamics of modern systems (e.g., IoT systems, fog computing systems, web services, services in service-oriented architectures) and also with the data volume growth, which can intensify its management complexity. In these environments, with data coming from potentially unreliable data sources, or originating from complex applications where the software changes very often (possibly holding residual bugs), it is easy to understand that the quality of the data will eventually decrease.

With the society and industry increasingly relying on computer systems, the effects of a failure caused by poor quality data can be critical. This should have a high impact on the way applications are built, as developers are greatly interested in programming systems that are reliable, even in the presence of poor quality data. The problem is that although the analysis and improvement of data quality have gathered plenty of attention (e.g., to carry out data cleaning operations) from practitioners and researchers [6, 24–28], and despite the well-known impact of poor quality data in critical data-centric systems [29], 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 software verification activities (e.g., software testing) is essential.

In a previous work [23], we researched the state of the art in data quality classification and data quality problems with the goal of identifying representative poor data quality issues that could be integrated in a test-based methodology. The goal of the work, besides providing a broad vision of the state of the art in this domain, was to support the definition of an approach that could use such knowledge to define effective tests for assessing data-centric applications in presence of poor quality data. We presented, in our previous work [16], an initial proposal for assessing the behavior of relational database applications in presence of poor quality data. The approach is based on the bytecode instrumentation of a JDBC driver, which allows us to intercept calls to the database. Whenever data is accessed by the application, our instrumentation code (based on the type of call and on the value of the data being delivered to the application) returns a case of poor quality data. Externally, the behavior of the application is observed and analyzed with the goal of identifying failures. In the present work, we generalize the approach to support not only relational but also NoSQL document database applications.

The main idea of this paper revolves around the application of software fault-injection techniques, which work by deliberately injecting software faults to potentially trigger failures (and their consequences) in the software. In practice, software fault-injection techniques aim to understand how well the software tolerates faults, which may include observing the behavior of failure detection and recovery mechanisms [30]. In this context, the closest studies in the domain of this paper actually come from the robustness testing area [10–12]. Although the perspective is quite different, in the end, the goal is to observe how the application deals with erroneous inputs, and this kind of concept also applies to other domains (e.g., fuzzing or even penetration testing). In the case of robustness testing, the goal is to understand the behavior of a particular system in presence of invalid input or stressful conditions [11]. The robustness tests stimulate the system with the objective of exposing possible internal errors, allowing developers to correct the identified problems. This technique can be used to distinguish systems according to their robustness and depending on the number and severity of the problems uncovered. It has been mostly applied to public interfaces, from a black-box perspective [11, 12]. The interaction points between different independent systems have 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 [11] is a tool for testing robustness that combines acceptable and exceptional values on calls to kernel functions of POSIX 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 are used to classify each system in terms of its robustness, according to the CRASH scale [8], which distinguishes several failure modes. In [31], the authors present the results of executing Ballista-generated tests on system calls of Windows 95, 98, CE, NT, 2000, and also Linux. The tests were able to trigger system crashes in Windows 95, 98, and CE. The other systems also revealed robustness problems, but not full system crashes.

MAFALDA [12] is another robustness testing tool that allows characterizing the behavior of microkernels in the presence of faults. It applies fault injection to parameters of system calls and also to the memory segments that implement the microkernel being tested. In a previous work, we defined an approach to assess the behavior of web services in the presence of tampered SOAP messages [10]. It consists of a set of robustness tests based on invalid web service call parameters. The services are classified according to the failures observed during the execution of the tests and using an adapted version of the CRASH scale [11]. More recently, robustness tests on web services have been extended to consider the state of the system being tested [32], which is modeled using symbolic transition systems.

Fuzzers are black-box tools that are typically used to discover vulnerabilities in applications, which, from a

code perspective, in essence refer to the presence of some software fault (e.g., a software bug) that allows an external fault (e.g., a malicious input) to harm the system [33]. Although the domain is typically security, many times these tools operate by providing erroneous data (random or semi-random) to the applications' public interfaces, thus being also partially related to the approach discussed in this paper. Fuzzers fit a few different types. Some target file formats (e.g., FileFuzz); others target network protocols (e.g. Querub, TAOF); others are designed to be more general and besides file and network can also use custom I/O interfaces, such as RPC or SOAP (e.g., Peach, Fuzzled); and finally, custom fuzzers exist and specialize in one specific format or protocol (e.g., AxMan, Hamachi) [34]. The simplicity of the technique used by fuzzers and the simplicity of its application are a great advantage of these tools.

Mutation testing is a method related to ours that was originally designed to understand how good test suites are at detecting faults [35]. Tests are executed on one version of the program that has been generated by applying a single fault, and the goal is to understand if the tests are able to detect that a mutant is being evaluated or not [35]. The faults used in mutation testing typically represent real coding errors made by developers, while in our approach, we use faults that represent typical data quality problems (which are essentially the effect of common operator errors mixed with the effect of developer mistakes or bad practices).

The impact of invalid data on the reliability of web services has been the object of research in [36]. The approach is based on a set of steps, which include building an architecture view of the system being tested, using a tool to measure the data quality or validity, 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.

Testing applications using poor data injection

This section describes our approach to test database applications in the presence of poor quality 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.

Injecting poor quality data

Our approach is based on the presence of an instrumented data access driver (e.g., the MongoDB CRUD driver or a JDBC driver) that we place between the application, which we generally designate as a service application, and a database server (e.g., a database management system). The ultimate goal is to understand if the service application can handle poor quality data coming from the database server 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). Any service application should be able to handle poor data, and this is especially true for applications deployed in critical environments. As mentioned, the presence of poor data in a storage system is known to increase with the age of applications, and in the 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. A simplified view of our approach, applied to a typical database application system, is depicted in Fig. 1 and explained in the next paragraphs.

As we can see in Fig. 1, there are three main parts that compose a typical database application system: (i) the service application, which concentrates business logic; (ii) a database server that is responsible for managing the data and accesses to the data; and (iii) a data access driver, which is used by the service application (via some

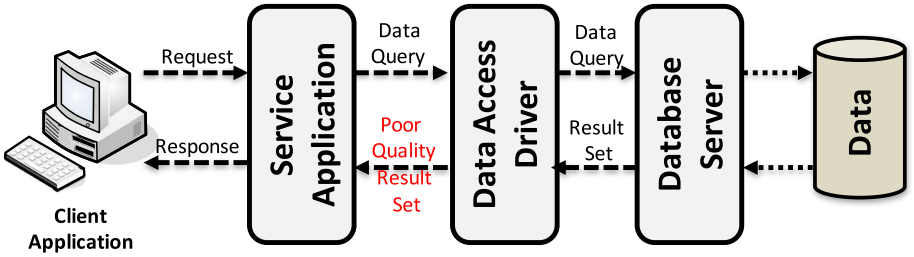

Fig. 1 The approach used for testing services using poor quality data, based on the presence of (i) the service application, which concentrates business logic; (ii) a database server for managing the data and accesses to the data; and (iii) a data access driver which is used by the service application for reading/writing data from/to the database. Data coming from the database is replaced with poor quality data by an instrumented data access driver

API) and allows it to read/write data from/to the database. The core concept behind our approach is that the driver is able to intercept all calls to the database management system and simulate the presence of poor quality data, by replacing the original data coming from the database with poor quality data.

In our approach, no changes to the service application code, database management system, or database are required. In practice, we do not modify the data access driver code directly; we rely on its external API and on code instrumentation libraries (e.g., Byteman, AspectJ) to intercept calls to well-known methods that are used by the application to access the data [37, 38]. By purely relying on the API to perform the aspect-oriented instrumentation, we can apply our approach to any data access driver. It is important to mention that, despite the specificities of this setup, the concepts involved in this approach are generic and are present in other mainstream programming languages and in other types of databases (e.g., graph databases). Although in this paper we mostly use a Java-based scenario, a similar setup could be used, for instance, with Python or C# .NET, as we can find code instrumentation libraries that allow achieving the same goals. Overall, this core mechanism is used in a set of distinct phases in our approach, explained in the next section. Despite the multiple phases, using this mechanism is very easy, essentially requiring a simple replacement of the data access driver with our instrumented version.

Approach execution phases

Our approach involves the execution of three phases, named warm-up, injection, and analysis. During all these three phases, we assume the presence of a workload generation client that places valid requests on the service, which allows exercising different service operations, each possibly making several data accesses at different code points. In addition to this scenario, during the injection phase, we also inject faults, which are specific cases of poor data, as observed in the literature and explained in the next paragraphs. Fig. 2 depicts the temporal execution of the three phases. For clarity, the figure presents the full details only for the injection phase.

During the *warm-up phase*, the instrumented data access driver performs like a regular driver. The 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. During this phase, several operations are exercised, as triggered by the workload generation client, which results in possibly several data access points being reached. Each of these data access points are locations in the code where there is a call to the database services (via the data access driver), in particular, where there is access to data retrieved from the database (e.g., an access to a particular column of a row resulting from the execution of an SQL query).

As mentioned, we assume the presence of a workload generation client which is responsible for defining the work that the service must fulfill during the execution of the tests. In practice, the workload can be one of the following three types: (i) real, (ii) realistic, and (iii) synthetic. Real workloads are the ones produced or consumed by applications in real environments, which means that they are representative of real scenarios (e.g., they may result in a representative coverage of the application code). In some scenarios, it is not possible to use real workloads (e.g., in the case the real environment is too complex to be understood or accurately reproduced for testing, and the workload may lose its meaning in a different environment). Realistic workloads are essentially a subset of representative operations that take place in real systems. They are

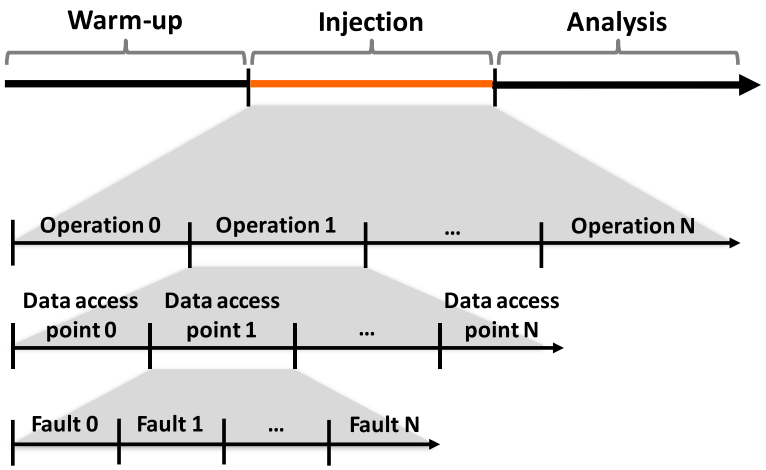

Fig. 2 Approach execution phases. The warm-up, injection, and analysis phases in detail

still quite representative and tend to be more portable than the real workloads (i.e., they apply easier to different systems, as they are less coupled to a particular system). The last type is the synthetic workload, which is an artificial (many times random) selection of calls (operations and respective input) to the system. In practice, this leads to higher repeatability and portability, when compared to the former workload types, but is also less representative. Regardless of the type of workload used, it should try to reach as much data access code points as possible and, at the same time, it should ideally represent the system's real operating conditions.

Passing from the warm-up phase to the injection phase is typically triggered by time, but any other criteria that are relevant for the system under test (or for the user performing the tests) can be used. For instance, allowing the system to reach a particular memory state or making sure the data storage is populated with specific data might allow reaching different data access points (or reaching previously visited ones in different state conditions) which potentially allows injecting different faults or obtaining different effects after injection. In the case of our tool, passing on to the injection phase can be manually set by the tester, by changing a Boolean configuration value at the driver.

The *injection phase* is central to the approach. During this phase, we replace genuine data coming from the database with data that represents a poor data quality problem, for the particular data type and value being accessed. In order to define which types of problems should be included in our testing approach, 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) in a previous work [23]. 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. Consequently, in this paper, we use this kind of data quality problems, in particular the ones applicable to single values, to build a fault model that is used during testing. By using single value mutations, we might actually be able to emulate some data quality problems that refer to multiple values (e.g., violation of referential integrity, violation of functional dependency). Still, defining specific combinations or ways to combine multiple values to represent more complex issues is left for future work and is out of the scope of this paper. Table 1 presents a partial example of data quality problems used in our approach. The complete model of the identified problems is available online (please refer to [19]).

A relevant aspect is that, although it is very infrequent in relational databases, when the target of the tests is a NoSQL document database application, it is not rare to have documents being stored or retrieved from the database (e.g., JSON documents). Considering the functions provided by the most popular APIs in major NoSQL document databases (e.g., MongoDB and Apache CouchDB), there are essentially two cases of document retrieval at the application level. The first one refers to the retrieval of data that is mapped to a POJO (e.g., the *query* method in *org.lightcouch.View* for CouchDB [39]), and the other one refers to the retrieval of data that is mapped in a container that is essentially a Map structure (e.g., the *get* method in *org.bson.BSONObject* for MongoDB [40]). Thus, in these cases, a list of all non-complex attributes in the object (either attributes in the POJO or keys in the Map) should be built with the purpose of randomly selecting a simple type (e.g., int, string, long). Once identified, the respective mutation should be applied and the object delivered to the application.

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 (limited to the number of data access points executed at runtime), 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 (other 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 this aspect is something out of the scope of the present work. In some applications, data access points may be shared by different operations. Even in these cases, it is 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 Fig. 2.

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 work, we manually determine that a test should stop when the user action

Table 1 Partial example of poor data quality faults

Data type	Issue description	Example
String	Replace by null	*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	John Locke ➜ Jon Locke
	Replace with an imprecise value. Chooses a single random word and replaces it by the respective acronym or abbreviation (dictionary-based)	Doctor John ➜ Dr. John
	Replace by homonym (dictionary-based, randomly selects a word from the original string)	allowed ➜ aloud
	Replace by synonym (dictionary-based, randomly selects a word from the original string)	happy ➜ cheerful
	Add whitespace in a leading or trailing position, or between words (random choice)	John Locke ➜ John Locke
	Remove whitespace in a leading or trailing position, or between words (random choice)	John Locke ➜ JohnLocke
	Add extraneous data in leading, trailing, or random position (random choice)	John Locke ➜ John Locke.
	Add substring (randomly selected and inserted in the beginning, random middle, or end of the original string)	John Locke ➜ Johnn Locke
	Remove substring (initial position and length are randomly chosen)	John Locke ➜ John Lock
	…	…
Integer	Set to zero	1904 ➜ 0
	Add one	1904 ➜ 1905
	Subtract one	1904 ➜ 1903
	Add one random numeric character	1904 ➜ 19004
	Remove one random numeric character	1904 ➜ 190
	Flip sign	1904 ➜ -1904
	…	…
…	…	…

has 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 a set of different tasks that basically allow to describe the behavior of the system and help in correcting any observed problem. First, the tester should observe any deviation from correct service. The most obvious way to do this is by analyzing service responses and detecting any deviation from the specifications. Also, in some cases, it might be helpful to observe any anomalous behaviors (e.g., high CPU usage or memory allocation at the server), which might be useful to, for instance, optimize code or system design. When the system under test is of large dimension, it might be difficult to carry out this task (especially considering that this kind of tests will then produce large amounts of data), as the tester may have many cases to analyze. Still, there are a few methods that can alleviate this task that, depending on the system being analyzed and on the tester's goals and budget, might apply (e.g., using machine learning algorithms to automatically identify incorrect responses [41]). Upon detection of a failure, the tester is very likely interested in further describing the failure, by, for instance, classifying the failure. A possibility is the use of the CRASH scale, which classifies the failures observed in different levels that represent distinct cases of failure severity. The levels are, from the highest to the lowest severity, Catastrophic, Restart, Abort, Silent, and Hindering [11]. This step is recommended if there is, for instance, the need for comparing systems or prioritizing bug fixing.

The next obvious task is to identify the location of the problem (i.e., in which software layer it occurs). As we are instrumenting calls to particular functions, any code that uses those instrumented functions can trigger a failure when handling poor quality data. In this context,

such code is at higher system layers that directly or indirectly need to execute the instrumented functions. This includes the application code itself but also libraries, such as Object mapping frameworks that abstract the details of the database system being used (e.g., Hibernate ORM for relational databases, Morphia for MongoDB). This task is of particular relevance when the goal is to characterize a bug in the context of a whole system.

Finally, for the cases where the tester is interested in correcting the problem (or simply further understanding it, so that it is not repeated in a new version of code), it is important to identify the origin of the problem, i.e., understanding its root cause. This includes pinpointing the exact location in the code where the problem exists (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). Again, this might become a difficult and time-consuming task to carry out, which the user might be interested in automating by using, for instance, machine learning methods [42].

During all phases, any request 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. 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.

Applying the approach

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

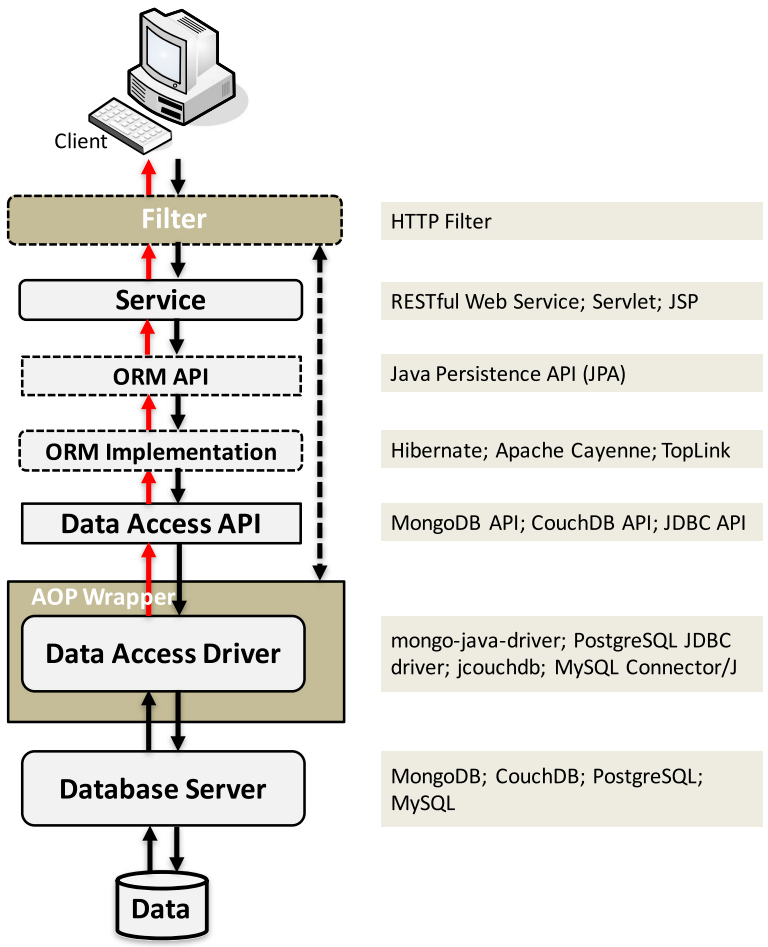

Fig. 3 Detailed view of the components involved in the tests. The elements involved in a typical services interaction are represented in gray boxes, while in light brown, we have the components that are part of our approach. All components in solid lines are mandatory, whereas the dashed components are optional (their use depends on the application being tested and on the tester requirements)

represented in gray boxes, while in light brown, we have the components that are part of 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 the 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, the JDBC and MongoDB drivers) 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 1). In order to rely only on the AOP Wrapper to support the core of the approach, we start by identifying the functions of the driver's API that access the database. These functions may differ depending on the system used (i.e., a relational database vs a NoSQL document database). In the case of relational databases, there are standard APIs (e.g., JDBC), which do not exist yet for document databases, such as MongoDB or Apache CouchDB. For these latter cases, each individual API has to be considered and the exact methods that allow accessing data from the database must be identified. Despite this additional step, the differences between the different cases are usually small, as, in general, they all provide access to the same data types. Table 2 shows an excerpt of the calls that provide applications with data access and that need to be intercepted in the JDBC API, with MongoDB's native driver and with the LightCouch driver for CouchDB, which are among the most popular NoSQL database engines [43].

Using the above function signatures as reference, we then instrument the driver being used by the system, so that any calls to the identified functions run through our injection code *after* actually running through the driver's code but before data is delivered to the application. Our injected code will allow us, at runtime, to replace any data retrieved from the database by our own poor quality data (according to specific rules presented in the next section). This kind of setup is easily achieved with the use of code instrumentation libraries (e.g., Byteman, AspectJ), as long as we provide the signatures of the functions to be intercepted. An example of the application of this technique is shown in Fig. 4, which uses AspectJ to intercept calls to a MongoDB method that extracts a string from a database object and replaces the returning string by a faulty string.

The remaining components above the instrumented driver are the typical components of a service application, namely, the service code and any additional middleware. In terms of typical middleware, the use of an Object Mapping library is relatively frequent (e.g., an Object Relational Mapping framework, such as Hibernate or an Object Document Mapper, such as Morphia).

Finally, in terms of overall setup and although not mandatory, we recommend the presence of a service *Filter*. In this context, a Filter is a component that is executed at two moments: (1) before processing each client request and (2) after the client request has been processed (and immediately before the response is delivered to the client). Thus, in the case of a typical service (e.g., a SOAP/REST web service or a web application), this can simply be an HTTP Filter or any other component that is executed at the two moments referred (e.g., an HTTP proxy). It is worth mentioning that the major web serving platforms allow configuring HTTP Filters, as it is actually a requirement for any Java EE compliant platform. In the case of our tool, we deploy an HTTP Filter for this purpose.

The HTTP Filter also allows a fine-grained 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

Table 2 Example of API methods intercepted

JDBC API	MongoDB API	CouchDB API
java.sql.ResultSe	org.bson.BasicBSONObject	org.lightcouch.View
String getString(String columnLabel)	String getString(String key)	String queryForString()
long getLong(String columnLabel)	long getLong (String key)	long queryForLong()
int getInt(String columnLabel)	int getInt (String key)	int queryForInt()
boolean getBoolean(String columnLabel)	boolean getBoolean(String key)	boolean queryForBoolean()
Date getDate(String columnLabel)	Date getDate(String field)	–
T getObject(String columnLabel, Class<T> type)	Object org.bson.BSONObject.get(String key)	List<T>query(Class<T> classOfT)
...	...	...

```
@Around("execution(* org.bson.BasicBSONObject.getString(..))")
public synchronized Object interceptGetString(ProceedingJoinPoint joinpoint) throws Throwable
{
        String result;

        Object obj = joinpoint.proceed();
        result = injectStringFault(obj);

        return result;
}
```

Fig. 4 Aspect-oriented code for intercepting a MongoDB string access function. Aspect-oriented (AspectJ) code to intercept calls to a MongoDB method that extracts a string from a database object and replaces the returning string by a faulty string

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 [19], all of these server-side components are assembled in a single unit (i.e., one jar file) that implements our approach, which will replace the original data access driver (i.e., the JDBC or MongoDB 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 replace the original data access driver with our own and it will be ready to perform injection of poor data for any Java web application that uses a data access driver. With our current configuration, any JDBC compliant driver will do (e.g., Oracle database, MariaDB, PostgreSQL) and typical drivers for the NoSQL document databases MongoDB or CouchDB are also contemplated. However, in practice, any data access driver will serve this purpose, as long as we write the names of the required dependencies in our maven project descriptor file, identify the names of the functions to intercept, and finally recompile and package the project using *mvn package*. No further implementation or configuration is needed.

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.

Experimental setup

We selected two cases for carrying out the tests, a relational database application and a NoSQL application, which we respectively name AppR and AppN. AppR is a well-known widely used commercial open source ERP business solution for enterprises. The company behind this ERP solution characterizes the product as a world leader counting, at the time of writing, with nearly 2.5 million downloads. It allows companies to manage their entire business and supports typical processes such as sales, manufacturing, or finance. The developers behind this application have obvious software quality concerns, as any software or security failure might impair the application user's business, which in turn will impact AppR's company business. AppR requires a database which we chose to be PostgreSQL 9.4 and a server for deployment for which we chose the well-known Apache Tomcat 7.0.68. The interaction with the database in this context is mediated by the PostgreSQL JDBC driver 9.4-1201 [44]. Regarding AppN, we selected an open source blog web application that makes use of MongoDB [45], which is currently one of the most popular NoSQL database engines available [43]. To interact with the database, AppN uses the MongoDB Java driver (v3.4.2) [46], which is the typical solution for Java-based applications that need to use this kind of storage. As we intended to repeat the tests, besides a regular browser, we recorded and later replayed user actions on the browser using SikuliX 1.1.0.

Selecting and executing the test cases

AppR is an application of huge dimensions and due to this, we selected a few test cases for testing. We considered the CRUD model [47] 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 be relevant as long as it accesses the database. Regarding AppN, the same rationale was applied and again we selected the four types of operations.

The execution of the experiments resulted in thousands of faults being injected, with each operation being executed 100 times (to allow going through the different faults, with different configurations). Due to the nature of the operations, the majority of the injected faults (about 95.5%) refer to the *string* type, followed by *Long Integer* faults with 2.4%, and Boolean faults with 1.1%.

The remaining types of faults were injected with a frequency lower than 0.1%.

Table 3 presents the operations selected for testing, their mapping to the CRUD model, and a reference to the selected failures uncovered in each operation during testing (these failures are discussed in the next section).

Results and discussion

As shown in Table 3, 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 data access driver code (i.e., the widely used PostgreSQL driver).

Table 4 presents an excerpt of the issues disclosed during the tests. We were able to disclose further issues, but we opted to discuss only a set of problems that manifested in different forms and at different structural locations of the system (as noticed in Table 4). 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 in AppR triggers a *TemplateModelException* that eventually is shown to the user in an alert box. In the case of AppN, whenever this failure was observed, a message mentioning the presence of an internal error was shown to the user.

Table 3 Results overview

Application	Operation		
	Name	Type (CRUD)	Failure Reference
AppR	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
AppN	Login	R	A, E, F
	Create a new user	C	F, G
	Create a new post	C	A, G
	Add comment	C	A
	Delete post	D	A
	Add like	U	H
	List posts	R	A
	List comments	R	A

AppR loads several classes dynamically, and *Failure B* occurs when one of those names is wrong (due to the application of a mutation). When 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 understand 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 new bug in Hibernate, the popular Object-Relational Mapping framework used by the AppR. 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, for which we filled a bug report [18] which triggered a correction by Hibernate developers in Hibernate 5.2.7. The bug was found to affect Hibernate 5.2.6 and 5.1.3 and could have been avoided if the developers had used our testing technique. This is actually very similar to the one described next (which has already received attention and a correction from the developer community).

Failure E occurs in AppR when adding characters that include a single quote to a string. This is a reported bug [17] 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

Table 4 Selected cases from the experiments

Ref	Root exception triggered	Location	Last mutation	External behavior
A	NullPointerException	Application	changeToOppositeCase (AppR) changeToNull (AppN)	*TemplateModelException* reported to the user (AppR). An internal error is reported to the user (AppN).
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 (AppR); application (AppN)	addCharactersToString (AppR), All cases except for changeToNull and changeToLongString (AppN).	No message displayed to the user An internal error is reported to the user (AppN)
F	MongoWriteException (key too large to index)	Application	changeToLongString	An internal error is reported to the user (AppN)
G	MongoWriteException (duplicate key error collection)	Application	All cases except for changeToLongString	An internal error is reported to the user (AppN)
H	MongoWriteException (modifiers operate on fields but we found type null instead)	Application	changeToNull	An internal error is reported to the user (AppN)

closing single quote and returns the position 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. In AppN, this failure occurs when replacing the encrypted password by any string without a comma character (the salted hashed password is supposed to have two parts separated by a comma: the hashed value and the salt value). The application fails to split these two parts from each other, because it does not check if the read data holds the comma character.

Failure F occurs when AppN attempts to insert a record including a long string. It occurs at two points: when a new session is inserted to the database (after a successful login) and when a new user is being registered. In both cases, the username is set to a long string after being read from the database (the read operation is done for checking the existence of the username). This failure could be prevented in the application by checking the length of data before insertion.

Failure G occurs when AppN attempts to insert a new user session, assigned to an existing key (username), into the database. This failure is uncovered by setting the username (key) to an invalid string. The application adds a new session into the database using this invalid string as a key without verifying its correctness. The failure occurs when a new user enters the system, and its username is set to the same invalid string. This failure could be avoided by checking the correctness of the value read from database.

Finally, *Failure H* occurs when AppN tries to update the value of a field with null. The application fails when the value of the data involved (the number of current "likes" in this case) is set to null. A simple null check would be sufficient to avoid this failure.

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 bugs 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.

There are a number of lessons to learn from these experiments. First, and foremost, most of the failures are due to the fact that developers do not see the database as an external system that is prone to hold poor quality data. Thus, the applications are programmed to trust the database inputs, which may lead to not only failures but also severe security problems. Validation at the entry points of the application is a crucial technique to use, and this applies not only to the application external interfaces (i.e., with clients) but, as we have seen, also to the application internal interfaces (i.e., with the database

system). Regarding the fault model used, its relevance is based on the fact that it holds types of faults that represent typical data quality problems (as identified in [23]). However, it is expectable that some faults may be more prone to cause visible problems, of which we highlight the faults that set null values, empty values, and also those that include query language (e.g., SQL) keywords or characters (e.g., apostrophes). While the former two types of faults show the lack of ability to handle unexpected data coming from an external system, the latter shows the lack of concern with malicious data coming from the database, potentially allowing for changes to the structure of subsequent data access queries. These observations are obviously coupled to the two systems being tested and, due to this, they cannot be generalized. However, we did perform tests on quite typical database applications in the presence of also typical data quality problems; thus, our observations can be seen as a valuable source of information regarding the errors made by developers when building their services.

Conclusion

This paper presented a testing approach for relational and NoSQL database applications, which is 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 in both types of systems, including bugs at the application, JPA implementation (Hibernate), and also the PostgreSQL JDBC driver used. Some of the bugs disclosed also represent security problems, which emphasizes the technique's ability to detect security issues in the system being tested. Although we were expecting to find a few problems at the application level (even in the very popular AppR application), the ability of the technique to disclose middleware level bugs (already recognized and corrected by the middleware developers) was quite surprising if we consider the widespread usage of this kind of middleware (e.g., Hibernate counts, at the time of writing with about 11 million downloads at *sourceforge.net*).

We showed that the application of the technique can be valuable not only for service providers, which are made aware of the software issues in their systems, but also for service developers which have the chance to improve the behavior of their services (or middleware) in the presence of poor quality data.

As future work, we intend to further extend the approach to other types of NoSQL engines, explore the state of the system being tested, and further explore additional configurations. We also intend to research ways of automating the tests, possibly resorting to machine learning algorithms to analyze the responses obtained and the system behavior.

Availability of data and materials
Not applicable

Authors' contributions
All authors read and approved the final manuscript.

Funding
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); by the project EUBra-BIGSEA, funded by the European Commission under the Co-operation Programme, Horizon 2020 grant agreement n° 690116; and by P2020 SAICTPAC/0011/2015 Project MobiWise: from mobile sensing to mobility advising.

Authors' information
Not applicable

Ethics approval and consent to participate
Not applicable

Consent for publication
Not applicable

Competing interests
The authors declare that they have no competing interests.

Publisher's Note

Author details
[1]CISUC, Department of Informatics Engineering, University of Coimbra, Coimbra, Portugal. [2]Polytechnic Institute of Coimbra - ISEC, Coimbra, Portugal.

Received: 19 June 2017 Accepted: 22 November 2017
Published online: 06 December 2017

References

1. Han J, Haihong E, Le G, Du J (2011) Survey on NoSQL database. In: 2011 6th international conference on pervasive computing and applications, pp 363–366
2. Gao J, Xie C, Tao C (2016) Big data validation and quality assurance—issuses, challenges, and needs. In: 2016 IEEE symposium on Service-Oriented System Engineering (SOSE), pp 433–441
3. Loshin D (2011) Evaluating business impacts of poor data quality. IAIDQ's Information and Data Quality Newsletter, Vol. 7 Issue 1. https://www.iqint.org/publication2011/doc/loshin-2011-01.shtml. Accessed 27 May 2017
4. Ge M, Helfert M (2007) A review of information quality research—develop a research agenda. Proceedings of the 12th international conference on information quality, MIT, Cambridge, MA, USA, pp 76–91
5. Pipino LL, Lee YW, Wang RY (2002) Data quality assessment. Commun ACM 45:211–218
6. Loshin D (2010) The practitioner's guide to data quality improvement. Morgan Kaufmann. https://www.elsevier.com/books/the-practitioners-guide-to-data-quality-improvement/loshin/978-0-12-373717-5
7. Quality ED (2015) The data quality benchmark report. Experian Data Quality. https://www.edq.com/globalassets/white-papers/data-quality-benchmark-report.pdf. Accessed 27 May 2017
8. Marsh R (2005) Drowning in dirty data? It's time to sink or swim: a four-stage methodology for total data quality management. J Database Market &#38 Custom Strategy Manag 12:105–112. doi:10.1057/palgrave.dbm.3240247
9. Singh R, Singh K et al (2010) A descriptive classification of causes of data quality problems in data warehousing. Int J Comp Sci Issues 7:41–50
10. Laranjeiro N, Vieira M, Madeira H (2012) A robustness testing approach for SOAP web services. JISA 3:215–232. doi:10.1007/s13174-012-0062-2

11. Koopman P, DeVale J (1999) Comparing the robustness of POSIX operating systems. In: Twenty-ninth annual international symposium on fault-tolerant computing, pp 30–37
12. Rodríguez M, Salles F, Fabre J-C, Arlat J (1999) MAFALDA: microkernel assessment by fault injection and design aid. In: The third European dependable computing conference on dependable computing. Springer-Verlag, Berlin, Heidelberg, pp 143–160
13. Sebastian-Coleman L (2013) Measuring data quality for ongoing improvement: a data quality assessment framework. Morgan Kaufmann Publishers Inc, San Francisco. https://dl.acm.org/citation.cfm?id=2500972
14. Saha B, Srivastava D (2014) Data quality: the other face of big data. In: Data Engineering (ICDE), 2014 IEEE 30th International Conference on. IEEE, pp 1294–1297
15. Ilyas IF, Chu X et al (2015) Trends in cleaning relational data: consistency and deduplication. Foundations Trends® Databases 5:281–393
16. Laranjeiro N, Soydemir SN, Bernardino J (2016) Testing web applications using poor quality data. Seventh Latin-American Symposium on Dependable Computing (LADC 2016), Cali, pp 139–144. doi:10.1109/LADC.2016.30
17. PostgreSQL JDBC driver—Issue 369 (2015). https://github.com/pgjdbc/pgjdbc/issues/369. Accessed 27 May 2017
18. Seyma NS [HHH-11134] StringIndexOutOfBoundsException in BooleanTypeDescriptor—Hibernate JIRA. https://hibernate.atlassian.net/browse/HHH-11134. Accessed 27 May 2017
19. Laranjeiro N, Seyma NS, Jorge B (2017) Poor data quality injector toolset and dataset. http://eden.dei.uc.pt/~cnl/papers/2017-jbcs.zip. Accessed 18 Jun 2017
20. Batini C, Palmonari M, Viscusi G (2014) Opening the closed world: a survey of information quality research in the wild. In: The philosophy of information quality. Springer International Publishing, Springer, Cham, pp 43–73. https://doi.org/10.1007/978-3-319-07121-3_4.
21. ISO/IEC (2008) Software engineering – software product quality requirements and evaluation (SQuaRE) – data quality model. ISO/IEC. https://www.iso.org/standard/35736.html
22. SWEBOK V3 Guide IEEE Computer Society
23. Laranjeiro N, Nur Soydemir S, Bernardino J (2015) A survey on data quality: classifying poor data. The 21st IEEE Pacific Rim International Symposium on Dependable Computing (PRDC 2015). doi: 10.1109/PRDC.2015.41
24. Caro A, Calero C, Mendes E, Piattini M (2007) 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. pp 143–153
25. Xiaojuan B, Shurong N, Zhaolin X, Peng C (2008) Novel method for the evaluation of data quality based on fuzzy control. J Syst Eng Electron 19: 606–610. doi: 10.1016/S1004-4132(08)60127-9
26. Bergdahl M, Ehling M, Elvers E, et al (2007) Handbook on data quality assessment methods and tools. Wiesbaden, European Comission Eurostat. https://unstats.un.org/unsd/dnss/docs-nqaf/Eurostat-HANDBOOK%20ON%20DATA%20QUALITY%20ASSESSMENT%20METHODS%20AND%20TOOLS%20%20l.pdf
27. Choi O-H, Lim J-E, Na H-S, et al (2008) An efficient method of data quality evaluation using metadata registry. In: Advanced Software Engineering and Its Applications, 2008. ASEA 2008. pp 9 –12
28. Galhardas H, Florescu D, Shasha D (2001) Declarative data cleaning: language, model, and algorithms. In: In VLDB, pp 371–380
29. Haug A, Zachariassen F, van Liempd D (2011) The costs of poor data quality. J Indust Engineer Manag 4(2):168–193. doi:10.3926/jiem.2011.v4n2.p168-193
30. Natella R, Cotroneo D, Madeira HS (2016) Assessing dependability with software fault injection: a survey. ACM Comput Surv 48(44):1–44:55. https://doi.org/10.1145/2841425
31. Shelton CP, Koopman P, Devale K (2000) Robustness testing of the Microsoft Win32 API. In: International Conference on Dependable Systems and Networks (DSN 2000). pp 261–270
32. Salva S, Rabhi I (2010) Stateful web service robustness. Fifth International Conference on Internet and Web Applications and Services. doi: 10.1109/ICIW.2010.32
33. Antunes J, Neves N (2012) Recycling test cases to detect security vulnerabilities. In: IEEE 23rd international symposium on software reliability engineering (ISSRE 2012). IEEE Computer Society, Washington, pp 231–240
34. Eddington M (2009) Demystifying Fuzzers. Black Hat USA, Las Vegas
35. Fraser G, Zeller A (2010) Mutation-driven generation of unit tests and oracles. In: Proceedings of the 19th international symposium on software testing and analysis. ACM, New York, pp 147–158
36. Musial E, Chen MH (2012) Effect of data validity on the reliability of data-centric web services. IEEE 19th International Conference on Web Services, Honolulu, pp 576–583. doi:10.1109/ICWS.2012.95
37. Eclipse Foundation (2006) The AspectJ Project. http://www.eclipse.org/aspectj/. Accessed 8 Apr 2016
38. Kiczales G, Lamping J, Mendhekar A, et al (1997) Aspect-oriented programming. 11th European Conference on Object-oriented Programming
39. LightCouch.org (2017) View (LightCouch 0.1.8 API). http://www.lightcouch.org/javadocs/org/lightcouch/View.html. Accessed 27 May 2017
40. MongoDB (2017) MongoDB BSONObject API. http://api.mongodb.com/java/current/org/bson/BSONObject.html. Accessed 27 May 2017
41. Laranjeiro N, Oliveira R, Vieira M (2010) Applying text classification algorithms in web services robustness testing. 29th IEEE international symposium on reliable distributed systems (SRDS 2010)
42. Wong WE, Gao R, Li Y et al (2016) A survey on software fault localization. IEEE Trans Softw Eng 42:707–740. doi:10.1109/TSE.2016.2521368
43. solid IT (2017) DB-Engines ranking—popularity ranking of database management systems. https://db-engines.com/en/ranking. Accessed 27 May 2017
44. The PostgreSQL Global Development Group (2017) PostgreSQL JDBC driver. https://jdbc.postgresql.org/. Accessed 27 May 2017
45. MongoDB (2017) MongoDB NoSQL Database. https://www.mongodb.com. Accessed 27 May 2017
46. MongoDB (2017) MongoDB Java Driver. http://mongodb.github.io/mongo-java-driver/. Accessed 27 May 2017
47. Martin J (1983) Managing the data base environment, 1st edn. Prentice Hall PTR, Upper Saddle River