# The Test Data Challenge for Database-Driven Applications

Klaus Haller
COMIT AG
Pflanzschulstr. 7
CH-8004 Zürich
Switzerland
klaus.haller@comit.ch

## ABSTRACT

Business applications rely typically on databases for storing and processing their data (database-driven applications, or DBAPs). Testing DBAPs requires testing the application logic *plus* the interaction between the application logic and the database. Thus, DBAP test cases consist of input and output parameter values, the function to be tested, *and* an initial database state (i.e., DBAP test data). Various test data provisioning methods exist, such as manual test data design, generators for synthetic test data, and live-system snapshots. Many criteria and factors influence which method is optimal for a given project setting, such as costs, quality, data privacy, etc. This paper presents our methodology for guiding software development projects towards the DBAP test data provisioning method best suited for them.

## Categories and Subject Descriptors

D.2.4 [**Software Engineering**]: Software/program verification, D.2.5 [**Testing and Debugging**]: Testing tools, coverage testing, H.0 [**GENERAL**]

## Keywords

Information systems, databases, testing, test data, test coverage

## 1. INTRODUCTION

Today, most business applications rely on databases for storing and managing data (DBAPs). The rise of object-relational mapping and persistency frameworks like Hibernate [1] eases the development of DBAPs. The frameworks allow many database details to become transparent. However, one truth has not changed: Fruitful testing of DBAPs is possible only if there is test data in the tables of the DBAP (DBAP test data). Projects often ignore this fact until it is too late, and project managers end up making ad-hoc decisions without a conceptual foundation. One reason might be the lack of a methodology for approaching the DBAP test data challenge. We address this need in this paper.

The role of input data and how it is derived is well understood for non-DBAP applications (see, e.g., the survey of Zhu, et al. [2] on test data and coverage). In contrast, DBAP test data is a niche topic. Even testing methodology books (e.g., Perry [3]) do not cover it. Only most recently has the combination of databases (or DBAPs) and testing received more attention. A key concept is the quintuple model for DBAP test cases (Willmor et al. [4]). The first three elements of the quintuple model are the procedure to be tested, the input parameters, and the expected output parameters. These elements are the same for normal test cases; however, two new elements exist: the initial database state (i.e. DBAP test data) and the expected result state.

*Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. To copy otherwise, to republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee.*

*DBTest'10, June 7, 2010, Indianapolis, Indiana, USA.*
*Copyright © 2010 ACM 978-1-4503-0190-9/10/06... $10.00*

Groundbreaking work in the field of DBAP test data is the AGENDA prototype (Deng, et al. [5]). It generates test data with techniques such as boundary analysis for testing single SQL statements or complete transactions. The work of Dai and Chen [6] falls into the same category. They follow a holistic approach covering the complete testing process from database schema and program sources and their analysis, to test adequacy criteria and automatic execution.

Houkjær et al. [7] describe how to generate DBAP test data based on database catalogue information, thereby dealing with foreign key relationships. Willmor and Embury [4] contribute a concept for intensional specification of DBAP test cases. They analyze the DBAP tables for matching data. If no matching data exists, test data is generated. Binning et al.'s work on Muli-RQP [8] follows the same vision. They generate a test database state based on declarative test case specifications. In a second paper, Binning et al. [9] address how to test the correctness of a database system. Their system takes one or multiple queries as input together with query results and a database schema. As a result, they get *one* possible suitable database state as output.

All of these papers focus on how to create data. In contrast, Suárez-Cabal and Tuya [10] assume having a large database. Their idea is to remove all rows from the database tables that do not contribute to additional test coverage. Test coverage is also a topic for Kapfhammer and Soffa [11]. They propose a coverage criterion based on a database interaction control flow graph.

The important research mentioned above focuses (mostly) on algorithms for generating DBAP test data. We bring in a different focus. We focus on developers and testers in commercial software development projects. They need to know which concrete DBAP test data provisioning method is optimal under which circumstances. Our work on DBAP test data is based on experience spanning commercial projects in areas such as core-banking system implementation projects, application management, and software development. In our previous work, we focused first on getting a holistic understanding of DBAPs and testing [12]. The next step was identifying important factors influencing the decision for a method for deriving DBAP test data [13]. This paper extends our work by analyzing the interplay between impact factors and concrete DBAP test data provisioning methods. It answers questions such as: how do privacy needs go together with using live-data?

We structure our paper as follows. Section 2 presents the sample application we then use throughout the paper. We introduce compliance levels for assessing DBAP test data quality in Section 3. Section 4 compiles the most popular methods for

providing DBAP test data. Section 5 describes our methodology and the impact factors driving the decision process. We link the impact factors to the different test data provisioning methods in Section 6. To the best of our knowledge, this is the first paper discussing the pros and cons of the various methods while considering different impact factors thoroughly and providing practical useful guidelines. In Section 7, we conclude with a short summary and point out the remaining questions.

## 2. Sample Application

Our sample application is a three-tier-application for credit rating (Figure 1). Banks use credit rating applications for estimating how likely a customer is to pay back the loan and pay the interest. The rating determines whether a customer gets loan and at which interest rate.

The presentation layer has two sample GUIs: One GUI is an input mask for new financial statements. It has input fields for the balance sheet date, the sum of all assets, the sum of all liabilities, and the earnings before interest and taxes (EBIT). If the user wants to save the financial statement, the validation procedure P_VALIDATE_FS checks whether assets and liabilities are equal. This check is part of the presentation layer. If the check succeeds, procedure P_STORE_FS writes the data into table T_FINSTATEMENT. The first column stores the company ID, which refers to a company in table T_CUSTOMER. The second column stores the balance sheet date; the third, the sum of all assets. Additional columns store the sum of all liabilities and the EBIT. The last column stores the rating class if it has been calculated. The second GUI is for approving the financial statement by a second or third person. The application layer has a second procedure: P_CALC_RATINGS. It is a batch job that executes overnight and calculates the rating for all companies. The workflow itself (not illustrated) also belongs to the business logic layer.

The table T_CUSTOMER stores customer details. It has three columns: a unique company ID, the company name, and whether the company is a corporation. Table T_TRANSLATION stores the GUI texts in German and English. Thus, the text shown in the masks can switch languages depending on the parameterization item "Language" in table T_PARAMETRIZATION. The parameterization table also contains the name of the bank shown in the masks. Two additional parameterization items are "Limit_Approval_2" and "Limit_Approval_3." The values specify for which balance sheet totals two or three persons have to approve the loan.

## 3. Test Data Quality

The quality of different test data sets can differ; we can compare their quality only if we have a notion of quality. Therefore, we rely on the concept of *test data compliance levels* [12]. There are four levels: The lowest level is **type compliance**. It requires that the data reflect the types of columns. If the ID column of table T_CUSTOMER is of type "NUMBER," the test data are numbers and not strings. We can generate type compliant test data easily. Aside from random generators for the different data types, we need table names, column names, and column types. The database catalogue provides this information. However, type compliance does not guarantee that we can load all rows into tables successfully that we prepare. If the tables have constraints, rows might be rejected.

If the test data respects all constraints, the database accepts all rows we want to load. This test data is **schema compliant**. Again, we can read all constraints from the database catalogue. Most of them are easy to deal with: primary key, foreign key, not null, and unique constraints. However, check constraints are a challenge because they can contain arbitrary conditions.

We can achieve the two previous compliance levels relying only on the database catalogue. The following two compliance levels need more information. If we look at table T_FINSTATEMENT, obviously, the two columns ASSETS and LIABILITIES should be the same. If not, procedure P_VALIDATE_FS rejects the financial statement. This is an example of dependencies between attributes enforced by the application and *not* reflected by constraints. Certainly, such dependencies can also exist between tables—that is, one table with financial statements and a table with profit and loss information. If the DBAP test data does not reflect such dependencies, the application might process data for which it is not specified. As a result, false

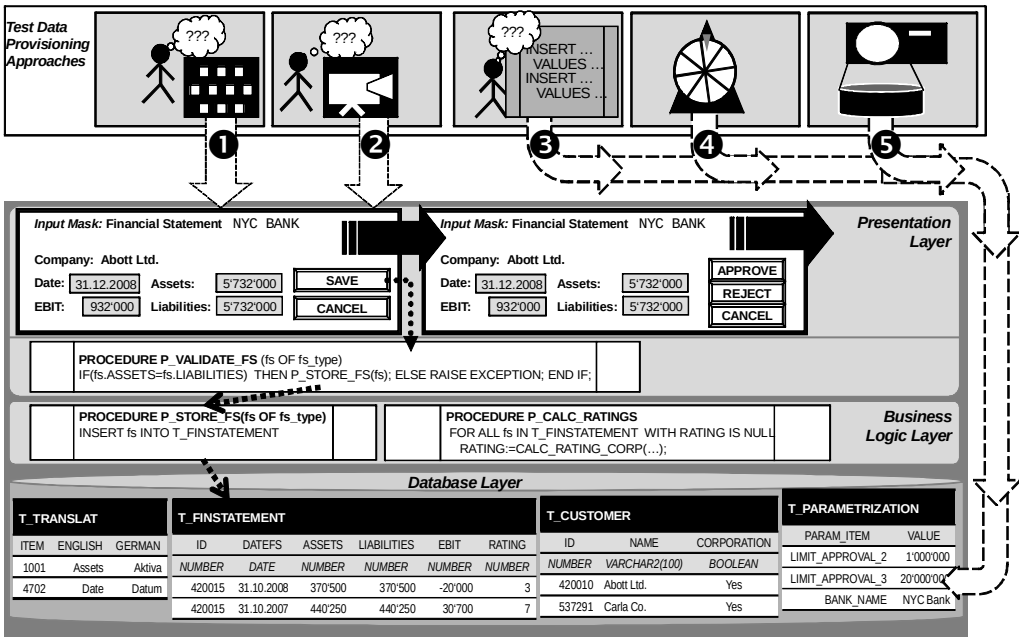


**Figure 1: Sample Application & Provisioning Approaches**

positives can appear. Only when test data could be the result of "normal" GUI input and data processing, no false positives appear. Then, the test data is **application compliant**.

Still, test data can become better. Let us assume we test the credit rating functionality for corporations in procedure P_CALC_RATINGS. This test case requires a financial statement of a corporation in table T_FINSTATEMENT; otherwise, we cannot test the functionality. If the DBAP test data is suitable for a test case, it is **path compliant**. Path compliance always refers always to a test case and the corresponding execution path of the application. Table 1 compiles the information for all four compliance levels.

| Compli-ance Levels | Informa-tion Needs | Generated = Loaded Data | False Positives | Intended Path Executed |
|---|---|---|---|---|
| ***Type Compliant*** | Schema | Not guaranteed | Yes | Not guaranteed |
| ***Schema Compliant*** | Schema | Yes | Yes | Not guaranteed |
| ***Application Compliance*** | Schema & application logic | Yes | No | Not guaranteed |
| ***Path Compliance*** | Schema & application logic & path | Yes | No | Yes |

**Table 1: Compliance Levels and their Characteristics**

## 4. Test Data Provisioning Methods

This section discusses the most popular methods for providing DBAP test data. We distinguish *how* to gain test data and how to get test data *into* the DBAP. There are three options for the *how* aspect: copying live data, designing data manually, or generating test data automatically. There are two options for the *into* aspect: Option one is the normal way, via GUI. The business logic layer manages how to write the GUI input into database tables. The second option is writing the data directly into database tables. There are six ways to combine the *how* and the *into*. There are out-of-the-box solutions for four of them (Table 2).

The idea of manual test data is that a tester (or developer) analyzes the application. She designs test cases based on the specification (black-box testing) or on the application structure (white-box testing). Next, she must get the data into the DBAP. She can insert the test data (e.g., financial statements) using the GUI (*GUI Input*, Figure 1, ❶) or via other interfaces. Or she can use *capture and replay* tools such as Selenium [14]. They capture the keyboard and mouse and mouse input and replay the input every time the DBAP is installed new again (❷). Writing INSERT scripts (e.g., in SQL) is option three. INSERT statements write data directly into database tables (❸).

| | Manual | Generated | Live-Data |
|---|---|---|---|
| ***Regular (GUI/ Interface)*** | GUI Input, Capture & Replay | No out-of-the-box solution | No out-of-the-box solution |
| ***DB direct*** | INSERT Scripts | Data Generation Tools | DB Snapshots |

**Table 2: Data Provisioning Methods**

*Test data generators* (❹) generate synthetic test data automatically and write the data directly into DBAP tables. The *snapshot approach* (❺) relies on live system data; it copies the database of a live system and loads the data into the database of a test system. Testers can use systems such as Oracle's data pump functionality [15]. The data generators and the snapshot approaches both work directly on the database tables. To the best of our knowledge, there are no commercial tools on the market that support data generation or snapshots with data input via GUI.

## 5. A Methodology for a Choosing a Provisioning Method

Commercial projects take quality, quality assurance, and test data quality seriously. But costs are relevant, too. Costs can be divided into one-time set-up costs and reoccurring maintenance costs. The decision for a provisioning method means estimating quality and costs for the different provisioning methods and choosing the one with the most adequate cost-quality profile.[1] Such a methodology requires first analyzing the different impact factors; impact factors can be either DBAP specific or context specific. DBAP factors reflect the domain and concrete requirements of the application such as how often the GUI changes. Context factors address organizational issues such as whether power users are responsible for the test data. Understanding these impact factors is the first step (Step 1, "Analysis," Figure 2).

Step 2, "Evaluation," is understanding how the impact factors relate to costs and quality for the different provisioning methods. A single formula returning a triple <quality, set-up costs, maintenance costs> for each method would simplify the decision process. However, commercial projects are more complex: There are hard constraints (such as privacy needs for banks) or the information is more vague than three numbers express. A methodology should never feign an exactness that does not exist

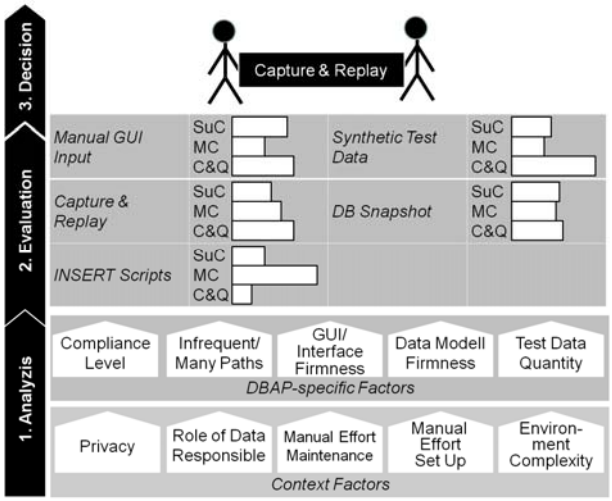


**Figure 2: DBAP Test Data Provisioning Methodology**
SuC: Set-up Costs, MC: Maintenance Costs, C&Q: Coverage Quality

[1] Certainly, bad test data quality can also be a cost-driver, such as in case of false positives. Software engineers have to analyze them until they figure out that it is a false-positive and not a "real" failure.

in reality. Therefore, we concentrate on qualitative statements (↑↗→↘↓) summed up in a guideline matrix. This is sufficient, because project managers can deal with vagueness. Finally, in Step 3, "Decision," one has to choose a concrete method.

The impact factors of the analysis step fall into two groups: Context impact factors and DBAP impact factors. **DBAP impact factors** address the specifics of a DBAP and its evolution over time. The first aspect of this group is the test data *compliance level*. The project has to determine which level it needs and what the differences between the levels are; that is, whether in a specific case schema compliant data is also application compliant. The second impact factor is whether there are *many or infrequent* (execution) *paths*. In other words, how likely is it that randomly selected data is sufficient for testing all paths, including the ones needing specific data constellations—or do we need some extra effort or a different data set for each execution path? Test data quantity concentrates on how much data we need for test cases. Are two rows sufficient or do we need thousands? The firmness factors relates to the maintenance of test cases. *GUI/Interface firmness* is about how often and how much the GUI or other external interfaces change from one DBAP version to the next. *Data model firmness* concentrates on how often and how much the data model in the database changes.

**Contexts impact factors** capture the influence from outside the project. Here, the first factor is the *role of the data responsible*. Testers have different backgrounds. Software engineers know development tools well. Test engineers have good testing methodology know-how. The effort for installing an environment (*set-up costs*) and keeping it up and the test cases à jour (*maintenance costs*) are further impact factors. *Privacy* is a concern e.g. for banks or hospitals. If it is a concern, it is mostly a killer criterion: No bank wants customer data finding the way into newspapers via leaks in the test process. Finally, there is the aspect of *environment complexity*. The question is whether the DBAP is a stand-alone application or connected with many satellite systems. In the latter case, one has to analyze, for example, whether the DBAP requires test data fitting to the satellite systems or whether loading test data into the satellite systems is an option.

# 6. Guideline Matrix

Our methodology should support commercial projects when choosing a DBAP test data provisioning method. The outcome of applying this methodology to a concrete project is a decision for using one method. Once we have analyzed all impact factors, our guideline matrix comes into play (Table 3). It is a qualitative analysis of which impact factors favor which method. The rows of the matrix represent the methods, and the columns are the impact factors. The arrows in the cells are qualitative characterizations of how favorable the impact factor is for choosing this method. In the following, we elaborate the matrix. Our discussion is structured according to the methods. In the end of this section, we illustrate the usage of the guideline matrix with a small example.

## 6.1 "Hand-crafted" Test Data

Hand-crafted test data is data a person designs after analyzing the DBAP and the testing needs. The data can be inserted manually via the GUI, using capture & replay technologies, or by coding INSERT scripts (Table 3, ❶, ❷, ❸). If the tester designs the data correctly, it is path compliant. If the tester makes mistakes, the data is at least application compliant if it is inserted via GUI. In case of INSERT scripts with mistakes, the data might be only type compliant (before the load) or schema compliant (if we look at the loaded data). If there are many paths to be considered, hand-crafted data becomes expensive. Further, the tester has to be aware of all the paths in which he is interested. This might lead to problems in case of infrequent paths. However, hand-crafted data is efficient for constructing test data for a limited number of workflows.

Hand-crafting test data becomes too expensive when many tables have to be filled with many rows (*mass test data*). *Privacy* is no concern, because the method does not use live system data. Writing SQL scripts is easy for software developers, and it is acceptable for test engineers. However, it is not suitable for power users. Often, the latter—or even the latter two—do not have the necessary tools installed and can miss access rights and database rights.

We see the typical influence of GUI and internal data model changes: Human testers deal with changes easily (❶). If we insert the data via the GUI automatically using capture & replay, there are adoption costs in case of GUI changes. Due to the GUI input, internal data model changes are irrelevant (❷). In contrast, data written directly into the database by SQL insert statements are not affected by GUI changes but by data model changes (❸).The set-up costs are mainly the efforts for specifying database test data for the different test cases. The set-up costs for the tools (and archiving the captured inputs) are added to that for the capture & replay method. In case of the manual GUI input, the follow-up costs are tremendous; one has to input the test data every time a new testing environment is set up. The capture & replay method is also relatively expensive due to the need to maintain the captured inputs and adapt it to changes. INSERT scripts are still not the least expensive option but are more stable and easier to use. Thus, all three methods that rely on hand-crafted data have similar but nevertheless slightly different characteristics.

## 6.2 Generators for Synthetic Test Data

Test data generators (Table 3, ❹) promise to reduce the manual effort for test data design. Many commercial tools are on the market, e.g. the Datanamic DB Data Generator [16]. Most are inexpensive with single-user licenses under $1,000. The process of choosing a tool is more expensive than the license itself. Tool usage typically consists of three steps: First, the tester chooses the tables for which she needs data. In the second step, she specifies how to generate the values for each column. This is a particular strength of the commercial tools. They come with predefined lists (ZIP codes for different countries, names, etc.), allow the user to create (and use) its own lists, or reuse values of other tables. They also provide random generators for simple numeric values or complex strings. The tools usually deal with not null, unique, primary, and foreign key constraints. Check constraints are hardly ever considered. In Step 3, the tools generate and load the data. Many of them determine the optimal insertion order needed due to foreign keys in this phase. However, cyclic dependencies are problematic. If the database schema has check constraints, the data is only type compliant. Type compliance implies that some rows might be rejected. A tester specifying that he wants 50 rows for a table might get only 45. If a tool continues generating rows

until 50 rows are loaded, a (nearly) infinite number of tries might be needed for difficult constraints.

If the tester wants application or path compliant data, he must specify carefully how to generate the values for each column. The tester must consider intra-table and inter-table dependencies *not* covered by constraints. If the specification is not 100% correct, the data is not application compliant. The result might be false positives. On the positive side, the tools generate a massive number of different rows quickly. If many rows are generated, and the test considers all data in the tables (batch-processing), even paths are covered nobody is aware of. If one needs data fitting a specific constellation (such as a real estate company with a balance sheet total of $10 million), it takes effort to identify the data. Finally, generated data does not raise privacy concerns even if testing is done offshore.

| | | Quality | | | | | Costs | | | | | | |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| | | | Many/ Infre-quent Paths | | | | Suitability | | | Firm-ness | | Manual Effort | |
| | *Methods* | Test Data Compliance Level | Batch | Workflow | Mass Test Data | Privacy | SW Engineers | Testers | Power User | GUI/ Interface | Data Model | Set-Up | Maintenance |
| ❶ | Manual GUI Input | a→p | ↘ | ↑ | ↓ | ↑ | ↑ | ↑ | ↑ | ↑ | ↑ | → | ↓ |
| ❷ | Capture & Replay | a→p | ↘ | ↑ | ↓ | ↑ | ↗ | ↑ | → | → | ↑ | ↘ | ↘ |
| ❸ | INSERT Scripts | t/s→ p | ↘ | ↑ | ↗ | ↑ | ↑ | → | ↓ | ↑ | ↘ | → | → |
| ❹ | Data Generator | t (→p) | ↗ | ↘ | ↑ | ↑ | ↑ | ↗ | ↘ | ↑ | ↗ | ↑ | ↑ |
| ❺ | Snapshot | a→p *) | ↑ | → | ↗ | ↓ | ↑ | → | ↓ | ↑ | ↗ | ↑ | ↑ |

**Table 3: Guideline Matrix for choosing a Provisioning Method**
[*) with respect to old release]

If—and only if! —the false positive challenge is solved and application or path compliant data is derived, a project benefits from the advantages of the tools. They are easy to use. Testers and developers can learn to use them efficiently in less than an hour. However, some understanding of the data model is needed, which can be a hurdle for testers and is unacceptable for power users. GUI changes have no impact. The tools deal easily with schema changes. The latter holds true as long as few inter-table or intra-table dependencies exist that are not reflected, for example, by foreign keys. Lastly, the tools have low set-up and maintenance costs for test data specifications.

## 6.3 Life System Snapshots

From the moment the first user works with the DBAP, its database tables fill up with data. The data originates from "normal" usage with the DBAP's GUI. Thus, the data is application compliant.[2] Even better, the database contains (path-compliant) data for all standard and exceptional situations of "real" usage. No tester has to be aware of them. This is interesting for testing batch processes such as the rating calculation procedure P_CALC_RATINGS. If one tests specific workflows, suitable rows must be identified first in the large data set. A particular strength of life system snapshots is test data provisioning for complex systems such as ERP systems. They have hundreds or thousands of tables. Only snapshots provide sensible data with limited effort. However, from a quality point of view, there are two aspects to consider: the version and the privacy challenge.

Privacy is a serious concern for banks or hospitals. There are academic approaches (e.g., [17] and [18]) or commercial tools (e.g. Datamaker Data Mask [19]) addressing the privacy challenge. However, often one uses the term "anonymization" though it is more a kind of "veiling" (see [20] for example). *Anonymization* guarantees that nobody can figure out to whom the data belongs, for example, to which company a rating belongs. *Veiling* means that it is not obvious to whom the data belongs. Veiling is often sufficient for internal testing. The first one is a prerequisite for off-shore tests with bank data. In contrast to veiling, anonymization is difficult or even impossible. Assume that one replaces the names in table T_CUSTOMER with a random string. This looks like anonymization. In fact, it is veiling. A financial statement is a fingerprint for a company. But if we change the financial statement to achieve anonymization, we have to recalculate the ratings. Otherwise, the data is inconsistent and could never appear this way in the application. The data would not be application compliant and might cause false positives. As a consequence, one should analyze carefully whether one needs veiling or anonymization and what one really gets.

The version challenge addresses that snapshot data comes from a DBAP version *in use*. When users work with version 1.0, we get version 1.0 data. If we want to reuse version 1.0 data for version 1.1 tests, we must upgrade the data. This looks like extra costs but is usually free. A vendor must develop such upgrade functionality for its customers. A bank never accepts that it looses the rating history when upgrading to version 1.1 of a credit rating application. The challenge is developing the upgrade early enough. False positives appear if the upgrade mechanism is not correct. Then, the data becomes corrupt. The DBAP might crash. The important point is that a false positive is a false positive only regarding testing the new release. It is a "real" failure of the upgrade process. It has to be solved anyway.

Life system snapshots are easy to use for developers, especially if they have database experience. Testers might not be too familiar with database import and export tools like Oracle's data pump. They need more training and support in unforeseen situations.

[2] It might even be too much data, e.g. in case of billing systems of telecommunication companies or data warehousing. Thus, one has to shrink the data without loosing coverage. Suárez-Cabal, et al. [10] present ground-breaking ideas for doing this automatically. In practice, it is currently a manual step.

Snapshots are not feasible for power users. GUI changes are irrelevant because snapshots work on the database level. Data model changes are also irrelevant, because the vendor need anyway upgrade procedures coping the changes. Finally, the set-up and maintenance costs for such a solution are low.

## 6.4 Example

We conclude this section with applying the results of the discussion (summarized in Table 3) to our sample credit rating sample application. We make some further assumptions about the application. First, the data model and GUI are stable. Second, testing is done offshore. Third, we test due to new rating business logic: a new workflow and modified calculations. The first column of Table 3 lists the provisioning methods: "Manual GUI Input," "Capture & Replay," "INSERT Scripts," "Data Generator," and "Snapshot." We want to test workflows, so we need path compliant data. Column "Quality" indicates that "Data Generator" produces (in the best case) schema compliant data. Thus, we exclude this method. It is a banking application tested offshore and privacy is a concern, so column "Privacy" indicates that "Snapshots" is also not an option. Therefore, three options remain: "Manual GUI Input," "Capture & Replay," and "INSERT Scripts." Deciding whether "Manual GUI Input" or "Capture & Replay" is better depends on the GUI stiffness and the set-up costs for "Capture & Replay." Our assumption was that the GUI does not change often; thus, "Capture & Replay" is better than "Manual GUI Input." Now, two options are left: "Capture & Replay" and "INSERT scripts." Our final decision would depend on the knowledge of the tester: How familiar is she with the internal data model and the validations?

## 7. Summary and Outlook

One of the basic problems of testing DBAPs is the need for DBAP test data. The main methods are easy to list: designing test data manually and (a) inserting it manually via GUI, (b) capture and replay a manual insertion via GUI, or (c) code an INSERT script writing data straight into database tables. Other methods are generators for synthetic test data or using database snapshots. Many factors decide which method is the most suitable. The challenge is balancing quality and costs.

Test data compliance levels—type, schema, application, and path compliance—allow assigning test data a concrete quality level. The cost estimation is more complex. We contribute a compilation of the main influence factors. The factors fall into two groups: DBAP-specific factors (e.g., firmness of GUIs and internal data model) and context factors. Sample context factors are privacy needs or whether power users or testers are responsible for test data. To make the abstract decision process with the various impact factors useful for commercial projects, we introduced the guideline matrix. It concentrates on the most relevant influence factors and provides a qualitative statement of which factors favor which provisioning method.

However, our methodology does by no means address all DBAP test data challenges. Many questions remain. One example is to move on from a qualitative guideline matrix to a quantitative matrix. Obviously, this would require broad empirical research. A second open challenge comes when broadening the view. This paper deals with database test data for *one* DBAP. The application landscapes in companies consist of hundreds of DBAPs, and workflows often span many of them. Therefore, we also need solutions for consist test data for systems with many DBAPs. Obviously, this is a big challenge.

**Acknowledgements:** The author would like to thank Michael Mlivoncic for the valuable discussions.

## 8. REFERENCES

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

[2] Zhu, H., Hall, P., May, J.: Software Unit Test Coverage and Adequacy, ACM Computing Surveys, Vol. 29, No. 4, 1997

[3] Perry, W. E.: Effective Methods for Software Testing, 3rd edition, Wiley Publishing, Indianapolis, IN, 2006

[4] Willmor, D. and Embury, S.: An Intensional Approach to the Specification of Test Cases for Database Systems, ICSE'06, Shanghai, China, May 20-28, 2006

[5] Deng, Y., Frankl, P. G., Chays, D.: Testing database transactions with AGENDA. ICSE'05: 15-21 May, 2005, St Louis, MO

[6] Dai, Zh., Chen, M.-H.: Automatic Test Generation for Database-Driven Applications, SEKE'07, July 9-11, 2007, Boston, MA

[7] Houkjær, K., Torp, K., Wind, R.: Simple and realistic data generation, VLDB'06, September 12-15, 2006, Seoul, Korea

[8] Binning, C., et al.: MultiRQP – Generating Test Databases for the Functional Testing of OLTP Applications, DBTest'08, Vancouver, Canada, June 13, 2008

[9] Binning, C., Kossmann, D., Lo, E.: Towards Automatic Test Database Generation, IEEE Bulletin on Data Engineering, Vol. 31(1), 2008

[10] Suárez-Cabal, M., Tuya, J.: Using an SQL Coverage Measurement for Testing Database Applications, SIGSOFT'04/FSE-12, Oct. 31–Nov. 6, 2004, Newport Beach, CA

[11] Kapfhammer, G., Soffa, M.: A Family of Test Adequacy Criteria for Database-Driven Applications, ESEC/FSE'03, September 1–5, 2003, Helsinki, Finland

[12] Haller, K.: White-Box Testing for Database-driven Applications: A Requirements Analysis, DBTest'09, Providence, RI, 29.6.2009

[13] Haller, K.: Test Data Provisioning for Database-Driven Applications, BNCOD'10, Dundee, UK, June 29 – July 1, 2010

[14] Selenium. http://seleniumhq.org/

[15] Data Pump in Oracle® Database 11g: Foundation for Ultra High-Speed Data Movement Utilities, Oracle, 2009

[16] datanamic DB Data Generator, http://www.datanamic.com/datagenerator/index.html

[17] Terrovitis, M., Mamoulis, N., Kalnis, P.: Privacy-preserving Anonymization of Set-valued Data, VLDB '08, August 23-28, 2008, Auckland, New Zealand

[18] Zhong, Sh., Yang, Zh., Wright, R.: Privacy-Enhancing k-Anonymization of Customer Data, PODS 2005, June 13-15, Baltimore, MD

[19] Datamaker Data Mask, www.grid-tools.com

[20] Anderson, N.: "Anonymized" data really isn't—and here's why not, Ars Technica, www.arstechnica.com, 8.9.2009