10

June 2010

testing experience

The Magazine for Professional Testers

printed in Germany

print version 8,00 €

free digital version

www.testingexperience.com

ISSN 1866-5705

Performance Testing

The Test Data Challenge for Database-Driven Applications

by Klaus Haller

It was the summer of 2008. Testing suddenly emerged as *the* topic in my professional life. I became responsible for the database back-end of a credit rating application. Banks use such applications for estimating whether companies pay back their loans. This impacts whether they get a loan and for which interest rate. So I was interested in how to test "my" application. But the application was different to the ones you read about in testing literature. The application was built on a database (a database-driven application, or, short, a DBAP). The DBAP output of an action not only depends on your input, but also on the history of the DBAP's usage. The history manifests in rows and data stored in the tables. And this history influenced the present and future behavior of "my" application. What sounds like a philosophical question is of high practical relevance. Four questions sum up what I had to answer myself: What do I need DBAP test data for? When is a DBAP "correct"? What is a DBAP test case? And, finally, which test data is "good"?

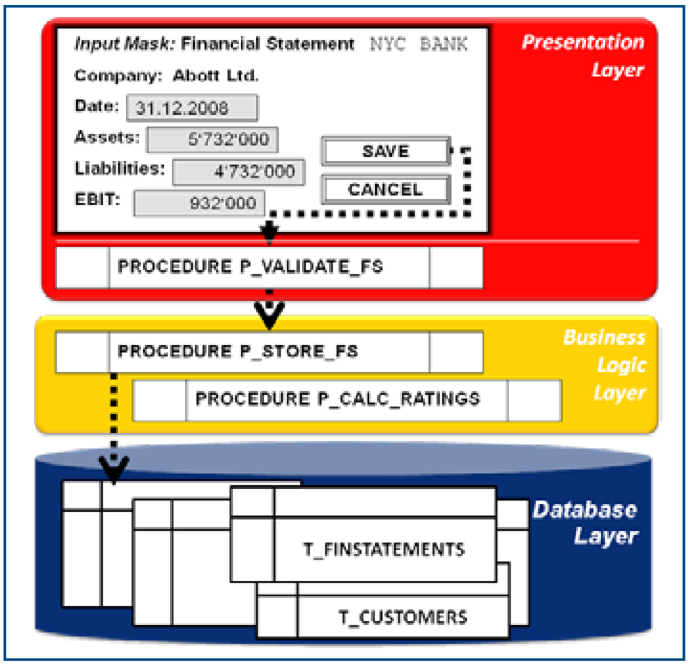


Figure 1: Sample Credit Rating Application

These questions are the roadmap for this article. A simplified credit rating application (Figure 1) will serve as an example. It is a three-tier application. The database layer stores the data persistency in tables. Table T_CUSTOMERS contains the information about the customer companies. Table T_FINSTATEMENTS stores financial statements for the various companies. The middle layer contains the business logic. It has two procedures. Procedure P_STORE_FS stores a financial statement in the database. Procedure P_CALC_RATINGS calculates the rating for all customers. On top, there is the presentation layer. It provides the GUI input form for manual input of financial statements. Also, it comprises the procedure P_VALIDATE_FS. The procedure checks whether the financial statement inserted into the GUI is "sensible", e.g. whether the sum of all assets and liabilities are equal.

DBAP Test Data Cube

Conventional tests know only one kind of data: data used as input parameters for the procedure to be invoked[1]. It is the idea of the DBAP test data cube (Figure 2) to visualize why and what kind of test data testers need. The cube has three dimensions: test trigger, test stage, and data purpose. The first dimension is the **test trigger**. It represents why we test. This issue is discussed intensively in literature. So we point out only five main reasons. It can be a completely *new application* that has never been tested before. When our credit rating application comes to the market the first time, the trigger is "new application". The subsequent releases trigger tests for reason two: a *new release* of an existing application. We test the new functionality and do regression tests for the existing functionality. Thus, if we add rating functionality, we need data for testing the new functionality as well as data for regression testing the existing functionality. A new application or a new release are two test triggers for software vendors. Customers buying software have other reasons for testing: parameterization and satellite system interaction. *Satellite system interaction* reflects that most of today's applications run in complex application landscapes. Processes span many applications. Various applications have to interact. Thus, we test the interaction of an application with its satellite systems. The fourth test trigger is *parameterization*. Standard software such as SAP or Avaloq allows adapting the software and the workflows according to specific customers' needs. One might parameterize e.g. that three persons have to check the rating for loans over ten millions. Whereas the customer can trust the standard software, the customer has to test whether its parameterization works as intended. Finally, the underlying *infrastructure* can trigger regression tests, e.g. when operating systems or compilers change.

The second dimension of the DBAP test data cube represents the

1 *„Invoking a procedure" is a terminology typical for unit tests. However, it is meant in an inclusive way, i.e. this term also includes GUI based actions with input and output values inserted or presented via the GUI.*

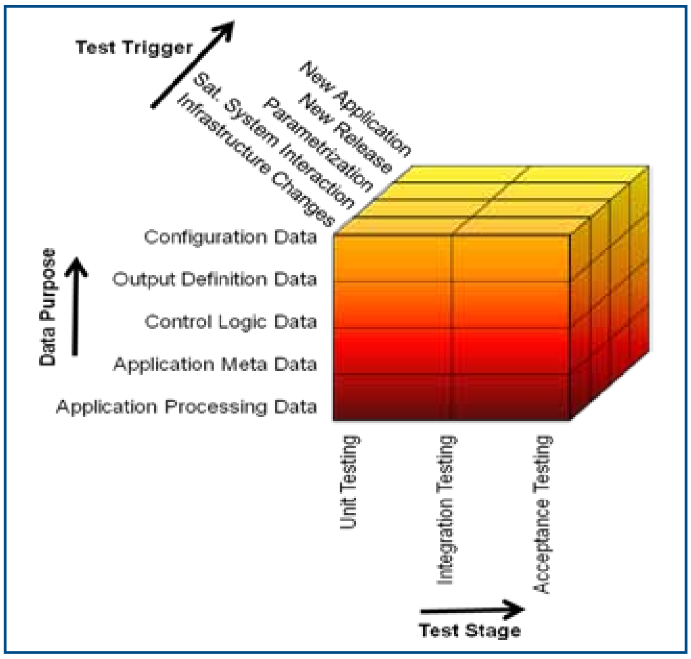


Figure 2: DBAP Test Data Usage Cube

test stages. Test stages are the different steps in testing addressing different needs and can be done by different units of the company. Typical stages are unit tests, integration tests, and acceptance tests. The exact steps and their names often depend on the process of the company.

The last dimension is the **data purpose**. It is the role the data plays in the DBAP. The actual roles might depend on the concrete architecture. We have identified the following data purposes as especially important in our applications: *Application processing data* is data everybody is aware of at first glance. Customers and their financial statements are examples in the context of our credit rating application. *Application meta data* is data which is more or less stable, but influences the "normal" data processing. An example is tables with credit pricing data. The data determines the risk-adjusted interest rate based on the credit score of a company. If the company has a scoring of "medium", the interest rate might be 9.2%. If the score is "excellent", the interest rate might be 3.7%. *Control logic data* influences the execution of processes and workflows. An example would be the ten million limit for loans. Loans over ten million demand three persons to check the rating. *Output definition data* defines the design and appearance of reports and customer output. The bank name "NYC BANK" or a bank logo are examples. Finally, *configuration data* deals with the IT infrastructure the application is deployed to. Examples are the configuration of interfaces, e.g. to SWIFT.

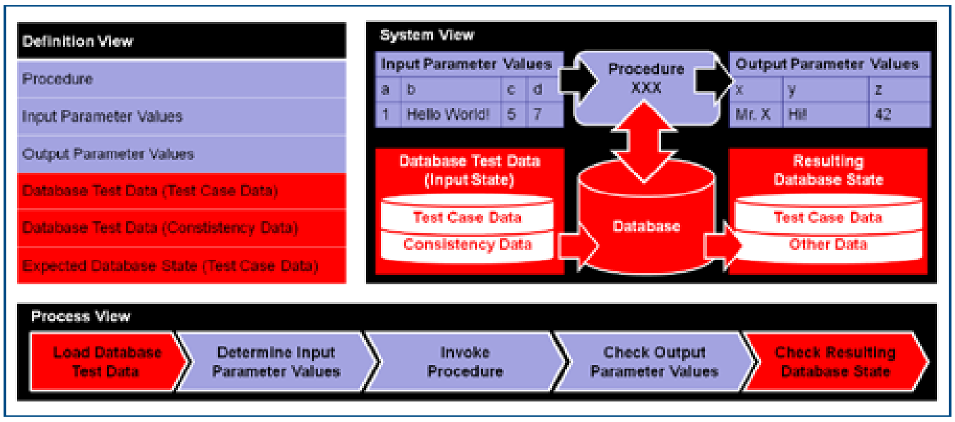


Figure 4: Understanding DBAP Test Cases (conventional test case: blue, DBAP extension: red)

DBAP Correctness

Testing aims at finding as many bugs as possible as early as possible in the development life-cycle. The DBAP shall be or shall become "correct". But correctness in the context of DBAPs has various meanings. A data architect, a database administrator, a software developer, and a tester might focus on completely different aspects. To point this out, we present three possible DBAP correctness concepts (Figure 2): schema correctness, conformity correctness, and application correctness.

Schema correctness focuses on the database schema the DBAP uses for storing its data (green). Schema correctness understands correctness as having (a) a specification that reflects the real world and (b) an implementation reflecting the specification and the real world. Our credit rating application stores financial statements in the database schema. Schema correctness means in this context: First, there is one table (or more) for storing the financial statements. Second, the table has attributes for all the information provided by financial statements. Third, the financial statements must refer to the companies they belong to.

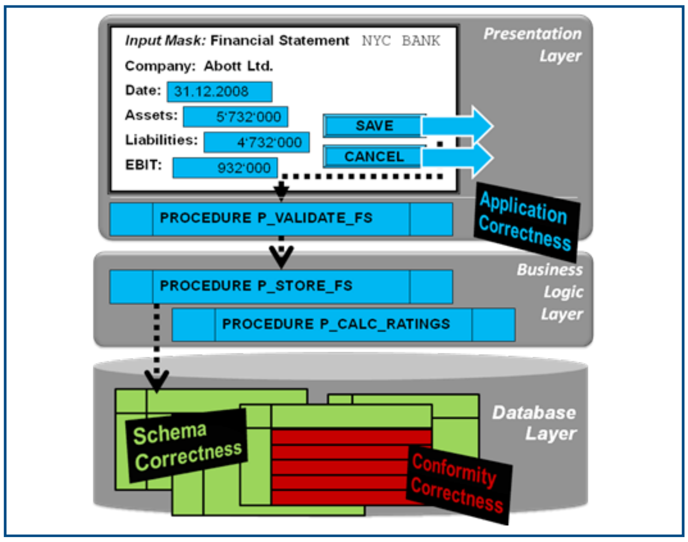


Figure 3: DBAP Correctness Criteria

Conformity correctness (brown) focuses on constraints or dependencies which are not part of the schema. The dependencies between balance sheet positions and profit-and-loss accounts are a good example. They are too complex to be reflected by database constraints. In our example, there are also no constraints in our database enforcing that the sum of all assets and all liabilities are equal. The data (and the DBAP) is only conformity-correct if it reflects also these non-schema-enforced constraints. Conformity correctness is similar to the concept of assertions in programming languages such as Java. Assertions do not improve the quality by looking at the result of actions, but by ensuring that the preconditions are as they have to be. Whereas these two correctness criteria focus only on the database, the third criterion, **application correctness**, looks at the complete DBAP behavior as observed e.g. via the GUI. However, it makes sense not to concentrate only on GUI interactions. Also batch processes such as the rating calculation procedure P_CALC_RATINGS are relevant. Application correctness is the most intuitive DBAP correctness criterion. Thus, we rely on it for discussing DBAP test cases.

Test Cases

Literature defines test cases based on three elements: the procedure to be invoked and the input and output parameter values. This model is suitable for stateless applications such as a calculator. A calculator returns *always* "7" after pressing "2", "+", and "5."

We need to adopt the test case concept for DBAPs, because they are not stateless. Procedure P_CALC_RATINGS needs financial statements to operate on. Procedure P_STORE_FINSTATEMENT needs a customer who a new financial statement refers to. So we need an initial database state. Also, these procedures show that we cannot rely only on the GUI output for deciding whether a procedure works correctly. We have to check the database whether a new financial statement was added to the database (P_STORE_FINSTATEMENT), or whether the rating calculations are correct (P_CALC_RATINGS). So a DBAP test case consists of five elements: input parameter values, an input database state, a procedure to be invoked, output parameter values, and a resulting database state.

This model was first presented by Willmor and Embury. We extended it for practical usage for complex systems by distinguishing two parts of the input state. ERP systems or core-banking-systems have hundreds or thousands of tables. One table might be relevant for our test case, but we need the other hundreds or thousands to be filled such that we can perform our test case. Thus, we divide our input state into two parts, test case data and consistency data. We illustrate this with procedure P_CALC_RATINGS. Here, we want to test whether the rating function works correctly, e.g. whether a bankrupt company gets rating "0". So we need test case data in table T_FINSTATEMENTS. This test data must contain a financial statement of a bankrupt company. However, we can add such a financial statement if we can link it to a customer in table T_CUSTOMERS. Thus, the customer would be consistency data. After execution of the procedure, we might have to check whether the rating is as expected. Thus, we look at data in the database. Again, there are two kinds of data. There is data we are interested in (test case data), e.g. the rating information in T_FINSTATEMENTS. We can ignore all other tables in this particular case, because it is not in the focus of this test case.

Figure 4 compares a conventional test case and a DBAP test case. *Blue* reflects what a conventional test case definition contains, the involved system components, and which actions (input parameter selection, invocation, checking the outcome) have to be done. The needed extensions for a DBAP test case are shown in red. These are the input and output states, the database, loading the database before the test case execution, and, potentially, checking the resulting state.

Quality

All roads lead to Rome. And in projects many ways lead to DBAP test data. One can design them by analyzing the specification. One might use commercial data generation tools. The decision often depends (besides on costs) on the test data quality. If we want to compare the quality of different DBAP test data, we need a notion of quality. In other words: We have to understand what DBAP test quality means and how it can differ.

T_FINSTATEMENTS			T_CUSTOMERS	
OWNER_ID	SUM_AS	SUM_LI	ID	NAME
NUMBER	NUMBER	NUMBER	NUMBER	VARCHAR2(100)
67	120000	12000	55	ALICE CORP.

CONSTRAINTS:
PRIMARY KEY(OWNER_ID);
FOREIGN KEY (OWNER_ID)
REFERENCES T_CUSTOMERS(ID);
SUM_AS NOT NULL;
SUM_LI NOT NULL;
CHECK(SUM_AS>0);
CHECK(SUM_LI>0);
			67	BETTY LTD.

Figure 5: Sample Tables with Constraints

Therefore, we rely on the concept of *test data compliance levels*[2]. The compliance levels (Figure 5) are like a stair with four steps. It requires effort to get to the next step, but you gain quality. The lowest level is **type compliance**. Type compliance considers the data type of the columns. Table T_FINSTATEMENTS has three columns: one stores the sum of all assets, one the sum of all liabilities; an ID column refers to the customer ID in table T_CUSTOMER. The reference refers to the company that the financial statement belongs to. Type compliance demands that we insert only rows for which all attributes have the right type.

We take a look at the following three INSERT statements for table T_FINSTATEMENTS (Figure 5):

```
(1) INSERT T_ FINSTATEMENTS(OWNER_ID, SUM_AS, SUM_LI)
      VALUES('ALICE CORP.', 'ONE MILLON', 'NO INFORMATION');
(2) INSERT T_ FINSTATEMENTS(OWNER_ID, SUM_AS, SUM_LI)
      VALUES (55, -50'000,  NULL);
(3) INSERT T_ FINSTATEMENTS(OWNER_ID, SUM_AS, SUM_LI)
      VALUES (32, 23'000, 20'000);
```

Statement (1) does not reflect that the attributes must have the data type NUMBER. It is not type compliant. Statements (2) and (3) are type-compliant. However, statement (2) does not make sense. It does not reflect the schema constraints. A NULL value is not allowed for attribute SUM_LI. Also, there is no customer with ID 55 in table T_CUSTOMERS. Next, the check constraints demand that the values for the sum of all assets and liabilities are positive. The problem from a testing perspective is that all rows not complying with a constraint are rejected by the database. So if we prepare 50 type-compliant rows, we do not know whether 50, 45, or 0 rows make it into the database. However, statement (3) reflects this requirement, as does statement (4). Thus, we use the term **schema compliance** for statements (3) and (4). The advantage compared to only type-compliant data is the guarantee that all rows are loaded into the database.

```
(4) INSERT T_ FINSTATEMENTS(
           OWNER_ID, SUM_AS, SUM_LI)
     VALUES (32, 20'000, 20'000);
```

We can achieve the two previous compliance levels "type compliance" and "schema compliance" relying only on information of the database catalogue[3]. The two highest compliance levels need more information. From an application point of view, the sum of all assets and liabilities is always equal. SUM_AS and SUM_LI must be equal. This is not reflected by the database schema. In the case of GUI input, procedure P_VALIDATE_FS ensures this. Otherwise, the procedure rejects the GUI input. So we have dependencies between attributes, which are enforced by the application and not reflected by constraints. Such dependencies can also exist between tables, e.g. one table with financial statements and a table with profit and loss information. The problem with dependencies not reflected by schema constraints is that there might be data that has been inserted in the database which does not reflect these dependencies. Thus, the DBAP might be in a state that was not specified. The consequence can be unexpected behavior of the application. If errors emerge for unspecified circumstances, they are false positives[4]. Such circumstances would never appear under normal usage. So our third level is **applica-**

2 *K. Haller: White-Box Testing for Database-driven Applications: A Requirements Analysis, Second International Workshop on Testing Database Systems, Providence, RI, 29.6.2009*

3 *The database catalogue are dedicated tables in a database which store all information about the content of the database: users, tables, constrains, views, access rights etc.*

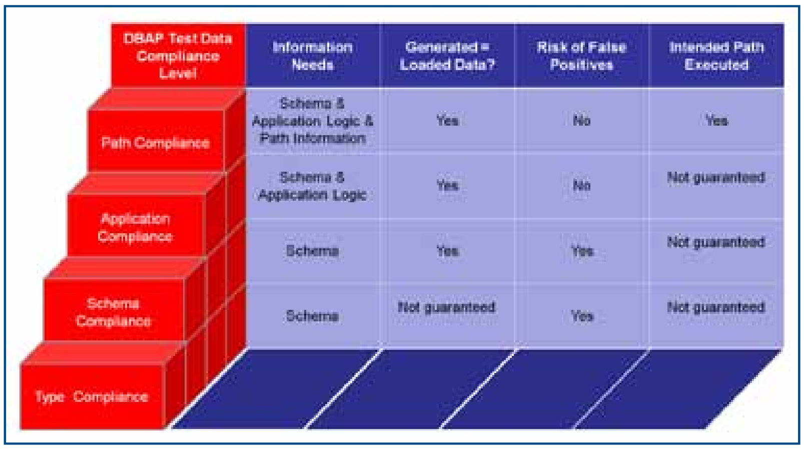
4 *False positive means that testers seem to have found a failure. After an analysis by the testers or software developer the failure turns out not be a "failure". Certainly, they are costly. If they appear too often, one risks that software engineers might stop taking "failures" seriously. They might assume all failures to be false positives and stop analyzing potential failures in a sensible way.*

Database Constraints restrict which data can be stored in the tables of a database. There are the following constraints: *Unique constraints* state that this attribute must be different for all rows of a table. ID columns are a good example. *Primary key* constraints allow identifying a row. They can span more than one attribute. *Foreign keys* refer to primary keys of a different table. They ensure that there is a fitting value in the different table, e.g. a financial statement refers always to an existing (!) customer ID in the customers table. *Not null constraints* demand that there is a value provided for this attribute, e.g. that all financial statement have a sum of assets or a sum of liabilities. *Check constraints* allows formulating nearly arbitrary conditions.

tion compliance. Application compliance means that the DBAP input state could be the result of "normal" GUI input and data processing. Statement (4) is a perfect example. Now no false positives can appear. But still, test data can be better. Let us assume a test case shall test the credit rating functionality for large corporations with assets and liabilities of more than 1'000'000. We are interested whether procedure P_CALC_RATINGS considers specific risks of large companies. This test case requires a financial statement with "higher" values such as statement (5).

```
(5) INSERT T_ FINSTATEMENTS(OWNER_ID, SUM_AS, SUM_LI)
      VALUES (32, 1'500'000, 1'500'000);
```

The difference between statements (4) and (5) is that the latter allows us to test the execution path (or test case) we are interested in: the credit rating for large corporations. If the DBAP test data is suitable for a test case, it is **path-compliant** (the term refers to the path coverage criterion). Path compliance bases always on a test case and a specific execution path of the application. This is the highest level we can achieve. However, it also makes clear that different test cases might need different test data sets. Figure 5 compiles the information about all four compliance levels.



DBAP Test Data Compliance Level	Information Needs	Generated = Loaded Data?	Risk of False Positives	Intended Path Executed
Path Compliance	Schema & Application Logic & Path Information	Yes	No	Yes
Application Compliance	Schema & Application Logic	Yes	No	Not guaranteed
Schema Compliance	Schema	Yes	Yes	Not guaranteed
Type Compliance	Schema	Not guaranteed	Yes	Not guaranteed

Figure 6: Overview Compliance Levels

In this article, we explained the typical problems of testing database-driven applications. Instead of providing simple answers to complex problems, we concentrated on explaining the different challenges for projects. Our goal is to foster discussions within projects to find the best solutions for their particular problems. Thus, we explained the most important concepts: the test data cube, correctness, test cases, and quality for DBAPs.

Biography

Klaus Haller is a Senior Consultant with COMIT, a subsidiary of Swisscom IT Servies. COMIT focuses on projects and application management services for the financial industries. Klaus' interest is the architecture and management of information systems. His experiences span from process engineering, architecture and implementation of data migration solutions for core-banking systems to the responsibility for the database backend of a credit rating application. Klaus studied computer science at the Technical University of Kaiserslautern, Germany. He received his Ph.D. in databases from the Swiss Federal Institute of Technology (ETH) in Zurich, Switzerland. He welcomes your feedback at klaus.haller@comit.ch