12

December 2010

testing experience

The Magazine for Professional Testers

printed in Germany

print version 8,00 €

free digital version

www.testingexperience.com

ISSN 1866-5705

Open Source Tools

Release Upgrades for Database-Driven Applications: A Quality Assurance Perspective

by Klaus Haller

1. Motivation

The majority of today's business applications are database-driven applications (DBAPs). DBAPs rely on databases for storing and processing data. When a vendor plans a new release, business analysts, software engineers, and testers collaborate to incorporate new features and to solve bugs. "The more, the better" is a typical slogan. Thus projects often postpone one task for as long as possible: the development and testing of upgrades. Upgrades, however, are of crucial importance. Customers expect a smooth transition to the new release. They do not want to lose any data. This demands for high quality assurance standards. In practice, however, not only developers but also testers treat upgrades as unloved appendices. One reason might be that the literature does not provide guidelines. This article fills this gap. It provides insights into DBAP upgrades, their risks, and adequate testing strategies.

2. Understanding DBAPs and DBAP Upgrades

The article relies on a sample DBAP, a credit risk application, to discuss challenges in a practical context. Banks use credit rating applications when they decide whether a company gets a loan and at which interest rate. Figure 1 provides a simplified model of the application. In the middle, there is the current release 8.5, release 9.0 is on the right. Shown on the left is a generic DBAP model. Release 8.5 has two input forms for financial statements. One form is for small and medium enterprises (SMEs), one for large corporations. The forms provide input fields such as "assets and liabilities" and "EBIT". The business logic calculates the company's credit risk, i.e., the likelihood that the company might not pay back the loan, based on the financial statement.

The graphical user interface (GUI) of the application with the input forms and the business logic form together the DBAP **application logic** component (Figure 1, left). It runs typically in an application/web server (**classic application logic**). Certain parts, e.g. triggers or stored procedures, are in the database (**database application logic**). DBAPs rely on three components for storing data persistently in the database. First, there is the **table structure**. It defines the data model on the database layer. The tables are a kind of container for storing customer and vendor data. **Customer data** is data relevant only for one customer. The financial statements in tables T_FINSTATEMENT_CORP and T_FINSTATE-

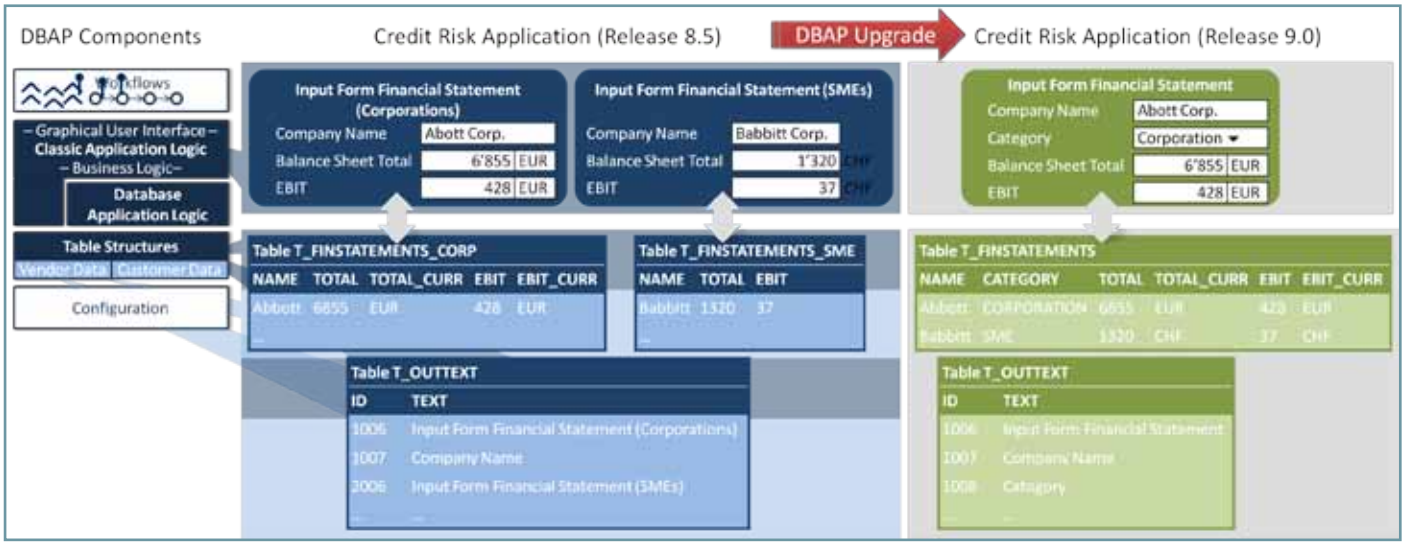


Figure 1: DBAP components (left) and sample application (release 8.5 - middle - and release 9.0 - right) with GUI (upper part) and financial statement input forms (middle and right, upper part) and corresponding database tables (lower part).

MENT_SME are examples. Table T_OUTTEXT stores the GUI names and descriptions for the input fields. These texts are shown on the GUIs. They are the same for all customer installations. This illustrates the idea of **vendor data**.

Besides release 8.5, Figure 1 incorporates also release 9.0. A **DBAP upgrade** allows switching to the new release, e.g. from release 8.5 to release 9.0. Release upgrades must overcome the differences between the two versions. Release 8.5 has two input forms, one for SMEs and one for large corporations. Both forms store their data in specific tables. In release 9.0, there is only one input form, and one table stores all financial statements. The application logic changes, too. Also, attributes, tables, and database schemas can be added, removed, or modified in a new version. A DBAP upgrade ensures that the customer can still use his old data with the new release.

3. Upgrade Correctness

Testing compares observed behavior with behavior defined to be correct. In "normal" application testing, the tester uses the applications and, thereby, enters data into input forms. If the application does not crash and the tester sees the expected output, the test is successful. Upgrades, and especially DBAP upgrades, are different. The upgrade routine must not crash. It should also return whether the upgrade succeeds. However, this does not mean that the upgrade is correct. An upgrade is only correct, if (a) the application works correctly after the upgrade, and (b) the old data is preserved and can still be used. This is an intuitive understanding, but testers need a precise understanding in order to test. The term *upgrade correctness* covers data and application component(s) aspects. Regarding most of the components, upgrade correctness bases on a comparison. It compares the upgraded DBAP with a **reference installation**. A reference installation is an installation of the new release (release 9.0 in our example). It is an installation as installed by new customers not having run any old version.

In more detail, **upgrade correctness** has four aspects (Figure 2):

1. The *application logic* of the upgraded DBAP equals the one of the reference installation. This must hold true for database application logic (triggers, stored procedures etc.) and the classic application logic, e.g., in Java/J2EE.
2. The *database tables* (the data model of the persistent data) are the same as for a reference installation.
3. The *vendor data* equals the data of a reference installation.
4. The *meaning of the customer data* is unchanged. In our example, all financial statements are still there. Also, their semantics must remain unchanged.

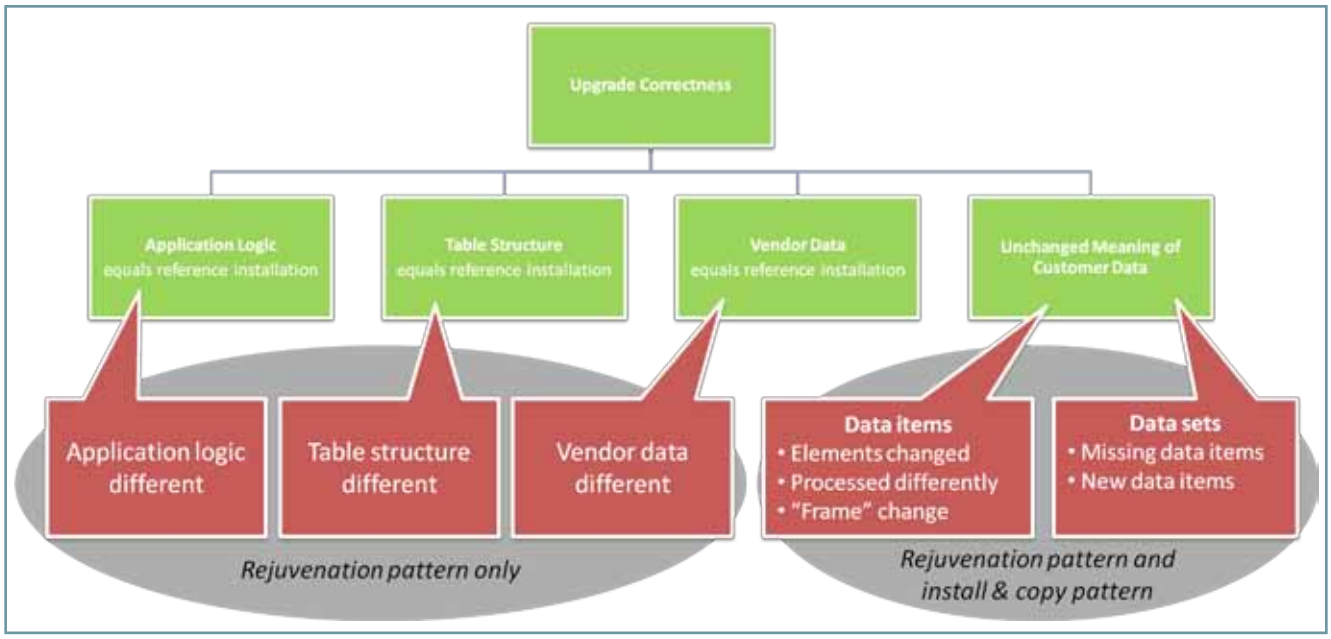


Figure 2: Upgrade correctness and upgrade risks

4. Upgrade Patterns

Software developers can use various patterns for implementing upgrades (see [1] for more details and a theoretical foundation). They have different risks. Testers need to understand the patterns and their risks for focused tests. This section presents the two main upgrade implementation patterns, the rejuvenation pattern and the install & copy pattern.

The **install & copy pattern** installs the new DBAP version in step 1 (Figure 2, 1). This is done the same way as for new customers. In step 2, the upgrade copies the data from the old installation. It transforms the data and loads it into the new installation (2). As a result, there are now two installations. One is the old one (release 8.5 in our example). It can be shut down and removed from the server in step 4. Before that, in step 3, the users switch to the second, the new installation. The only upgrade risk of this pattern is that the meaning of the customer data might be changed. All other components (application logic, vendor data, and table structure) are installed from scratch. Thus, they are correct by definition.

The **rejuvenation pattern** adopts the application logic. It removes obsolete and deploys new Java classes (Figure 2, A). It deletes, modifies, or adds database application logic such as triggers and stored procedures (B). It modifies database tables and adopts customer data accordingly (C). Finally, it updates the vendor data (D). So the rejuvenation pattern transforms the old installation into a new one. The pattern has various risks. First, the meaning of the customer data might change. This risk is the same as for the previous pattern. Other risks are that the application logic, the table

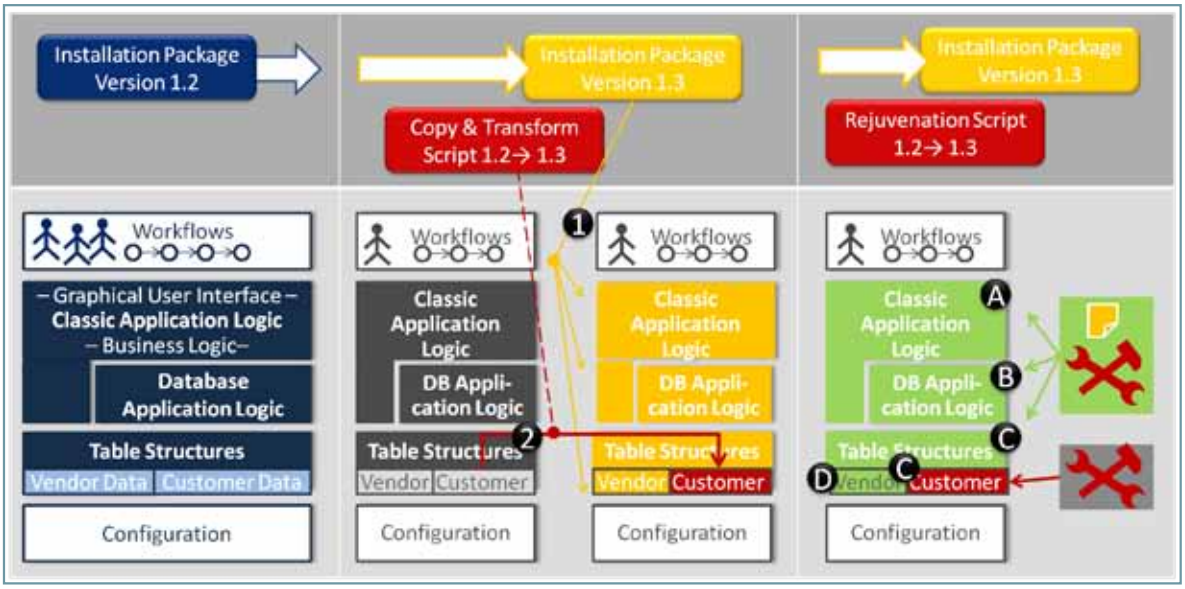


Figure 3: Upgrade patterns with the old version (left), the copy & transform upgrade pattern (middle), and the rejuvenation upgrade pattern (right)

structure, or the vendor data are modified incorrectly. Then, not all components of the DBAP equal the reference installation. The DBAP is inconsistent. It might return wrong results (e.g., wrong credit ratings) or simply crash.

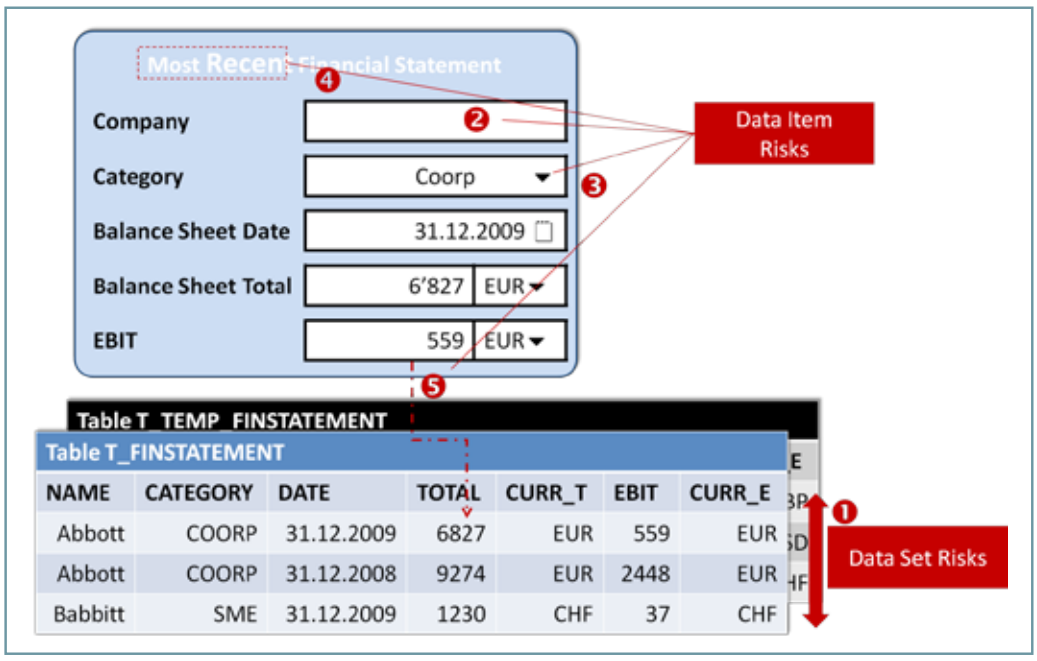
5. Understanding DBAP Upgrade Risks

The previous section mentioned briefly the upgrade risks. Three are easy to describe. Application logic, table structure, and vendor data must be the same as for a reference installation. If not, the upgrade is not correct. Customer data-related risks are more complex. They fall into two sub-categories. First, there are two **data set risks** (Figure 4, 1):

- *Missing data items*. One or more data items might get lost during the upgrade. There could be twenty financial statements in release 8.5, from which one is missing when the bank starts working with release 9.0 after the upgrade.
- *New data items*. The upgrade must not "create" financial statements which did not exist before the upgrade. This could happen, e.g. when certain financial statements are duplicated unintentionally.

The **data item risks** deal with the semantics of a single data item, e.g. one financial statement. There are three risk types:

- *Element changed*. One or more attributes are changed incorrectly. Attributes might get lost during extraction and copy. If the upgrade uses an INSERT-SELECT-SQL statement, it might not cover all needed attributes (2). This seems unlikely for our example with table T_FINSTATEMENTS_CORP or T_FINSTATEMENTS_SME. However, if there are one hundred or more tables, forgetting one attribute is not so unlikely any more. Also, the table structure of the new release can change till the last second. This requires changing the upgrade always accordingly. Regarding incorrect changes, an example would be if the currency for all SME financial statements is not set to CHF during the upgrade, but to USD.
- *Processed differently*. The category field is a good example how an additional attribute influences the processing of otherwise unchanged data (3). If one copies old financial statements for release 8.5 and does not fill up the new attribute CATEGORY, the business logic might use the wrong algorithm and calculate wrong credit ratings, or it might crash.
- *Frame change*. The data is the same on the database level and is (still) correct. However, the semantics can change due to (slightly) different GUI texts ("most recent financial statement" vs. "last year's financial statement" – 4), due to field name changes (e.g., "total" instead of "EBIT"), or due to being linked to a new GUI or differently to the existing one (5).



6. Testing in Practice

Vendors and customers test for different reasons. Vendors want to improve the quality of the upgrade. It must run with all customer installations, even if various installations differ sub-

Figure 4: Interpretation-related risks: data item risks and data set risks

tly. Customer-site tests ensure that the customer company can continue working. Companies cannot risk that an ERP system is down for a day or two. Banks cannot approve loans without credit rating. So upgrades are done in maintenance windows. The application management tests whether the company can *work* with the upgraded DBAP. If not, it restores the state before the upgrade. Thus, the business is not affected even if the upgrade fails. The good news is that the customer and vendor tests might have different objectives, but the needed techniques overlap.

The DBAP upgrade risks are the base for deriving the techniques and strategy for testing DBAP upgrades. They guide the testers when preparing test plans and running tests. Table 1 compiles a DBAP upgrade test strategy. It links the risks to the tasks to be performed. It states also whether automation is an option for vendor-site or customer-site tests.

Table 1: DBAP Upgrade Test Strategy

Patterns	Risk Type	Risk Subtype	Test Approach	Automation (Vendor-site)	Automation (Customer-site)
Rejuvenation only	*Application Logic Differences*	"Classic"	*As for normal upgrades*		
		Database	Schema comparison	Yes, full automation possible	Yes, full automation possible
	Table Structure Differences		Schema comparison		
	Vendor Data Differences		Counting rows/ calculating hash values		
Install & Copy and Rejuvenation	*Data Set Risks*	*Missing Data Items*	Reconciliation		
		New Data Items	Reconciliation		
	Data Item Risks	*Element change*	Checking all representatives	Questionable cost-benefit ratio	No (tests only needed once)
		Frame change	Checking all representatives	No	*Irrelevant*
		Processed differently	Execute *all* workflows with as much data as	Questionable cost-benefit ratio, option: smoke	Part of "normal" tests after setting up a system

There are the three risks specifically relating to the rejuvenation pattern: application logic differences, table structure differences, and vendor data differences. The database catalog helps addressing two: **database application logic differences** and **table structure differences**. It contains a list of all existing tables and their attributes. The vendor extracts this information from a reference installation and incorporates it into the upgrade. At customer-sites, when the upgrade has finished with all transformations, two validations take place. The first validation compares (i) the (database) application logic and table structure after the upgrade with (ii) a reference installation. It queries the database catalog of the DBAP for (i) after the upgrade. The information for (ii) is incorporated in the upgrade. A second validation checks **vendor data difference** risks. It counts, e.g. the rows per table and calculates hash values. The vendor benefits from the validations in two ways. First, developers get quick feedback during the upgrade development. Second, if run at customer-site, the vendor knows that the upgrade ran as expected. If the vendor has not considered all nuances of a customer installation, they are either detected by the checks, or they are irrelevant. If upgrade problems are not detected early, this can ruin the vendor reputation. Without remote access to the customer environment, the vendor support has no chance identifying bugs which are a result of such upgrade "hiccups".

Testing **data set risks** means matching data items of the old and new system. Data items without matching counterparties indicate errors. It is like looking for a (missing or an dditional) needle in a haystack. It is not about whether the needle head is blue or red. The checks do not consider any semantics. So automation is an option. A reconciliation is the technical solution. A *reconciliation* is a list comparing, per table, identifiers before and after the upgrade as Table 2 illustrates. The top four rows contain rows about loans. They are identified by their IDs. There are no matches for account 200540. It exists only in the new version. Contrary, account ID 755543 exists only before the upgrade.

In practice, a reconciliation must consider three advanced aspects:

- Certain data items are not relevant for upgrades, e.g. logs. Also, if modules or functionality is removed, corresponding data becomes obsolete. They are not looked at by the reconciliation.
- Changes of the table structure must be addressed. When the data of the two tables T_FINSTATEMENT_CORP and T_FINSTATEMENT_SME is copied into one table T_FINSTATEMENTS, the reconciliation must consider this. The artificial table name "OBJ_FINST" reflects this. It is used for elements in all three tables.
- If a table has no identifier, a good one can often be constructed by concatenating attributes, e.g. a company name and the balance sheet total in the example.

The data set risks are so crucial that vendors must address them during their tests. In case of business-critical applications, customers (or their auditors) might insist on such checks when the upgrade takes place at customer-site. Thus, the vendor should design a reconciliation as delivery object for the customer and run them after/with each upgrade. The concept of a reconciliation is discussed in more detail in [2], e.g., on how to integrate more attributes.

Risks which involve semantics are difficult to automate. They depend on human testers. This applies to two data item risks, the element change risk and the frame change risk. They are done at customer-site and vendor-site. Obviously, it is not possible to check every data item, but only a small selection. Thus, it is important to choose representatives. Testers must have a list of the ty-

Table 2: Reconciliation

Table T_FINSTATEMENTS			
TAB_OLD	ID_OLD	TAB_NEW	ID_NEW
		T_LOAN	200540
T_LOAN	422784	T_LOAN	422784
T_LOAN	522423	T_LOAN	522423
T_LOAN	755543		
OBJ_FINST	Abbott6855	OBJ_FINST	Abbott6855
OBJ_FINST	Babbitt1320	OBJ_FINST	Babbitt1320

pes of data items in the system, e.g. financial statements, customers, loans, etc. If there are subtypes with different behavior or processing, e.g. financial statement of a SME company versus financial statements of large corporation, this must be considered. Then, testers choose representatives for each data item type. The more complete and varied the set of representatives, the likelier it is that failures are detected. Practical experience proves, that all attributes of all input and output forms must be covered at least once.

Testers compare the data before and after the upgrade manually. They print out the relevant data before the upgrade and compare the print-outs with the data in the upgraded DBAP. In case of the install & copy pattern, testers can also log into the old and the new systems simultaneously. Then, they compare the data items by opening the relevant forms in both systems. This takes time, i.e. it is an expensive task. Especially for the upgrade development project, recurrent element change tests might be automated. This requires test data in the old installation, which is tested at GUI level after the DBAP upgrade. There is no such option for frame change tests.

The processed differently risk is trivial when one is aware of it. However, at the very end of a project, a tester had compared all representatives thoroughly. There was not a single error left when looking at the data in the GUI. By chance, one developer pushed a button starting some data processing of/with the upgraded DBAP. The whole application crashed immediately. The reason was a new mandatory attribute in a table. The GUI presented a default value if the attribute was missing. The application logic crashed. This illustrates that all workflows should be executed respectively tested at least once by a tester manually. Then, the tester can also check whether the returned results are as expected. This approach turned out to be highly effective. In case of longer upgrade projects, implementing smoke-tests for all important workflows can speed up the development process.

***

Developing and testing upgrades are crucial for customer satisfaction, especially for DBAPs. Customers expect, first, that the application runs as smoothly as before the upgrade. Second, all data must (still) be available. This is challenging. After reading this article, testers should understand three aspects better. These are, first, the DBAP upgrade patterns, second, the risk associated with the upgrade patterns (precisely: data set risks and data item risks), and, third, what kind of tests are needed. This knowledge makes testing DBAP upgrades more focused, more effective, and, hopefully, more fun.

References

[1] K. Haller: *On the Implementation and Correctness of Information System Upgrades*, International Conference on Software Maintenance (ICSM), 12-18 September, Timișoara, Romania

[2] K. Haller: *Data Migration Project Management and Standard Software*, Data Warehousing 2008: St. Gallen, Switzerland

Biography

Klaus Haller 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. Klaus has worked now for more than five years mainly in the financial industry. His focus is on information system architecture & management and business-IT alignment.