Transform!

Patterns for Data Migration

Andreas Rüping

Sodenkamp 21 A, D-22337 Hamburg, Germany
www.rueping.info
andreas.rueping@rueping.info

Abstract

When an existing application is replaced by a new one, its data has to be transferred from the old world to the new. This process, known as data migration, faces several important requirements. Data migration must be accurate, otherwise valuable data would be lost. It must be able to handle legacy data of poor quality. It must be efficient and reliable, so as not to jeopardise the launch of the new application. This paper presents a collection of patterns for handling a data migration effort. The patterns focus on the design of the migration code as well as on process issues.

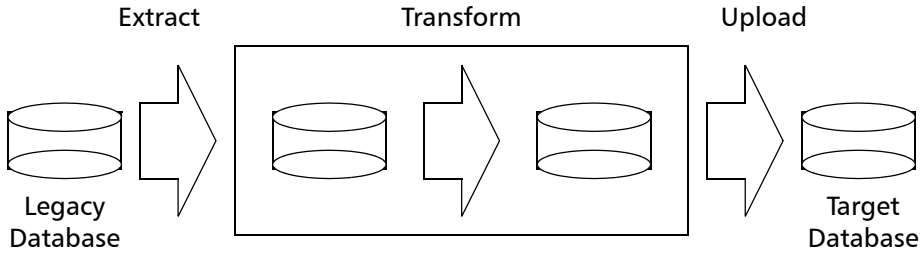
Introduction

There are many reasons that may prompt an organisation to replace an existing application, usually referred to as the legacy system, by a new one. Perhaps the legacy system has become difficult to maintain and should therefore be replaced. Perhaps the legacy system isn't even that old, but business demands still require some new functionality that turns out difficult to integrate. Perhaps technological advances make it possible to develop a new system that is more convenient and offers better usability.

Whatever reason there is for the development of a new system, that system cannot go operational with an empty database. Some existing data has to be made available to the new application before it can be launched. In many cases the amount of data will be rather large; for typical business applications it may include product data, customer data, and the like. Since this data is valuable to the organisation that owns it, care must be taken to transfer it to the new application accurately.

This is where data migration enters the scene. The data models of the old world and the new will probably not be the same; in fact the two could be fundamentally different. The objective of data migration is to extract data from the existing system, to re-format and re-structure it, and to upload it into the new system (Morris 2006, Bisbal Lawless Wu Grimson 1999, Haller 2008, Haller 2009, Matthes Schulz 2011, Matthes Schulz Haller 2011).[1]

Migration projects typically set up a migration platform in between the legacy system and the target system. The migration platform is where all migration-related processing takes place, as the following diagram illustrates. Similar diagrams can be found in the literature (Matthes Schulz 2011, Haller 2009).



The technical basis can vary a lot:

- The migration platform often contains a copy of the legacy database (as indicated in the diagram), so that the live database remains undisturbed from any migration efforts. An alternative strategy is to extract the legacy data into flat files.
- The migration platform may also contain a copy of the target database.
- Various technologies can be used for the actual transformation, including Java programs, PL/SQL scripts, XML processing and more.

While database vendors make tools available that cover most of the extraction and uploading functionality, the actual transformation usually requires custom software. The transformation depends heavily on the data models used, and so differs from one migration effort to the next.

Migration projects involve quite a few risks. According to the literature (Morris 2006, Matthes Schulz 2011, Matthes Schulz Haller 2011, Fowler 2008, Keller 2000), the most common risks include the following:

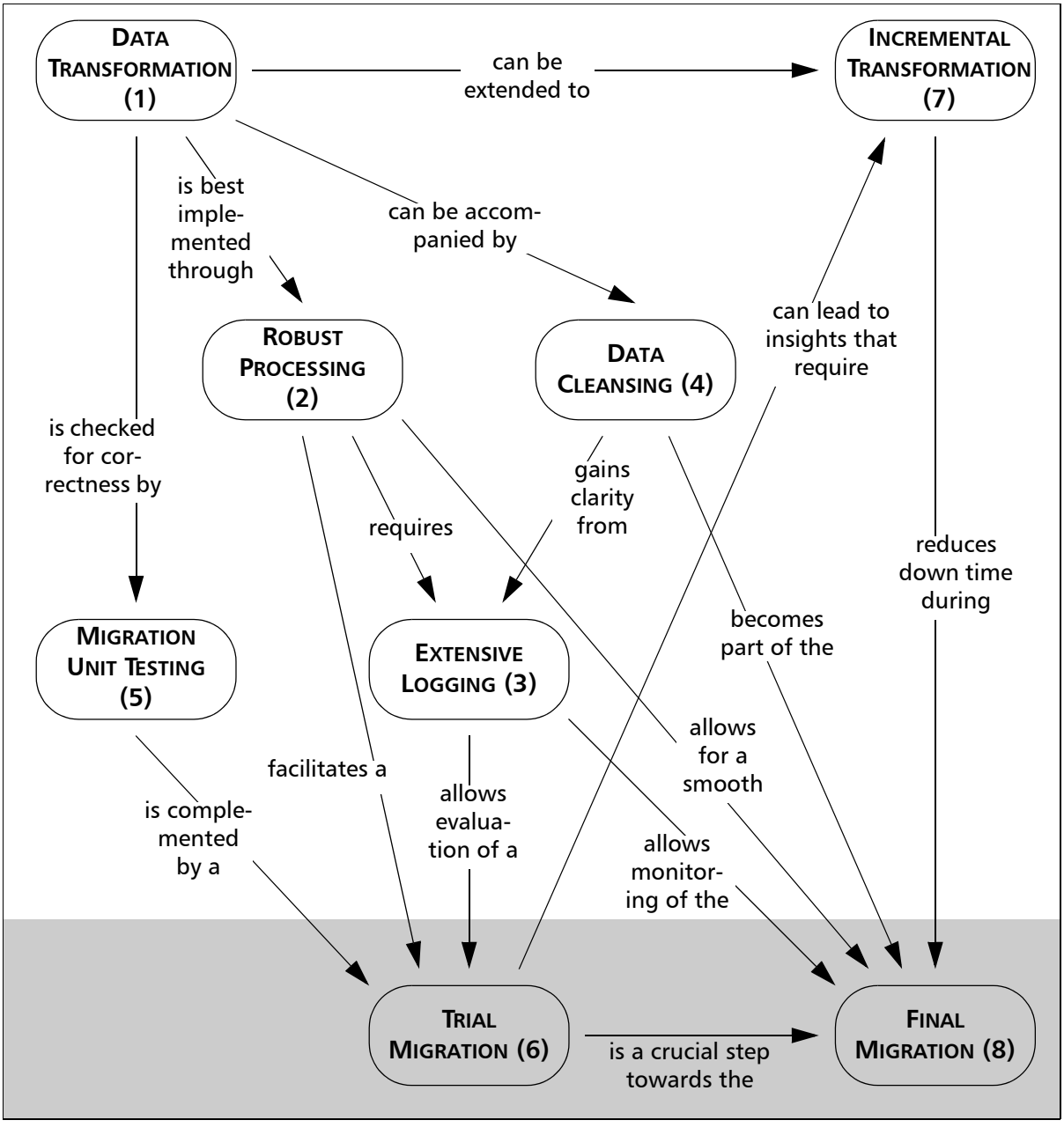
---

1. Data migration is different from database migration. *Database migration* refers to the replacement of one database system by another, which may make some changes to database tables necessary for technical reasons. Database migration is outside the scope of this paper However, *data migration* includes the transfer of data from one data model to another. This is what this paper is about.

- The legacy data might be complex and difficult to understand.
- The legacy data might be of poor quality.
- The amount of data can be rather large.
- The target data model might still be subject to change.

As a consequence, care must be taken for a migration project to be successful. A failed data migration could easily delay the launch of the new application.

The patterns in this paper address these requirements. They demonstrate techniques and strategies that help meet the typical requirements of a data migration project. The patterns are targeted at software developers, architects and technical project leads alike. The following figure gives an overview of the patterns and briefly sketches the relationships between them. Six patterns address the design of the migration code, while two patterns (illustrated with a grey background) focus more on the data migration process.



I have mined these patterns from three migration projects in which I was involved as developer and consultant. The first was the data migration made necessary by the introduction of a new life insurance system. The second was the migration of the editorial content for an online catalogue for household goods from one content management system to another. The third was the migration of customer data and purchase records for a web shop from an old application to a new one. Although the application domains were different, the projects showed some remarkable similarities in their requirements and in their possible solutions. The patterns in this paper set out to document these similarities.

Throughout this paper I assume relational databases, as this is by far the most widespread technology. With a little change in terminology, however, the principles described in this paper can also be applied to migration projects based on other database technology.

I'll explain the patterns with a running example that is inspired by (though not taken directly from) the web shop project mentioned above. The example consists of a web shop where customers can make a variety of online purchases. The system keeps track of these purchases and maintains the contact information for all customers. The overall perspective is to migrate customer data and purchase records onto a new platform. I'll explain the details as we go.

1 Data Transformation

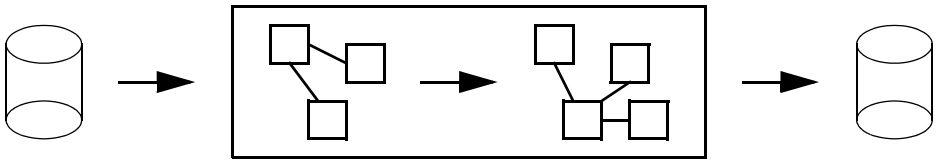
Context A legacy system is going to be replaced by a new application. The legacy application data will have to be migrated.

Problem **How can you make legacy data available to the new system?**

Forces The new application will almost always use a data model that is different from the legacy system's data model. You cannot assume a 1:1 mapping from database table to database table. Moreover, the legacy system's data model might be difficult to understand.

Nonetheless, data imported into the new system's database will have to express the same relationships between entities as the original system. References between entities, expressed through foreign key relationships, will have to be retained.

Solution **Implement a data transformation that establishes a mapping from the legacy data model to the target data model and that retains referential integrity.**



The data transformation will be embedded into the overall migration process, which in most migration projects consists of three major steps (Haller 2009): first, all relevant data is exported from the legacy system's database; next, the data transformation is performed; finally, the transformation results are imported into the new application database.

The actual transformation consists of the following steps:

- The transformation iterates over database tables, reading one entity after the other, while taking all its related entities into account as well.
- In each iteration, related entities are transferred into an object structure that matches the new application's data model. Because related entities are processed together, references between entities can be established and referential integrity is maintained.
- Some data models are too complex to allow the transformation to work this way, especially when cyclical references occur. In such a case, the transformation process needs to be extended, for instance by splitting up the transformation and storing intermediate results in temporary files.

A data transformation can be implemented in different ways. One option is to operate directly on database records, for instance with a PL/SQL script. Because running migration scripts on the live legacy database is almost always a bad idea, the original legacy data has to be exported into a database within the migration platform where the actual transformation can then be performed.

An alternative is to export data from the legacy database into a file-based representation, also within the migration platform. In this case the legacy data can be processed by Java programs, XML processors and the like.

In any case, the resulting objects represent the new system's data model. The transformation process stores them in a format that an import mechanism of the target database understands.

Example

In our web shop data migration, all relevant data records are exported from the legacy database into flat files, one for each database table. These files are read by a Java component that implements the transformation by iterating over all customers. For each customer, it takes the customer's purchases into account as well, as these maintain a foreign key relationship to the customer.

The transformation process creates a customer object for every legacy customer entity and a new purchase object for each legacy purchase entity. In addition, the transformation process creates address objects for all a customer's addresses, which in the legacy system were stored within the customer entity.

After a fixed number of customers, say 10.000, have been processed, the customer, address and purchase objects created so far are stored in the file system from where they can later be imported into the new database.

Benefits

The new application is provided with the initial data it needs.

Relationships between entities are maintained. Referential integrity is retained throughout all application data.

Liabilities

Implementing the data transformation requires a good deal of domain knowledge (Morris 2006). It's next to impossible to map an old data model onto a new one without understanding the domain logic behind all this data. It's therefore crucial to involve domain experts into the migration effort. It's a good idea to use their knowledge for powerful MIGRATION UNIT TESTING (5).

Establishing a correct data transformation can still be difficult, especially if the legacy system's data model is flawed or the two data models differ a lot. You may have to apply DATA CLEANSING (4) in order to solve possible conflicts. You should use EXTENSIVE LOGGING (3) whenever the data transformation encounters any problems.

Depending on the overall amount of data and the transformation complexity, a data migration can require a long execution time. In practice, several hours or even several days are possible.

The transformation process can have significant memory requirements, especially if large groups of data have to be processed together due to complex relationships between entities.

If the overall amount of data turns out to be too large to be processed in one go, you may opt to migrate the data in batches. A common strategy is to MIGRATE ALONG DOMAIN PARTITIONS (Wagner Wellhausen 2010), which means to apply vertical decomposition to the overall application and to migrate one subsystem after the other.

2 Robust Processing

Context You have set up the fundamental DATA TRANSFORMATION (1) logic necessary to migrate data from a legacy system to a new application. It's now time to think about non-functional requirements.

Problem **How can you prevent the migration process from unexpected failure?**

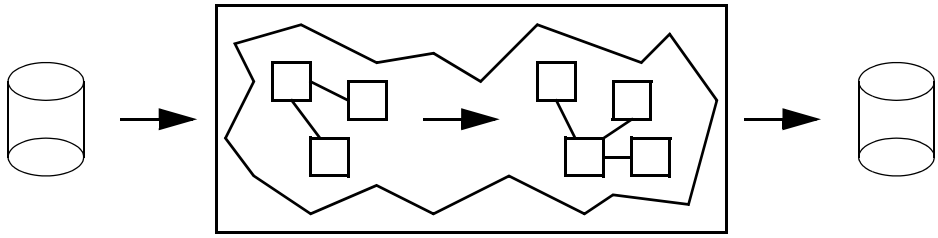
Forces Legacy data is sometimes of poor quality. It can be malformed, incomplete or inconsistent. Certain inconsistencies can in principle be avoided by the introduction of database constraints. However, legacy databases often lack the necessary constraints.

Despite all this, the transformation process must not yield output that, when imported into the new system, leads to database errors such as unique constraint violations or violations of referential integrity. (For the new database the relevant constraints will hopefully be defined.)

Moreover, the migration process should not abort due to flawed legacy data. While it's true that a crashed TRIAL MIGRATION (6) tells you that a specific entity is problematic, you don't get any feedback regarding the effectiveness of the migration code in its entirety. For a serious TRIAL MIGRATION (6) this is unacceptable.

For the FINAL MIGRATION (8) robustness is even more important. The FINAL MIGRATION (8) is likely to be performed just days or even hours before the new application will be launched. If unexpected problems caused the migration process to abort, the launch would be seriously delayed.

Solution **Apply extensive exception handling to make sure that the transformation process is robust and is able to cope with all kinds of problematic input data.**



The most common cases of problematic input data include the following:

- Missing references (violations of referential integrity in the legacy database).
- Duplicate data (unique constraint violations in the legacy database).
- Illegal null values (non-null constraint violations in the legacy database).
- Technical problems (illegal character sets or number formats, and the like).

Exception handling can take different forms depending on the technology you use to implement the DATA TRANSFORMATION (1). Exception handling mechanisms are available in many programming languages, including Java and PL/SQL.

Sometimes you won't be able to detect invalid data by evaluating entities in isolation, but only by evaluating entities in their relational context. In some cases, if you discard a specific entity, you will have to discard some related entities as well — entities that the DATA TRANSFORMATION (1) processes together.

In the aftermath of a migration run you will have to analyse what exceptions have occurred. In the case of a TRIAL MIGRATION (6) this will tell you where the migration code needs improvement. During the FINAL MIGRATION (8) (directly before the new application is launched) ideally no exceptions should occur. If they do, the problematic data will have to be handled manually in the target database.

Example

The web shop data migration applies exception handling to detect any illegal data formats. For example, names and addresses should consist of valid characters and prices should be non-negative numbers. If illegal values occur, an exception is caught and the flawed entity is discarded. The migration process won't crash.

The migration also discards purchases that refer to a non-existent customer, at least for the time being. In principle, such purchases shouldn't exist, but unfortunately there are some dangling references to customers.

As the migration code is gradually improved, some DATA CLEANSING (4) mechanisms are added so that exceptions are largely avoided; the few remaining problems are handled manually.

Benefits

Reliability of the migration process is increased as invalid data is prevented from causing the migration process to crash. As a TRIAL MIGRATION (6) will process all input data, regardless of possible data quality issues, it can give you valuable feedback regarding the effectiveness and the efficiency of your migration code. A single TRIAL MIGRATION (6) can detect a multitude of problems, not just a single problematic entity.

You can be confident that the FINAL MIGRATION (8) will take place as planned and without delay before the new application is launched.

You can also be sure not to run into constraint violations when importing data into the target database.

Liabilities

The migration process must never skip any invalid legacy data without further notice. Whenever problematic data occurs EXTENSIVE LOGGING (3) must document what entities have been discarded.

If there is a chance that flawed legacy data can mended, you can plan to apply DATA CLEANSING (4), which reduces the amount of exceptions that will occur and will need to be handled.

3 Extensive Logging

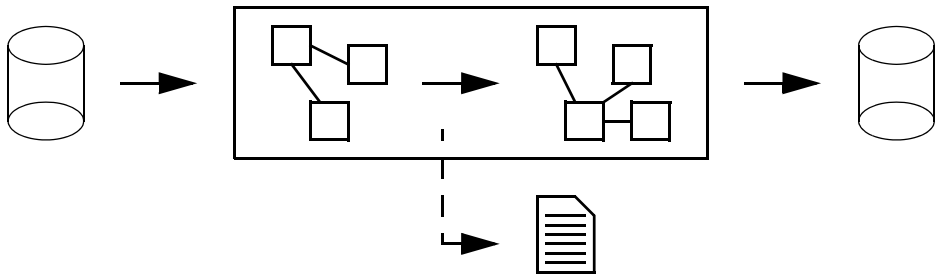
Context You have implemented the DATA TRANSFORMATION (1) logic required for your migration project. ROBUST PROCESSING (2) ensures that problematic input data is discarded if otherwise it would cause runtime exceptions.

Problem **How can you facilitate the analysis of problems that may occur during the transformation of possibly large amounts of data?**

Forces A data migration usually involves a huge amount of data — clearly too much data to monitor manually. Moreover, the migration process can last for several hours or days. You need to keep track of the current status while the migration is up and running.

Because ROBUST PROCESSING (2) is usually required, illegal data may have to be discarded during the DATA TRANSFORMATION (1) process. However, it's crucial to know what data had to be deleted and for what reasons.

Solution **Enhance your transformation process with extensive logging mechanisms.**



Relevant logging includes the following:

- Count every entity that undergoes a DATA TRANSFORMATION (1), classified by type (legacy database table).
- Count every entity that is made available to the import into the new database, classified by type (new database table).
- Log every entity that is discarded in the process of ROBUST PROCESSING (2). Also log the reason why the entity had to be discarded, for instance the exception that was thrown.
- If the migration code includes aspects of DATA CLEANSING (4), log every entity that is corrected or otherwise modified.

In addition, log status messages at certain intervals throughout the migration process. These status messages should explain how many entities of what type have been processed so far. They can also include the current memory usage or other parameters that may prove useful for system configuration.

The technical details of logging depend on the technology you use for the DATA TRANSFORMATION (1). A transformation implemented in PL/SQL will probably use a specific database table for logging. A Java-based transformation might use Log4J or any other filed-based logging mechanism. Either way it's important to make sure that logs won't be lost in case the system crashes.

Example The web shop data migration logs the number of legacy customers and legacy purchases that are processed, as well as the number of customers, addresses and purchases that are created for the new database.

In addition, comprehensive log entries are written for each entity that has to be discarded (ill-formatted entities, but also purchases without customer). These log entries include as much information about the discarded elements as possible to make a straightforward problem analysis possible.

After a migration run the logs are analysed thoroughly. Special attention is given to the purchase records that has to be discarded due to non-existing customers. Domain experts look into the data and analyse what needs to be done.

Benefits Status logs allow you to monitor the DATA TRANSFORMATION (1) process while it's running.

After the DATA TRANSFORMATION (1) process has finished, you know how many entities of which kind have been successfully migrated.

In addition, you know what data had to be discarded as a consequence of ROBUST PROCESSING (2), and why. If the migration code is still under development, a problem analysis can help you find out what parts of the code still need improvement. Once the migration code is final, the log files tell you what data may require manual treatment.

Liabilities Log files can easily use up a significant amount of disk space.

Logging can make the migration process slower.

4 Data Cleansing

Context You have implemented the DATA TRANSFORMATION (1) logic for your data migration project with some ROBUST PROCESSING (2) that handles illegal data from the legacy system. Still, the legacy system might contain data that isn't exactly illegal, but isn't useful for the new application either.

Problem **How can you prevent the new application from being swamped with useless data right from the start?**

Forces A legacy system's data base often contains outdated or incomplete data; sometimes it's appalling how poor the data quality is (Morris 2006, Fowler 2008). Some problems with data quality are of technical nature — problems that could in principle be avoided by the definition of database constraints, which, however, are sometimes lacking in legacy systems. Other problems are caused by data that is invalid in terms of the application domain.

For an existing application invalid data is rarely ever corrected. Many legacy systems work reasonably well despite a low data quality, so people sometimes feel the least expensive way to handle poor data quality is simply to ignore it.

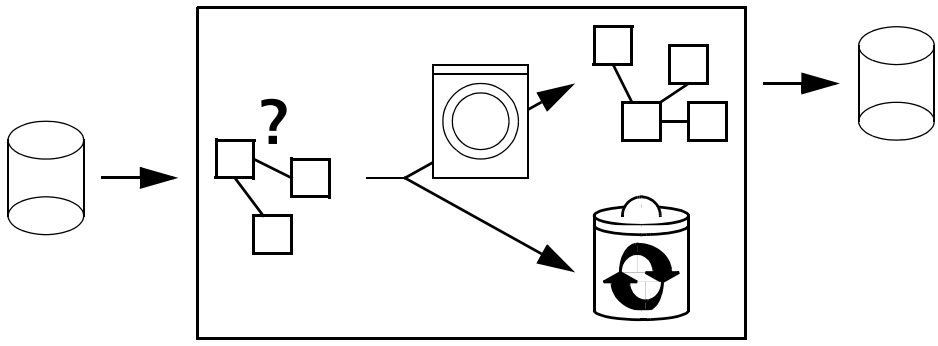
However, you can no longer ignore data quality issues when introducing a new application. First, launching a new application based on flawed data would be unsatisfactory. Second, there are technical reasons why it may be impossible to migrate flawed legacy data. If, for instance, the target database makes better use of constraints than the legacy database did, object that violate referential integrity couldn't be imported successfully.

Handling flawed data is a process known as data cleansing (Morris 2006). Data cleansing can take place at different points in time within the overall migration process. One approach is to handle flawed data early on in the legacy database, which offers the advantage that the DATA TRANSFORMATION (1) need not be concerned with data cleansing issues (Matthes Schulz 2011, Matthes Schulz Haller 2011).

However, when legacy data is migrated, every single entity has to be checked for validity anyway to make ROBUST PROCESSING (2) possible. Handling flawed legacy data during the DATA TRANSFORMATION (1) is therefore a powerful option too. In addition, this option bears the advantage that the data cleansing can be applied to groups of related entities, as these entities are typically migrated together.

Solution

Enhance your transformation processes with data cleansing mechanisms.



Data cleansing can either mean to remove the invalid data or to try to correct it. Concrete solutions depend on the individual situation. Ultimately it's a business decision what data should be cleansed. Typical examples include the following:

- Data that violates constraints of the application domain.
- Data that is outdated and no longer relevant.

As data cleansing requires a good deal of application logic, it's usually a good strategy to encapsulate it into dedicated methods, which can then be invoked from the overall DATA TRANSFORMATION (1) process.

Apply EXTENSIVE LOGGING (3) to all cases of data cleansing, so that it's clear what changes to the application data have been made.

Example

In the past, the software team in charge of the web shop never got round to improving the quality of customer data. Now, however, the chance has come to do just that.

On the one hand, data cleansing addresses the problem with orphaned purchase records that point to non-existing customers. A function is added that tries to retrieve the missing customer data for purchases too recent to be simply ignored. The function consults a database table that stores transaction details — a table that so far wasn't considered.

On the other hand, data cleansing also aims to get rid of data that is no longer needed. Customers are removed if they haven't made a purchase within the last 7 years. In addition, the process looks up potential customer duplicates — distinct customers, though with the same name and address. Moreover, postal codes in customer addresses are validated and, if possible, corrected automatically.

Looking up customer duplicates makes the transformation process somewhat more complex. It's necessary to maintain the set of customers processed so far in order to detect duplicates. If a duplicate is detected, the two entities are merged into one.

Benefits Data quality is improved: technical and domain-driven constraints are met to a larger extent. Fewer exceptions will therefore have to be caught in the exception handling mechanisms introduced with the goal of ROBUST PROCESSING (2).

Also as a consequence of improved data quality, the new application can be expected to work more reliably.

Liabilities Data cleansing requires addition development effort.

Data cleansing can lead to a longer execution time.

Data cleansing can become so complex that more than one pass can be required to perform the necessary corrections.

5 Migration Unit Testing

Context You have implemented a DATA TRANSFORMATION (1) characterised by ROBUST PROCESSING (2) and DATA CLEANSING (4). You have added EXTENSIVE LOGGING (3) to monitor the transformation process. It's crucial for the success of your migration project that the transformation you have implemented is indeed correct.

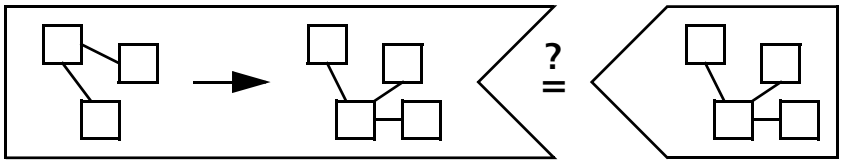
Problem **How can you catch errors in your data transformation process?**

Forces Correctness is essential for any data migration effort. If the DATA TRANSFORMATION (1) was misconstrued, the new application would be initialised with flawed application data, which would give it a bad start.

However, the mapping from the old data model to the new one can be complex and difficult to understand. Detailed knowledge of both the old and the new system is required to define a correct mapping.

Solution

Write unit tests based on a representative test data in order to check the transformation process for correctness.



Ideally, the migration team will DEVELOP WITH PRODUCTION DATA (Wagner Wellhausen 2010), which suggests that the test data should be an excerpt of the legacy system's live database. The set of test data must be large enough to guarantee sufficient coverage, while on the other hand it should be compact enough to be still manageable.

- The unit tests should compare the outcome of the DATA TRANSFORMATION (1) with the expected results. Run the test suite automatically with each change you make to the migration software while it's still under development.
- In case the DATA TRANSFORMATION (1) applies DATA CLEANSING (4), an automated test can check that flawed data is properly mended.
- In addition, tests can verify that the EXTENSIVE LOGGING (3) works fine: the log files must contain the expected entries regarding invalid legacy data.
- Finally, you can import the results of the DATA TRANSFORMATION (1) into the new application and see if you can perform typical uses cases on the migrated data.

Much in the vein of test-driven development (Beck 2002), it's essential to apply unit tests right from the start of your migration effort. This way, you get early feedback regarding the accuracy of your migration software.

Example

Representative test data consists of a set of several customers along with their purchase records. The customers should differ especially with respect to the number of their purchases, although other attributes (such as address information, preferred payment method, etc.) should vary too. A unit test can check that customers and purchases are migrated correctly, that their relationships are established, that the correct address entities are created, etc.

The test data also includes candidates for discarding, such as inactive customers, orphaned purchases, and the like. Testing can verify that the invalid data does not cause the transformation process to abort, that the invalid legacy entities are indeed discarded, and that the necessary log file entries are written.

Benefits

Unit testing ensures that the transformation process works as it should. You get a good impression of how smoothly the new application will work on migrated data.

If there are logical problems with the transformation — problems that probably result from a lack of understanding of the application domain — at least you'll become aware of these problems as quickly as possible.

Liabilities Representative test data can be difficult to obtain. The test data should be compact, yet it must be self-contained (in terms of referential integrity).

Even if tests based on representative data run smoothly and show that the DATA TRANSFORMATION (1) works as it should, there is no guarantee that the overall migration will work reliably. A TRIAL MIGRATION (6) can shed more light on the overall process.

6 Trial Migration

Context You have successfully applied MIGRATION UNIT TESTING (5) to the DATA TRANSFORMATION (1) mechanisms you have implemented. The overall process migration should be characterised by ROBUST PROCESSING (2).

Problem **How can you avoid problems with the processing of mass data during the execution of your data migration process?**

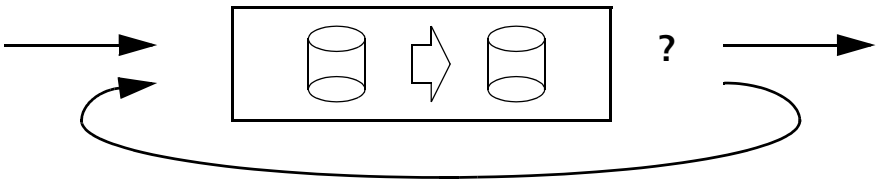
Forces MIGRATION UNIT TESTING (5) gives you a good idea regarding the accuracy of your DATA TRANSFORMATION (1).

However, unit tests cannot prove that your migration software meets its robustness requirements. You simply cannot anticipate all possible risks. Even if you plan to discard illegal legacy data, there might be instances of illegal data you didn't foresee.

You cannot foresee the exact system requirements for your migration process either. What if the process fails due to lack of memory, or because there was no disk space left when trying to store a file? Keep in mind that in practical cases a DATA TRANSFORMATION (1) can easily require several gigabytes of memory and that EXTENSIVE LOGGING (3) can lead to log files that amount to hundreds of gigabytes. You have to ensure ROBUST PROCESSING (2) despite these risks.

Moreover, you cannot start to plan the FINAL MIGRATION (8) unless you know how long it will take to complete. MIGRATION UNIT TESTING (5) cannot answer this question, only a realistic test using a complete set of legacy data can.

Solution **Run a series of trial migrations that each process a complete set of legacy data. Perform measurements on the process and the results and improve the data transformation logic until the results are satisfactory.**



The simplest way to obtain a complete set of legacy data is to create a dump of the legacy system's live database. A trial migration comprises the following tasks:

- Migrate the complete database dump.

- Test whether the overall migration process completes without aborting.
- Measure the time span the trial migration requires.
- Measure the amount of disk space required. This includes files generated by EXTENSIVE LOGGING (3), but also temporary files the DATA TRANSFORMATION (1) may require.
- Compare the number of legacy entities processed with the number of new entities that are created. Test whether these numbers are plausible.

Any trial migration benefits from EXTENSIVE LOGGING (3). You can evaluate log files or log tables to obtain the numbers of entities that were processed or discarded. A simple script can usually check the log files for plausibility and MEASURE MIGRATION QUALITY (Wagner Wellhausen 2010).

It's important to understand that a single trial migration isn't enough. Instead, the idea is to establish a process model that involves repeated trial migration runs (Matthes Schulz 2011, Matthes Schulz Haller 2011). If you manage to run trial migrations on a regular basis, you'll be aware of any effects that changes to the migration code or to the target data model may have. Once the trial migrations run consistently smoothly, you're ready for the FINAL MIGRATION (8).

Example The trial migration uses a dump of the web shop's live data base, so it should cover all kinds of data that the real migration will have to face. Reliable conclusions regarding robustness become possible. In addition, the test shows how long the migration process will take and what size the log files will be.

A small shell script performs a few plausibility checks. For instance, the number of migrated customers and discarded customers are extracted from the log files. They must add up to the number of customers exported from the legacy database.

Benefits Reliability is increased as trial migrations verify that your overall migration process is robust and fail-safe.

You get a concrete idea of the time and space requirements of your migration process when applied to realistic amounts of data. This helps you plan the FINAL MIGRATION (8): it tells you what kind of machines you need (with regard to memory and disk space) and also tells you how many hours or days you'll have to reserve for the data migration before the new application is launched.

Domain experts can use the results of a trial migration to perform business acceptance tests. Confidence in the migration software is increased as the trial migrations yield expected results.

Liabilities In some organisations, getting hold of a dump of the live database can prove difficult. If the migration involves sensitive live data, it may have to be anonymized before it can be used for testing purposes.

If a trial migration reveals that the overall DATA TRANSFORMATION (1) requires more time than expected, you may have to think about strategies to reduce application down time during the FINAL MIGRATION (8). A powerful strategy is to apply INCREMENTAL TRANSFORMATION (7).

7 Incremental Transformation

Context A TRIAL MIGRATION (6) has revealed how long the migration of the complete legacy data might take.

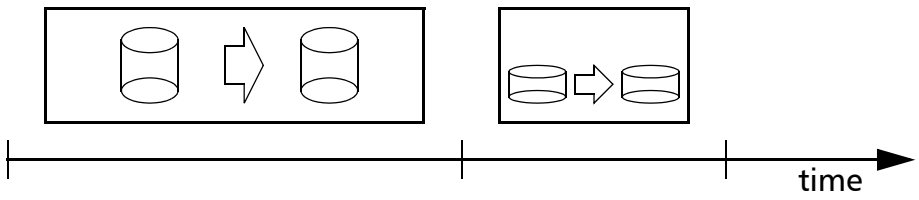
Problem **How can you avoid unacceptable down times of your application while the data migration takes place?**

Forces A straightforward migration strategy is to migrate all data in one go. In this scenario, you have to shut down the legacy application before the data migration starts, since otherwise changes could be made to the legacy data that would not be reflected by the results of the migration process.

Depending on the amount of legacy data and the complexity of the DATA TRANSFORMATION (1), the overall migration process can take quite some time. Experience shows that, for large applications, data migration can require several hours or even several days. During this time neither the legacy system nor the new application will be available.

This may or may not be acceptable. A short down time is in many cases ok, but taking a business-critical application off-line for several days is almost always impossible.

Solution **If necessary, enhance the data transformation mechanism with the ability to process data records only if they haven't been processed before or have changed in the meantime. Such a transformation mechanism makes it possible to migrate the legacy data in batches and to keep the legacy system operational while the bulk of data is being processed.**



Technically, the idea is to introduce a slightly more sophisticated DATA TRANSFORMATION (1):

- As with a standard DATA TRANSFORMATION (1), there is a mapping from the old data model onto the new one.
- Entities are processed only if they have been created or updated since the last data migration. A timestamp on the database can be used to make that decision.
- When data is imported into the target database, it isn't automatically created anew. If an entity already exists, it is updated instead.
- Entities aren't deleted in the legacy database after the initial migration, but are only marked for deletion. The version that was migrated before will be deleted in the target database during the next migration run.

This incremental approach[2] allows you to keep the legacy application operational while a large amount of data is migrated in a first migration run, which is likely to take some time. The second migration run will be much faster as it has to process only data that has been updated in the meantime. If necessary, there can be a third increment that will require even smaller time spans, and so on. Shutting down the legacy system will be necessary only while the last increment is being processed.

Because an incremental transformation is more complex than a simple DATA TRANSFORMATION (1), you should only opt for this strategy if there is a real benefit to it. There is hardly a justification for implementing an incremental transformation unless a TRIAL MIGRATION (6) has shown that migrating all legacy data in one go would take longer than you can afford to take the application off-line.

Example

A TRIAL MIGRATION (6) of the web shop's database has taken almost two days and it's definitely impossible to take the shop off-line for so long. There is a chance that the run time can reduced by making the code more efficient, but this alone won't solve the problem. The decision is made to apply incremental migration so that much of the migration process can take place a few weeks before the new application is launched.

The existing transformation code is enhanced so that it only processes new or updated entities. Database timestamps are used to identify entities that need to be migrated.

The next TRIAL MIGRATION (6) shows that there is reason for optimism. The first migration run still takes two days, but an incremental migration performed a week later completes successfully after only a few hours. This is a time span that is acceptable as an application down time during the FINAL MIGRATION (8).

Benefits

Down times are clearly reduced. The time span between the shutdown of the legacy application and the launch of the new application, during which no application data can be written, can be kept relatively short.

Some projects have reported that risk is also reduced provided you do a series of incremental migration runs. First, much of the data is migrated some time before the launch of the new application and, second, the closer you get to that date, the more familiar you are with migrating live data as you're doing one increment after the other (Drupal 2010).

---

2. The term *incremental migration* is used in the literature in two entirely different ways. First, it is used in the context of application migration and refers to a strategy of migrating one application and its underlying data after the other (or one subsystem after the other if a large application can be broken down into independent subsystems), as opposed to a big bang scenario where several new applications are launched at once. This approach is often chosen as a means of risk reduction (Cimitile De Carlini De Lucia 1998), but can also be the consequence of an incremental software development method (Fowler 2008). The second meaning of *incremental migration* refers to techniques of migrating only data that hasn't been migrated before, usually with the aim of reducing down times in mind. The latter is what that this pattern is about.

Liabilities Implementing an incremental transformation is more complex than implementing a standard DATA TRANSFORMATION (1). For instance, the last modification date needs to be known for each entity, which may or may not be easy to figure out. Also, handling deleted elements may turn out difficult to implement. As said before, an incremental transformation should be implemented only if the benefits in terms of reduced down times justify the additional effort.

Testing an incremental migration is equally more complex. It certainly requires several TRIAL MIGRATION (6) runs, consisting of two or more increments each. Whenever you make changes to the logic underlying the incremental transformation, you have to re-test the whole process, starting with the bulk migration and continuing with the smaller increments.

8 Final Migration

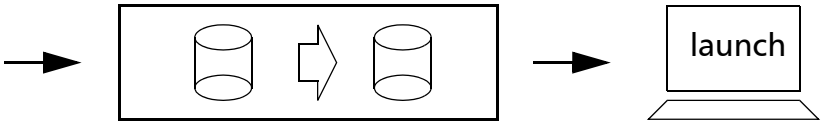
Context You're approaching the launch of the new application. A series of TRIAL MIGRATION (6) runs has shown that the migration code is ready to go live. You may or may not have decided to apply INCREMENTAL TRANSFORMATION (7) in order to reduce system down time.

Problem **How can you avoid trouble when the new application is launched?**

Forces You have tested the data migration throughout a series of TRIAL MIGRATION (6) runs, but still you can't be one hundred percent sure that there aren't going to be problems. The risk of running into trouble may be very, very small, but if problems occur the consequences could be serious. The launch of the new application is at stake: if the legacy data cannot be migrated successfully, the new application isn't going to be launched.

It's not just possible problems that you need to be thinking of. At some point the legacy system will have to be shut down and from that point on the application will not be available until the new system is launched. From the latest TRIAL MIGRATION (6) you know how long the down time is going to be. It's wise to think about whether that down time should be on a specific date, a specific day of the week, a specific weekend, during the day or the night time, and the like.

Solution **Set up a checklist that includes all things that need to be done in the context of the final data migration, by whom, and at what point in time.**



Some preparations are necessary to come up with such a checklist:

- You have to choose a migration date that is least problematic with regard to the expected system down time. You'll probably need to reach an agreement with many stakeholders to make that decision.

- In case you apply INCREMENTAL TRANSFORMATION (7), the final data migration will consist of several increments. Only the last increment will be take place immediately before the new application is launched. You'll have to set up a timeline from migrating the first bulk of data to the last small increment.
- You have to decide what hardware you're going to use. Ideally you'll use exactly the same hardware for the final migration that you've already used for the latest TRIAL MIGRATION (6). If this isn't possible, make sure to use equivalent hardware that is configured identically. In any case, keep a list of the commands necessary to execute the final migration tasks.
- Implement a fall-back strategy in case the data migration fails (or anything else goes astray). If nothing else works, you must at least be able to revert to using the legacy system.
- The final migration will produce EXTENSIVE LOGGING (3) just as the latest TRIAL MIGRATION (6) did. Keep a list of the log files or database tables that you may have to inspect during the final migration for monitoring purposes.
- Plan to do a few final tests in the live environment-to-be before the new application goes live. There cannot be extensive testing because the new application will have to be launched soon, but there is probably time for a few simple tests that completely rely on migrated data.

Last but not least, make sure the necessary team members are available when the final migration takes place.

Example In order to avoid significant down times, the bulk of legacy data for the web shop is migrated a couple of weeks before the launch of the new application.

The decision is made to launch the new application on a Monday night, as this is when the web shop creates the smallest revenue. On that Monday night, only the last increment of legacy data is migrated. Since the migration includes only entities that were changed recently, the legacy database export, the transformation and the target database import together take only a few hours to complete.

A few final tests in the new live environment show that all necessary data has been migrated completely. The new application can be launched.

Benefits The benefit of being well-prepared is very likely a smooth ride through the final data migration process.

Liabilities There is no way to deny that the preparation takes quite some effort.

Conclusions

Data migration doesn't happen automatically. In a certain sense, a data migration effort constitutes a project in its own right. After its completion the migration software is expected to be executed only once, but nonetheless, or perhaps because of this, the requirements especially on correctness, robustness and efficiency are quite high.

In my experience the effort necessary to perform a successful data migration is often underestimated. If you are, or will be, involved in a data migration project, the patterns in this paper should give you a reasonable idea of what needs to be done and how, and should also give you a realistic feel for the underlying complexity.

Acknowledgements

I'd like to thank Filipe Correia who, as the EuroPLoP 2010 shepherd for this paper, offered valuable feedback on both its form and content. His detailed suggestions clearly helped me improve the paper.

Thanks are also due to the participants of the EuroPLoP 2010 workshop in which this paper was discussed. The workshop spawned many good ideas and helped me fine-tune the paper. Special thanks go out to James Noble, Jim Siddle and Tim Wellhausen for feedback at an amazing level of detail; James Noble also provided the winning suggestion for the title of this paper.

I'd also like to thank Christopher Schulz for the in-depth discussion on data migration we had in February 2012. This discussion provided a lot more insight and lead to a number of improvements.

Last but not least, thanks are due to the anonymous TPLoP reviewers for their comments and suggestions that helped me to round off this paper.

References

There isn't much literature available on data migration. The contributions listed below discuss related topics.

The paper by Martin Wagner and Tim Wellhausen also contains patterns on data migration. Their paper and mine were parallel efforts, written independently though at the same time. There are a few overlaps. In general, their paper places emphasis on process and management issues, while the focus of this paper is mostly on technical aspects. Ultimately, the two papers complement each other.

Beck 2002

Kent Beck. *Test-Driven Development — By Example.* Addison-Wesley, 2002.

Bisbal Lawless Wu Grimson 1999

Jesús Bisbal, Deirdre Lawless, Bing Wu, and Jane Grimson: "Legacy Information Systems: Issues and Directions" in *IEEE Software*, September/ October 1999 (vol. 16 no. 5).

Cimitile De Carlini De Lucia 1998

A. Cimitile, U. De Carlini, A. De Lucia: "Incremental Migration Strategies: Data Flow Analysis for Wrapping", in 5th Working Conference on Reverse Engineering, Honolulu, USA, 1998. IEEE, 1998.

Drupal 2010

The Economist.com data migration to Drupal. Drupal, October 2010.

http://drupal.org/node/915102

Fowler 2008

Martin Fowler. *Incremental Migration.* July 2008.

http://martinfowler.com/bliki/IncrementalMigration.html

Keller 2000

Wolfgang Keller. "The Bridge to the New Town — A Legacy System Migration Pattern", in M Devos, A. Rüping (eds.), *EuroPLoP 2000 — Proceedings of the 5th European Conference on Pattern Languages of Programs,* Universitätsverlag Konstanz, 2001.

Haller 2008

Klaus Haller. "Data Migration Project Management and Standard Software: Experiences in Avaloq Implementation Projects", in: Barbara Dinter, Robert Winter, Peter Chamoni, Norbert Gronau, Klaus Turowski (eds.), *DW 2008 - Synergien durch Integration und Informationslogistik*, *St. Gallen, Switzerland.* LNI 138, Gesellschaft für Informatik, 2008.

Haller 2009

Klaus Haller. "Towards the Industrialization of Data Migration: Concepts and Patterns for Standard Software Implementation Projects", in P. van Eck, J. Gordijn, R. Wieringa (eds.), *CAiSE 2009 — Proceedings of the 21st International Conference on Advanced Information Systems,* Lecture Notes in Computer Science LNCS, Vol. 5565, Springer-Verlag, 2009.

Matthes Schulz 2011

Florian Matthes, Christopher Schulz: "Towards an integrated data migration process model - State of the art and literature overview". Technische Universität München, Fakultät für Informatik, Technical Report, 2011.

http://wwwmatthes.in.tum.de/file/attachments/wikis/sebis-article-archive/ms11-towards-an-integrated-data-migration/tb_DataMigration.pdf

Matthes Schulz Haller 2011

Florian Matthes, Christopher Schulz, Klaus Haller: "Testing & quality assurance in data migration projects", in: *27th IEEE International Conference on Software Maintenance (ICSM)*, Williamsburg, USA. IEEE, 2011.

Morris 2006

John Morris. *Practical Data Migration.* British Computer Society, 2006.

Wagner Wellhausen 2010

Martin Wagner, Tim Wellhausen. "Patterns for Data Migration Projects", in M. Weiss, P. Avgeriou (eds.), *EuroPLoP 2010 — Proceedings of the 15th European Conference on Pattern Languages of Programs,* ACM Digital Library, 2011.