Patterns for Data Migration Projects

Martin Wagner
martin.wagner@tngtech.com
http://www.tngtech.com

Tim Wellhausen
kontakt@tim-wellhausen.de
http://www.tim-wellhausen.de

March 17, 2011

Introduction

Data migration is a common operation in enterprise computing. Whenever a new system is introduced and after each merger and acquisition, existing data has to be moved from a legacy system to some hopefully more suitable target system.

A successful data migration project needs to manage several key challenges that appear in most such projects:

First, you don't know what exactly is in the legacy system. The older the legacy system is, the less likely it is that its original developers are still members of the development team or that they are at least available to support the data migration project. Additionally, even if the legacy system has been well documented in the beginning, you often cannot rely on the correctness and completeness of the documentation after years of change.

Second, data quality might be an issue. The quality constraints on the data in the old system may be lower than the constraints in the target system. Inconsistent or missing data entries that the legacy system somehow copes with (or ignores) might cause severe problems in the target system. In addition, the data migration itself might corrupt the data in a way that is not visible to the software developers but only to business users.

Third, it is difficult to get decisions from the business side. Business experts typically already struggle to get their daily work done while, at the same time, working on the requirements for the new system. During the development of the data migration code, many decisions need to be made that may affect the quality and consistency of the migrated data.

Fourth, development time is restricted. The development of the data migration project cannot start before the target system's domain model is well-defined and must be finished

before the go-live of the new system. As the main focus typically is on the development of the target system, the data migration project often does not get as many resources as needed.

Fifth, run time is restricted. Large collections of legacy data may take a long time to migrate. During the development of the migration code, a long round trip cycle makes it difficult to test the data migration code regularly. At go-live it may not be feasible to stop the legacy system long enough to ensure that the legacy data to migrate is both constant and consistent during the complete migration process.

This paper presents several patterns that deal with these issues:

- Develop with Production Data: Use real data from the production system for tests during the development of the migration code.
- Migrate along Domain Partitions: Divide and conquer the migration effort by migrating largely independent parts of the domain model one after another.
- Measure Migration Quality: Implement code that collects and stores all sorts of information about the outcome of the migration during every run.
- Daily Quality Reports: Generate detailed reports about the measured quality of the migrated data and make it available to all affected stake holders.

In parallel to this paper, Andreas Rüping wrote a collection of patterns with the title *Transform!* [5]. His patterns cope with the same forces as described here. Among these patterns are:

- Robust Processing: To prevent the migration process to halt from unexpected failure, apply extensive exception handling to cope with all kinds of problematic input data.
- Data Cleansing: To prevent the new application from being swamped with useless data right from the start, enhance your transformation processes with data cleansing mechanisms.
- Incremental Transformation: Perform an initial data migration a while before the new application goes live. Migrate data that has changed since then immediately before the new application is launched.

The intended audience for this paper are project leads, both on the technical and the business side, and software developers in general who need to get a data migration project done successfully.

Test infrastructure

A data migration effort needs a supporting infrastructure. How you set up such an infrastructure and what alternatives you have is out of the scope of this paper. Rather, the paper assumes the existence of an infrastructure as outlined in figure 1:

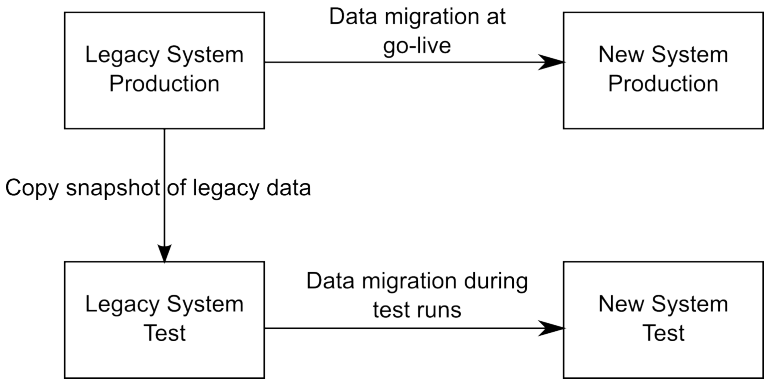

Figure 1: Data flow between productive and test systems during data migrations

The final data migration takes place from the legacy system into the new system in the production environment. Once this migration has been successfully done, the project has succeeded. To perform test runs of the data migration at development time, test environments for both the legacy and the new system exist and are available to test the migration code. To set up the tests, legacy data from the production system needs to be copied into the corresponding test system. After a migration test run, the new system can be checked for the outcome of the test.

Running example

Throughout the paper, we will refer to an exemplary migration project to which the patterns are applied. The task at hand is to migrate data about customers and tickets from a legacy system to a new system with a similar yet somewhat different domain model.

The legacy system contains just one large database with all information about the customers and the tickets that they bought. For repeated ticket sells to existing customers, new entries were created. The new system splits both customer and ticket data into several tables. Customers are now explicit entities and may have several addresses. For tickets, current and historical information are now kept separately.

Develop with Production Data

Context

You need to migrate data from a legacy system to a new system whose capabilities and domain model differ significantly from the legacy system. The exact semantics of the legacy data are not documented.

Problem

Once the data migration has started on the production environment, the migration code should run mostly flawless to avoid migrating inconsistent data into the target system.

How do you ensure that the migration code covers all cases of the legacy data?

Forces

The following aspects make the problem difficult:

- *The legacy system's exact behavior is lost in history.* Its developers may not be available anymore or do not remember the exact requirements determining the legacy system's behavior.
- *All existing states in the data must be revealed.* During the development of the migration code many decisions must be made how to handle special cases. Missing knowledge about some cases data must not go unnoticed for a long time. At least you should know which cases exist that you may safely ignore.
- *Wrong assumptions.* Unit tests are a good means to verify assumptions about how the migration code should work. Such tests cannot guarantee, however, that these assumptions are correct.
- *Insufficient test data.* Testing the migration code with hand-crafted test data does not guarantee that all corner cases are covered because some corner cases might just not be known.

Solution

From the beginning on, retrieve snapshots of the legacy data from the production system and develop and test the migration code using real data.

Try to get a complete snapshot of the legacy data from the production system, or, if that is not feasible because of the data size, try to get a significantly large amount of legacy data to cover most corner cases. If the legacy data changes during the development,

regularly update your snapshot and always develop against the most current available data. All developers should use the same data set to avoid any inconsistencies.

Whenever your code makes wrong assumptions, the migration is likely to fail. Even though you still have no documentation at hand for the exact semantics of the legacy system, you are informed early on about gaps in your knowledge and therefore in the migration code.

If the amount of legacy data is too huge to work with on a daily base, try to define a working set that most likely covers all cases. You could reduce the amount of data, for example, by copying only those data sets that have been used (read/modified) in the production system lately.

Production data often contains sensitive information that must not be available to everyone because of legal reasons, business reasons or privacy protection. Examples are credit card information from customers or data that has a high business value such as financial trade data. In such cases, production data needs to be modified (e.g. anonymized) before the development team is allowed to use it.

If you test your migration code on production data, you get a good feeling for how long the migration might take at go-live.

Consequences

Applying this pattern has the following *advantages*:

- *Detecting corner cases.* When production data is fed into the migration code from the beginning on, it is more likely that all boundary conditions are detected during test runs.
- *Increasing the understanding of data semantics.* Developers improve their understanding of the implicit meaning and the quality of the existing legacy data. If they are forced to run their code with real data, they cannot decide to silently ignore corner cases.

You need also to consider the following *liabilities*:

- *Slow progress in the beginning.* If there are many corner cases to handle, the development efforts may slow down significantly in the beginning because you cannot as easily concentrate on the most common cases at first.
- *More effort.* Taking snapshots of legacy data might involve extra effort for extracting it from the legacy system and loading it into a test system. Setting up such a test system might be a time-consuming task on its own.
- *Too much data.* The complete production data may be too voluminous to be used in test runs. In that case, you need to find out whether you can restrict testing to a subset of the production data. If you Migrate along Domain Partitions, you may be able to restrict test runs to subsets of the full legacy data.

Example

In our example CRM system, we set up jobs triggered by a continuous integration system. The jobs perform the following task at the push of a button or every night:

- Dump the legacy database to a file storage and restore the database dump to a test legacy system.
- Set up a clean test installation of the new system with an empty database only filled with master data.
- Run the current version of the migration code on the test systems.

Migrate along Domain Partitions

Context

You want to migrate data from a legacy system to a new system. The legacy system's capabilities cannot be matched one-to-one onto the new system but the domain of the target system resembles the legacy system's domain.

Problem

The migration of a data entry of the legacy system may affect multiple domains of the target system. Moreover, even multiple business units may be affected. The complexity of migration projects can be overwhelming.

How can you reduce the complexity of a migration effort?

Forces

The following aspects make the problem difficult:

- *The legacy system's domain model may be ill-defined.* This may be the result of inadequate design, or of uncontrolled growth within the system's domain model. It is unclear how data found in the legacy system is structured and interrelated.
- *Different users have contradicting views on the legacy system.* Every business unit employing the legacy system may have a different conceptual model of what the legacy system does. It becomes very hard to analyze the legacy system in a way that provides a consistent view on its domain model.
- *The technical view on the legacy system is often counter-intuitive.* The development team more often than not does only have a view on database tables or similar artifacts of the legacy system. Semantics associated by business users with special states in raw data is not visible. However, a successful migration relies on eliciting the intricacies of the legacy system.

Solution

Migrate the data along sub-domains defined by the target system.

Introducing a new system allows for defining a consistent domain model that is used across business units [2]. This model can then be partitioned along meaningful boundaries, e.g. along parts used primarily by different business units.

The actual migration then takes place sub-domain by sub-domain. For example, in an order management system there may be individual customers, each having a number of

orders, and with each order there may be tickets associated with it. It thus makes sense to first migrate the customers, then the orders, and finally the tickets.

For each sub-domain, all necessary data has to be assembled in the legacy system, regardless of the legacy system's domain model. For example, the legacy system could have been modeled such that there was no explicit customer, instead each order contains customer information. Then in the first migration step, only the customer information should be taken from the legacy system's order domain. In the second migration step, the remaining legacy order domain data should be migrated.

Consequences

To Migrate along Domain Partitions offers the following *advantages*:

- *The data migration becomes manageable.* Due to much smaller building blocks it becomes easier to spot problems. Furthermore, individual migration steps can be handled by more people simultaneously.
- *Each step of the data migration is testable in isolation.* This reduces the overall test effort and may increase the quality of tests if additional integration tests are provided.
- *Verifying the migration results can start earlier.* As soon as a single sub-domain migration is ready, business units affected by this sub-domain can verify the migration results. It is no longer necessary to wait until all migration code is ready.

To Migrate along Domain Partitions also has the following *liabilities and shortcomings*:

- *The migration run takes longer.* It is very likely that data from the legacy system has to be read and processed multiple times.
- *Errors in migrations propagate.* If there is some problem with the migration of a sub-domain performed in the beginning, many problems in dependent sub-domain migrations will likely result. Thus, the earlier the migration of a sub-domain occurs, the higher its quality must be. This necessitates an iterative approach to implementation and testing.
- *Redundant information might be introduced.* If data of the legacy system is read multiple times, there is the possibility of introducing the same information into the target system multiple times. This may be necessary if distinct artifacts have had the same designator in the legacy system, but may also be highly problematic. Care has to be taken to avoid any unwanted redundancies in the target system.

Example

In our example CRM system, the following data partitions might be possible:

- Customer base data
- Customer addresses
- Ticket data
- Ticket history

The data migration process first processes all entries to generate the base customer entities in the new system. Then, it runs once more over all entries of the legacy system to create address entries as necessary. In the next step, all current information about sold tickets are migrated with references to the already migrated customer entries. As last step, historical ticket information is retrieved from the legacy table and written into the new history table.

Measure Migration Quality

Context

You need to migrate data from a legacy system to a new system. The legacy system's capabilities cannot be matched one-to-one onto the new system. A staging system with production data is available so that you can Develop with Production Data at any time. The migration code is regularly tested against the production data.

Problem

Resources are limited. Thus, there is often a trade-off between migration quality and costs, leading to potential risks when deploying the migration code.

How do you prevent the migration code from transforming data wrongly unnoticed?

Forces

The following aspects make the problem difficult:

- *Some changes cannot be rolled back.* Although you may have data backups of the target system available, it may not be easy or possible at all to roll back all changes to the target system. You therefore need to know the risk of migrating corrupted data.
- *Down-times need to be avoided.* Even if it is possible to roll back all changes, a migration run often requires all users of both the source and the target system to stop using them. Such down-times may be very costly and therefore need to be avoided.
- *Know when to stop improving the quality* The 80-20 rule applies to many migration efforts. Business may not want to achieve 100% quality because it might be cheaper to correct some broken data entries manually or even leave some data in an inconsistent state.
- *Small technical changes in the migration code often lead to significant changes in the migration quality.* Yet, the people designing and implementing the migration code are usually not able to fully assess the implications of their changes.

Solution

In collaboration with business, define metrics that measure the quality of the migrated data and make sure that these metrics are calculated regularly.

Integrate well-designed logging facilities into the migration code. Make sure that the result of each migration step is easily accessible in the target system, e.g. as special database table. For example, there may be an entry in the database table for each migrated data set, indicating if it was migrated without any problems, or with warnings or if some error occurred that prevented the migration.

Add validation tasks to your migration code suite, providing quantitative and qualitative sanity checks of the migration run. For example, check that the number of data sets in the legacy system matches the number of data sets in the new system.

Make sure that at least the most significant aggregated results of each migration test run are archived to allow for comparing the effects of changes in the migration code.

Consequences

To Measure Migration Quality offers the following *advantages*:

- *Results of migration become visible.* By checking the logged information, you can quickly see what the migration actually did.
- *Quick feedback cycles.* The effects of each change in the migration code upon the migration quality metrics are visible immediately after the next migration test run. This allows the development team to quickly verify the results of their decisions.
- *Implicit assumptions on structures in the legacy system are revealed.* Whenever a developer makes implicit assumptions on some properties of the legacy system that are not correct, it is very likely that the migration will produce errors or warnings for at least some data sets.
- *Potential for legacy data clean-up is revealed.* Problems that are not coming from the migration code itself, but from data quality issues in the legacy system, will be revealed before the actual migration in the production system. This provides the unique opportunity to clean up the data quality in a holistic fashion.

To Measure Migration Quality also has the following *liabilities and shortcomings*:

- *Additional up-front effort necessary.* Setting up test suites to measure the quality of the outcome requires additional effort from the beginning on.
- *Challenge to find suitable metrics.* You have to be careful to create meaningful metrics. If there is a warning for every single data set, it is very likely that people will ignore all warnings.
- *Too much transparency* Not everyone may appreciate data quality problems detected by thorough quality analysis in the legacy system. Pointing people to required efforts might cause pressure against the overall migration project, along the lines of "if we have to do this much data cleaning to migrate, it is not worth the effort".

Example

In our example legacy CRM system, there are no database foreign key relationships between a ticket and a customer the ticket refers to. Instead, user agents were required to fill in the customer's ID in a String field of the ticket table. In the target system, referential integrity is enforced.

The results of the migration are stored in a logging table, having the following columns:

- *Source value* designates the primary key of an entry in the legacy system.
- *Target value* designates the corresponding primary key of an entry in the target system.
- *Result* is either OK or ERROR.
- *Message* gives additional explanations about the migration described by the log entry.

As part of writing the migration code, we check if the target system's database throws an error about a missing foreign key relationship whenever we try to add a migrated ticket entity to it. If it does, we write an *ERROR* entry along with a *Message* telling *Ticket entity misses foreign key to customer. Customer value in legacy system is XYZ.* into the migration log table, otherwise we write an *OK* entry.

Daily Quality Reports

Context

You need to migrate data from a legacy system to a new system. You Measure Migration Quality to get results on your current state of the migration after every test run and are therefore able to control the risk of the data migration. Migration tests are run regularly, e.g. every night in an effort of Continuous Integration [1]. You want to involve business experts into testing the outcome of the data migration early on.

Problem

IT cannot fathom the quality attributes of the domain. Therefore, the decisions defining the trade-offs are made by business units and not the IT department implementing the migration. However, it is not always easy to get feedback on the effect of some code change onto the migration quality.

How can you closely involve business into the data migration effort?

Forces

The following aspects make the problem difficult:

- *People are lazy.* If there is any effort involved in controlling the current state of the migration quality metrics, it is likely that business won't notice any significant changes.
- *Business has little time to spare.* You cannot expect business to analyze and understand measured data about the migration quality in detail.

Solution

After every migration test run, generate a detailed report about the state of the migration quality and send it to dedicated business experts.

To make the reports easily accessible to business, provide aggregated views of the migration results to allow for trend analysis. For example, provide key statistics indicating the number of successfully migrated data sets, the number of data sets migrated with warnings, the number of data sets that could not be migrated due to problems of data quality or the migration code and the number of those not migrated due to technical errors.

It is important to start sending daily quality reports early enough to be able to quickly incorporate any feedback you get. It is also important not to start these reports too early to avoid spamming the receivers with mails that they tend to ignore if the reports

are not meaningful to them. It also crucial to find business experts that are willing and interested in checking preliminary migration results.

Make sure that the migrated data is written into a database that is part of a test environment of the new system. Business experts who receive quality reports may then easily check the migrated data in new system themselves.

Consequences

Daily Quality Reports offer the following *advantages*:

- *The state of the migration code becomes highly transparent.* If the results of each migration test run are made accessible to business units, they can closely track the progress of the development team charged with the migration.
- *Easy to obtain feedback.* By triggering an automatic migration run on the test environment, it becomes comparatively easy to receive immediate feedback on any changes to the migration code.
- *Risk is delegated to business.* By receiving feedback early and often, business experts are able to evaluate the risk of the data migration effort. If necessary, managers can be involved to decide, for example, whether some quality issues should ignored or resolved even if that influences the project plan.

Daily Quality Reports also have the following *liabilities and shortcomings*:

- *Efforts necessary for generating reports.* Additional up-front effort is required to set up the reporting structures. To ensure comparability between different test runs, the reporting structures have to be defined before the actual migration code gets done.
- *Political issues.* The high degree of transparency might lead to political issues in environments that are not used to it. If corporate culture dictates projects to report no problems, it might be a bad idea to let people from outside the migration development team peek into the current, potentially problematic state.
- *Possible delays because of data volume.* If there is much data to be migrated, the execution time of a full migration test run may be very long. This prevents tight feedback loops.
- *Regular updates of legacy data may be needed.* If the data in the legacy production system changes regularly (for example because business users clean up the existing legacy data), you may need to also automate or at least simplify transfering production data into the migration test system so that business is able to check the effect of their data cleaning on the new system.

Example

To provide daily quality reports in our example system, we add some code at the end of a migration test run that aggregates the number of *OK* and *ERROR* entries and reports the numbers such that trend reports can be built using Excel and provided to business people for reporting purposes. If necessary, the generation of high-level trend reports can be implemented as part of the migration code, and a continuous integration server's mail capabilities are used to send the reports to business.

Also, lists are generated that show all *source values* leading to *ERROR* entries. These lists are given to user agents to manually correct them and to the IT development team to come up with algorithms that cleverly match lexicographically similar references. After each subsequent test run, the procedure is repeated until a sufficient level of quality is reached.

Related Work

Regardless the importance of data migration projects, there is currently surprisingly few literature on this topic.

The pattern collection *Transform!* by Andreas Rüping [5] contains several patterns for data migration projects that nicely complement the pattern collection presented here.

The technical details of data migration efforts have also been described in pattern form by Microsoft [6]. Also, Haller gives a thorough overview of the variants in executing data migrations [4] and the project management tasks associated with it [3].

Acknowledments

The authors are very thankful to Hugo Sereno Ferreira who provided valuable feedback during the shepherding process.

The workshop participants at EuroPLoP 2010 also provided lots of ideas and suggestions for improvements, most of which are incorporated into the current version.

Special thanks also go to Andreas Rüping for his close cooperation after we discovered that we are working on the same topic at the same time.

References

[1] Paul M. Duvall, Steve Matyas, and Andrew Glover. *Continuous Integration: Improving Software Quality and Reducing Risk*. Addison-Wesley Longman, 2007.

[2] Eric J. Evans. *Domain-Driven Design: Tackling Complexity in the Heart of Software*. Addison-Wesley Longman, 2003.

[3] Klaus Haller. Data migration project management and standard software experiences in avaloq implementation projects. In *Proceedings of the DW2008 Conference*, 2008.

[4] Klaus Haller. Towards the industrialization of data migration: Concepts and patterns for standard software implementation projects. In *Proceedings of CAiSE 2009, LNCS 5565*, pages 63–78, 2009.

[5] Andreas Rüping. Transform! - patterns for data migration. In *EuroPLoP 2010 - Proceedings of the 15th European Conference on Pattern Languages of Programming*, 2010.

[6] Philip Teale, Christopher Etz, Michael Kiel, and Carsten Zeitz. Data patterns. Technical report, Microsoft Corporation, 2003.