Patterns in Data Conversion

Allard Muis
May 2009

Master Thesis
Computer Science
University of Twente

Under supervision of:

University of Twente
Dr. M.L. Ponisio
Dr. P.A.T. van Eck

Quinity B.V.
Drs. J. Snijders
J. Kleerekoper, MSc.

Abstract

When a legacy information system is replaced by a new information system, the data inside the legacy system needs to be moved to the new system. This process is called data conversion. In the data conversion the data is exported, transformed to fit the new systems data structure and imported in the new system.

Making a design for a data conversion is a complex and time consuming task. Designers feel the need to improve the efficiency of the design process. They would like to share and reuse experience of data conversion design in other projects. There is currently no structured way of doing this.

This thesis proposes the use of design patterns in data conversion design in order to enable designers to reuse experience. These data conversion patterns describe a common, recurring problem in data conversion design and give a solution to resolve the problem that has proven to work effectively in the past. Five data conversion patterns are provided.

The patterns are structured in a pattern language. This language groups patterns that give a solution for the same problem. Using this, a designer can quickly find multiple alternative solutions for a data conversion design problem. New patterns can easily be added to the pattern language.

A designer needs to decide which solution fits the context of a particular problem best. In order to make this decision easier, every pattern is weighted on eight quality attributes. Depending on the context and situation, a good score on some attributes is more important than others. The quality attributes also give very clear insight in the tradeoffs of a pattern.

The data conversion patterns were evaluated by a discussion with several experienced designers. They recognize the problems and solutions described in the patterns and support this solution for enabling reuse is data conversion design. Comments of the experts were used to improve the patterns further.

Data conversion patterns allow designers to share design experience with each other and reuse this experience in other projects. Using patterns designers can make better, explicit design decisions based on experience recorded in the past. This will lead to improved efficiency of the design process, resulting in better design while needing fewer resources.

The research presented in this thesis forms a good basis for further research. The quality attributes could form a basis of assessing design quality. The effect of data conversion patterns should be tested in an experimental setting. Expanding the number of data conversion patterns will increase the usefulness of the patterns.

Acknowledgements

Before you lies the conclusion of the final project of my study Computer Science. It has taken a long time and sometimes a lot of struggle with myself to get it finished. I would like to thank those that helped me get results and shape this thesis to what it is now. My university supervisors Laura and Pascal who always managed to motivate me to go on and asked those questions that made me think in the right direction. Jeroen and Jacob from Quinity who patiently explained their opinions and always made time for me to help me out. All the other people at Quinity that shared their experience and helped me with the information I needed for my research. And also thanks to my follow students at Quinity with whom I shared the room and a lot of time playing football matches.

Table of contents

Chapter 1

Introduction

1.1 Background

Data conversion is the act of moving data from one information system to another information system. Since two information systems are usually structured differently the data needs to be converted. The process of data conversion comprises three parts: exporting data from the source system, restructuring the data and importing the data in the target system. In this thesis, we will concentrate on the restructuring part of data conversion.

A data conversion cannot be done ad-hoc, but must be designed first. Such a design contains decisions about what data is moved where, how this data is restructured and how this can be executed. For systems that have a very elaborate datamodel (i.e. a large number of tables, columns and relations between them), like systems for insurance and banking organizations, designing the way the data is converted can be a time-consuming task.

A conversion designer needs to make a lot of design decisions. He comes across many problems that he needs to design a solution for. For example, every time a single field from the source system needs to be mapped to a single field in the target system, he needs to decide whether the value of the target field should be looked up in a table or that maybe an algorithmic function would be better. Or maybe still another way? The designer needs to find the up and downsides of each approach and judge which solution fits the situation best.

Another example of such an issue is the order in which entities are converted. Should we start with converting every insurance policy, and then all coverages of the policies? Or is it better to convert one policy, then convert all coverages that policy has and move on to the next policy and its coverages?

Later on in the design or in the next project, if the same situation occurs, the designer will need to think again about the possibilities and find the best solutions. While he could fall back on the way he solved it the previous time, it may not easily be recognizable as a similar case. It may not be obvious to the designer that he is solving the same thing again. Even if he notices, a slightly different situation calls for a new evaluation of the possible solutions.

Creating a data conversion design is a highly repetitive task. The same kinds of problems are encountered very often in different projects and even multiple times within a single project. This repetitive nature calls for a way of reusing design and design experience.

The concept of design experience is difficult to define. According to WordNet Online (WNO), experience is "*the accumulation of knowledge or skill that results from direct participation in events or activities*". Data conversion design experience is thus the knowledge and skills you learn from creating data conversion designs. Documenting such experience in order not to forget and share it with other designers is important but difficult.

Design patterns have emerged over the last decade as a good way to reuse solutions and design experience. Many different types of patterns have been described to enable reuse at different places and at different levels of abstraction. Examples are technical design patterns (GHJV95)(YA04), user interface design patterns (WVE00), analysis patterns (Fow97)(Fow02) and functional design patterns (Sni04).

The research reported in this thesis has partly been performed at Quinity B.V. Quinity is a company that develops e-business solutions and information systems. For the largest part, these systems are administrative applications for insurance companies in the Netherlands. Such systems have large databases containing customer information, insurance policies and claims. When a new application is delivered, the data from the old, obsolete system must be converted. Having done many of such projects, Quinity has considerable experience in data conversion.

Quinity already reuses design experience in the design of new systems. Functional Design Patterns (Sni04) are used in the functional design in order to reuse functionality such as workflows and time dependent information. Technical design patterns from Gamma et. al. (GHJV95), among others, are used in the technical design. During implementation, reuse is stimulated with a framework and standard components. Because of their experience with reuse through patterns, Quinity engineers noted that in practice there is currently no good way of reuse in data conversion design.

1.2 Problem statement

Making a design for a data conversion project is often complex, expensive and time consuming. This does not mean that designers are unable to make quality conversion designs but that they feel the need for improved efficiency, in order to make good designs using fewer resources or better designs with the available resources.

Researchers are acknowledging this need for improved efficiency for a long time already and research into this topic goes back as far as the 1970s. Shu, Housel and Lum (SHL75) defined the languages 'DEFINE', which can be used to describe data structures and 'CONVERT', which can describe the mapping between two data structures. These two languages were intended to serve as a means to specify a data conversion. A little more recently, Cluet et al. (CDSS98) presented the language 'YATL' which has a similar goal but is better suited for modern relational databases. Languages like 'CONVERT' and 'YATL' enable a designer to effectively specify a data conversion design and communicate this to other stakeholders.

An abundance of commercial tools are available to help the data conversion designer. Oracle Migration Workbench (ORA) helps migrating database systems to the Oracle platform. DataBridger (TAU), targeted at real-time data conversion, allows mapping data from source

to target system using a graphical editor. Graphical data mapping is also possible with MapForce (ALT) which supports converting data from and to a wide variety of data sources.

The field of data warehousing is related to data conversion. Data warehouses are used to store and analyze vast amounts of data from different sources. Data coming from these different sources needs to be transformed in order to be combined and stored (SV08). While a data conversion project is a one-time transformation and a data warehouse continuously receives and transforms data, the transformation is similar. Data warehousing is supported by ETL (extraction, transformation, loading) tools. Vassiliadis et al. summarize the tasks of ETL tools: identifying and extracting data from source systems, integrating the data from multiple sources into a common data format, cleaning the data and loading the result into the data warehouse (VVS+01). Deciding what data should be combined and what transformations should be applied on the data still needs to be done by a human designer.

Each of these approaches can indeed increase efficiency of data conversion design. At the same time, there is still room for improvement as none of these approaches enables designers to reuse and share design experience they gained in the past. Therefore, the problem statement of this research is:

> "It is currently only possible to reuse design experience of data conversion in an ad-hoc, suboptimal way. There is no structured, effective way to reuse design experience."

1.3 Research questions

The problem statement leads to the following main research question:

> "How can we improve the efficiency of making data conversion design?"

In order to construct an answer to this we need to research the following questions:

- What criteria should the solution meet for it to indeed increase efficiency?
- What are the elements of a data conversion design?
- How should data conversion design experience be documented?
- What design experience is available and can be recorded?

1.4 Approach

This section explains the way we will analyze and solve the problem discussed in the previous sections. The problem statement of this research is a practical problem. Wieringa (Wie07) defines a practical problem as "(…) a difference between the current state of the world and the desired state of the world." This opposed to a knowledge problem, which is solved by acquiring knowledge but does not change the state of the problem domain. Hevner et al. (HMPR04) call a practical problem a design problem. According to them, solving a design problem involves the design of an artifact that will address this design problem, thus reducing the difference between the current state and desired state of the world.

The engineering cycle, shown in figure 1.1, is a well known way of solving a design problem. In this thesis we will use the engineering cycle as the approach in solving the problem statement. The first step of the engineering cycle is the problem investigation. In this step we analyze the problem discussed in section 1.2 and make a list of acceptance criteria which the solution should meet. If a solution does not satisfy these criteria (to a certain extend), it does not really solve the problem at hand.

In the second step, the solution design, we design a solution to the problem. This design is validated in the third step, solution validation. Here the design of the solution is compared to the acceptance criteria and we check to what extend the solution satisfies these criteria.

In the fourth step the solution is implemented: "realized and placed in its use environment" (PER08). Using the solution proposed in this thesis and starting to reuse design experience is outside the scope of this research. Therefore, solution implementation is not part of this thesis.

The engineering cycle closes with the evaluation of the solution. Does it really solve the problem as intended? What new issues arise within the altered state of the world? We will investigate such questions in the last part of the thesis.

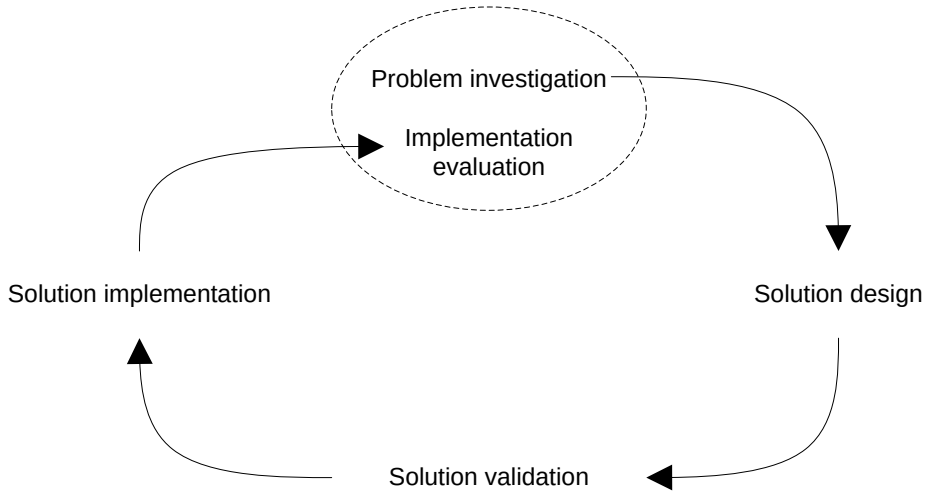


Figure 1.1: Engineering cycle (adapted from (PER08) and (Wie07))

1.5 Thesis structure

This thesis is structured in five chapters. The following chapters contain the results of each of the steps of the engineering cycle.

In **Chapter 2** we analyze the problem and the problem domain of data conversion by looking into the elements of data conversion design. We give a broad overview of data conversion and zoom in on parts where we can reuse design experience. This analysis is based on literature on the subject, discussions with data conversion designers and study of design documentation of projects executed by Quinity. At the end of the chapter we discuss the acceptance criteria of the solution that will be proposed.

Chapter 3 describes the proposed solution. We discuss how we can most effectively document data conversion design experience. Based on discussions with designers and design documentation we propose a library of design experience that can be reused.

In **Chapter 4** we validate and evaluate the proposed solution. We ask ourselves whether the solution proposed in chapter 3 matches the acceptance criteria of chapter 2 and we discuss the solution with data conversion designers.

In **Chapter 5**, the conclusions, we look back at the research questions and discuss what we have learned by performing the research.

Chapter 2

Data Conversion

This chapter explores data conversion in more detail. We look at when you need data conversion, what data conversion looks like, the phases a data conversion project consists of and the design of data conversion.

2.1 System replacement

Many companies are using old information systems to support their business processes. These systems were designed and created with technology that may have been state of the art at that time, but is now obsolete. Systems like this are often called *legacy systems*. Brodie and Stonebraker (BS95) define a legacy system as "any information system that significantly resists modification and evolution".

Data conversion is necessary when an organization decides to replace a legacy system with a new information system. While developing a new system is expensive, maintaining a legacy system may be even more expensive in the long run (Vis01, BCV03). It is increasingly hard to find personnel with the necessary skills in the aging techniques and it is very hard to evolve the information system to keep supporting the changing needs of the business processes. Replacement is a good opportunity to implement new functionality that was not supported by the legacy system.

Replacing it is not the only technique to cope with the challenges of legacy systems (Ben95). For example, encapsulating a legacy system with modern technology can be a useful and less expensive technique. There is no need for data conversion in this case because the data remains in its original database.

2.2 Data conversion

When the datamodels of the legacy system and the new system differ, fields in both systems have different names or meanings, or the new system imposes stricter rules on the data, the data needs to be restructured before it can be imported into the new database. This process of exporting, restructuring and importing is called *data conversion*.

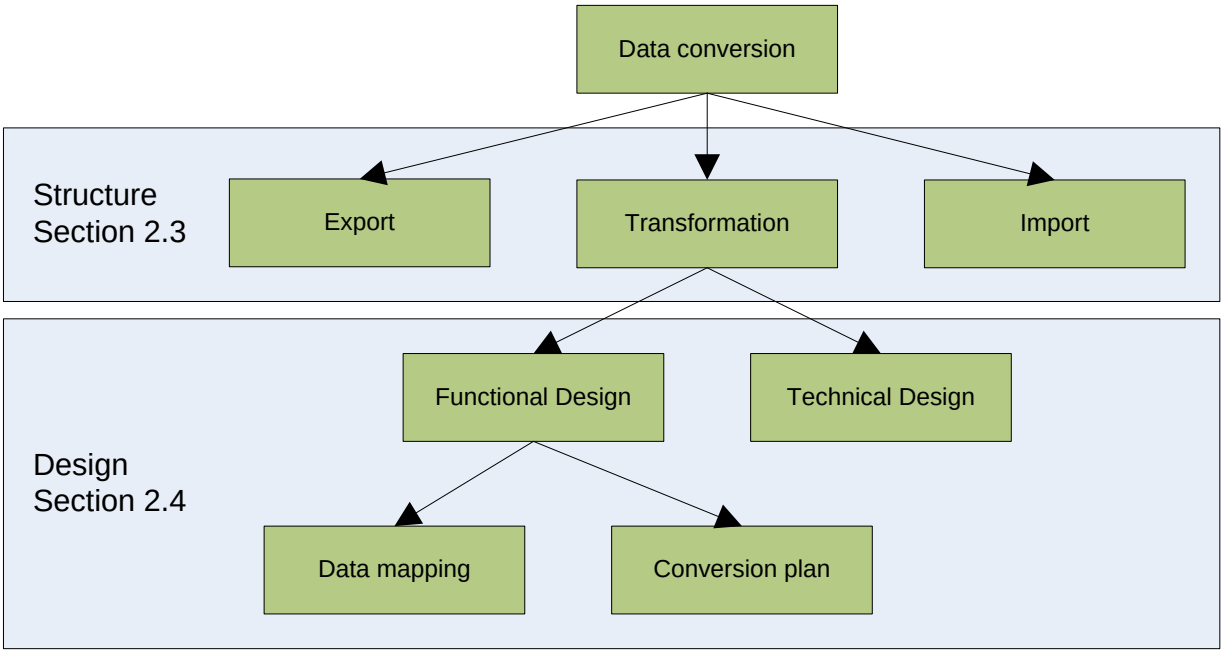


Figure 2.1: Data conversion aspects

Figure 2.1 gives an overview of the aspects of data conversion. It is divided in two parts: structure and design. The structure of data conversion will be discussed in section 2.3 while the design of data conversion is the subject of section 2.4. Not all parts shown in the picture are analyzed equally thoroughly, the focus lies on the parts that are candidates of reusing design experience.

Data conversion is possible between any two information systems. As long as there is data in one system you can convert it to another system, it does not matter what kind storage technology the system employs. Very old legacy systems are still using text-file based storage systems, while XML and object based technology is on the rise. The majority of information systems are built on a relational database, however, and it looks like relational databases will be the dominant force for some time at least. Therefore, we will be focusing on relational databases in this thesis.

2.3 Structure of data conversion

Figure 2.2 depicts data conversion. The system which data is being converted is called the *source system*; the system that receives the converted data is called the *target system*.

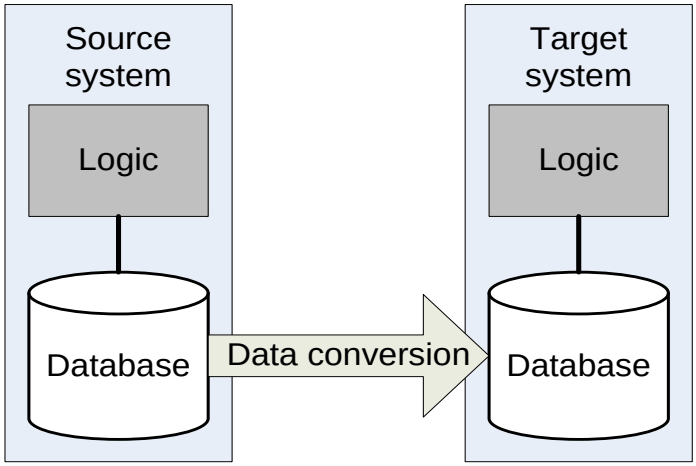


Figure 2.2: Data conversion

Data conversion does not concern the design or creation of any part of the target system, just the conversion of the data itself from the source system to the target system.

Conversion is done in multiple steps. First, the data is exported from the database to a set of datafiles. These files can then be used exclusively by the conversion. The source datafiles are then converted to the target datafiles in an 'in between' step. Finally, the resulting target datafiles are imported in the database of the target system. Figure 2.3 shows this process. In each of these three steps part of the conversion can be performed. Some types of operation are best done in one of the three steps. For example, removal of redundant information is best done during the 'export' step, while introducing new redundancy in the target system should be done in the 'import' step.

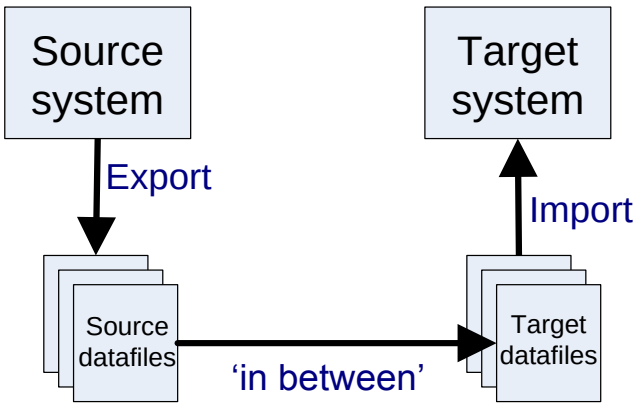


Figure 2.3: Conversion process in three steps

Exporting the data from the source system can be a challenge in itself. It can be easier to make use of the existing interfaces the system has. Interfaces are meant to communicate with other systems. If an interface of the source system can be used to get all or most data that must be converted, it can be used to get the data out of the system to fill the source datafiles. Thus, there are two basic ways to get data out of the source system: directly from the database or through the source systems' logic. This is shown in figure 2.4.

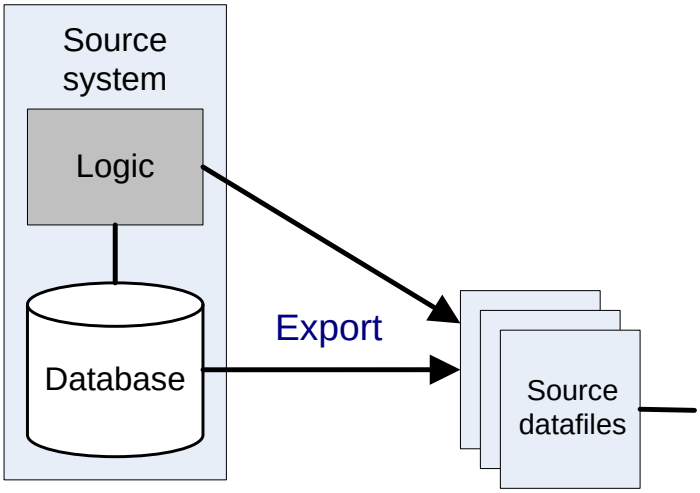


Figure 2.4: Export using an interface

Such an interface can be used for the target system as well. Data from the target datafiles can be imported through the logic of the target system or to its database directly, as can be seen in figure 2.5.

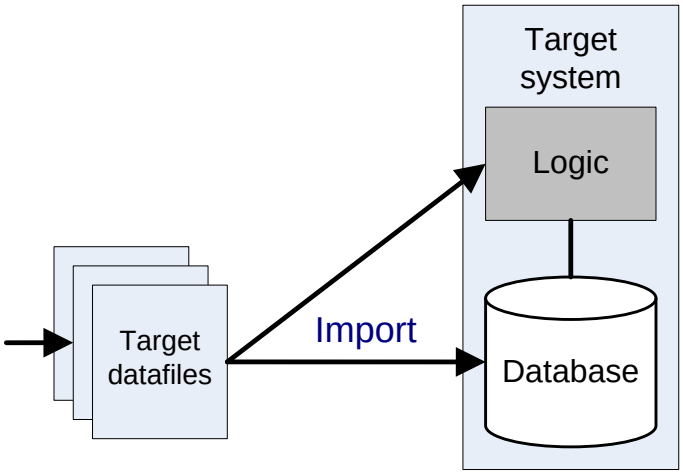


Figure 2.5: Import using an interface

It is not necessary to do the 'in between' step at once. The step can be divided in multiple smaller steps. The intermediate results are stored in intermediate datafiles that are converted further to the target datafiles. An example of this process with one intermediate step is shown in figure 2.6.

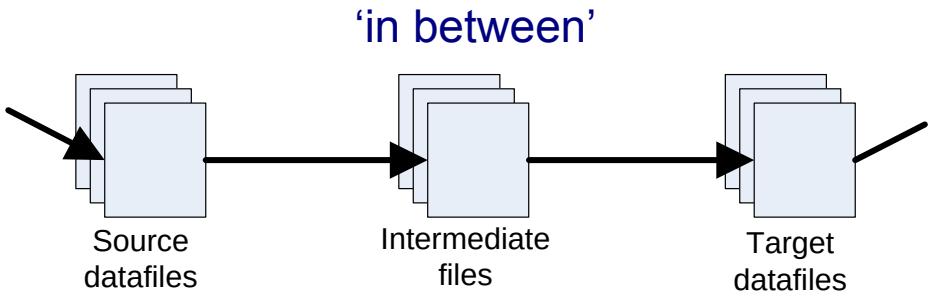


Figure 2.6: 'in between' conversion in multiple steps

2.4 Data conversion design

According to Purba's Data Management Handbook (Pur99), a data conversion project is comprised of several phases. After these phases have been discussed, we will look how they relate to the design of data conversion.

1. Determining if conversion is required

In some cases, Purba claims, conversion is not required because the data is available at other locations. The data may also have such a low quality, i.e. much erroneous or missing information, that conversion is not worth its cost.

2. Analysis of source datamodel

This step should give the developer understanding of the source systems' data model. When there is very limited documentation about the system available and the systems' owner can provide only limited information, this step can be difficult and time consuming. But without a good understanding of the datamodel and the meaning of the data, it can never be successfully converted to the target system.

3. Investigating data quality

It is important to know whether the data is consistent with the datamodel. Inconsistencies will lead to problems during the conversion or a lot of data that cannot be converted automatically. For example, many legacy systems make extensive use of free text fields. Users of the systems often abuse these fields to store other data than the field was intended for. By investigating the quality of the data an estimation of these problems can be made.

4. Analysis of target datamodel

The developers of the conversion also need to understand the datamodel of the target system. The target system is often better documented than the source system as it is brand new. This analysis is therefore not as difficult as the analysis of the source system.

5. Determining the data mapping

When the developers understand both the source and target datamodel they can design the mapping between the models. This mapping specifies for each field in the target datafile from what field in the source datafile the data must be taken. Often, this involves difficult steps where fields are filled depending on the value of other fields, where data from a single field must be split to multiple other fields or where the value of a field in the target datafile depends on the values of many fields in the source datafile. A lot of research has been done into data quality and data cleaning in the field of data warehousing, for example see Rahm and Do (RD00).

6. Determining how to treat missing information

Sometimes, data that is needed to fill the target database is not available in the source database. In these cases it is unclear how to fill the target fields. A solution must be found, for example creating dummy values or skipping those parts of the conversion.

7. Programming the conversion

When it is decided how to convert the data from the source to the target system, software to execute the conversion must be written and tested.

8. Running the conversion

When the conversion program is created, it must be executed. This will actually fill the target datafiles.

9. Verifying conversion

When the conversion is finished, it is important to verify that all data is indeed converted. For example, it could be unacceptable to have 'forgotten' to convert a few customers. So it must be ensured that the target database contains all data, and that all data is correct.

There are two forms of design of data conversion: functional design and technical design. The functional design of data conversion describes all the issues concerning the conversion on a conceptual level. This design discusses the result of phases two to six: analysis of source and target data models, investigation of data quality, the mapping of the data and the way missing information is treated. It also determines what data is converted and what not. For example, it may be decided not to convert customer information that is over ten years old as there is no purpose for this information anymore.

While all the parts of the functional design have their own problems and difficulties, designers especially recognize the data mapping (step 5) as a place where they encounter many reoccurring problems. This leads us to believe that this is the place where documenting and reusing design experience would be the most efficient. Therefore, we will concentrate on data mapping problems for the rest of the thesis. Research on reuse of design experience of other phases, for example determining how to treat missing information, may very well be worthwhile but is outside the scope of this thesis and is considered future research.

The technical design concerns the technical issues of the data conversion and contains the design for the phases seven, eight and nine. Technical design has a lot in common with technical design of 'normal' software. Reuse of technical design experience has been studied at length (see for example (GHJV95)) and will not be discussed further in this thesis.

2.4.1 Data mapping

The data mapping is part of the functional description of the conversion. It describes which fields or entities in the source system are converted to which fields or entities in the target system and what transformations are applied to this data. The way this is described varies according to how detailed the conversion designers need to document the mapping. For example, it can be done in a large table, where for every field in the target system is recorded from which fields in the source system it is taken or what formulas are used to calculate its value. It also can be done in a textual, less formal fashion: "the age of the customer is calculated using the current date and the date of birth in the source system".

There are two points of view from which a mapping can be described: from the source system or the target system. In the first case the mapping describes where to bring information to that

is present in the source system. In this thesis we will call this way the ‘push’ method, since data is pushed out of the source system to the target system. The push method describes for one or more fields in the source system to what fields in the target system the data is moved. Using this way, it is easier to verify that all data that is present in the source system is converted to the target system.

In the second case the mapping describes how the fields in the target system are filled. From now on we will call this the ‘pull’ method. The pull method describes for one or more fields in the target system what fields in the source system are used to fill these target fields. The pull method makes it easier to verify that all fields in the target system are filled.

In a single conversion design, both pull and push methods can be used. The entire mapping does not have to be described using the same method. For some fields it may be better to use the pull method while for other parts it is better to use the push method. When the push method is used for *every* field in the source system and the pull system is used for *every* field in the target system, one can ensure that all data from the source system is being converted and all fields in the target system are filled, i.e. the conversion is complete. Because completeness is an important aspect of data conversion, it is good practice to describe the conversion using both push and pull methods. However, the descriptions need to be consistent and should not contradict each other. Without tool support it may be difficult to guarantee this consistency.

The data mapping consists of a lot of small parts each describing how one or a few fields are mapped. Sometimes it is useful to describe the conversion of multiple fields at the same time, but if you describe too many fields at once it will become difficult to comprehend. These mappings are described in either the push or pull fashion. Each description has one of the following cardinalities.

One to one

A single field in the source system is converted to a single field in the target system. This is a very simple case, although a complicated transformation may be used to convert the value of this source field to the target field.

One to many

A single field in the source system is converted to multiple fields in the target system. In this case the value of this single source field solely determines the value of multiple fields in the target system. This is often the case when certain data that is stored in a single field in the source system is split over multiple fields. For example, the source system may store an address in a single field, while the target system uses two fields, one for the street name and one for the house number. Such a mapping is best described using the push method since it describes how to fill multiple fields using a single source field. A one to many cardinality can be used to split non-atomic data into multiple atomic values.

Many to one

Multiple fields in the source system are converted to a single field in the target system. Here, several source fields together determine the value of one target field. This can be the case when information from multiple source fields is put together in the target field, such as when the source system stores street name and house number separately while the target system

stores the address in a single field. But it can also be more complicated, for example when someone's age, gender and address are used to calculate the risk category someone is in for determining an insurance premium. Mappings with this cardinality are best described using the pull method.

Many to many

Multiple fields from the source system are converted to multiple fields in the target system. This is the most complicated case where in both the source system and the target system multiple fields are used in a single data mapping.

In most cases it is possible to split the 'many to many' mapping in multiple 'many to one' mappings as these are easier to understand. Sometimes, for example when multiple fields are used in a calculation which result must be stored in multiple target fields, splitting is not possible or leads to the same complex calculation being done multiple times.

Zero to one

The target system may have fields that store data that was not stored in the source system. This data is not really converted, but something has to be filled in in these target fields. Often this will be simply dummies or default values but it needs to be described in the design of the data conversion. While this case can hardly be called a mapping, it is natural to describe it in the same place. This cardinality can only be described in a pull method as no source fields are being used.

One to zero

A one to zero mapping effectively means the data from the source system is not converted to the target system. It can be useful to include such a description to make it clear it was a conscious decision not to convert the data. Clearly, only the push method can be used for describing data that is not taken to the target system.

2.4.2 Conversion plan

The data mapping describes what source fields are used to fill the target fields. But in reality only one piece of data can be read from the source datafiles and copied to the target datafiles at a time: actually converting the data is a sequential process.

We call the way this sequential process is executed the 'conversion plan'. The design of this plan concerns the order in which fields in the source system are read, when new records in the target system are created and when data is written to those records. So the conversion plan describes how the data mapping can be executed.

The following example clarifies this: an information system of an insurance company holds information about their customers and the insurance policies these customers bought. This data has to be converted to a new system and the data mapping has already been made. The conversion plan describes how this mapping is realized: "We start reading the table in the source system holding customer information. For every customer we find, we create a new record in the target systems' customer table and we move the data to the new record. When all customers are converted, we move on to the insurance policies. For every insurance policy we

find in the source table we create a new record in the insurances table in the target system. We look up the customer it belongs to and link the policy to that customer. Policies in the target system have a reference to its latest version; we leave that empty for now. Then we read the table holding the policy versions. For every policy version we create a new record in the versions table in the target system. We look up the policy it belongs to and link the version to that policy. When all policy versions are converted we create a reference from the policies to their latest version."

There are a few difficult issues regarding the conversion plan. The first is dealing with references between entities, i.e. the foreign keys in the database tables. Primary keys are usually not converted from the source system to the target system. When a record in the target system is created, a new key is made in the form of a unique id. This means that foreign keys referring to primary keys have to be changed as well. For example, when an insurance policy in the source system refers to customer with id 3456, you need to know what the id of that customer is in the target system before you can convert the policy. This means you need to keep track of how primary keys in the target system relate to primary keys in the source system.

The second issue is the ordering in which tables are converted. It is not possible to convert a table that has a foreign key relation with another table that has not been converted yet. However, when table A has a foreign key relation with table B and table B has a foreign key relation with table A, one of the tables must be done first. In this case, the column holding the foreign key must be filled with dummy values and updated after the other table has been converted. Once again, this requires careful registering of how keys in the source system relate to keys in the target system.

The third difficult issue is that of memory management. When a large database is being converted, it is often not possible to hold all the datafiles in memory continuously. This means that, when data from multiple tables is needed at the same time, these tables need to be available at the same time. This influences the order in which tables should be converted.

We identified two fields of data conversion design were design experience is important: data mapping and conversion plan. In order to reduce the complexity of this research we will focus on data mapping and reuse of experience of conversion plans will be left for future research. The choice for data mapping was made because data conversion experts at Quinity could share more experience in data mapping than in conversion plan design and we could identify more recurring problems in data mapping.

2.5 Acceptance criteria

Based on what we now know about data conversion and reasoning about how design experience can be reused, we will look at the acceptance criteria that the proposed solution should meet. In chapter 4 we will look at whether the proposed solution indeed meets these criteria.

As described in this section 2.4, most repeating problems occur in data mapping. While other phases of data conversion could hold reusable design experience as well, data mapping is the most promising place to start. Therefore, the first acceptance criterion is:

AC 1: The solution should be able to document design experience about data mapping.

In the problem statement we discussed that the goal of this research is to increase the efficiency of making a design for data conversion. This means that it should be possible to make a better design using the same amount of resources, or using fewer resources to make a design that is as good. We must avoid a solution that actually takes more time to use. Thus, designers should be able to search for the experience they need:

AC2: The library with design experience should be searchable.

For novice designers it is difficult to oversee the consequences of the design decisions they make. Handing them solutions to problems is not enough as they will still have trouble understanding the consequences of these solutions.

AC3: The solution should give insight of the consequences of design decisions.

While reusing design experience is useful, it should not restrict the freedom of the designer. Instead of telling designers how to do things, it would be better to present a number of alternatives. Designers can than compare the alternatives and decide which one they prefer, given the situation and context.

AC4: The solutions should help to objectively compare alternatives and select the best one in the current context.

Making a design is teamwork. Multiple designers work together on a design and have to communicate with other stakeholders like managers, customers and users of the data. Facilitating the communication will increase efficiency of the designing process.

AC5: The solution should facilitate communication between designers and other stakeholders.

Design experience can be very specific or less specific and broadly applicable. Design experience that is very specific and can only be applied in very rare cases does not help much. Reusable experience has the right balance between being broadly applicable and being specific.

AC6: Design experience should not be too specific but have an adequate level of granularity.

Chapter 3

Data Conversion Patterns

3.1 Introduction

In the introduction of the book "Design Patters" (GHJV95), Gamma et al. describe the following problem:

"*Yet experienced object-oriented designers do make good designs. Meanwhile new designers are overwhelmed by the options available and tend to fall back on non-object-oriented techniques they've used before. It takes a long time for novices to learn what a good object-oriented design is all about. Experienced designers evidently know something inexperienced ones don't."*

The authors recognize that design experience is the most important factor determining whether a designer can make a good design or not. Gamma et al. are talking about object-oriented design here, not about design of data conversion. But despite that, the problem they try to solve in their book is remarkably similar to what was described in the problem statement earlier.

"We all know the value of design experience. (...) However, we don't do a good job of recording experience in software design for others to use. The purpose of this book is to record design experience in designing object-oriented software as ***design patterns****."*

Today, design patterns are a huge success in software engineering. The quotes above were taken from the 32nd printing of the book "Design Patterns". There are numerous other books that contain design patters of many different sorts: technical design patterns, conceptual design patterns, patterns specifically for Smalltalk. It could be argued that design patterns themselves are a best practice in software engineering: if you want to record design experience, use design patterns.

That is exactly the approach we take here. We will try to bring design experience to the designer of data conversion with design patterns. From now on, we will call patterns specifically targeted for this purpose 'data conversion patterns'.

In the next sections we will look into data conversion patterns. Section 3.2 defines data conversion patterns and describes how we can record data conversion design experience in patterns. Section 3.3 provides an example set of data conversion patterns.

3.2 Data Conversion Patterns

In this section we determine what information should be included in a Data Conversion pattern and how such a pattern should be described.

Fowler (Fow97) defines a pattern as "an idea that has been useful in one practical context and will probably be useful in others". Gamma et al. (GHJV95) use a more restrictive definition of the word pattern and call it a description of a problem which occurs over and over again and a solution to that problem. We use this same definition, which makes a data conversion pattern a description of a data conversion problem that occurs over and over again and a solution for that problem.

The core of a data conversion pattern consists of two parts. The first is a description of a *problem*. The word 'problem' can use different things. We use the definition given by WordNet Online (WNO), Princeton University's online lexical database: "a state of difficulty that needs to be resolved". This does not imply that a conversion problem is extremely difficult or even an impossibility, but just that we want to convert data and have to find a way to do this.

In a data conversion pattern, the problem shows a source system and a target system and explains why converting the data from the source to the target is problematic in such a situation. For example, the data from the source system may not fit in the structure of the target system.

The second core part is the *solution*. The solution describes a way to overcome the problem. It shows a way how to convert the data to the target system, how to fit it in the different structure of the target system or what to do with the data to get around the situation. Thus, the solution is the resolution to 'the state of difficulty'.

When a designer is looking for a pattern, the problem description serves as a way to identify the patterns as the one he is looking for: he should recognize the problem and see it resembles the problem he is currently solving. The solution should then be applicable to his current situation and provide a solution for his problem.

Data conversion patterns, or any other kind of design pattern, cannot be constructed out of nothing, but should contain design experience that has been gained in practice. The solutions provided in the pattern should not be new; a design pattern must be based on solutions that have been successful in real systems, so their quality has been proven.

3.2.1 Quality attributes

For any given problem in data conversion, a designer can invent multiple solutions. Every solution has good sides and bad sides. In general, there is no single 'best' solution for any conversion problem. Which solution would be the best approach depends on the particular circumstances of the problem, i.e. the context. Thus, for a single problem we can write multiple patterns, each describing a different solution. Each solution is based on past experience and proved to work well in certain situations. But for a designer it is difficult to determine which patterns would work best in his current situation.

Therefore, it is not enough to give a designer access to design experience of data conversion, a set of solutions that worked well in the past for a particular problem. We must help the designer choose between these different options and select the one that fits his situation best.

In order to give the designer a tool to compare solutions, we identified eight important quality attributes of solutions to conversion problems. These attributes were selected based on opinions of data conversion experts working at Quinity. Designers already discuss design choices and alternative option along these attributes. There is no explicit list of these attributes yet, but during interviews with designers they came up as being important factors of the quality of design choices.

A pattern receives a score on each of the attributes. The values of these scores are '+' for good, '□' for mediocre and '-' for poor. A perfect solution would score well on every attribute; unfortunately, there will most likely not be many perfect solutions. Solutions will be a trade-off between these attributes; scoring well on some of them and scoring badly on others. It will be up to the designer to decide how important each attribute is in the particular context and decide whether a pattern provides an acceptable solution.

A widely accepted model for software quality was published by the International Organization for Standardization (ISO) in ISO-9126 (ISO01). The model defines six quality attributes and 27 sub-attributes. However, this model is intended for measuring software quality and cannot be applied to data conversion patterns. Many of the attributes and sub-attributes are irrelevant to data conversion patterns. Important quality attributes of data conversion patterns are not included in ISO-9126.

The eight quality attributes are each described below. Where possible, we will draw a comparison to a quality attribute from the ISO-9126 standard.

Completeness

Completeness describes whether all data from the source system is converted to the target system. This means that all data that was stored in the source should be stored in the target system after the conversion. This does not mean it has to be stored the same way, in the same structure or be available as easily.

Ideally, we would like to convert all information that is available in the source system to the target system. Sometimes it can be a better choice to do an incomplete conversion when the costs of a complete approach are high and missing some information is acceptable.

When a solution is incomplete it is very important to check exactly what information will be lost and whether that information is indeed not critical.

Loosing data can occur in different forms. The most extreme form is really loosing knowledge about something, like not converting some customers or not converting the addresses of the customers. This is generally not acceptable as a solution.

The less extreme form is converting information to a lower resolution. For example, imagine a system that stores music genres of singles. When this system is converted to another, similar system, resolution would be lost when the genres 'swamp rock' and 'pagan rock' would both be converted to 'rock'. Loosing resolution can be acceptable in certain cases, but never desirable.

Patterns can score a ‘+’ on completeness when it provides a mechanism to convert all data in the problem description completely, or a ‘-‘ when applying the pattern means losing some data.

Completeness can be seen as an interpretation of the ‘functionality’ attribute of the ISO-9126 standard. The functionality provided by a data conversion pattern is a method of converting data from the source to the target system. The better this functionality is, the more complete the solution provided by the pattern is.

Technical impact on target system

The attribute ‘technical impact on target system’ describes whether the pattern requires the target system to be changed or not.

Some solutions need the target system to be changed. This is not always possible, and even when it is, changing the target system is a major downside. It is a lot of work to design these changes, make sure the changes will not break the target system and implement the changes. Of course the target system needs to be tested because it is often difficult to assess the exact impact changes may have.

Patterns score a ‘+’ on this attribute when they require no change in the target system, a ‘□’ when the pattern requires a small change in the target system and a ‘-‘ when it requires a big change in the target system.

There is no attribute in ISO-9126 that corresponds to this quality attribute. Change in target system is very specific to data conversion.

Functional impact on target system

‘Functional impact on target system’ describes the need to change the target system functionally. After changing a system functionally it behaves differently than before. This contrasts with a technical change which has no influence on the behavior of the system, i.e. a user will not notice a technical change while he will notice a functional change.

The need to make changes to the target system is a downside in itself, it is even worse when these changes have an impact on the systems’ functionality. Changing functionality involves the end user having to adapt to the new functionality and the systems’ interaction with other systems to be influenced.

A pattern scores a ‘+’ on this attribute when the pattern requires no functional change of the target system, a ‘□’ when it requires a small change in the systems’ functionality and a ‘-‘ when it requires a big change in the functionality.

ISO-9126 has no attribute corresponding to change in functionality of target system.

Maintainability

Maintainability describes how easy or difficult it is to maintain the data in the target system after is has been converted.

Some solutions do convert the data from the source system, but the resulting data in the target system becomes hard to maintain. This is specifically the case when data is converted to fields that were not designed to keep that data. For example, when data from multiple source fields is converted to a free text field called ‘comments’, it may be hard to interpret this data later

on. When a solution requires the target systems data structure to be changed it will likely become less maintainable as design decisions are made in two different locations: the original design of the target system and the conversion design. Low maintainability may become a costly affair in the long run even when it has little impact early on.

A '+' is given to a pattern when it does not results in maintainability issues. When there are small maintainability issues the pattern gets a '□' and a '-' when the pattern introduces serious maintainability issues.

Maintainability is described in ISO-9126 as well. In the standard, maintainability concerns keeping the entire software system running correctly, while in data conversion it concerns only the maintainability of the data.

Effort to implement

The effort to implement attribute describes how much work it is to implement the solution provided in the pattern. Patterns that describe a simple mapping between the data will be quick and inexpensive to implement while solutions that require lots of calculation or changes in the target system take a lot of effort and will increase cost.

When a solution is easy and quick, it will be rated '+' on effort, when it requires medium effort it will be rated '□' and when the solution takes a lot of effort it is rated '-'.

The ISO-9126 standard gives quality attributes that can be used for running software. Effort to implement is important before the software is made and is therefore not part of the standard.

Understandability

The attribute 'understandability' describes how easy it is to understand the solution provided by the pattern. A solution that requires a complicated calculation or is dependent on many different factors is hard to understand.

In some projects clients or management are closely involved in the conversion design. In this case it is important to communicate the chosen solution to those stakeholders. It may be hard to explain complex, technical solutions to non-technical parties. When they don't understand the chosen solution they cannot give feedback or their approval on the solution.

A solution that is hard to understand is also likely to be implemented incorrectly. Testing whether the conversion was successful becomes much more complex if the design contains hard to understand decisions.

A pattern scores a '+' when it is easy to understand, a '□' when it is of medium difficulty and a '-' when it is hard to understand.

Understandability is one of the attributes of ISO-9126. We use it in the same way as in the standard.

Data integrity

Data integrity describes whether the data's integrity is guaranteed by the pattern. Data integrity is very important for a database. Some solutions may have the risk of creating data of low integrity. It largely depends on the effects of corrupt data whether one could take such a risk.

Patterns score a ‘+’ when they have no risk of compromising data integrity, a ‘□’ when they have a risk of compromising data integrity and a ‘-‘ when they have a high risk or are sure to decrease data integrity.

There is no attribute in ISO-9126 that directly corresponds to data integrity. It could be seen as a form of reliability, which is part of the standard.

Traceability
After the data conversion we should be able to check whether all data has successfully been converted. When problems are found, it should be possible to find out where it went wrong. This is expressed by the attribute ‘traceability’.

Traceability is rated ‘+’ when the patterns’ solution makes it possible to trace back values in the target system to their origin in the source system. When this is possible, we can detect records that were not converted or find out the reason behind fields that contain wrong values. A ‘□’ means that values are traceable to some extent, a ‘-‘ indicates that values are hard or impossible to trace back to their origin in the source system.

There is no corresponding attribute in the ISO-9126 standard. Traceability can be found in process quality models like CMMI, developed by the Software Engineering Institute (SEI).

3.2.2 Pattern template

The template we will use for Data Conversion patterns is based on the template of the design patterns from Gamma et al. (GHJV95) as it is the most widely used pattern template. It is adapted to make it able to describe the problem, solution and quality attributes discussed above. Each of the sections in the template will be described here.

Name
A good name it very important for a design pattern. It is the first clue in identifying which pattern to use in which situation and the name becomes part of the design vocabulary.

Intent
A very short description of the goal of the pattern, what kind of problem it addresses and what the main focus of its solution is.

Motivation
A motivation of the pattern is a sketch of a situation where the pattern would apply well and why the pattern would fit that situation.

Applicability
A description of the context in which the patterns should be applied. This will often refer to the quality attributes. For example: “use this pattern when good maintainability later is more important than low effort now”.

Solution
Here the solution to the problem is presented. It consists of a general description of the solution and shows how the solution would be applied on the example of the problem.

Example
The solution is applied on the example provided earlier in the pattern. This gives a better understanding of the presented solution.

Implementation
This section describes how to implement the pattern. In a few steps a general plan for the implementation is given. This is followed by an implementation of the patterns example. Such an example gives a designer better understanding of how the solution works and proves the solution can indeed be implemented and does work. The implementations in the patterns provided in this thesis are pieces of Java code.

Quality attributes
A discussion on how the pattern is rated for each quality attribute. It contains not only the scores, but also why the pattern is rated in this way. This enables a designer to think about the impact of the up- and downsides of the pattern.

Related patterns
Gives pointers to other patterns that could be of interest to the user of the pattern as well. Often, this will lead to other patterns solving the same problem in a different way.

3.3 A pattern language

As noted before, for one data conversion problem there may be multiple solutions. For each of those solutions we can create a data conversion pattern. While each of these patterns may be fine on its own, they would be even more useful when a designer could see the relations between these patterns. A designer should be able to see which patterns solve the same problem and he should be able to select the pattern that fits his situation best.

Therefore, we propose a pattern language of data conversion patterns. Avgeriou et al. (APRS03) define a pattern language as "a set of related patterns that collaborate inside the boundaries of an application domain". The way the patterns are related within a language varies. Gamma et al. (GHJV95) use a pattern language to show which patterns can be used in combination with other patterns, while Roberts and Johnson (RJ96) use the relationship to show the order in which patterns should be applied. We will follow the approach of Keller (Kel97) who structures his language according to the problem structure and provides alternative patterns for every problem.

The language consists of a list of problems of data conversion. For each problem it gives a description of the problem and an example that illustrates the problem, one or more patterns that solve this problem and a comparison of the patterns. This comparison is done using a table that contains the patterns' scores on each of the eight quality attributes. Using this table, designers can quickly get an impression of the tradeoffs of each pattern and on which points one pattern scores better than the others. New problems and patterns that address that problem can simply be added in the future, and new patterns holding new solutions can be added to existing problems to give designers new ways of solving the problem.

An important aspect of making design patterns is worded by Rising (Ris99) as follows: "Patterns are not theoretical constructs created in an ivory tower; they are artifacts that have

been discovered in multiple systems. It is an important element of patterns that the solution is one that has been applied more than twice. This 'Rule of Three' ensures that the pattern documents tried and true application, not just a good idea without real use behind it."

In short, you cannot invent a pattern, you have to discover it. A pattern should describe a proven successful solution. The patterns that are given on the next pages all have their roots in practical experience of data conversion designers working at Quinity, the solutions described in the patterns have been used successfully in real systems. The experience documented in the patterns was obtained by interviews with designers. They shared experiences and knowledge about data conversion design. This experience was analyzed and searched for problems and solutions that reoccurred and were generic; they could be described and explained without referring to the particular case and context they occurred in and could be applied in other contexts. These problems and solutions were worked out and grew into design patterns which were again discussed with the designers to check whether they indeed described the experience they had. After that, the patterns were evaluated with a different group of experts, as is discussed in chapter 4.

The patterns were also tested by using them in simple test systems. The solutions of the patterns were implemented in Java code. We put some example data in a test database and ran the program on this data. The resulting data had to match the expected result for the test to be accepted. The important parts of the Java code are included in the patters and the test data is the same as the data in the example used to explain the solution in the pattern. These tests show the patterns indeed work as they should and the Java code can help a designer better understand the pattern it the way it can be implemented.

The database that was used for the test was PostgreSQL version 8.3 and the test system was implemented in Java SE 6 update 11. All tests worked as expected without problems.

On the next pages the pattern language is given. It contains five patterns addressing two problems. Both problems are data mapping problems; they describe a difficulty of mapping data from the source system to the target system (see section 2.4). We do not claim these patterns contain all the reusable design experience about data conversion or data mapping, this is merely the experience that that we could find during our time at Quinity. Spending more time at Quinity, or somewhere else, will result in more potential data conversion patterns.

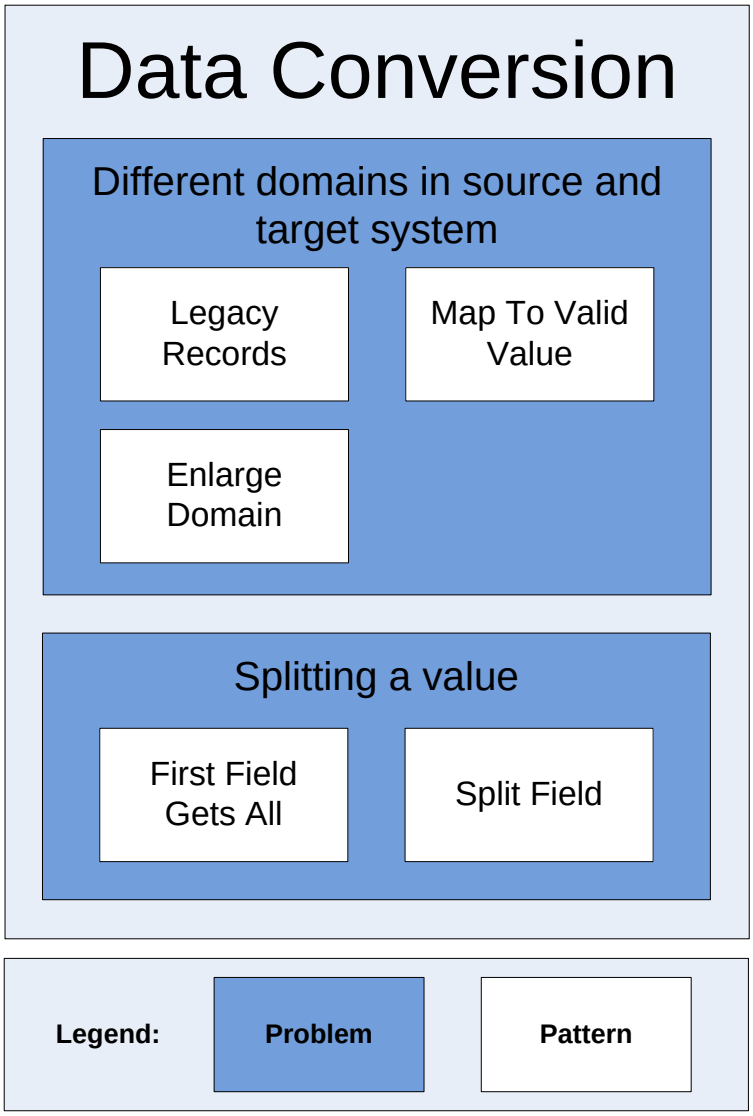


Figure 3.1: Data Conversion pattern language

Problem 1: Different domains in source and target system

Example

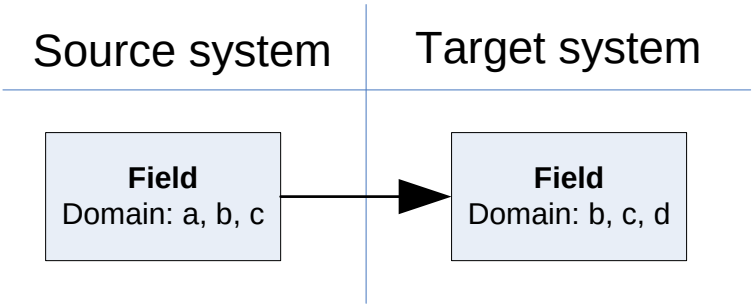
A university has an information system that stores the courses its students have completed and what marks the students received for the exams they made. For every exam the database contains a record containing the student number, the course number and the mark the teacher has given. The mark is stored as a natural number and can have values of zero or above.

When the university develops a new system to replace the old information system, they design the storage of marks a little differently. The designers reason that marks can only have values between 1 and 10. Zero or anything above 10 should not be given. To make sure this cannot be done by accident the database restricts the allowed value of marks to 1 to 10.

For the conversion designers this is a problem. The source system could contain values of 0 and greater than 10, but these cannot be stored in the target system.

Abstraction

In this problem, a single field from the source system is converted to a single field in the target system. The domain of the field in the source system differs from the domain of the field in the target system. This means that in the source system, this field can have values that are not allowed as values in the corresponding field in the target system.



Remark about cardinality

For simplicity, this problem is presented with cardinality ‘one to one’, but cases with other cardinalities would be very similar. In case of a ‘many to one’ cardinality, the domain in the source system is not relevant, but the domain that emerges when the fields of the source system are combined is. In a ‘one to many’ or ‘many to many’ case this problem may occur for each of the target fields. Different solutions from different patterns may be applied to the different target field where the problem of mismatching domain occurs. For example, the domain of one of the target fields may be enlarged (Enlarge Domain) while the invalid values may be mapped onto the target domain for another target field Map To Valid Value).

Patterns that solve this problem are:

Legacy Records provides a method that can guarantee completeness.
Enlarge Domain focuses on completeness with less effort.
Map to Valid Value focuses on effort and requires no changes in the target system.

The following table summarizes the scores of the patterns on the eight quality attributes:

	Completeness	Technical impact	Functional impact	Maintainability	Effort to implement	Understandability	Data integrity	Traceability
Legacy Records	+	-	□	-	-	-	+	+
Enlarge Domain	+	□	-	+	□	+	+	+
Map To Valid Value	-	+	+	+	+	□	□	-

Legacy Records

Intent

Convert fields from the source system that contain values that are not allowed in the target system (has a different domain) without losing information by marking records as being 'legacy'.

Motivation

Consider the mark registration system of the example. Suppose the university is required by law to store all marks of students for a number of years after their graduation. So when the marks are converted to the new marks registration system, all marks have to be brought to the target system without modification.

However, the older marks are only stored for archiving purposes. It is unlikely they have to be accessed very often, and they do not have to be changed later on. So the old marks have to be stored, but they do not have to be stored in the same way as new marks, and do not need to be as accessible as new marks.

With Legacy Records all marks can be converted unchanged, without altering the domain (allowed values) for new marks.

Applicability

Use Legacy Records when completeness of the conversion is of importance and loosing data is not an option. Legacy Records can only be applied when changing the target system is possible.

Solution

In this solution, we do not change the values that the target field can hold. Instead, we introduce the concept of a *legacy record*. A record in the target system becomes a legacy record if a field in the source system is converted to a field in the record but contains a value that is not valid for that field in the target system.

In order to store this information, two new fields are added: 'isLegacy' and 'legacyDetails'. The first stores whether the record is a legacy record, the second stores the invalid information that cannot be stored in the field that should contain that information. When the source field contains a value that is also valid in the target system, we simply set the target field to that value. Additionally, the 'isLegacy' field is set to 'false' and the 'legacyDetails' field is left empty.

When the source field contains a value that cannot be converted to the target field, the target field is left empty. The 'isLegacy' field is set to 'true' and the field 'legacyDetails' is set to the value of the source field.

Note that, because new fields are added to the target system, a 'one to one' mapping becomes a 'one to many' mapping when applying Legacy Records. Such a mapping may be best described using the 'push' fashion.

Example

In the example of the marks registration system, this solution means that we add columns 'isLegacy' and 'legacyDetails' to the table holding the marks. For all marks between 1 and 10, we copy the mark to the target field, set the value of 'isLegacy' to false and leave the field 'legacyDetails' empty. When we encounter a mark of 0 or above 10, we leave the mark in the target system empty, set 'isLegacy' to true and copy mark to the field 'legacyDetails'.

Now all marks, including invalid ones, can be converted to the target system, without removing the restriction that new marks have to be between 1 and 10.

Suppose the source system stored the marks in the following database table:

courseId	**studentId**	**mark**
c123	s074589	7
c123	s045944	4
c123	s044846	11
c123	s089657	0

The target system has uses the same columns, but we add 'isLegacy' and 'legacyDetails'. The information from the table above would then be converted to the following table in the target system:

courseId	**studentId**	**mark** constraint: value >=1 and value <= 10	**isLegacy**	**legacyDetails**
c123	s074589	7	false	null
c123	s045944	4	false	null
c123	s044846	null	true	11
c123	s089657	null	true	0

Implementation

The following steps are needed to implement the Legacy Records pattern in Java:

- Two classes (SourceRecord and TargetRecord) are defined for the source table and the target table. These classes hold instance variables that mirror the columns of the database tables.
- For every record in the source table, an object of type SourceRecord is constructed.

- For every SourceRecord object the appropriate TartetRecord object is constructed. The TargetRecord isLegacy field is set to true if the SourceRecord holds values that are not allowed in the target system. In that case the legacyDetails field is filled with that value.
- Every TargetRecord object gets written to the target database.

The example could be implemented as follows:

```
public void runConversion()
{
   Vector<SourceRecord> sourceRecords = DataFile.getSourceRecords();
   Vector<TargetRecord> targetRecords = new Vector<TargetRecord>();

   for( int i = 0; i < sourceRecords.size(); i++ )
   {
      TargetRecord tr = transformation( sourceRecords.get( i ) );
      targetRecords.add( tr );
   }
   DataFile.storeTargetRecords( targetRecords );
}

public TargetRecord transformation( SourceRecord sr )
{
   int courseId = sr.getCourseId();
   int studentId = sr.getStudentId();
   int mark = sr.getMark();
   boolean isLegacy = false;
   int legacyDetails = 0;

   if( mark < 1 || mark > 10 )
   {
      isLegacy = true;
      legacyDetails = mark;
      mark = 0;
   }

   return new TargetRecord( courseId, studentId, mark, isLegacy,
                legacyDetails );
}
```

Quality attributes

Completeness: The upside of this solution is that we are able to convert all information from the source system, also the invalid information, without losing any data. Score: '+'.

Technical impact on target system: The target system needs to be changed significantly. First of all, the fields 'isLegacy' and 'legacyDetails' must be introduced in the database. The system must be altered so it can handle these new fields and stores information in the database correctly. Finally, there must be some way to view the legacy information which requires changes in the logic and user interface. Score: '-'.

Functional impact on target system: This solution has an impact on the target systems functionality. It does require changes to the user interface, but new

information is not affected by the changes: it is still bound by the limitations the system was designed with.

A problem arises when the target field cannot be empty or when it is used in calculation by the system because when using Legacy Records, the field may remain empty when the record is marked 'isLegacy'. Solving this problem may introduce changes to the functionality of the target system. Score: '□'.

Maintainability: The solution introduces extra fields in the database, which also introduces extra maintenance. In the future, the use of legacy fields may seem odd and unwanted. Score: '-'.

Effort to implement: This solution is a lot of work to implement. The system needs to be changed in various locations. Score: '-'.

Understandability: The solution is complicated and may be difficult to explain. Score: '-'.

Data integrity: There are no integrity issues with this solution. Score: '+'.

Traceability: All the original values from the source system are copied to the target system, either to the target field or the 'legacyDetails' field. Thus, all records and values can be traced back to their origin. Score: '+'.

Related patterns

The patterns Enlarge Domain and Map to Valid Value provide solutions to the same problem. Enlarge Domain focuses on completeness with less effort while Map to Valid Value minimizes the impact on the target system.

Enlarge Domain

Intent

Convert fields from the source system that contain values that are not allowed in the target system (has a different domain) without losing information.

Motivation

Suppose the marks from the marks registration system in the example all need to be converted unchanged, i.e. the completeness of the conversion of these marks is very important. Because some of the marks may have values that are outside of the domain of the target system are of students that are not yet graduated, these marks need to be fully accessible; new marks and old marks cannot be treated differently. The restricted domain in the target system, limiting allowed marks to 1 to 10, is not crucial to the operation of the target system. So, the completeness of the data conversion takes priority over the functionality of the target system.

Applicability

Use Enlarge Domain when completeness of the conversion is of importance and loosing data is not an option. Enlarge Domain can only be applied when changing the target system is possible and functional changes to the target system are allowed.

Solution

The problem is that the field that needs to be converted can have values in the source system that are invalid in the target system. This can be solved by making these values valid in the target system. This means that the domain (set of valid values) in the source system must be a subset of the domain (set of valid values) in the target system.

This means that any constrains in the database limiting the allowed values in that field have to be removed, but also that in the systems' logic such constrains must be altered. The user interface must be able to show values other than 1 to 10 and the user must be able to enter values other than 1 to 10.

Example

Suppose that the marks registration system from the example stores the marks in the following database table:

courseId	studentId	mark
c123	s074589	7
c123	s045944	4
c123	s044846	11
c123	s089657	0

The target system uses a similar table, except that a database constraint limits the 'mark' column to values greater or equal to 1 and smaller or equal to 10. Applying the solution means removing this constraint. The target system now accepts any numeric value for the marks. Now, every mark can be converted to the target system because any value in the source system is also valid in the target system. The resulting table in the target system looks the same as the original table in the source system.

Implementation

This pattern does not contain an implementation as there is no need to actually transform the data. After removing the constraint limiting the allowed values from the target system, the data can be moved from source to target unmodified.

Quality attributes

Completeness: With this solution, all data can be moved to the target system. The solution is therefore complete. Score: '+'.

Technical impact in target system: For this solution to be implemented, the target system needs to be changed. The change is not very big as only the list of allowed values needs to be enlarged. Score: '□'.

Functional impact on target system: This solution means that the target system becomes functionally different from how it was designed in the first place. The restriction on valid values was put into place for a reason; in the example the designers wanted to only allow marks between 1 and 10 so teacher could not register invalid marks by accident. This safety measure would have to be removed to enable the conversion with this solution. Score: '-'.

Maintainability: The solution has no maintainability issues. Score: '+'.

Effort to implement: This solution is moderately difficult to implement. It is important to check whether changing the number of allowed values has any impact on the rest of the system. Other parts may be expecting only the original range. Such issues must be resolved. Score: '□'.

Understandability: The solution is easy to comprehend and can by explained without problems. Score: '+'.

Data integrity: There are no integrity issues with this solution. Score: '+'.

Traceability: Traceability is excellent since the data can be copied directly from the source system to the target system. Score: '+'.

Related patterns

The patterns Legacy Records and Map to Valid Value provide solutions to the same problem. Legacy Records provides another method that can guarantee completeness with a smaller impact on the target systems functionality. Map to Valid Value focuses on effort and requires no changes in the target system.

Map To Valid Value

Intent

Convert fields from the source system that contain values that are not allowed in the target system (has a different domain) without changing the target system.

Motivation

Consider the marks registration system of the example. The target system restricts marks to the values 1 to 10, the source system does not. But in fact, marks were always supposed to be between 1 and 10. It is determined that only very few marks break this rule, and all of these marks belong to students that are already graduated. It is decided that these marks are not very important, and that the quality of the new system has priority over the correctness of these few exceptions.

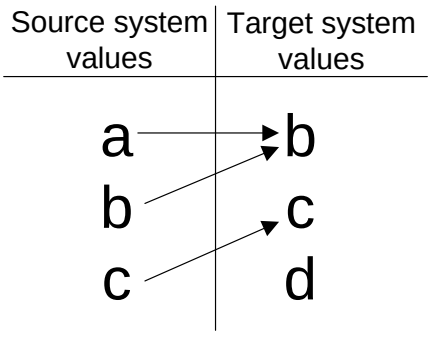
Applicability

Use Map To Valid Value when changes in the target system are not possible or when a quick and easy solution is more important than completeness.

Solution

This solution means that we convert all instances that have a value in the source system that is also valid in the target system without change. For every value that could occur in the source system but is invalid in the target system we choose another value that is valid in the target system. So every value in the domain of the source field is mapped on a value in the domain of the target field.

What values these invalid values should be mapped on is often an arbitrary decision. When the values have an ordering, such as the marks in the example, it is best to choose the closest valid value. When there is no such closest value, the designer should make a choice.



Mapping values

Be aware that Map To Valid Value may lead to internal inconsistencies in the target system when the source field is used for other mapping as well. Suppose, for example, that the mark '11' is mapped onto the value 'unknown'. But for another field containing whether the student has passed the course it is reasoned that any value higher than 5 is considered a 'pass'. In that case students can be registered passed with unknown marks in the target system. That may be an undesirable situation. If these mappings are described in a 'push' fashion, there is less risk for such accidental inconsistencies.

Example

In the example, all registered marks with values between 1 and 10 can be converted to the target system without any problem. Only those marks that have different values cannot be converted. The number of problematic marks will likely be very low, as by far the greatest number of marks are in the 1 to 10 range.

Therefore, it could be argued that it's not such a big loss to lose a bit of information on only those marks that are not between 1 and 10. Thus we could solve this problem by converting all marks with values below 1 in the source system to a value of 1 in the target system and converting all marks with values above 10 in the source system to 10 in the target system. All marks between 1 and 10 are converted without change.

Suppose that the marks registration system from the example stores the marks in the following database table:

courseId	**studentId**	**mark**
c123	s074589	7
c123	s045944	4
c123	s044846	11
c123	s089657	0

The target system uses a similar table, but has a constraint that limits the 'mark' column to values between 1 and 10. Converting the data from the table above to the target system would give the following table:

courseId	**studentId**	**Mark** Constraint: value >= 1 and value <= 10
c123	s074589	7
c123	s045944	4
c123	s044846	10

c123	s089657	1

Implementation

The following steps are needed to implement the Map To Valid Value pattern in Java:

- Two classes (SourceRecord and TargetRecord) are defined for the source table and the target table. These classes hold instance variables that mirror the columns of the database tables.
- For every record in the source table, an object of type SourceRecord is constructed.
- For every SourceRecord object the appropriate TartetRecord object is constructed. Values that are not allowed in the target system are mapped on values that are valid.
- Every TargetRecord object gets written to the target database.

The example could be implemented as follows:

```
public void runConversion()
{
   Vector<SourceRecord> sourceRecords = DataFile.getSourceRecords();
   Vector<TargetRecord> targetRecords = new Vector<TargetRecord>();

   for( int i = 0; i < sourceRecords.size(); i++ )
   {
      TargetRecord tr = transformation( sourceRecords.get( i ) );
      targetRecords.add( tr );
   }
   DataFile.storeTargetRecords( targetRecords );
}

public TargetRecord transformation( SourceRecord sr )
{
   int courseId = sr.getCourseId();
   int studentId = sr.getStudentId();
   int mark = sr.getMark();

   if( mark < 1 )
      mark = 1;
   else if( mark > 10 )
      mark = 10;

   return new TargetRecord( courseId, studentId, mark );
}
```

Quality attributes

Completeness: A consequence of this solution is the loss of some data. If values ‘a’ and ‘b’ in the source system are both mapped to the value ‘b’ in the target system, it is impossible to differentiate between them later. So only use this solution when the affected data is not important. Score: ‘-’.

Technical impact on target system: The strong point of this solution is that there is no need to make changes to the target system at all. Score: '+'.

Functional impact on target system: The target systems functionality remains unchanged. Score: '+'

Maintainability: The solution has no maintainability issues. Score: '+'.

Effort to implement: This solution is easy to implement. All that is needed is the mapping of values from the source to the target system. Score: '+'.

Understandability: The understandability of Map To Valid Value depends on the complexity of the mapping. The example used a simple mapping, but when hundreds of different values are mapped to hundreds of other values in the target system it becomes very hard to understand. Good knowledge of the meaning of all the values is required to understand the mapping. Score: '□'.

Data integrity: There are no integrity issues with this solution, as long as the mapping is well thought out. Mapping a value to another value could result in conflicting information if it is mapped to a wrong value. Score: '□'

Traceability: The target system contains different values than the source system. These cannot be traced back to their original values in the source system. Score: '-'.

Related patterns

The patterns Legacy Records and Enlarge Domain provide solutions for the same problem without sacrificing completeness.

Problem 2: Splitting a value

Example

An insurance company stores the insurance policies it has sold in an information system. In this system, an insurance policy has one or more coverages. Each coverage contains part of the provisions of the policy. For every coverage the system stores a certain premium. The total premium the customer has to pay for the policy is the sum of the premiums of the coverage.

This data has to be converted to a new information system. This new system also stores one or more coverages for each insurance policy. But, the target system has other coverages than were used in the source system. Not every coverage of the source system is converted to a single coverage in the target system, some source coverages are converted to multiple target coverages.

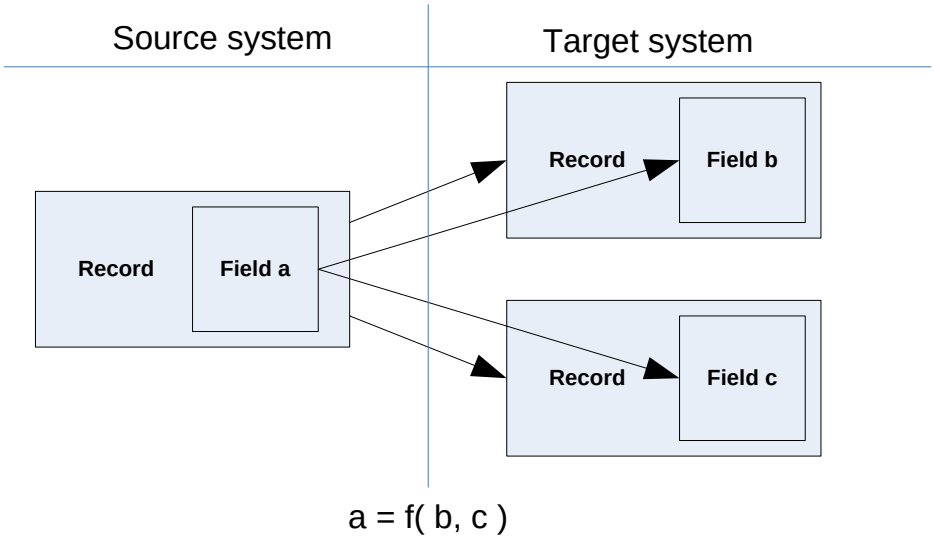
This leads to the question how to convert the premiums of the coverages from the source system. When one source coverage is converted to one target coverage, it seems the most logical choice to make the target coverage have the same premium as the source coverage. But what to do when a source coverage is converted to multiple target coverages?

Abstraction

In this problem we have a field with a certain value in the source system. This field is part of a record that is converted to multiple records in the target system. So, this field in the source system is converted to multiple fields in the target system.

In many cases, the value of the source field can be copied to all the target fields. But in this case the field will be used in some combination, for example a summation. This combination of the target fields in the must give the same result as the field in the source system had.

This means that the value of the source field has to be ‘split’ in some way over the target fields.



Patterns that solve this problem are:

First Field Gets All is a good solution but it does introduce a lot of empty fields.
Split Field does not work well on non-numeral values.

The following table summarizes the scores of the patterns on the eight quality attributes:

	Completeness	Technical impact	Functional impact	Maintainability	Effort to implement	Understandability	Data integrity	Traceability
First Field Gets All	+	+	+	+	+	+	□	□
Split Field	+	+	+	-	□	□	□	-

First Field Gets All

Intent

First Field Gets All provides a simple way to split a field from the source system over multiple fields in the target system.

Motivation

Suppose that in the insurance database of the example, having some coverages without a premium is not a problem. Because, for example, customers are only presented the premium of the coverages combined. So the manner in which the original premium is split over the target coverages is not important., only the sum of the premiums is. Splitting the premium is undesirable because it limits the traceability: the original premiums of the source system are not present in the target system.

Applicability

Use this pattern when a field must be split and it is not a problem that certain fields in the target system remain empty.

Solution

A simple way to solve this problem is to copy the value of the source field to the target field of the first record, and leave the other target fields empty. The restraint that the combination of the values of the target fields must result in the value of the source field will be satisfied for almost all combinations, for example for summation or concatenation.

Example

In the example of the insurance company, a single coverage was converted to multiple coverages in the target system. Applying this solution, the first coverage in the target system would get the same premium as the premium of the coverage in the source system while the other coverages will get a premium of zero. The sum of the premiums in the target system will have the same value as the premium in the source system.

Suppose the coverages are stored in the following database table in the source system:

coverageId	type	premium
123	A	100
124	B	50
125	AB	200
126	AB	100
127	AB	51

All the coverages that have a 'combination' type (AB) are split into two coverages (A and B). This results in the following table in the target system:

coverageId	type	premium
123	A	100
124	B	50
125a	A	200
125b	B	0
126a	A	100
126b	B	0
127c	A	51
127d	B	0

As is easily seen, the sum of the premiums of two coverages is the same as the original one in the source system.

Implementation

The following steps are needed to implement the First Field Gets All pattern in Java:

- Two classes (SourceRecord and TargetRecord) are defined for the source table and the target table. These classes hold instance variables that mirror the columns of the database tables.
- For every record in the source table, an object of type SourceRecord is constructed.
- For every SourceRecord object one or more TartetRecord objects are constructed. If a SourceRecord should not be split, only one TargetRecord is made, holding the same values as the SourceRecord object. If the SourceRecord should be split, multiple TargetRecord objects are made. The first TargetRecord hold the entire value of the split field, the others are set to zero.
- Every TargetRecord object gets written to the target database.

The example could be implemented as follows:

```
public void runConversion()
{
   Vector<SourceRecord> sourceRecords = DataFile.getSourceRecords();
   Vector<TargetRecord> targetRecords = new Vector<TargetRecord>();

   for( int i = 0; i < sourceRecords.size(); i++ )
   {
      Vector<TargetRecord> trs = transformation( sourceRecords.get(i) );
      targetRecords.addAll( trs );
   }
   DataFile.storeTargetRecords( targetRecords );
}

public Vector<TargetRecord> transformation( SourceRecord sr )
{
   int coverageId = sr.getCoverageId();
   String type = sr.getType();
   int premium = sr.getPremium();

   char[] types = type.toCharArray();
   Vector<TargetRecord> trs = new Vector<TargetRecord>();

   for( int i = 0; i < types.length; i++ )
   {
      int trCoverageId = coverageId*10+i;
      String trType = ""+types[i];
      int trPremium;
      if( i == 0 )
         trPremium = premium;
      else
         trPremium = 0;

      trs.add( new TargetRecord( trCoverageId, trType, trPremium ) );
   }
   return trs;
}
```

Quality attributes

Completeness: The original value of the source field is copied to the target field, so all data is preserved. Score: '+'.

Technical impact on target system: This solution does not require any changes to the target system. Score: '+'.

Functional impact on target system: This solution does not have any impact on the target systems' functionality. Score: '+'.

Maintainability: This solution does not have maintainability issues. Score: '+'.

Effort to implement: Copying the original value to one field and leaving the other fields empty can be described in a simple data mapping. Therefore, this solution does take little effort. Score: '+'.

Understandability: This solution is simple to understand. Score: '+'.

Data integrity: A downside of this solution is that it leaves many fields in the target system empty or at zero, which is often an unnatural situation. In the example of the insurance company, a customer would get an insurance policy with one expensive coverage and one or multiple coverages that are for free. If the customer would cancel the expensive coverage a policy with only free coverages would remain, certainly an undesirable situation. This is not a problem in all cases, it largely depends on the specific situation the conversion designer encounters. Score: '□'.

Traceability: The fields that hold the original value can be traced back to their origin in the source system. The fields that remain empty, however, cannot. Score: '□'.

Related patterns

Split Field solves the same problem in a different way.

Split Field

Intent

Split Field provides a simple way to split a field from the source system over multiple fields in the target system.

Motivation

Consider the insurance database outlined in the example. Because customers can cancel single coverages while keeping other coverage unchanged, it should not be possible to have coverages that do not have a premium, i.e. free coverages. In every coverage records, the premium field should be filled with some value. Because a premium is an integer, splitting it is not difficult.

Applicability

Use this pattern when a field must be split and it is technically and functionally feasible to split the original value of the source field.

Solution

The value of the source field is split evenly over the target fields. Each target field gets a part of the value of the source field. The way this splitting is done depends on the combination that is used: the splitting has to be done in such a way that the combination of the parts results in the original value. The designer has to create an algorithm that splits the value. Special care should be given to avoid rounding errors when the target fields can only hold integer values.

This approach is difficult when it is unknown upfront in how many parts a record is split. When this number is not known it cannot be determined what the value of each of the targets fields must be.

When applying Split Field, the mapping can only be described in the ‘push’ fashion. When the mapping would be described in ‘pull’ fashion, it cannot be guaranteed that the correct fields get their share of the split value from the source system.

Example

In the example, the premium of the coverage in the source system would be split evenly over the coverages in the target system. If there are three target coverages, the premium of each of these will be one third of the original coverage. Summing these values will of course result in the original premium.

Suppose the coverages are stored in the following database table in the source system:

coverageId	type	premium
123	A	100
124	B	50
125	AB	200
126	AB	100
127	AB	51

All the coverages that have a 'combination' type (AB) are split into two coverages (A and B). This results in the following table in the target system:

coverageId	type	premium
123	A	100
124	B	50
125a	A	100
125b	B	100
126a	A	50
126b	B	50
127a	A	26
127b	B	25

Note that the premium of coverage 127 was not split evenly in order to avoid rounding errors.

Implementation

The following steps are needed to implement the Split Field pattern in Java:

- Two classes (SourceRecord and TargetRecord) are defined for the source table and the target table. These classes hold instance variables that mirror the columns of the database tables.
- For every record in the source table, an object of type SourceRecord is constructed.
- For every SourceRecord object one or more TartetRecord objects are constructed. If a SourceRecord should not be split, only one TargetRecord is made, holding the same values as the SourceRecord object. If the SourceRecord should be split, multiple TargetRecord objects are made. All TargetRecords hold an equal part of the value of the split field. The first or last of those receives the rounding difference.

- Every TargetRecord object gets written to the target database.

The example could be implemented as follows:

```
public void runConversion()
{
   Vector<SourceRecord> sourceRecords = DataFile.getSourceRecords();
   Vector<TargetRecord> targetRecords = new Vector<TargetRecord>();

   for( int i = 0; i < sourceRecords.size(); i++ )
   {
      Vector<TargetRecord> trs = transformation( sourceRecords.get(i) );
      targetRecords.addAll( trs );
   }
   DataFile.storeTargetRecords( targetRecords );
}

public Vector<TargetRecord> transformation( SourceRecord sr )
{
   int coverageId = sr.getCoverageId();
   String type = sr.getType();
   int premium = sr.getPremium();

   char[] types = type.toCharArray();
   int newPremium = premium/types.length;
   int correction = premium - newPremium*types.length;
   Vector<TargetRecord> trs = new Vector<TargetRecord>();

   for( int i = 0; i < types.length; i++ )
   {
      int trCoverageId = coverageId*10+i;
      String trType = ""+types[i];
      int trPremium = newPremium;
      if( i == types.length-1 )
         trPremium += correction;

      trs.add( new TargetRecord( trCoverageId, trType, trPremium ) );
   }
   return trs;
}
```

Quality attributes

Completeness: The original value of the source field is not preserved in the same form, but the information it carried is converted to the target system completely. Score: '+'.

Technical impact on target system: This solution does not require any changes to the target system. Score: '+'.

Functional impact on target system: This solution does not have any impact on the target systems' functionality. Score: '+'.

Maintainability: If one of the target fields is removed later on, combining the resulting fields may give a badly formed result. This solution requires greater care when manipulating the resulting data. Score: '-'.

Effort to implement: The effort it takes to split a value depends on the complexity of the algorithm that calculates the different values. Score: '□'.

Understandability: This solution can be more difficult to understand when non-numeric values are concerned. Splitting and combining such values can be complicated. Score: '□'.

Data integrity: The upside of this solution is that every target field gets its 'share' of the source field. None of the target fields will be empty, resulting in a natural situation in the target system. As a downside, spitting a value can be dangerous. For example, if the source field contains a comma separated list, it is possible to split this list and divide it over the target fields. Every field will contain a part of the comma separated list. When these partial lists are combined, the result depends on the order in which they are concatenated. If it cannot be guaranteed that this order is always the same, it is uncertain that the combination will result in the same value as the source field had. Score: '□'.

Traceability: When the values are split, it becomes difficult to trace them back to their origin on the source system. Score: '-'.

Related patterns

First Field Gets All provides a solution for the same problem where the original value is not split.

Chapter 4

Validation and Evaluation

This chapter contains the validation and evaluation of the solution proposed in chapter 3. In the validation we will discuss whether the solution satisfies the acceptance criteria of section 2.5. In the evaluation we will ask ourselves if this solution really solves the problem we presented in the problem statement (section 1.2).

4.1 Validation

In section 1.2 we discussed that the problem with data conversion is the lack of reusability of design experience. Because data conversion design is comprised of making similar decisions over and over again, we need a way to document and reuse the things we learn in order to make data conversion design more efficient.

Although Pree and Sikora in 1997 already thought to see elements of a hype in the spread of design patterns (PS97), the use of patterns has not seen a decline. On the contrary, patterns are everywhere and it is widely accepted that design patterns are a solid way of enabling reuse of design experience that leads to better designs in less time. As Gamma et. al. put it: "Design patterns help a designer get a design right faster" (GHJV95). Shaloway and Trott also explain how the use of design patterns lead to a higher quality design (TS02). But does this also apply to the use of design patterns in data conversion? We are confident that, since data conversion design is a process with many repeating design decisions, it is a field where design patterns can be applied especially well.

In this section we compare the proposed solution with the acceptance criteria from section 2.5. We discuss to what extend the solution complies with these criteria one by one.

AC 1: The solution should be able to document design experience about data mapping.

We saw in chapter 2 that data mapping is the best place for successful documentation and reuse of data conversion design experience. In chapter 3 we provided five patterns that address data mapping problems. Although it is likely possible to make patterns for other areas of data conversion as well, the solution certainly captures data mapping design experience.

AC2: The library with design experience should be searchable.

Each pattern has a clear name and short introduction sections that explains the intent of the pattern. The problem description have names as well that can be used to quickly find the right problem and pattern. If more patterns are developed, searching for the right pattern will become more difficult, however. This criterion is therefore reached partly; it is possible to search the library with design experience, but it may become more difficult to search effectively in the future.

AC3: The solution should give insight of the consequences of design decisions.

Quality attributes give insight in the consequences of applying the solution given in a pattern. It enables designers to predict what strong and weak point their implementation of the solution will have.

AC4: The solution should help to objectively compare alternatives and select the best one in the current context.

By providing multiple patterns giving solutions to each problem and developing quality attributes to describe the strong and weak points of each of those solutions, designers can select the solution that would fit their particular problem best.

AC5: The solution should facilitate communication between designers and other stakeholders.

"Design patterns provide a common vocabulary for designers to use to communicate, document, and explore design alternatives" (GHJV95). The data conversion patters presented here will also provide a common vocabulary for designers, making it easier to communicate about their design decisions. Whether this will also affect communication with other stakeholders, like users of the system and managers, is questionable. The patters could be a means of explaining design decisions to them and involving them in the design process.

AC6: Design experience should not be too specific but have an adequate level of granularity.

The patterns presented in chapter 3 contain general design experience that is not specific to design methods, programming or modeling languages, or problem domains of the converted data. The design experience captured in the patterns can be applied in a wide range of projects.

Based on the discussion of the acceptance criteria above, we can conclude that the solution presented in chapter 3 satisfies the acceptance criteria almost completely. The largest shortcoming is the lack of a mechanism to aid in searching the pattern library if it grows in the future. While looking through the pattern names and searching for keywords works well to some extent, it is not perfect.

4.2 Evaluation

This section is about the last step in the engineering cycle: the evaluation. While we saw in section 4.1 that the solution of data conversion patterns does satisfy the acceptance criteria almost completely, here we ask ourselves the question whether the solution will really solve the problem. Are the patterns acceptable for conversion designers, will it really enable them to reuse design experience and will it improve the efficiency of designing data conversion?

We answer these questions by presenting the data conversion patters to a group of data conversion experts and discussing their view of the patterns.

4.2.1 Evaluation session

The data conversion designers that gave their opinion for this evaluation are all working for Quinity, the company where this research was performed. However, they had not been involved in the development of data conversion patterns, prior to this session they had had no influence on the research. Each of the designers had been working on data conversion design for over a year and would, when asked, qualify him- or herself as 'expert'. The designers had extensive experience with design patterns in other areas. Especially Functional Design Patterns as described by Snijders (Sni04) were well known.

The goal of this evaluation session was twofold: getting feedback in order to improve on the result presented to the experts and evaluating data conversion patterns. The results of these two objectives are presented separately in sections 4.2.2 and 4.2.3. Note that the feedback given by the experts has already been included in the results presented in chapter 3. The version presented to the experts did not structure the patterns in a pattern language but contained an unstructured set of patterns. The patterns did not include the quality attribute 'traceability' and the names and values of some other quality attributes were slightly different. These changes will be discussed in more detail in section 4.2.3.

During the session, the experts were asked a set of questions. We chose this setup so the experts could react on each other's statements and opinion. Discussion between the experts was encouraged. We expect that we would get more results in this way than through interviewing the experts separately or through surveys.

During the session the following questions were asked:

1. Do you think the problem statement of this thesis, namely the lack of reuse in data conversion design, is indeed a problem that needs to be addressed?
2. Do you think that patterns could be used to enable reuse of design experience on data conversion?
3. Do you think the patterns as presented in chapter 3 contain the right information to enable reuse of experience?
4. What is your opinion about the use of quality attributes?
5. Did we include all the important quality attributes and are all included attributes relevant?
6. Do you think the proposed method will help you make better and faster designs?

For each pattern, the following questions were asked:

7. Dou you recognize the problem addressed in this pattern?
8. Can you relate to the solution provided by the pattern?
9. Were the scores of quality attributes chosen correctly?

10. Do you think this pattern is useful and helpful to a data conversion designer in practice?

With question 1 we tried to find out if the experts agreed on the existence of the problem statement. If they did not see the problem, they could not evaluate the solution either. Questions 2 to 5 were intended to get their opinion on the different parts of the proposed solution. Question 6 had to find out whether the solution indeed improved efficiency of data conversion design, the goal of this research. The quality of the separate patterns was assessed with questions 7 to 10.

4.2.2 Expert opinions

When asked whether they recognized the problem addressed by this research, each of the designers was convinced that the problem indeed existed. They have all encountered and acknowledged the problem of lack of reuse of experience. They were genuinely enthusiastic that a solution to that problem was presented. They thought that the use of design patterns in this area would be a good idea and did not have doubts that design patterns could indeed provide a way of reusing design experience. They said that sharing design experience was currently often done at lunch, and a structured method for doing this would indeed improve design quality and speed. They mentioned another reason for using patterns: when using the same solutions in every design, it becomes easier for programmers to implement the design.

The designers were content with the pattern template. They agreed that it provides all the information they need to use the pattern. While they would not have problems learning to use a new pattern type, they noted that they would be interested in some information that is usually included in Functional Design Patterns. We will look into this further in section 4.3.

The use of quality attributes is appreciated by the designers. One of them noted "this really makes such things explicit, you do not have to pick them up between the lines." With the addition of 'traceability' the designers agreed that all important quality attributes were there. All quality attributes were deemed relevant.

When asked about the individual data conversion patterns, the designers recognized the problems that were addressed in the patterns and most could relate to the solutions provided in the patterns and agreed that they could be applied in a data conversion project. The designers stressed the importance of updating the patterns when used in practice as such values can never be set precisely right at the time of writing.

Most of the designers indicated that they would immediately start using the data conversion patterns if they had the opportunity.

4.2.3 Improvements after evaluation session

The experts had several pointers to improve the data conversion patterns as presented to them. In this section we will discuss the changes that were made to the solution after the evaluation session. Note that these changes are already incorporated in the solution presented in chapter 3.

Most notably, the designers saw a weak point in a lack of structure between the patterns. In the version they received the patterns were five unrelated documents. They were not yet grouped according to the problem type they solved. The experts missed oversight to see how the patterns were related and which patterns solved the same problem. To address this critique the patterns were embedded in a pattern language, grouping the patterns according to the problem they solve and making explicit what the strengths and weaknesses of the patterns are compared to the other patterns solving the same problem.

The experts missed one quality attribute. They indicated that it is important to trace data back from the target system to the source system in order to check whether all data has been converted and to find out what went wrong in case of a problem. The attribute 'traceability' was added to meet this request.

The name of two of the quality attributes was changed to better fit with the jargon used by data conversion designers. The attribute "change in target system" was renamed to "technical impact on target system" and "change in target systems' functionality" was changed in "functional impact on target system".

In the pattern 'Map To Valid' the experts did not agree with the scores on the quality attribute 'Understandability'. They argued that this pattern can be difficult to understand when the mapping gets very large. The score on this attribute has been lowered from '+' to '□'. In the pattern 'Split Field' the score on 'Effort to implement' has been lowered from '+' to '□' because the experts explained that this pattern can take more effort to implement when the splitting algorithm is complicated.

4.2.4 Conclusion on expert opinions

After we improved a few weaknesses pointed out by the experts, they approve of the introduction of patterns in the domain of data conversion design and believe that it will help them make better designs faster. The weaknesses that were improved were making explicit the relations between the patterns, the introduction of the quality attribute 'traceability', the renaming of two quality attributes, and a slight change in the scores of a few quality attributes in some patterns.

The idea of presenting data conversion patterns as functional design patterns would have such a big impact on the presented solution that it was decided not to fully work out this idea. In section 4.3 we will explore this idea but we will not implement it as rewriting all of the patterns would take too much time.

The positive feedback from the designers strengthens the confidence that the proposed use of data conversion patterns indeed helps to solve the problem of lack of reuse in data conversion design.

4.3 Functional Design Patterns

Quinity already makes use of design patterns in the form of Functional Design Patterns. These patterns, described by Snijders (Sni04), enable the reuse of functionality. These patterns are of great help to Quinity. The information systems Quinity develops, mainly administrative

systems for insurance companies, share a lot of functionality. For example, they all work with personal information about customers, they all have systems to calculate monthly premiums, they all have workflows with different stages of acceptance of applications and they all have intermediaries and provisions. Functional Design Patterns provide a way to document such functionality and reuse abstract concepts in the functional design.

Functional Design Patterns share many of the goals of Data Conversion Patterns. Both try to enable reuse of design experience on a conceptual level rather than on a technical level like most pattern types. In the last few years, multiple studies into Functional Design Patterns have presented additions to the patterns that could be useful for Data Conversion Patterns as well.

In this section we look into the possibility of presenting a Data Conversion Patterns as a Functional Design Pattern and analyze the upsides and the drawbacks. We do this by investigating the differences between Functional Design Patterns and Data Conversion Patterns and discussing what information is missing in the Data Conversion Patterns that should be present in Functional Design Patterns. In each of the coming sections we look into one such difference.

If data conversion patterns could be presented as Functional Design Patterns, doing so could make it easier for Quinity to use patterns in the design of data conversion. They already have experience in using Functional Design Patterns and their engineers are familiar to this type of pattern.

4.3.1 Domain and aspect level

As part of their research into Functional Design Patterns, Kleerekoper (Kle07) and Bosman (Bos07) both describe how functional design patterns can be categorized in aspect level and domain level patterns.

Domain level patterns capture knowledge that is specific to a certain domain or problem area. They describe functionality that is bound to a certain domain and cannot be applied in applications in another domain. For example, Snijders (Sni04) gives a set of patterns for pension calculations. Another domain level pattern could describe how to design the stacking of insurance policy versions.

Aspect level patterns are not bound to a specific domain, but describe functionality that can be applied in any application, regardless of its domain. The conversion patterns described in section 4.3 can be classified as aspect level patterns: they do not capture domain knowledge.

Documenting domain knowledge in Data Conversion Patterns is an interesting idea. Different information systems in the same domain will often store the same kind of information. For example, insurance databases will always store information about intermediaries. We could make a conversion patterns specifically for converting intermediaries to a target system. Such a pattern can be much more detailed than a general pattern without domain knowledge can be. Thus, domain level Data Conversion Patterns are most likely possible and useful. Not every Data Conversion Pattern needs domain knowledge to be useful, but it should be possible to create Data Conversion Patterns that do include relevant domain knowledge. Validating this by actually creating domain level data conversion patterns is outside the scope of this research.

4.3.2 Technical and implementation details

Functional Design Patterns describe functionality on a conceptual level. Van Montfort (Mon06) explores the addition of technical details and implementation details to Functional Design Patterns. He argues that reoccurring functionality will often lead to the same technical design decisions and even to similar source code. When a Functional Design Pattern is applied in the functional design of an application, its technical details could be applied in the technical design of the application and the implementation details could be applied in the realization phase. While this is not always possible, for some patterns it is a useful addition.

The same reasoning can be used for Data Conversion Patterns. Certain conversion patterns may describe a relationship between data in the source and target systems that always requires the same technical considerations. For example, when the mapping of a certain field depends on the value of a field in a different table, that table must be accessible at that time (i.e. the source datafile containing the table must be read and be in memory). Adding such a detail to a Data Conversion Pattern could be a great help for the designer.

Source code could also be added to a Data Conversion pattern. Whether the code can actually be reused in a project depends on the programming language used and coding standards used in the project, but could otherwise form an example of how to implement the solution provided by the pattern. Adding code to a conversion pattern makes it somewhat like a standard component. This may work in some cases but will not be explored further in this thesis.

4.3.3 Incorporation in software development

Bosman (Bos07) presents a method to incorporate Functional Design Patterns in software development. This is interesting as patterns are often used without explicitly thinking about how they can be used within a certain software development method.

The processes of data conversion and developing an application are similar; they share many steps and phases: requirements gathering, design, realization, testing (Pur99). Therefore, it is likely that Bosman's method for incorporating Functional Design Patterns in software development can be adapted to incorporating Data Conversion Patterns in data conversion. However, this requires a good definition of the development process of data conversion which has not been explored and documented in great detail.

4.3.4 Language of Functional Design Patterns

Kleerekoper (Kle07) defines a language for Functional Design Patterns. The language consists of three parts: key terms, diagrams and the structure of a Functional Design Pattern.

Every Functional Design Pattern starts with a 'key terms' section. This section contains definitions of all the important notions that a reader should know before he can understand the pattern. For data conversion patterns, the key terms section could be used to define parts of the problem; what fields or records do the source and target system contain, what information is important in the pattern. This leads to a more formal definition of the problem of the data

conversion pattern. Some key terms could be used in every data conversion pattern, for example the relationships between records, fields and values.

The second part of the language is the description of diagrams. Multiple diagram types are defined, like flow diagram and timeline diagram. These could be used for data conversion patterns as well. There may be a need for more types of diagrams to express certain data conversion patterns but these could be added. The language already allows the addition of other diagram types to a pattern.

The last part of the language describes the structure of a Functional Design Pattern. The structure contains five sections. The first is the introductory section. This contains the goal, applicability, relation with other patterns and time estimate. Except time estimate, all of these were already included in the data conversion patterns. A time estimate may be hard to make, but it could be included in a data conversion pattern.

The second section is the covering section. This describes the key terms, parameters and graphical additions to the diagrams that are not normally used in Functional Design Patterns. As described above, the key terms section could be used for data conversion patterns. The parameters section could describe table names and columns used in the pattern. Graphical additions may be needed for some data conversion patterns that require special diagrams.

The third section is the functional design section. This contains the information function segments, the data pattern and the user interactions. This section can be seen as the solution provided by the pattern. The solution of the data conversion pattern can be presented here.

The fourth section is the realization section. This section provides technical design details and implementation details. As discussed in section 4.4.2, these may be worthwhile additions to data conversion patterns.

The fifth and last section is used to provide proof of certain properties of the pattern. One of the purposes of the example and the implementation section in data conversion patterns is to convince the reader that the solution really works. The proof section could therefore be used to provide examples and the implementation.

Concluding, the language for Functional Design Patterns is suited for holding data conversion patterns. Some sections of the template are not very relevant to data conversion and a new section is needed to contain the quality attributes.

4.3.5 Conclusion on Functional Design Patterns

The previous paragraphs each discuss an important part of the theory about Functional Design Patterns that differs from Data Conversion Patterns. Each paragraph concludes that is could be applied on Data Conversion patters as well, but with some restrictions and changes.

Therefore, with some adaption, a Data Conversion pattern could be represented as a Functional Design pattern if the stakeholders of the pattern are more familiar with that type.

Chapter 5

Conclusions

This section summarizes the results presented in this thesis. First we look back at the research questions we asked ourselves in chapter 1. After that we look at the conclusions we can draw based on the research presented in this thesis and the openings for future research.

5.1 Research question

In chapter 1 we asked ourselves a list of research questions. Here we look back at those questions and discuss how we answered those questions in the previous chapters. We start with the supporting research questions:

- What criteria should the solution meet for it to indeed increase efficiency?

In chapter 2 we developed a list of seven acceptance criteria that the solution would have to meet. In chapter 4 we saw that the solution presented in chapter 3 indeed satisfies these criteria.

- What are the elements of a data conversion design?

We explored and described data conversion in detail in chapter 2. We divided data conversion in export, data transformation and import. The different kinds of design were discussed and the main parts of the functional design of data conversion were explained. We focused on data mapping and decided to document design experience of data mapping during this research.

- How should data conversion design experience be documented?

In chapter 3 we explained that design patterns are a well established way of documenting design experience. We argued that design patterns would likely work in data conversion as well. The pattern template described in which form the design experience should be documented and we developed quality attributes to support a designer in selecting the pattern that the situation best.

- What design experience is available and can be recorded?

During this research project we looked for design experience about data mapping at Quinity that could be documented in design patters. This lead to a pattern language, presented at the end of chapter 3.

The main research question:

- "How can we improve the efficiency of making data conversion design by documenting design experience of data conversion?"

We answered this question by developing data conversion patterns that record experience about data conversion design.

5.2 Main conclusion

Data conversion design experience can be documented in data conversion patterns as described in this thesis. The patterns enable designers to reuse this design experience in other data conversion designs. This makes the design of data conversion more efficient: it leads to better designs or requires fewer resources in order to make a good design.

5.3 Further conclusions

Data conversion patterns as Functional Design Patterns

In section 4.2 we saw that we could present data conversion patterns as Functional Design Patterns. This enables us to include some of the parts that are already used in Functional Design Patterns such as the difference between domain and aspect level patterns and technical details. Such information can be relevant for patterns specifically targeted for a certain application domain or developer, although not all patterns will benefit from such information. It can also help designers that are already familiar with this type of pattern to learn to use data conversion patterns more easily.

Data conversion patterns describe "what", not "how"

In a data conversion project you need to design two things: what needs to be converted and how is that conversion being done. The data conversion patterns provided here describe the "what"; they describe what data should be converted to what place in the target system, and what modifications should be done during this conversion.

The patterns give no further insight in how this can be done. In one way, this is limiting because the patterns only enable reuse of a part of the design experience that designers have. Experience in how to execute data conversion is not recorded. On the other hand, this allows the patterns to be applied by many different designers working at different places. The patterns are not limited to certain development methods or conversion tools. The problems described in the patterns will be encountered by every data conversion designer and the solutions described can be use in any development method or tool. This focus on the "what" somewhat limits the usefulness of the patterns but also means the patterns can be used and applied in a broad context.

Patterns that describe “how” to implement data conversion will be more difficult to make than “what” patterns. Situations on a more technical level will not repeat as commonly. For example, when splitting records from the source system, there are many different places where information about which records should be split could be available. It could be in large tables supplied by the customer, or it could be determined by an algorithm, or it could be based on other information in the source system. A single “what” pattern could be made for all of these cases, while separate “how” patterns are needed for each case.

“How” patterns cannot be applied as broadly as “what” patterns and it will be more difficult to recognize repeating “how” solutions to base patterns on.

Structure is important

A long list of patterns has limited usefulness. Designers should be able to see the relation between patterns and the problems they solve. The more patterns are developed, the more structure is needed to enable designers to find the right patterns and decide what patterns to use in a particular context.

Wide applicability of patterns

Data conversion design has problems that come back very often. For example, almost every large data conversion project will somewhere have the problem that the source system contains values that are not allowed in the target system. This means that some data conversion patterns could be used very often, in nearly every project. We believe that the patterns given in this thesis are such patterns, describing problems and solutions that will keep coming up. Design patterns in the field of data conversion could be even more useful that design patterns in other fields, like technical design, have already proven to be.

Storage technology

The research in this thesis has been limited to the field of relational databases. Data conversion is not limited to relational databases, however. Data can very well be converted between hierarchical data storage, like XML databases, or object oriented databases, as well as between systems built on different storage technology.

While different tools and approach may be necessary when converting data to or from such systems, the idea of data conversion patterns applies just as well as to relational databases. Some data conversion patterns will be relevant to other storage technologies as well, while some patterns will be applicable with only one technology. The solutions in many patterns will be relevant to other technologies only with modifications. For example, the idea behind Legacy Records can be applied to XML data very well, but the way data is marked as being legacy differs. Instead of adding a column, a simple attribute to the node containing legacy data is sufficient.

Reuse already happens informally

Reuse and sharing of design experience of data conversion already happens informally. This was confirmed by the design experts in the evaluation session. Designers talk about their design decisions and show each other their ways of solving a problem. This clearly shows that designers indeed feel the need of reusing design experience. We believe that this informal

sharing and reuse of design experience is a good thing and that design patterns will not fully replace it, but add a stronger form of reuse next to it.

5.4 Future research

While this thesis includes five data conversion patterns, it would be much better for designers if there were more patterns available. We expect that the low availability of patterns will discourage designers from making use of the existing patterns because their usefulness has to outweigh the time investment in learning how to use and apply them. Creating new patterns will have a big impact on the usefulness of the pattern library and thus affect the balance between investment and return.
During this research we investigated design experience of a limited number of designers at a single company. Other designers in different places will certainly have different design experience which makes it likely that more patterns can be discovered.
Of course, having too many patterns will make it difficult for designers to find the right pattern and will make maintaining the pattern library a burden, but we believe that the sweet spot in the number of patterns has not yet been reached.

The patterns that exist today should be validated in practice, as this is the only real way of testing a design pattern. While the implementation of the patterns show they can work, only when applied on a real design will the shortcomings and inaccuracies become visible.

We did not investigate interaction between patterns, i.e. what happens when we apply multiple patterns in one location. For example, will it work if we split values using the pattern ‘first field gets all’ and put the result in the target system using ‘legacy records’? An interesting issue here are the quality attributes, is the combination of two patterns that both score ‘+’ on a certain quality attribute guaranteed to score well on that attribute as well? More knowledge in this area will allow designers to make better design decisions.

Data conversion patterns could be validated further in an experimental setting. We could let multiple teams create designs for the same data conversion project, half of the teams using data conversion patterns while the other half does not. Such a ‘test and control’ experiment would be difficult to perform, however. Firstly, how do you select multiple teams with the same level ‘design skill’? Secondly, to our knowledge, there is currently no method of measuring the quality of a data conversion design so comparing the results of the different teams would not be straightforward.

The quality attributes described in chapter 3 could be a starting point for a method to determine data conversion design quality. It would be interesting to have a way of measuring quality of these designs.

References

(ALT) Altova MapForce 2008, created by Altova GmbH, Wien, Austria. http://www.altova.com/products/mapforce/data_mapping.html

(APRS03) P. Avgeriou, A. Papasalouros, S. Retalis and M. Skordalakis: "Towards a Pattern Language for Learning Management Systems" in Educational Technology & Society, volume 6, issue 2, 2003, p. 11-24.

(BCV03) A. Bianchi, D. Caivano and G. Visaggio: "Iterative Reengineering of Legacy Systems", in IEEE Transactions on Software Engineering, volume 29, number 3, March 2003, p. 225-241.

(Ben95) K. Bennett: "Legacy Systems: Coping with Success", in IEEE Software, volume 12, issue 1, January 1995, p. 19-23.

(Bos07) Y.M. Bosman: "Incorporating Functional Design Patterns In Software Development", Master thesis, University of Twente, 2007.

(BS95) M. Brodie and M. Stonebraker: "Migrating Legacy Systems: Gateways, Interfaces and the Incremental Approach", Morgan Kaufmann, San Francisco, 1995.

(CDSS98) S. Cluet, C. Delobel, J. Siméon and K. Smaga: "Your mediators need data conversion!" in Proceedings of the 1998 ACM SIGMOD international conference on Management of data, p.177-188.

(Fow97) M. Fowler: "Analysis Patterns: Reusable Object Models". Addison-Wesley Professional, 1997. ISBN: 0201895420.

(Fow02) M. Fowler: "Patterns of Enterprise Application Architecture". Addison-Wesley Professional, 2002. ISBN: 0321127420.

(GHJV95) E. Gamma, R. Helm, R. Johnson and J. Vlissides: "Design Patterns. Elements of Reusable Object-Oriented Software". Addison-Wesley, 1995. ISBN: 0201633612.

(HMPR04) A.R. Hevner, S.T. March, J. Park and S. Ram: "Design Science in Information Systems Research" in MIS Quarterly, volume 28, number 1, March 2004, p. 77-105.

(ISO01) ISO/IEC-9126-1:2001, Information technology – Product Quality – Part1: Quality Model, International Standard ISO/IEC 9126, International Standard Organization, June, 2001.

(Kel97) W. Keller: "Mapping Objects to Tables: A Pattern Language" in Proceedings Of European Conference on Pattern Languages of Programming (EuroPLOP), 1997.

(Kle07) J. Kleerekoper: “Design of a Pattern Definition Language”, Master thesis, Utrecht University, 2007.

(Mon06) J.J.E. van Montfort: “Functional Design Patterns: de implementatie van model-gedreven functionaliteit in een object georiënteerde omgeving”, Master thesis, Technische Universiteit Eindhoven, 2006. *Dutch*.

(ORA) Oracle Migration Workbench, created by Oracle Corporation, Redwood Shores, USA. http://www.oracle.com/technology/tech/migration/workbench/index.html

(PER08) M.L. Ponisio, P. van Eck and L. Riemens: “Using Critical Problem Solving to Plan Inter-Organisational Co-operation in e-Customs” in Proceedings of the International Association for Development of the Information Society (IADIS) International Conference e-Commerce, July 2008, p. 28-39

(PS97) W. Pree and H. Sikora: “Design patterns for object-oriented software development”, in Proceedings of the 19th international conference on Software Engineering, 1997, p. 663-664.

(Pur99) S. Purba: “Data Management Handbook”. Published by CRC Press, 1999. ISBN: 0849398320.

(RD00) E. Rahm and H.H. Do: “Data Cleaning: Problems and Current Approaches” in IEEE Bulletin of the Technical Committee on Data Engineering, volume 23 number 4, December 2000.

(Ris99) L. Rising: “Patterns: a way to reuse expertise” in IEEE Communications Magazine, volume 37, issue 4, April 1999.

(RJ96) D. Roberts and R. Johnson: “Evolving Frameworks: A Pattern Language for Developing Object-Oriented Frameworks” in Proceedings of the Third Conference on Pattern Languages and Programming, 1996.

(SEI) Software Engineering Institute: CMMI – Capability Maturity Model Integration. http://www.sei.cmu.edu/cmmi/

(SHL75) N.C. Shu, B.C. Housel and V.Y. Lum: “CONVERT: a high level translation definition language for data conversion” in Communications of the ACM, Volume 18, issue 10, October 1975, p. 557-567.

(Sni04) J. Snijders: “Functional Design Patterns”, Master thesis, Utrecht University, 2004.

(ST02) A. Shalloway and J. Trott: “Design Patterns Explained: A New Perspective on Object-Oriented Design”. Addison-Wesley Professional, 2002. ISBN: 0201715945.

(SV08) A. Simitsis and P. Vassiliadis: “A method for the mapping of conceptual designs to logical blueprints for ETL processes” in Decision Support Systems, volume 45, issue 1, April 2008, pages 22-40.

(TAU) DataBridger, created by Taurus Software, Redwood City, USA. http://www.taurus.com/products/databridger.htm

(Vis01) G. Visaggio: "Ageing of a data-intensive legacy system: symptoms and remedies", in Journal of Software Maintenance: Research and Practice, volume 13, issue 5, September-October 2001, p. 281-308.

(VVS+01) P. Vassiliadis, Z. Vagena, S. Skiadopoulos, N. Karayannidis and T. Sellis: "Arktos: towards the modeling, design, control and execution of ETL processes", in Information Systems, volume 26, issue 8, December 2001, pages 537-561.

(Wie07) R. Wieringa: "Research and Design Methodology for Software and Information Engineers". University of Twente internal report, 2007.

(WNO) Word Net Online, Princeton University. http://wordnetweb.princeton.edu/

(WVE00) M. van Welie, G.C. van der Veer and A. Eliëns: "Patterns as Tools for User Interface Design", in Tools for Working with Guidelines: Annual Meeting of the Special Interest Group, 2000, p. 313-324.

(YA04) S.M. Yaboub and H.H. Ammar: "Pattern-Oriented Analysis and Design: Composing Patterns to Design Software Systems". Addison-Wesley Professional, 2004. ISBN: 0201776405.